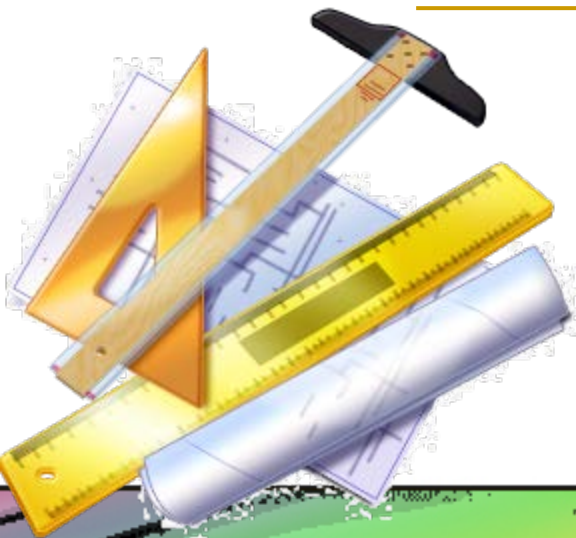


# Generic

## Lecture 12

Yaowadee Temtanapat

เยาวดี เต็มชนาภัตร์



# วัตถุประสงค์ของการเรียนในวันนี้

- เรียนรู้แนวคิดเกี่ยวกับการโปรแกรมเจเนอริก
- เรียนรู้การประกาศและใช้คลาสเจเนอริก
- เรียนรู้การประกาศและใช้เมทอดเจเนอริก
- ชนิดพารามิเตอร์ที่มีขอบเขต
- การใช้งานอินเทอร์เฟสเจเนอริก

# แนวความคิดการโปรแกรมเจเนอริก

## ■ การโปรแกรมเจเนอริก

- ❑ การโปรแกรมคลาสหรือเมทอดให้ทำงานกับชนิดที่ยังไม่ทราบแน่นอนได้
  - ลดข้อจำกัดของการเขียนที่ต้องกำหนดชนิดที่จะทำงานด้วยอย่างแน่นอน
- ❑ จาวาทำโดยกำหนดชนิดพารามิเตอร์ (parameter type) เขียนคร่อมด้วย < >

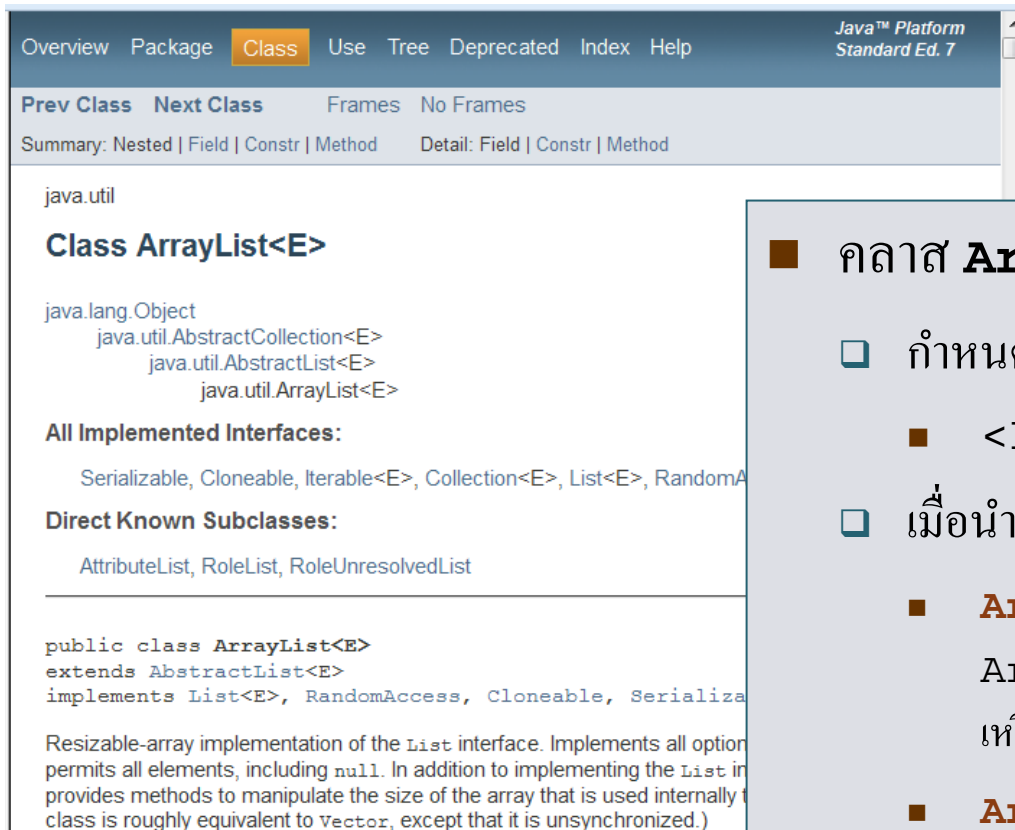
## ■ ชนิดพารามิเตอร์(*Parameter Type*)

- ❑ ชนิดที่ยังไม่ทราบแน่นอน กำหนดเป็นพารามิเตอร์ของคลาสหรือเมทอด
- ❑ ชนิดพารามิเตอร์จะถูกระบุ/แทนด้วยชนิดที่แท้จริงเมื่อตอนใช้งาน

## ■ คลาสที่มีชนิดพารามิเตอร์ เรียก *คลาสเจเนอริก (Generic Class)*

## ■ เมทอดที่มีชนิดพารามิเตอร์ เรียก *เมทอดเจเนอริก (Generic Method)*

# ตัวอย่างคลาสเจเนอริก ArrayList<E> ใน Java API



The screenshot shows the Java Platform Standard Ed. 7 API documentation for the `ArrayList<E>` class. The navigation bar includes tabs for Overview, Package, Class (selected), Use, Tree, Deprecated, Index, and Help. Below the navigation bar, there are links for Prev Class, Next Class, Frames, and No Frames. The summary section shows the class hierarchy: `java.util` package, `java.lang.Object`, `java.util.AbstractCollection<E>`, `java.util.AbstractList<E>`, and `java.util.ArrayList<E>`. The **All Implemented Interfaces:** section lists `Serializable`, `Cloneable`, `Iterable<E>`, `Collection<E>`, `List<E>`, and `RandomAccess`. The **Direct Known Subclasses:** section lists `AttributeList`, `RoleList`, and `RoleUnresolvedList`. The **Source Code** section shows the following code snippet:

```
public class ArrayList<E>
    extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, Serializable {
    // ...
}
```

Below the code snippet, there is a description: "Resizable-array implementation of the List interface. Implements all optional operations except `addIndex`. It permits all elements, including `null`. In addition to implementing the List interface, it provides methods to manipulate the size of the array that is used internally to store the elements. This class is roughly equivalent to `Vector`, except that it is unsynchronized."

- คลาส **ArrayList<E>** เป็นคลาสเจเนอริก
  - กำหนดคลาสโดยมีชนิดพารามิเตอร์ **<E>**
    - **<E>** เป็นชนิดสมาชิกที่ยังไม่รู้แน่นอน
  - เมื่อนำไปใช้ กำหนดชนิดที่แน่นอนแทน **<E>**
    - **ArrayList<Coin>** ประกาศชนิด ArrayList สำหรับชนิด Coin เพื่อใช้เป็นที่เก็บเหรียญ
    - **ArrayList<String>** ประกาศชนิด ArrayList สำหรับชนิด String เพื่อใช้เป็นที่เก็บสายอักขระ

# ตัวอย่างการใช้งานคลาสเจนเนอริก ArrayList<E>

- การใช้งาน ทำเช่นเดียวกับคลาสอื่น แต่ต้องกำหนดชนิดที่แท้จริงที่จะทำงานด้วย
- การประกาศตัวแปรและการสร้างวัตถุสำหรับเป็นที่เก็บเหรียญ
  - ประกาศตัวแปร coinList ชนิด ArrayList<Coin>  
`ArrayList<Coin> coinList;`
  - สร้างวัตถุ ArrayList สำหรับเก็บเหรียญ และให้อ้างถึงโดยตัวแปร coinList  
`coinList = new ArrayList<Coin>();`
- การเรียกใช้เมทอดต่าง ๆ ของวัตถุ ArrayList<Coin>
  - การใช้งานเพื่อเพิ่มเหรียญใน coinList  
`coinList.add(new Coin("10 bahts", 10.0));`
  - การใช้งานเพื่อดึงเหรียญลำดับที่ 0 ใน coinList  
`Coin coin = coinList.get(0);`

# ตัวอย่างปัญหา: คลาสเกมที่มีห้อง 1 ห้อง

```
public class Game {  
    private Room room;
```

ในห้องมีวัตถุชนิดตามที่ต้องการ

```
    public Game() {  
        this.room = new Room(new คลาสวัตถุที่ต้องการ() );  
    }
```

```
    public void play() {  
        คลาสวัตถุที่ต้องการ mons = room.getObject();  
  
        boolean sleep = mons.isSleep(); //ใช้งานตามความสามารถ  
        System.out.println("Monster sleep: " + sleep);
```

```
// ...
```

```
}
```

# ปัญหา: คลาส Room ทำงานกับชนิดตายตัว

- คลาส Room สำหรับเกมที่มีห้องมี Monster

```
public class Room {  
    private Monster object;  
  
    public Room(Monster object) {  
        this.object = object;  
    }  
  
    public Monster getObject() {  
        return object;  
    }  
}
```

ทำงานได้กับเฉพาะห้องที่มี Monster

- คลาส Room สำหรับเกมที่มีห้องมีวัตถุที่ยังไม่ทราบชนิดอย่างแน่นอน

```
public class Room {  
    private Object object;  
  
    public Room(Object object) {  
        this.object = object;  
    }  
  
    public Object getObject() {  
        return object;  
    }  
}
```

ทำงานได้กับเฉพาะห้องที่มีวัตถุใด ๆ แต่ต้องทำ type casting กลับเป็นชนิดที่ถูกต้องเอง มีแนวโน้มผิดพลาดได้ง่าย

# คลาสเจเนอริก Room<T>

## ■ ประกาศคลาสเจเนอริก Room

```
public class Room<T> {  
    private T object;  
  
    public Room(T object) {  
        this.object = object;  
    }  
    public T getObject() {  
        return object;  
    }  
}
```



# การใช้งานโดยคลาสเกมที่มีห้อง ภายในมี Monster

กำหนดชนิดที่ต้องการใช้งานกับ Room

```
public class Game {  
    private Room<Monster> room;
```

ในห้องมีวัตถุชนิดตามที่ต้องการ

```
    public Game() {  
        this.room = new Room<Monster>(new Monster());  
    }
```

```
    public void play() {  
        Monster mons = room.getObject();  
  
        boolean sleep = mons.isSleep(); //ใช้งานตามความสามารถ  
        System.out.println("Monster sleep: " + sleep);  
  
        // ...  
    }
```

# การใช้งานโดยคลาสเกมที่มีห้อง ภายในมี Dice

กำหนดชนิดที่ต้องการใช้งานกับ Room

```
public class Game {  
    private Room<Dice> room;
```

ในห้องมีวัตถุชนิดตามที่ต้องการ

```
    public Game() {  
        this.room = new Room<>(new Dice());  
    }
```

```
    public void play() {  
        Dice dice = room.getObject();  
        double value = coin.getValue(); //ใช้งานตามความสามารถ  
        System.out.println("A dice gave value: " + dice.roll());  
  
        // ...  
    }
```

# Syntax: ประกาศคลาสเจเนอริก

## ■ Syntax:

```
public class ClassName<GenericTypeList> { ... }
```

เมื่อ GenericTypeList เป็นชื่อพารามิเตอร์ ถ้ามากกว่า 1 ตัวคั่นด้วย comma ,

## ■ ตัวอย่างเช่น:

ชนิดพารามิเตอร์ตัวเดียว <T>

```
public class Room<T> {  
    private T object;  
    // ...  
}
```

ชนิดพารามิเตอร์หลายตัว <T, S>

```
public class Room<T, S> {  
    private T creature;  
    private S treasure;  
    // ...  
}
```

## ■ จุดมุ่งหมาย เพื่อประกาศคลาสเจเนอริก ซึ่งเมื่อนำไปใช้กำหนดชนิดวัตถุที่แท้จริง (หมายเหตุ: ไม่สามารถใช้กับชนิดพื้นฐานได้)

# การกำหนด identifier สำหรับเป็นชื่อชนิดพารามิเตอร์

- การตั้งชื่อ แนะนำให้ใช้ชื่อหนึ่งตัวอักษร โดย

ชื่อชนิดพารามิเตอร์	ความหมาย
E	ชื่อชนิดสำหรับเป็นชนิดของสมาชิกในกลุ่มข้อมูล (Collection)
K	ชื่อชนิดสำหรับชนิดที่ใช้เป็นคีย์
N	ชื่อชนิดสำหรับชนิดที่เป็นตัวเลข (Number)
V	ชื่อชนิดสำหรับชนิดที่จะเป็นค่า
T, S, U, ฯลฯ	ชื่อชนิดทั่วไป

# เมทอดเจเนอริก

- กำหนดเมทอดให้เป็นเจเนอริกได้เช่นเดียวกับคลาส

- Syntax:

```
modifier <T1, ..., Tn> returnType methodName(parameterList) {  
    // ตัวแปรท้องถิ่นและประโยคการทำงานของเมทอด  
}
```

- T<sub>i</sub> เป็นชนิดพารามิเตอร์ที่เมทอดใช้เป็นชนิดของพารามิเตอร์เข้าของเมทอด
  - parameterList เป็นรายการพารามิเตอร์ประกาศเช่นเดียวกับเมทอดทั่วไป
- หมายเหตุ กรณีที่เมทอดอยู่ในคลาสเจเนอริก และชนิดพารามิเตอร์เป็นของคลาส ไม่จำเป็นต้องระบุ <T<sub>i</sub>> ที่อยู่หน้า returnType

# ตัวอย่างเมทอดเจเนอริก

- เมทอดเพิ่มสมาชิกของ ArrayList จากข้อมูลที่อยู่ในอาร์เรย์

```
public static <T> void add(ArrayList<T> list, T[] data) {  
    for (T ele : data) {  
        list.add(ele);  
    }  
}  
  
public static <T> void print(String title, ArrayList<T> data) {  
    System.out.println(title);  
    for (T ele : data) {  
        System.out.print(ele + " ");  
    }  
    System.out.println();  
}
```

- การใช้งานเช่น

```
ArrayList<String> strList = new ArrayList<>();  
String[] data = { "monster", "pokemon", "coin", "dragon", "gold" };  
add(strList, data);  
print("String list", strList);
```

# ชนิดพารามิเตอร์แบบมีขอบเขต (Bounded Type Parameters)

- สามารถกำหนดชนิดพารามิเตอร์ให้มีขอบเขตว่า ต้องเป็น subclass ของบางชนิด
- ปัญหา: ต้องการให้ห้องมีวัตถุเฉพาะที่เป็นประเภทสัตว์ประหลาด (Monster และ subclass ของมัน) สามารถกำหนดขอบเขต

```
public class Room<T extends Monster> {  
    // รายละเอียดที่เหลือเช่นเดียวกับ Room เดิม ในหน้า 8  
}
```

- การใช้งานก็จะอยู่ในขอบเขตของกลุ่มประเภท Monster เท่านั้น
  - Room ไม่สามารถใช้กับชนิด Dice ได้อีกต่อไป

# คลาส Dragon เป็น subclass ของ Monster

```
public class Dragon extends Monster {  
    private int firePower;  
  
    public Dragon(int firePower) {  
        this.firePower = firePower;  
    }  
  
    public int getFirePower() {  
        return firePower;  
    }  
  
    public int getPower() {  
        return super.getPower() + firePower;  
    }  
}
```



# Game สามารถใช้งานกับ Dragon ได้

```
public class Game {  
    private Room<Dragon> room;  
  
    public Game() {  
        this.room = new Room<>(new Dragon(10));  
    }  
  
    public void play() {  
        Dragon mons = room.getObject();  
        boolean alive = mons.isAlive();  
        System.out.println("Dragon alive: " + alive +  
            " with firepower " + mons.getFirePower());  
    }  
}
```

ใช้งานความสามารถของ Dragon โดยตรงได้

ต่างจากกำหนด Room ให้มีชนิดวัตถุภายในเป็น Monster โดยตรง

# อินเทอร์เฟสเจเนอริก

- เช่นเดียวกับคลาส สามารถกำหนดอินเทอร์เฟสที่เป็นเจเนอริกได้
- อินเทอร์เฟสเจเนอริก (Generic Interface) ในจาวา API

```
public interface Comparable<T> {  
    int compareTo(T obj);  
}
```

□ โดยเมทอด compareTo คืนค่า

- ค่า 0 เมื่อวัตถุปัจจุบันมีค่าเท่ากับ obj
- ค่าลบเมื่อวัตถุปัจจุบันมีค่าน้อยกว่า obj
- ค่าบวก เมื่อวัตถุปัจจุบันมีค่ามากกว่า obj



# ตัวอย่างปัญหา

- ต้องการเมทอดเจเนอริก min ที่รับ 2 วัตถุเป็นพารามิเตอร์ เพื่อคืนวัตถุที่มีค่าน้อยกว่า

```
public static <T> T min(T obj1, T obj2) {  
    if (obj2 < obj1)  
        return obj2;  
    return obj1;  
}
```

- ปัญหาไม่สามารถเปรียบเทียบวัตถุใด ๆ ด้วยเครื่องหมายเปรียบเทียบ <

# การใช้งานอินเทอร์เฟซเจนเนอริก Comparable<T>

- ให้ min ทำงานกับคลาสของวัตถุที่มีความสามารถในการเปรียบเทียบ
- ให้ชนิดพารามิเตอร์ T มีขอบเขตภายใต้ Comparable<T>

```
public static <T extends Comparable<T>> T min(T obj1, T  
    obj2) {  
    if (obj2.compareTo(obj1) < 0)  
        return obj2;  
    return obj1;  
}
```

- เรียกใช้เมทอด compareTo จากวัตถุชนิด T ได้

# ประกาศคลาส Monster เป็น subtype ของ Comparable

```
public class Monster implements Comparable<Monster> {  
    private int power;  
    private boolean sleep;  
    public Monster() { revived(); }  
    public int getPower() { return power; }  
    public void revived() {  
        sleep = Math.random() > 0.5 ? true : false;  
        power = (int) (Math.random() * 10);  
    }  
}
```

@Override

```
public int compareTo(Monster other) {  
    return getPower() - other.getPower();  
}
```

```
}
```

# การใช้งานเมทอดเจเนอริก min

```
public static void main(String[] args) {  
    Monster mons1 = new Monster();  
    Monster mons2 = new Monster();  
    Monster minMons = min(mons1, mons2);  
    System.out.println("Monster1 power: " +  
                        mons1.getPower());  
    System.out.println("Monster2 power: " +  
                        mons2.getPower());  
    System.out.println("Min power: " +  
                        minMons.getPower());  
  
    System.out.println("Less (Apple and Banana) is " +  
                        min("Apple", "Banana"); ←  
}  

```

String implements  
Comparable อยู่แล้ว  
จึงสามารถใช้กับ min ได้

# Type Erasure

## ■ Type Erasure:

- เมื่อโค้ดเจเนอริก ผ่านการคอมไพล์แล้วชนิดพารามิเตอร์ที่ประกาศจะถูกเปลี่ยนเป็นชนิด Object หรือ ชนิดที่เป็นขอบเขตของชนิดพารามิเตอร์นั้น

## ■ มีผลและทำให้มีข้อกำหนดที่ต้องจำกัดในเรื่องการใช้งาน อาทิ

- ไม่อนุญาตให้สร้างอาร์เรย์ของคลาสเจเนอริก (เช่น `ArrayList<String>[]`)
- ไม่สามารถประกาศ Overloaded Method ที่มีลายเซ็นเหมือนกันหลังจากถูกลบชนิดแล้ว (คิดเสมือนไม่ชนิดพารามิเตอร์เลย)

เป็นต้น

# สรุปการเรียนรู้ในวันนี้

- แนวคิดเกี่ยวกับการ โปรแกรมเจเนอริก
  - เขียนโปรแกรมที่ยืดหยุ่นขึ้น เพื่อให้ทำงานได้กับชนิดที่ยังไม่ทราบ
- การประกาศคลาสเจเนอริก ทำโดยให้คลาสมีชนิดเป็นพารามิเตอร์
  - เมื่อใช้งาน แทนชนิดพารามิเตอร์ด้วยชนิดที่แท้จริง
- เรียนรู้การประกาศและใช้เมทอดเจเนอริก
- ชนิดพารามิเตอร์ที่มีขอบเขต
- เรียนรู้การใช้งานอินเทอร์เฟสเจเนอริก Comparable<T>