วาจาจาวา – ตอนที่ ๓

Class และ Interface

สมชาย ประสิทธิ์จูตระกูล

Encapsulation

- เปิดเผยแต่บริการที่ให้ (API)
  ปกปิดการทำงานภายใน (implementation)
  - ง่ายต่อการ develop, test, optimize, use, understand, และ modify
- ปกปิดให้มากที่สุดด้วย access control
  - private, package-private, protected, public
- อะไรที่เป็น public และ protected ถือเป็นสัญญาว่าจะให้บริการตลอดไป
- ยิ่งเปิดเผยมาก ยิ่งเปลี่ยนแปลงยาก

Encapsulation : public final อาจไม่ปลอดภัย

- final fields ที่เป็น public ได้
  - final primitives
  - final reference ไปยัง immutable object

```
public class Rectangle {
  public final Point upperLeft, lowerRight;

  Rectangle(Point ul, Point lr) {
    upperLeft = ul;
    lowerRight = lr;
  }
  ...
```

```
public class Point {
  public int x, y;
  ... // mutable
}
```

```
Rectangle r = new Rectangle(new Point(2,3), new Point(8,9));
...
r.upperLeft = new Point(0, 1); // compilation error
r.upperLeft.x = 0;   // change r's upperleft
r.lowerRight.y = 1;  // change r's lowerright
```

Encapsulation : getter ก็อาจไม่ปลอดภัย

- getter ที่คืน reference ไปยัง mutable object ไม่ปลอดภัย

```
public class Rectangle {
  private final Point upperLeft, lowerRight;
  ...
  public Point getUpperLeft() { return upperLeft; }
  public Point getLowerRight() {
    return (Point) lowerRight.clone();
  }
```

```
Rectangle r = new Rectangle(new Point(2,3), new Point(8,9));
...
r.getUpperLeft().x = 0;   // changes r's upperleft
r.getLowerRight().y = 0;  // does not change r's lowerright
```

```
class Point implements Cloneable {
  public int x, y;
  public Object clone() {...}
  ...
}
```

Encapsulation : public final array ไม่ปลอดภัย

```
public class A1 {
  public static final String[] DAYS =
     { "SU", "MO", "TU", "WE", "TH", "FR", "SA" };
  ...
```

```
public class A2 {
  private static final String[] PRIVATE_DAYS =
     { "SU", "MO", "TU", "WE", "TH", "FR", "SA" };
  public static final List DAYS =
     Collections.unmodifiableList(Arrays.asList(PRIVATE_DAYS));
  ...
}
```

```
public class A3 {
  private static final String[] PRIVATE_DAYS =
     { "SU", "MO", "TU", "WE", "TH", "FR", "SA" };
  public static String[] getDAYS() {
    return (String[]) PRIVATE_DAYS.clone();
  }
  ...
}
```

Immutable Class : ลักษณะ

- คลาสซึ่งผลิตออปเจกต์ที่เปลี่ยนแปลงไม่ได้อีก
- มีเมท็อดที่สร้างออปเจกต์ใหม่ซึ่งมี state ต่างจากเดิม
- ตัวอย่างใน Java API
  - String, BigInteger, BigDecimal, wrapper classes

```
String s1 = "Java";
String s2 = s1.concat(" Programming");
String s3 = s2.toLowerCase();
// Java,Java Programming,java programming
System.out.println(s1 + "," + s2 + "," + s3);
```

```
BigInteger bi = new BigInteger("1000000000000000000000000");
for (int i = 0; i < 10; i++) bi = bi.add(bi);
bi = bi.setBit(0);
System.out.println(bi); // 1024000000000000000000000001
```

Immutable Class : ตัวอย่าง

```
public final class Complex {
  private final double re, im;

  public Complex(double re, double im) {
    this.re = re; this.im = im;
  }

  public float realPart() { return re; }
  public float imaginaryPart() { return im; }
  public Complex add(Complex c) {
    return new Complex(re + c.re, im + c.im);
  }
  public Complex subtract(Complex c) {
    return new Complex(re - c.re, im - c.im);
  }
  public Complex multiply(Complex c) {
    return new Complex(re * c.re - im * c.im,
                       re * c.im + im * c.re);
  }
  ...
}
```

Immutable Class : ข้อดี ข้อด้อย

- ข้อดี : ผู้เขียน immutable class
  - เขียนง่าย มี state เดียวตั้งแต่เกิด
  - thread-safe แน่นอน ค่าไม่เคยเปลี่ยน
  - ไม่ต้องเขียน clone, ไม่ต้องเขียน copy constructor
- ข้อดี : ผู้ใช้ immutable class
  - ใช้เป็น key ใน map หรือ set ได้อย่างสบายใจ
  - ไม่ต้องคอยระแวง ไม่ต้องทำ defensive copying
- ข้อด้อย
  - ผลิตออปเจกต์บ่อย

```
String s = "Java";
s = s.concat(" 1.5").toLowerCase().substring(2, 6);
// "Java 1.5" -> "java 1.5" -> "va 1"
```

ต้องระวังเมื่อใช้ mutable object เป็น key

```
Collection ct = new TreeSet();
Date d = new Date(103, 11, 31); // Date is mutable
ct.add(new Date(104, 11, 31));
ct.add(d);
d.setYear(105);

Collection ca = new ArrayList(ct);

//[Sat Dec 31 00:00:00 ICT 2005, Fri Dec 31 00:00:00 ICT 2004]
System.out.println(ct);
System.out.println(ca);

System.out.println(ct.contains(d)); // false !!
System.out.println(ca.contains(d)); // true
```

Immutable Class : เขียนอย่างไร

- การตั้ง state ให้ออปเจกต์ต้องทำให้เสร็จใน constructor หรือ static factory method
- ใช้ defensive copy กับ field ที่เป็น mutable
- ต้องไม่ให้เมท็อดต่าง ๆ ถูก overriden
- มีแต่ private fields
- มีแต่ final fields
- ห้ามมี mutator

```
public final class A {
  private final int id;
  private final String name;
  private final Date date;
  public A(int i, String n, Date d) {
    id = i;
    name = n;
    date = (Date) d.clone();
  }
  public Date getDate() {
    return (Date) date.clone();
  }
  ...
```

Immutable Class : ใช้ออปเจกต์ร่วมกันได้

- static factory สามารถ cache ออปเจกต์ที่ใช้บ่อย ๆ หรือขอบ่อย ๆ เพื่อป้องกันการผลิตออปเจกต์ซ้ำได้

```
public final class Complex {
  public static final Complex ZERO = new Complex(0, 0);
  public static final Complex ONE = new Complex(1, 0);
  public static final Complex I = new Complex(0, 1);

  private final double re, im;

  public Complex(double re, double im) {
    this.re = re; this.im = im;
  }
  public static valueOf(double re, double im) {
    if (re == 0 && im == 0) return ZERO;
    if (re == 1 && im == 0) return ONE;
    if (re == 0 && im == 1) return I;
    return new Complex(re, im);
  }
  ...
}
```

Immutable Class : cache hashCode ได้

```
public final class Complex {
  private final double re, im;
  private int hashCode;

  public Complex(double re, double im) {
    this.re = re;
    this.im = im;
    long code = 17 + Double.doubleToLongBits(re);
    code = 37 * code + Double.doubleToLongBits(im);
    hashCode = (int) (code ^ (code >>> 32));
  }

  public boolean equals(Object o) {
    if (o == this) return true;
    if (!(o instanceof Complex)) return false;
    Complex c = (Complex) o;
    return (Double.doubleToLongBits(re) ==
            Double.doubleToLongBits(c.re)) &&
           (Double.doubleToLongBits(im) ==
            Double.doubleToLongBits(im));
  }
  public int hashCode() { return hashCode; }
  ...
```

Immutable Class : ไม่ให้เมท็อดถูก overriden

1. final class
2. final method : subclass เพิ่ม feature ได้
3. private หรือ package-private constructor แล้วมี public static factory methods

```
public class Complex {
  private final double re, im;
  private Complex(double re, double im) {
    this.re = re; this.im = im;
  }
  public static valueOf(double re, double im) {
    return new Complex(re, im);
  }
  public Complex add(Complex c) {
    return new Complex(re + c.re, im + c.im);
  }
  ...
}
```

Inheritance : ข้อควรระวัง

- เขียนคลาสใหม่
  - ที่ extends คลาสอื่น อาจมีปัญหาถ้าคลาสอื่นนั้นเปลี่ยน
  - ที่จะให้คลาสอื่น extends ต้องรอบคอบมาก ๆ
- ใช้ wrapper class ปลอดภัยกว่า

Inheritance : extends คลาสอื่นต้องระวัง

```
public class InstrumentedHashSet extends HashSet {
  private int addCount = 0;

  public InstrumentedHashSet() {}

  public boolean addAll(Collection c) {
    addCount += c.size();
    return super.addAll(c);
  }

  public boolean add(Object o) {
    addCount++;
    return super.add(o);
  }

  public int getAddCount() {
    return addCount;
  }
}
```

นับผิด เพราะ addAll ใช้ add

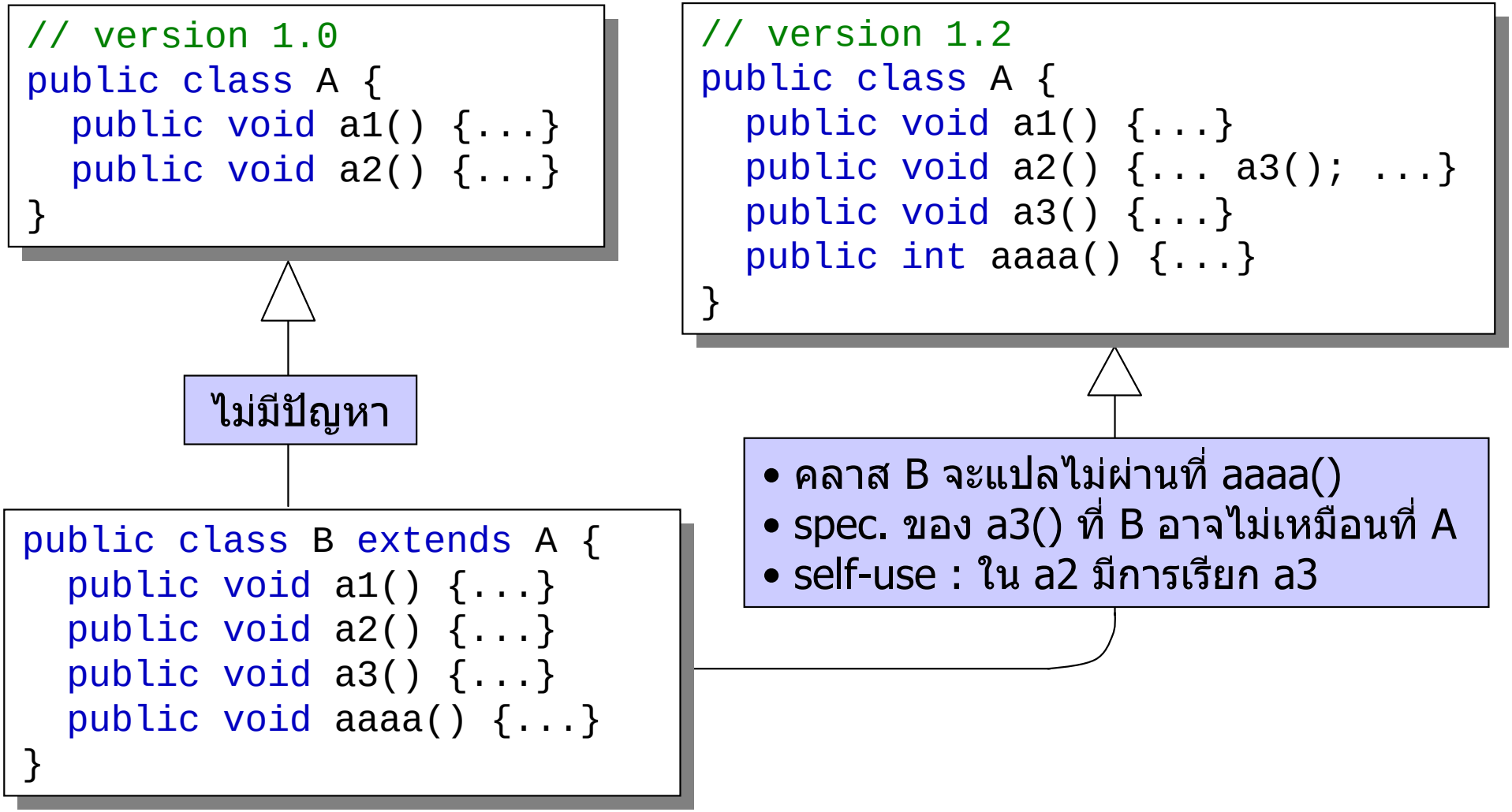
```
public boolean addAll(Collection c) {
  boolean modified = false;
  Iterator e = c.iterator();
  while (e.hasNext()) {
    if (add(e.next()))
      modified = true;
  }
  return modified;
}
```

Inheritance : ทำลาย encapsulation

```
public class InstrumentedHashSet extends HashSet {
  // The number of attempted element insertions
  private int addCount = 0;

  public InstrumentedHashSet() {}

  public boolean addAll(Collection c) {
    addCount += c.size();
    return super.addAll(c);
  }

  public boolean add(Object o) {
    addCount++;
    return super.add(o);
  }

  public int getAddCount() {
    return addCount;
  }
}
```

- ลบ addAll ทิ้ง ก็ถูกต้อง แต่...
- subclass ต้องรู้รายละเอียดการทำงานของ superclass (สูญเสีย encapsulation)
- ถ้ามีการเปลี่ยนการทำงานของ addAll ใน superclass อาจทำให้ subclass ทำงานผิด (fragile class)

Inheritance : fragile class



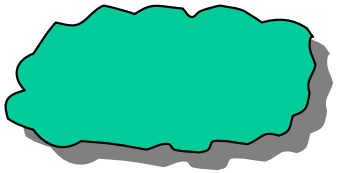
Inheritance : หลีกเลี่ยง self-use

- อยากเขียนคลาสให้ผู้อื่น extends ต้องรอบคอบ
  - หลีกเลี่ยง self-use overridable methods
    - overridable method ที่มีการเรียก overridable methods
  - ถ้ามี self-use ต้องเขียน javadoc comment ให้ชัดเจน

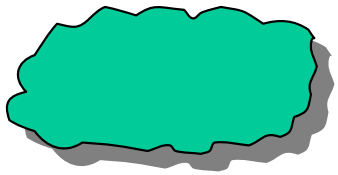
```
/**
 * ...
 * This implementation iterates over the collection
 * looking for the specified element.  If it finds the
 * element, it removes the element from the collection
 * using the iterator's remove method.
 * ...
 */
 public boolean remove(Object o) {
   ...
 }
```

Inheritance : ใช้ private helper method

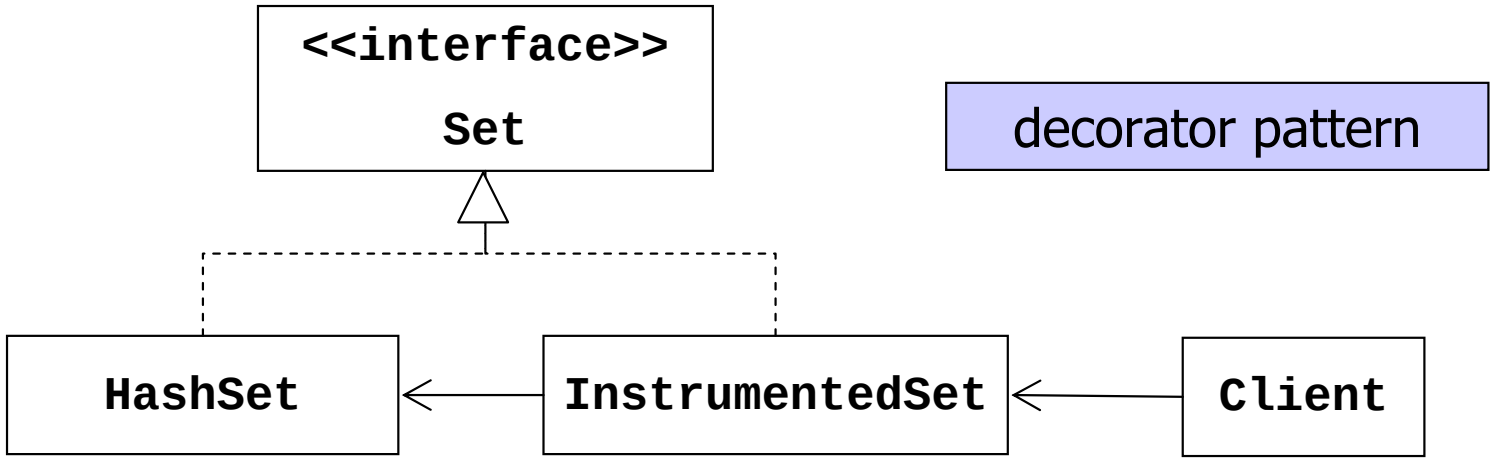
```
public class A {
  public A(int i) { b(i); } // danger : self-use
  public void a(int k) {
    for (int i = 1; i < k; i++) {
      b(i); // self-use
    }
  }
  public void b(int i) {
  }
}
```



```
public class A {
  public A(int i) { bHelper(i); }
  public void a(int k) {
    for (int i = 1; i < k; i++) {
      bHelper(i);
    }
  }
  public void b(int i) {
    bHelper(i);
  }
  private void bHelper(int i) {
  }
}
```



Inheritance : ใช้ wrapper class แทน




```
public class InstrumentedSet implements Set {
  private int addCount = 0;
  private Set s = 0;                              // composition
  public InstrumentedSet(Set s) {this.s = s}
  public boolean addAll(Collection c) {
    addCount += s.size();
    return s.addAll(c);                           // forwarding
  }
  public boolean add(Object o) { addCount++; return s.add(o); }
  public int getAddCount() { return addCount; }
  public void clear() { s.clear(); }
  public boolean contains(Object o { return s.containts(o); }
  ...
```

Inheritance : ข้อแนะนำ

- อย่า extends คลาสที่อยู่นอก package
- ข้อผิดพลาดของ superclass ส่งทอดให้ subclass
- superclass สูญเสีย encapsulation เพราะอาจต้องเปิดเผยวิธีการทำงานภายในเพื่อให้คนอื่น extends ได้
- subclass เปราะ เจ๊งง่าย เพราะถ้า superclass เปลี่ยนหรือเพิ่มพฤติกรรม อาจกระทบการทำงานของ subclass
- ให้ A extends B เมื่อมั่นใจจริง ๆ ว่า A "is-a" B และ A กับ B อยู่ใน package เดียวกัน
- หลีกเลี่ยงการใช้ overridable method ใน overridable method ด้วยกัน ใช้ private helper method ช่วย
- ห้ามเรียก overridable method ใน constructor, clone, และ readObject
- ใช้ wrapper class แทน inheritance จะดีกว่า

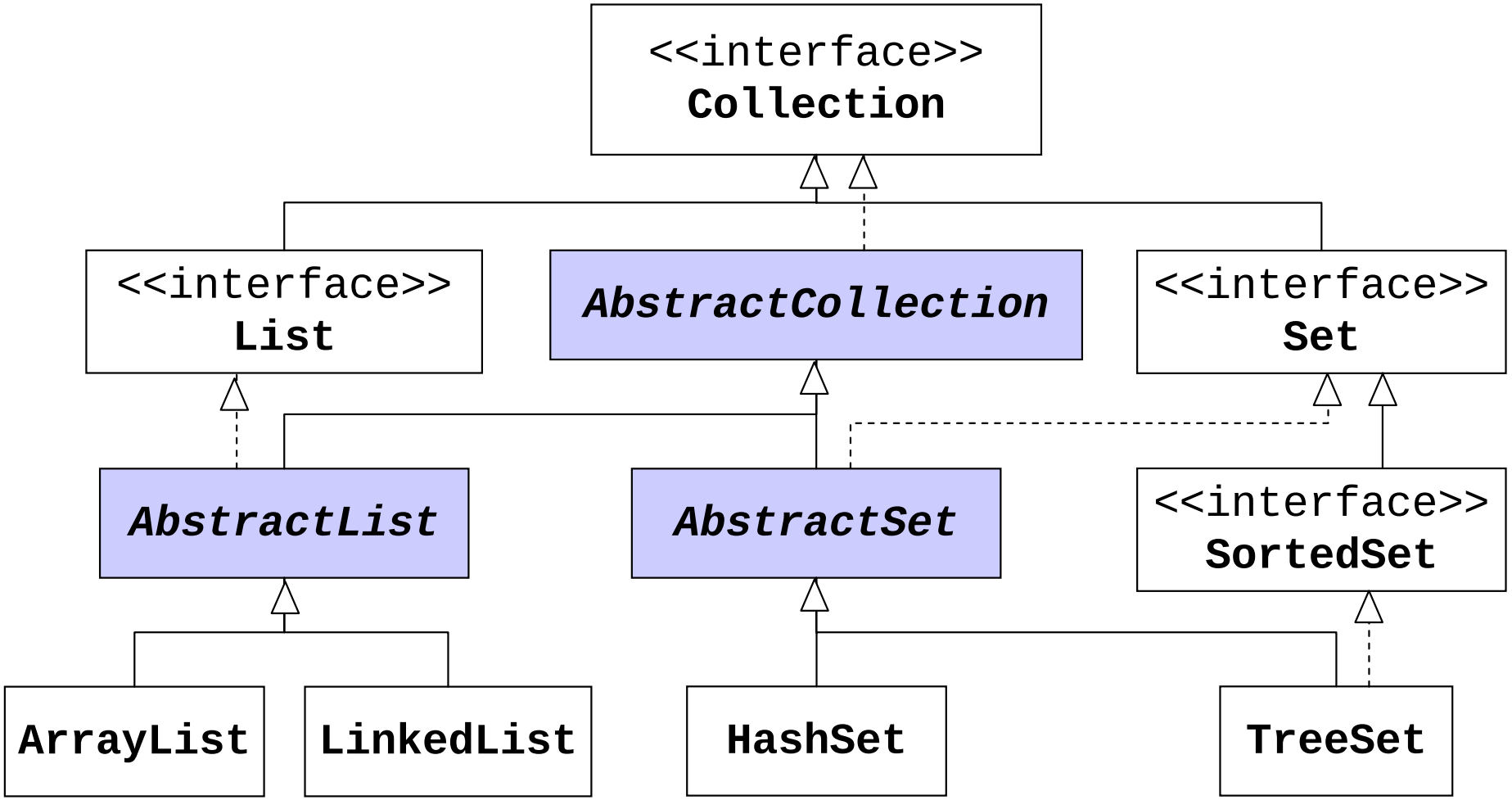
Interface vs. Abstract Class : ลักษณะ

- ใน interfaces มีแต่
  - public abstract methods
  - public static final fields
- ใน abstract class มี
  - อะไรก็ได้ที่ class มี และ
  - abstract methods
  - constructor ไว้ให้ลูกเรียก super(...)
- class หนึ่ง
  - implements ได้หลาย interfaces
  - extends ได้แค่หนึ่ง class หรือหนึ่ง abstract class

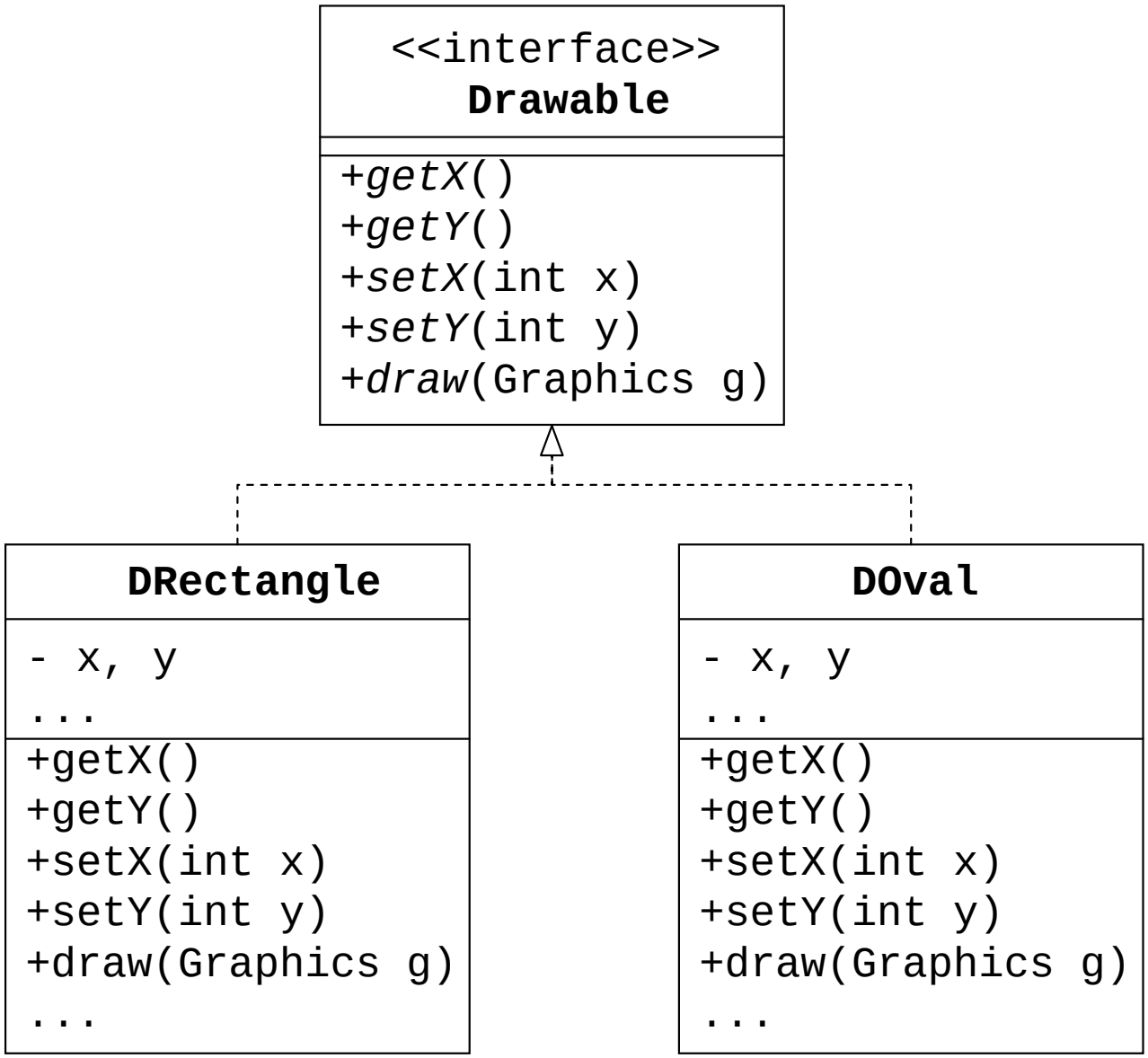
Interface vs. Abstract Class : การใช้งาน

- Interface
  - ใช้นิยามประเภทข้อมูลใหม่เพิ่มให้กับคลาส (mixin)
  - ใช้ interface ร่วมกับ wrapper class เพื่อสร้างคลาสใหม่โดยไม่ต้องใช้ inheritance
- Abstract class
  - ใช้เป็น skeletal implementation ของคลาสต่าง ๆ ที่ implements interface เดียวกัน

Interface vs. Abstract Class : ตัวอย่าง



Interface vs. Abstract Class : ตัวอย่าง



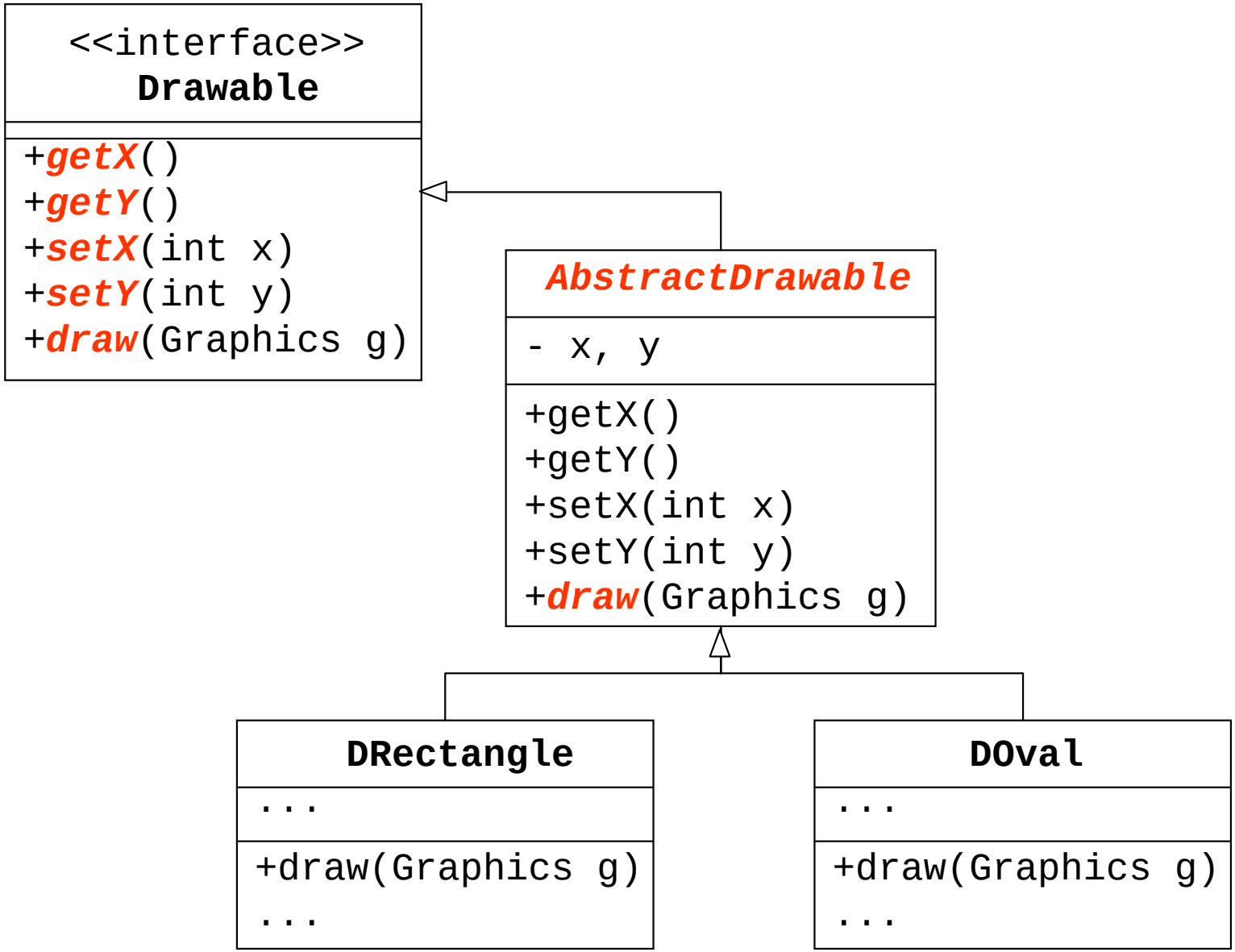
Interface vs. Abstract Class : ตัวอย่าง

```
interface Drawable {
  void draw(Graphics g);
  int getX();
  int getY();
  void setX(int x);
  void setY(int y);
}
```

```
class DRectangle implements Drawable {
  private int x, y;
  public void draw(Graphics g) {...}
  public int getX() { return x; }
  public int getY() { return y; }
  public void setX(int x) { this.x = x; }
  public void setY(int y) { this.y = y; }
  public void draw(Graphics g) {...}
  ...
}
```

```
class DOval implements Drawable {
  private int x, y;
  public int getX() { return x; }
  public int getY() { return y; }
  public void setX(int x) { this.x = x; }
  public void setY(int y) { this.y = y; }
  public void draw(Graphics g) {...}
  ...
}
```

Interface vs. Abstract Class : ตัวอย่าง



Abstract Classes : skeletal implementation

```
interface Drawable {
  int  getX();
  int  getY();
  void setX(int x);
  void setY(int y);
  void draw(Graphics g);
}
```

```
abstract class AbstractDrawable implements Drawable {
  private int x, y;
  public int  getX() { return x; }
  public int  getY() { return y; }
  public void setX(int x) { this.x = x; }
  public void setY(int y) { this.y = y; }
  public abstract void draw(Graphics g);
}
```

```
class DRectangle extends AbstractDrawable {
  public void draw(Graphics g) {...}
  ...
}
```

สำหรับคลาสที่ extends skeletal impl. ได้

```
class DOval extends AbstractDrawable {
  public void draw(Graphics g) {...}
  ...
}
```

Abstract Classes : sim. multiple inheritance

```
public class DRectangle extends Polygon implements Drawable {
  private int w, h;
  // simulated multiple inheritance
  private Drawable drawable = new AbstractDrawable() {
    public void draw(Graphics g) {
        g.drawRect(drawable.getX(), drawable.getY(),
                   Rectangle.this.w, Rectangle.this.h);
    }
  };
  public Rectangle(int x, int y, int w, int h) {
    drawable = this.w = w; this.h = h;
    drawable.setX(x); drawable.setY(y);
  }
  public int getX() { return drawable.getX(); }
  public int getY() { return drawable.getY(); }
  public void setX(int x) { drawable.setX(x); }
  public void setY(int y) { drawable.setY(y); }
  public void draw(Graphics g) { drawable.draw(g); }
  ...
}
```

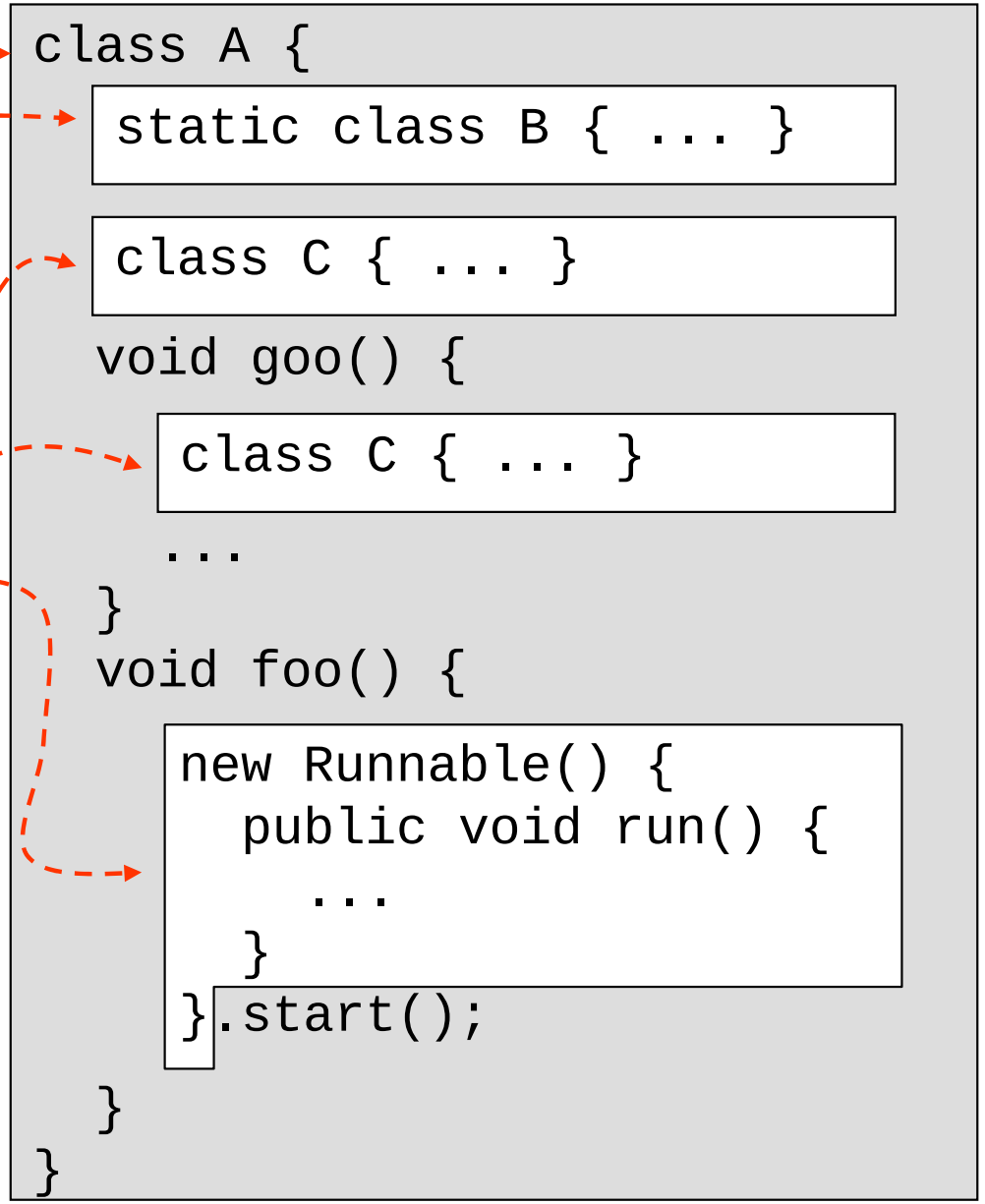
สำหรับคลาสที่มี superclass อยู่แล้ว
(ใช้ simulated multiple inheritance)

Interface vs. Abstract Class : ข้อแนะนำ

- ใช้ interface ในการนิยามประเภทข้อมูลใหม่ที่มีหลาย ๆ implementations
- อย่าใช้ interface เพียงเพื่อเก็บ constants
- การปรับเปลี่ยน abstract class ทำได้ง่ายกว่า interface
  - เพิ่มเมท็อดใน interface กระเทือนผู้ใช้หนัก
  - เพิ่มเมท็อดที่มี default implementation ใน abstract class เกิดผลกระทบน้อย
- เขียน public interface ต้องรอบคอบมาก ๆ เปลี่ยนแปลงในอนาคตลำบากมาก ๆ

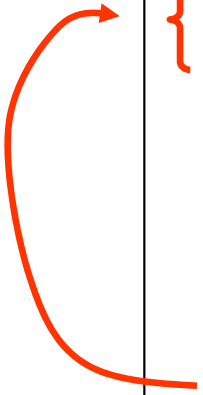
Nested Classes

- Top level classes
- Nested classes
  - Nested top-level
  - Member inner
  - Local inner
  - Anonymous inner



Nested Classes : Nested top-level classes

- อยู่ใน top-level หรือ nested top-level
- มี static กำกับ class
- อ้างอิง static members ของ outer class ได้เท่านั้น
- มีสมาชิกได้ทั้งแบบ static และ non-static

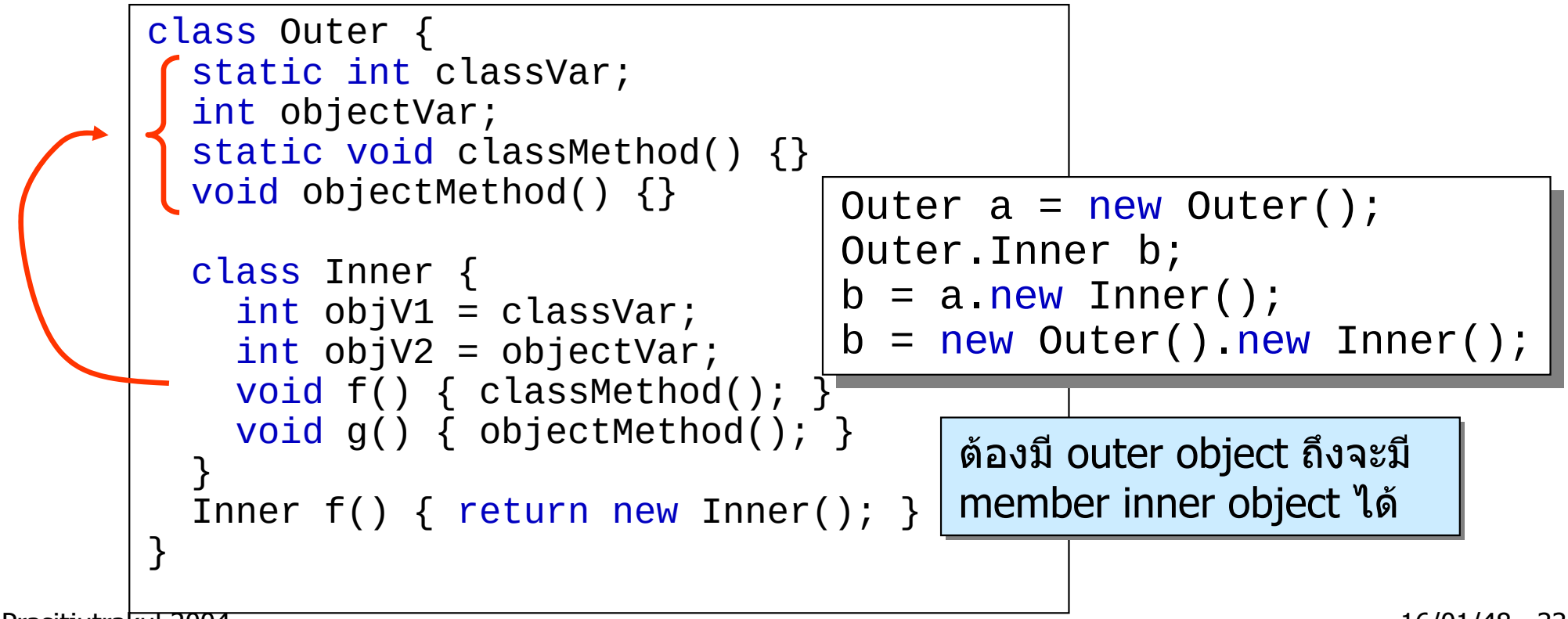


```
class Outer {
  static int classVar;
  static void classMethod() {}
  int objectVar;
  void objectMethod() {}

  static class NTL {
    static int cV = classVar;
    int objV = classVar;
    static void f() { classMethod(); }
    void g() { classMethod(); }

    static class NTL2 { ... }
  }
}
```

```
Outer a = new Outer();
Outer.NTL b = new Outer.NTL();
```

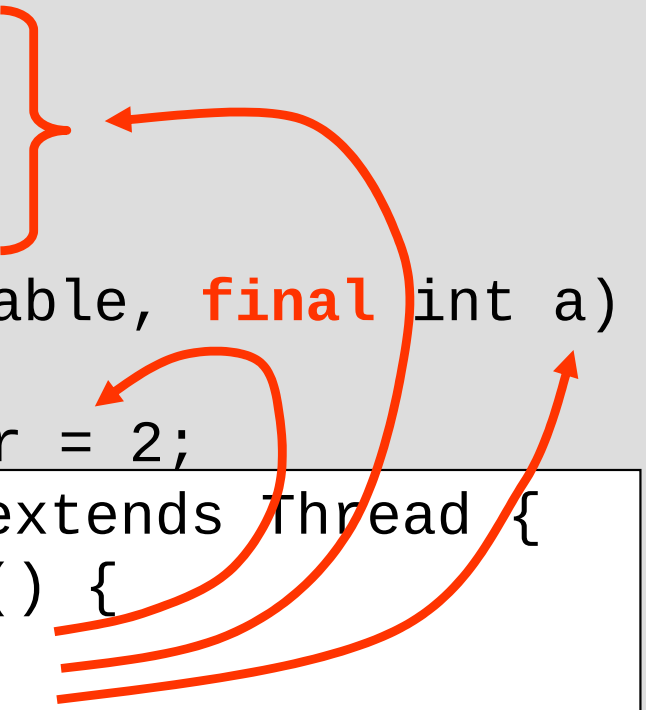
Nested Classes : Member inner classes

- อยู่ใน top-level, nested top-level, member inner
- ไม่มี static กำกับ class
- อ้างอิงทุก ๆ members ของ outer class ได้
- มีสมาชิกได้เฉพาะแบบ non-static



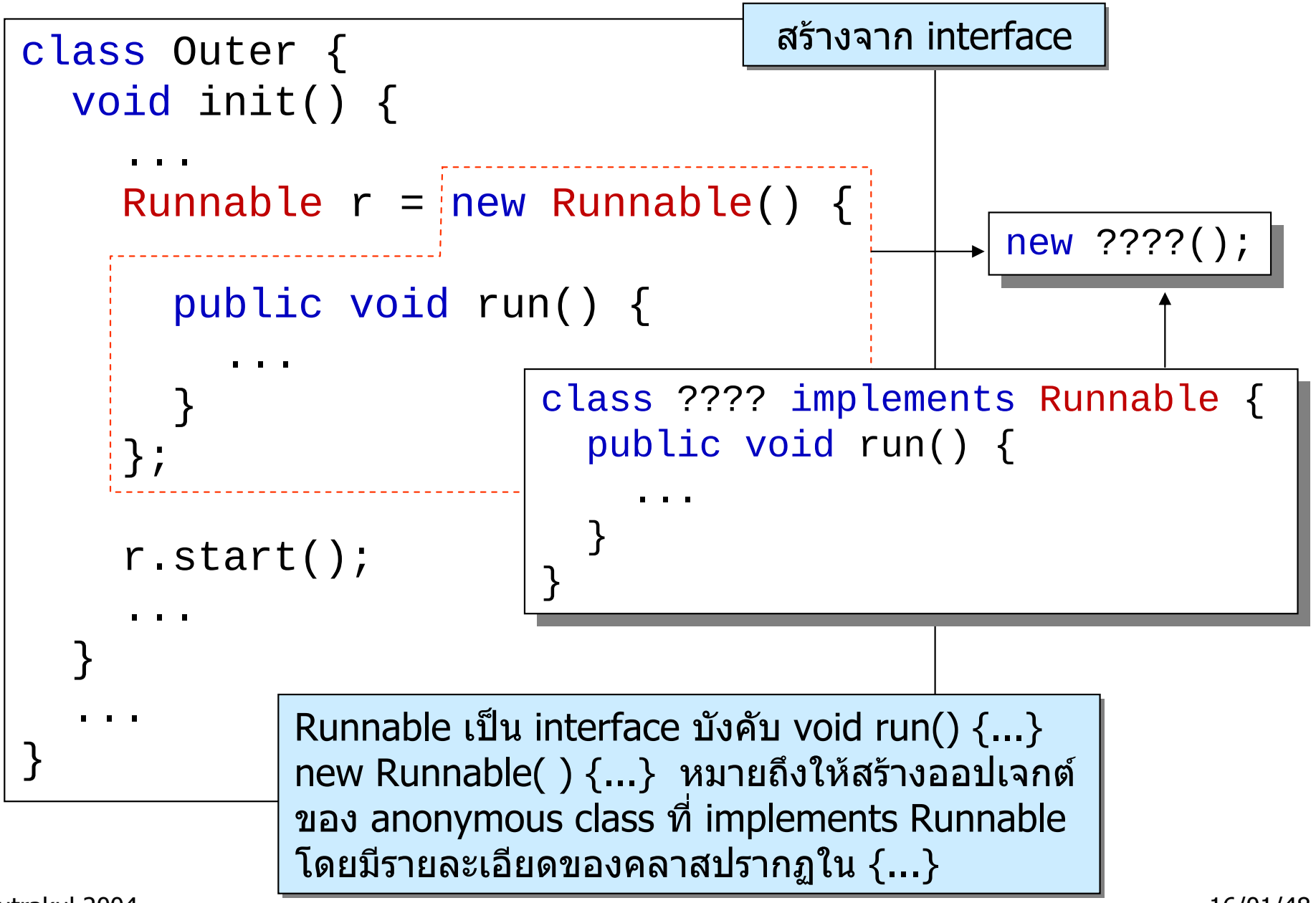
Nested Classes : Local Inner

```
class Outer {
  static int c;
  static void f(){}
  int v;
  void g(){}
  void init(boolean enable, final int a) {
    String s = “abc”;
    final int aLocalVar = 2;
    class LocalInner extends Thread {
      public void run() {
        ...
      }
    }
    new LocalInner().start();
  }
}
```

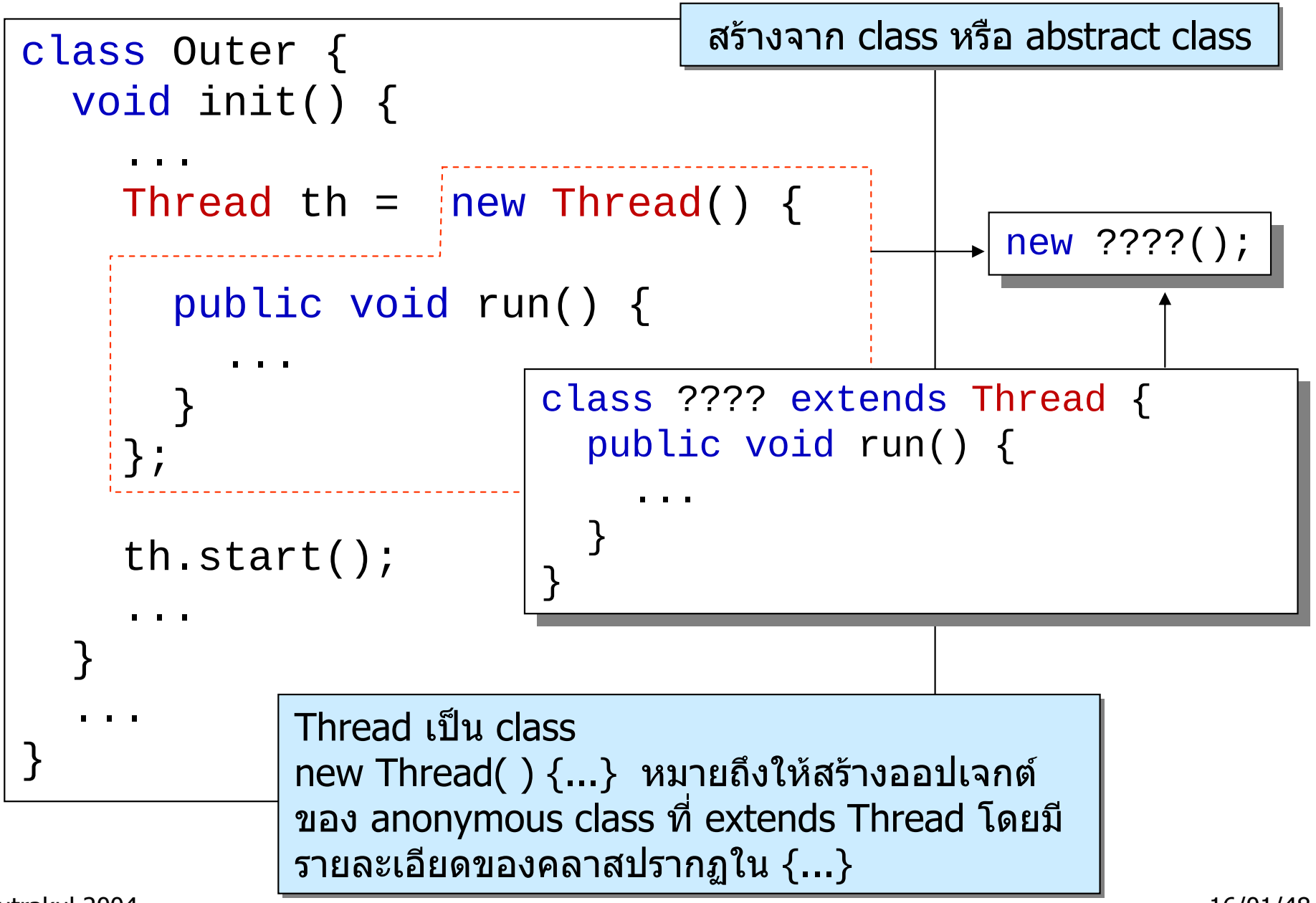


Local inner class คือ member inner class ที่เขียนนิยามใน method (อ้างอิง local var. ที่เป็น final ได้)

Nested Classes : Anonymous Inner

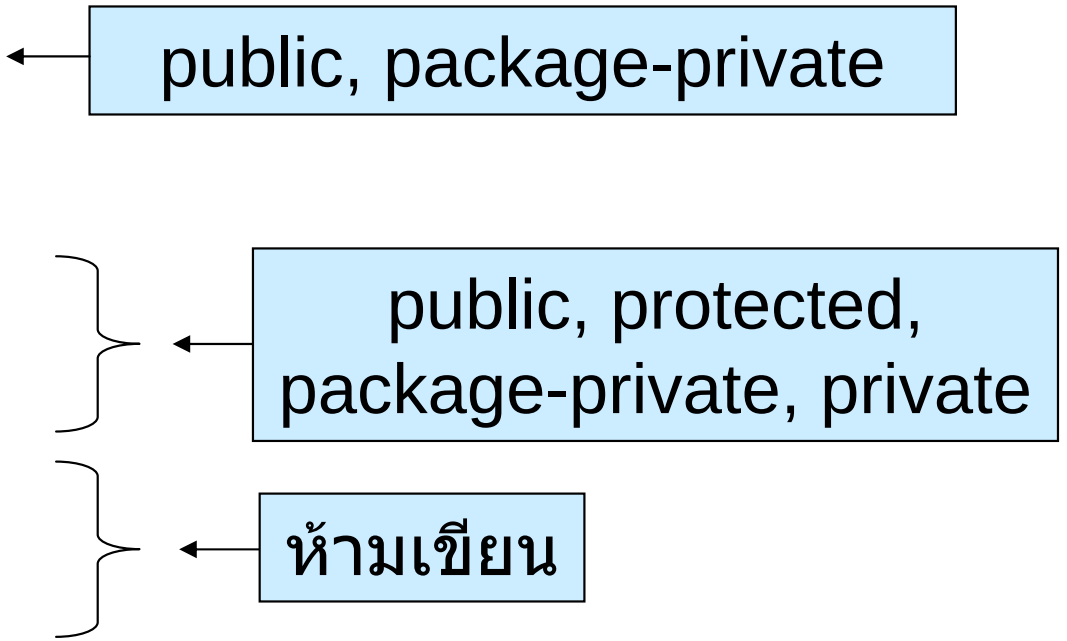


Nested Classes : Anonymous Inner



Nested Classes : Access Controls

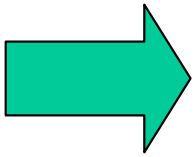
- Top level class
- Nested class
  - Nested top-level
  - Member inner
  - Local inner
  - Anonymous inner



ใช้ nested top-level class เมื่อไร ?

```
class A {
   ...
   B b = new B();
   ...
}
```

```
class B {
   ...
}
```



```
class A {
   ...
   B b = new B();
   ...

   private static
   class B {
      ...
   }

}
```

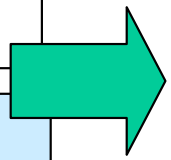
คลาส B ถูกใช้เฉพาะในคลาส A เท่านั้น ก็ให้ B เป็น nested top-level ซึ่ง private เฉพาะในคลาส A

locality : คลาสที่มีความสัมพันธ์กันควรเขียนอยู่ใกล้กัน

ใช้ member inner class เมื่อไร ?

```
class Bag {
 Object [] data;
 Itr iterator() {
  return new Itr(this);
 }
 ...
}
```

```
class Itr {
  private Bag bag;
  private int idx;
  Itr(Bag b) {
   this.bag = b;
  }
  Object next() {
    return bag.data[idx++];
  }
  ...
}
```



```
class Bag {
 private Object [] data;
 Itr iterator() {
  return new Itr();
 }
  class Itr {
    private int idx;
    Object next() {
      return bag.data[idx++];
    }
    ...
  }
  ...
}
```

ลดความซับซ้อน : member inner ใช้ members ของ outer ได้เลย

ใช้ anonymous class เมื่อไร ?

- คลาสมีขนาดเล็ก ใช้ที่ตำแหน่งเดียวในเมท็อด
- เป็น functor, process object, callback
- ใช้ใน static factory method

```
Arrays.sort(data, new Comparator() {
  public int compare(Object o1, Object o2) {
    return -((Comparable) o1).compareTo(o2);
  } } );
```

```
new Thread() {
   public void run() {
     veryTimeConsumingComputation();
   }
 }.start();
```

```
frame.addWindowListener(new WindowAdapter() {
  public void windowClosing(WindowEvent e) {
    System.exit(0);
  } } );
```

Nested Class : ข้อแนะนำ

- ใช้เพื่อจัดโครงสร้างคลาสให้มีระเบียบ
- ใช้เพื่อปกปิดคลาส
- ใช้เพื่อลดความซ้ำซ้อนของชื่อคลาส
- ถ้าใช้ nested top-level ได้ อย่าลืมเติมคำว่า static
  - เร็วกว่า ประหยัดกว่า
- ใส่ access control ให้เหมาะสม