เอกสารประกอบการสอนวิชา การโปรแกรมเชิงวัตถุ

สำหรับนักศึกษา สาขาวิชาวิศวกรรมคอมพิวเตอร์

คณะวิศวกรรมศาสตร์และเทคโนโลยีอุตสาหกรรม

มหาวิทยาลัยราชภัฏสวนสุนันทา

โดย อาจารย์รวิ อุตตมธนินทร์

บทที่ 1: บทนำสู่การโปรแกรมเชิงวัตถุ

1.1 แนวคิดพื้นฐานของการโปรแกรมเชิงวัตถุ (Object-Oriented Programming - OOP)

การโปรแกรมเชิงวัตถุ (Object-Oriented Programming หรือ OOP) เป็นกระบวนทัศน์การเขียนโปรแกรมที่ยึดหลักการจัดระเบียบซอฟต์แวร์โดยใช้ แนวคิดของ “วัตถุ” (Objects) ซึ่งอาจประกอบด้วยข้อมูล (Attributes) และฟังก์ชัน (Methods) ที่ทำงานกับข้อมูลนั้นๆ แนวคิดหลักของ OOP ได้แก่ Encapsulation, Inheritance, Polymorphism และ Abstraction

1.1.1 ความแตกต่างระหว่าง Procedural Programming และ Object-Oriented Programming

คุณสมบัติ	Procedural Programming	Object-Oriented Programming
แนวคิดหลัก	เน้นขั้นตอนการทำงาน (Sequence of instructions)	เน้นวัตถุ (Objects) และการโต้ตอบระหว่างวัตถุ
การจัดโครงสร้าง	โปรแกรมแบ่งเป็นฟังก์ชันย่อยๆ	โปรแกรมแบ่งเป็นคลาส (Classes) และวัตถุ (Objects)
ข้อมูล	ข้อมูลและฟังก์ชันแยกจากกัน	ข้อมูลและฟังก์ชันที่ทำงานกับข้อมูลถูกรวมอยู่ในวัตถุเดียวกัน (Encapsulation)
ความปลอดภัยของข้อมูล	เข้าถึงข้อมูลได้ง่ายจากทุกส่วนของโปรแกรม	ข้อมูลถูกซ่อนไว้ภายในวัตถุ (Data Hiding) และเข้าถึงผ่านเมธอดที่กำหนดเท่านั้น
การนำกลับมาใช้ใหม่	ทำได้ยากกว่า ต้องเขียนโค้ดใหม่หรือปรับแก้มาก	ทำได้ง่ายกว่าผ่าน Inheritance และ Polymorphism
การบำรุงรักษา	ซับซ้อนเมื่อโปรแกรมมีขนาดใหญ่	จัดการและบำรุงรักษาได้ง่ายกว่า
ตัวอย่างภาษา	C, Pascal, Fortran	Java, C++, Python, C#

1.2 ประโยชน์ของการโปรแกรมเชิงวัตถุ

การโปรแกรมเชิงวัตถุมีประโยชน์หลายประการที่ทำให้เป็นที่นิยมในการพัฒนาซอฟต์แวร์ขนาดใหญ่และซับซ้อน ได้แก่:

1. **ความสามารถในการนำกลับมาใช้ใหม่ (Reusability):** คลาสที่ถูกสร้างขึ้นสามารถนำไปใช้ซ้ำได้ในหลายๆ โปรเจกต์ ช่วยลดเวลาและค่าใช้จ่ายในการพัฒนา
2. **การบำรุงรักษาที่ง่ายขึ้น (Easier Maintenance):** การแบ่งโปรแกรมออกเป็นวัตถุอิสระ ทำให้การแก้ไขหรือปรับปรุงส่วนใดส่วนหนึ่งของโปรแกรมทำได้ง่ายขึ้น โดยไม่ส่งผลกระทบต่อส่วนอื่นๆ มากนัก
3. **ความปลอดภัยของข้อมูล (Data Security):** ด้วยแนวคิด Encapsulation ข้อมูลภายในวัตถุจะถูกป้องกันจากการเข้าถึงโดยตรงจากภายนอก ทำให้ข้อมูลมีความปลอดภัยมากขึ้น
4. **ความยืดหยุ่น (Flexibility):** Polymorphism และ Inheritance ช่วยให้โปรแกรมมีความยืดหยุ่นสูง สามารถปรับเปลี่ยนหรือเพิ่มฟังก์ชันการทำงานใหม่ๆ ได้ง่าย

5. **การจัดการความซับซ้อน (Managing Complexity):** OOP ช่วยให้สามารถจำลองโลกแห่งความเป็นจริงให้อยู่ในรูปแบบของวัตถุ ทำให้การออกแบบและพัฒนาโปรแกรมที่มีความซับซ้อนทำได้ง่ายขึ้น

1.3 หลักการสำคัญของ OOP

หลักการสำคัญ 4 ประการของ OOP ที่เป็นหัวใจของแนวคิดนี้ ได้แก่:

1. **Encapsulation (การห่อหุ้ม):** คือการรวมข้อมูล (data) และเมธอด (methods) ที่ทำงานกับข้อมูลนั้นๆ เข้าไว้ด้วยกันในหน่วยเดียวที่เรียกว่า “คลาส” (Class) พร้อมทั้งซ่อนรายละเอียดการทำงานภายในจากภายนอก เพื่อป้องกันการเข้าถึงข้อมูลโดยตรงที่ไม่เหมาะสม
2. **Inheritance (การสืบทอดคุณสมบัติ):** คือกลไกที่ทำให้คลาสหนึ่ง (Subclass หรือ Child Class) สามารถรับคุณสมบัติ (attributes) และพฤติกรรม (methods) ของอีกคลาสหนึ่ง (Superclass หรือ Parent Class) มาใช้งานได้ ช่วยให้สามารถสร้างคลาสใหม่ที่ต่อยอดจากคลาสเดิมได้ง่าย และส่งเสริมการนำโค้ดกลับมาใช้ใหม่
3. **Polymorphism (การพหุสัณฐาน):** คือความสามารถของวัตถุที่จะแสดงพฤติกรรมได้หลายรูปแบบ ขึ้นอยู่กับบริบทหรือประเภทของวัตถุนั้นๆ โดยมักจะเกี่ยวข้องกับการ Overloading (เมธอดชื่อเดียวกันแต่พารามิเตอร์ต่างกัน) และ Overriding (เมธอดชื่อเดียวกันในคลาสลูกที่เขียนทับเมธอดในคลาสแม่)
4. **Abstraction (การซ่อนรายละเอียด):** คือการแสดงเฉพาะข้อมูลที่จำเป็นและซ่อนรายละเอียดที่ไม่จำเป็นออกไปจากผู้ใช้งาน ช่วยให้ผู้ใช้สามารถโต้ตอบกับวัตถุได้โดยไม่ต้องรู้ว่าภายในวัตถุนั้นทำงานอย่างไร มักจะใช้ Abstract Classes และ Interfaces ในการ implement

---

บทที่ 2: พื้นฐานภาษาโปรแกรมเชิงวัตถุ

2.1 โครงสร้างพื้นฐานของโปรแกรม (Structure of a Program)

ก่อนที่จะเจาะลึกถึงแนวคิดเชิงวัตถุ สิ่งสำคัญคือต้องเข้าใจโครงสร้างพื้นฐานของภาษาโปรแกรมที่เราจะใช้ในการเรียนรู้ โดยในเอกสารประกอบการสอนนี้จะเน้นที่ภาษา Java และ C++ เป็นหลัก เนื่องจากเป็นภาษาที่นิยมใช้ในการสอนและพัฒนาโปรแกรมเชิงวัตถุ

2.1.1 โครงสร้างโปรแกรม Java เบื้องต้น

โปรแกรม Java ทุกโปรแกรมจะต้องมีอย่างน้อยหนึ่งคลาส และมีเมธอด `main` เป็นจุดเริ่มต้นของการทำงานของโปรแกรม ตัวอย่างเช่น:

```java
public class MyFirstProgram {
    public static void main(String[] args) {
        System.out.println("Hello, OOP!"); // แสดงผลข้อความออกทางหน้าจอ
    }
}
```

- `public class MyFirstProgram`: เป็นการประกาศคลาสชื่อ `MyFirstProgram` โดย `public` หมายถึงคลาสนี้สามารถเข้าถึงได้จากที่ใดก็ได้
- `public static void main(String[] args)`: เป็นเมธอดหลักที่โปรแกรมจะเริ่มทำงานจากตรงนี้
  - `public`: เมธอดนี้สามารถเข้าถึงได้จากที่ใดก็ได้
  - `static`: เมธอดนี้สามารถเรียกใช้ได้โดยไม่ต้องสร้างอ็อบเจกต์ของคลาส
  - `void`: เมธอดนี้ไม่มีการส่งค่ากลับ
  - `main`: ชื่อของเมธอดหลัก
  - `String[] args`: พารามิเตอร์สำหรับรับอาร์กิวเมนต์ที่เป็นสตริงจากบรรทัดคำสั่ง
- `System.out.println("Hello, OOP!");`: คำสั่งสำหรับแสดงข้อความออกทางหน้าจอ

2.1.2 โครงสร้างโปรแกรม C++ เบื้องต้น

โปรแกรม C++ ก็มีโครงสร้างที่คล้ายคลึงกัน โดยมีฟังก์ชัน `main` เป็นจุดเริ่มต้นของการทำงาน และมีการใช้ `include` เพื่อนำไลบรารีต่างๆ เข้ามาใช้งาน ตัวอย่างเช่น:

```cpp
#include <iostream> // สำหรับใช้งานฟังก์ชัน input/output

int main() {
    std::cout << "Hello, OOP!" << std::endl; // แสดงผลข้อความออกทางหน้าจอ
    return 0; // คืนค่า 0 เพื่อระบุว่าโปรแกรมทำงานสำเร็จ
}
```

- `#include <iostream>` : เป็นการนำไลบรารี `iostream` เข้ามาใช้งาน ซึ่งมีฟังก์ชันสำหรับการรับและแสดงผลข้อมูล
- `int main()` : เป็นฟังก์ชันหลักที่โปรแกรมจะเริ่มทำงานจากตรงนี้ โดย `int` หมายถึงฟังก์ชันนี้จะคืนค่าเป็นจำนวนเต็ม
- `std::cout << "Hello, OOP!" << std::endl;` : คำสั่งสำหรับแสดงข้อความออกทางหน้าจอ โดย `std::cout` คือ Object สำหรับแสดงผล และ `std::endl` คือการขึ้นบรรทัดใหม่
- `return 0;` : คำสั่งคืนค่า 0 เพื่อบอกระบบปฏิบัติการว่าโปรแกรมทำงานเสร็จสมบูรณ์โดยไม่มีข้อผิดพลาด

2.2 ชนิดข้อมูลและตัวแปร (Data Types and Variables)

ชนิดข้อมูล (Data Types) กำหนดประเภทของข้อมูลที่ตัวแปรสามารถเก็บได้ และตัวแปร (Variables) คือชื่อที่ใช้ในการอ้างอิงถึงตำแหน่งในหน่วยความจำที่เก็บข้อมูลนั้นๆ

2.2.1 ชนิดข้อมูลพื้นฐาน (Primitive Data Types)

ชนิดข้อมูล (Java)	ชนิดข้อมูล (C++)	คำอธิบาย	ตัวอย่าง
`byte`	`char` (1 byte)	จำนวนเต็มขนาดเล็ก	`byte b = 10;`
`short`	`short` (2 bytes)	จำนวนเต็มขนาดกลาง	`short s = 1000;`
`int`	`int` (4 bytes)	จำนวนเต็มที่ใช้บ่อยที่สุด	`int i = 100000;`
`long`	`long` (4 or 8 bytes)	จำนวนเต็มขนาดใหญ่	`long l = 10000000000L;`
`float`	`float` (4 bytes)	จำนวนทศนิยมความแม่นยำเดี่ยว	`float f = 3.14f;`
`double`	`double` (8 bytes)	จำนวนทศนิยมความแม่นยำสองเท่า	`double d = 3.14159;`
`boolean`	`bool` (1 byte)	ค่าตรรกะ (true/false)	`boolean b = true;`
`char`	`char` (1 byte)	อักขระเดี่ยว	`char c = 'A';`

2.2.2 การประกาศและกำหนดค่าตัวแปร (Declaring and Initializing Variables)

การประกาศตัวแปรคือการบอกคอมไพเลอร์ว่าเราต้องการใช้ตัวแปรชื่ออะไรและมีชนิดข้อมูลแบบใด ส่วนการกำหนดค่าคือการให้ค่าเริ่มต้นแก่ตัวแปรนั้นๆ

Java:

```
int age = 25; // ประกาศตัวแปร age ชนิด int และกำหนดค่าเริ่มต้นเป็น 25
String name = "Alice"; // ประกาศตัวแปร name ชนิด String และกำหนดค่าเริ่มต้นเป็น "Alice"
```

C++:

```
int age = 25; // ประกาศตัวแปร age ชนิด int และกำหนดค่าเริ่มต้นเป็น 25
std::string name = "Alice"; // ประกาศตัวแปร name ชนิด std::string และกำหนดค่าเริ่มต้นเป็น "Alice"
```

2.3 ตัวดำเนินการ (Operators)

ตัวดำเนินการคือสัญลักษณ์ที่ใช้ในการดำเนินการกับค่าหรือตัวแปร

2.3.1 ตัวดำเนินการทางคณิตศาสตร์ (Arithmetic Operators)

| ตัวดำเนินการ | คำอธิบาย | ตัวอย่าง |

`+`	การบวก	`5 + 3` ได้ `8`	
`-`	การลบ	`5 - 3` ได้ `2`	
`*`	การคูณ	`5 * 3` ได้ `15`	
`/`	การหาร	`5 / 3` ได้ `1` (จำนวนเต็ม) หรือ `1.66` (ทศนิยม)	
`%`	การหารเอาเศษ (Modulo)	`5 % 3` ได้ `2`	

2.3.2 ตัวดำเนินการเปรียบเทียบ (Relational Operators)

| ตัวดำเนินการ | คำอธิบาย | ตัวอย่าง |

`==`	เท่ากับ	`a == b`	
`!=`	ไม่เท่ากับ	`a != b`	
`>`	มากกว่า	`a > b`	
`<`	น้อยกว่า	`a < b`	
`>=`	มากกว่าหรือเท่ากับ	`a >= b`	
`<=`	น้อยกว่าหรือเท่ากับ	`a <= b`	

2.3.3 ตัวดำเนินการตรรกะ (Logical Operators)

| ตัวดำเนินการ | คำอธิบาย | ตัวอย่าง |

`&&` (Java), `and` (C++)	AND (และ)	`(a > 0) && (b < 10)`	
`||` (Java), `or` (C++)	OR (หรือ)	`(a > 0) || (b < 10)`	
`!` (Java), `not` (C++)	NOT (ไม่)	`!(a > 0)`	

2.4 คำสั่งควบคุมการทำงาน (Control Flow Statements)

คำสั่งควบคุมการทำงานใช้ในการกำหนดลำดับการทำงานของโปรแกรม

2.4.1 คำสั่งเลือกเงื่อนไข (Conditional Statements)

`if-else` Statement:

```java
// Java
int score = 85;
if (score >= 80) {
    System.out.println("Grade A");
} else if (score >= 70) {
    System.out.println("Grade B");
} else {
    System.out.println("Grade C");
}
```

```cpp
// C++
int score = 85;
if (score >= 80) {
    std::cout << "Grade A" << std::endl;
} else if (score >= 70) {
    std::cout << "Grade B" << std::endl;
} else {
    std::cout << "Grade C" << std::endl;
}
```

`switch` Statement:

```java
// Java
int day = 3;
switch (day) {
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
        break;
    default:
        System.out.println("Other day");
}
```

```cpp
// C++
int day = 3;
switch (day) {
    case 1:
        std::cout << "Monday" << std::endl;
        break;
    case 2:
        std::cout << "Tuesday" << std::endl;
        break;
    default:
        std::cout << "Other day" << std::endl;
}
```

2.4.2 คำสั่งวนซ้ำ (Looping Statements)

`for` Loop:

```java
// Java
for (int i = 0; i < 5; i++) {
    System.out.println(i);
}
```

```cpp
// C++
for (int i = 0; i < 5; i++) {
    std::cout << i << std::endl;
}
```

`while` Loop:

```java
// Java
int i = 0;
while (i < 5) {
    System.out.println(i);
    i++;
}
```

```cpp
// C++
int i = 0;
while (i < 5) {
    std::cout << i << std::endl;
    i++;
}
```

`do-while` Loop (Java/C++):

```java
// Java
int i = 0;
do {
    System.out.println(i);
    i++;
} while (i < 5);
```

```cpp
// C++
int i = 0;
do {
    std::cout << i << std::endl;
    i++;
} while (i < 5);
```

2.5 อาร์เรย์ (Arrays)

อาร์เรย์คือโครงสร้างข้อมูลที่ใช้เก็บข้อมูลชนิดเดียวกันหลายๆ ค่าในตัวแปรเดียว

2.5.1 การประกาศและใช้งานอาร์เรย์

Java:

```java
int[] numbers = new int[5]; // ประกาศอาร์เรย์ของจำนวนเต็มขนาด 5 ช่อง
numbers[0] = 10; // กำหนดค่าให้กับช่องแรก
System.out.println(numbers[0]); // แสดงผลค่าในช่องแรก

String[] names = {"Alice", "Bob", "Charlie"}; // ประกาศและกำหนดค่าเริ่มต้น
for (String name : names) { // Enhanced for loop
    System.out.println(name);
}
```

C++:

```cpp
int numbers[5]; // ประกาศอาร์เรย์ของจำนวนเต็มขนาด 5 ช่อง
numbers[0] = 10; // กำหนดค่าให้กับช่องแรก
std::cout << numbers[0] << std::endl; // แสดงผลค่าในช่องแรก

std::string names[] = {"Alice", "Bob", "Charlie"}; // ประกาศและกำหนดค่าเริ่มต้น
for (std::string name : names) { // Range-based for loop (C++11 onwards)
    std::cout << name << std::endl;
}
```

บทที่ 3: คลาสและอ็อบเจกต์

3.1 แนวคิดของคลาส (Class)

คลาส (Class) คือพิมพ์เขียว (Blueprint) หรือแม่แบบ (Template) ที่ใช้สำหรับสร้างวัตถุ (Objects) คลาสจะกำหนดโครงสร้างและพฤติกรรมของวัตถุที่สร้างขึ้นจากคลาสนั้นๆ โดยประกอบด้วย:

- **คุณสมบัติ (Attributes/Fields/Properties):** ข้อมูลที่อธิบายสถานะของวัตถุ เช่น ชื่อ อายุ สี ขนาด
- **เมธอด (Methods/Functions):** การกระทำหรือพฤติกรรมที่วัตถุสามารถทำได้ เช่น เดิน กิน นอน คำนวณ

3.1.1 การประกาศคลาส (Declaring a Class)

การประกาศคลาสในภาษา Java และ C++ มีรูปแบบที่คล้ายคลึงกัน:

Java:

```java
public class Dog {
    // Attributes
    String name;
    String breed;
    int age;

    // Methods
    public void bark() {
        System.out.println(name + " barks!");
    }

    public void eat() {
        System.out.println(name + " is eating.");
    }
}
```

C++:

```cpp
#include <string>
#include <iostream>

class Dog {
public:
    // Attributes
    std::string name;
    std::string breed;
    int age;

    // Methods
    void bark() {
        std::cout << name << " barks!" << std::endl;
    }

    void eat() {
        std::cout << name << " is eating." << std::endl;
    }
};
```

- `public class Dog` (Java) หรือ `class Dog` (C++): เป็นการประกาศคลาสชื่อ `Dog`
- `String name;`, `String breed;`, `int age;` (Java) หรือ `std::string name;`, `std::string breed;`, `int age;` (C++): เป็นการประกาศคุณสมบัติของคลาส
- `public void bark()` และ `public void eat()` (Java) หรือ `void bark()` และ `void eat()` (C++): เป็นการประกาศเมธอดของคลาส

3.2 แนวคิดของอ็อบเจกต์ (Object)

อ็อบเจกต์ (Object) คืออินสแตนซ์ (Instance) ที่ถูกสร้างขึ้นจากคลาส อ็อบเจกต์แต่ละตัวจะมีสถานะ (State) และพฤติกรรม (Behavior) เป็นของตัวเองตามที่คลาสกำหนดไว้

3.2.1 การสร้างอ็อบเจกต์ (Creating Objects)

การสร้างอ็อบเจกต์หรือการสร้างอินสแตนซ์ของคลาสทำได้โดยใช้คีย์เวิร์ด `new` ใน Java และ C++:

Java:

```java
public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog(); // สร้างอ็อบเจกต์ชื่อ myDog จากคลาส Dog
        myDog.name = "Buddy";
        myDog.breed = "Golden Retriever";
        myDog.age = 3;

        myDog.bark(); // เรียกใช้เมธอด bark ของอ็อบเจกต์ myDog
        myDog.eat();  // เรียกใช้เมธอด eat ของอ็อบเจกต์ myDog
    }
}
```

C++:

```cpp
#include <string>
#include <iostream>

class Dog {
public:
    std::string name;
    std::string breed;
    int age;

    void bark() {
        std::cout << name << " barks!" << std::endl;
    }

    void eat() {
        std::cout << name << " is eating." << std::endl;
    }
};

int main() {
    Dog myDog; // สร้างอ็อบเจกต์ชื่อ myDog จากคลาส Dog
    myDog.name = "Buddy";
    myDog.breed = "Golden Retriever";
    myDog.age = 3;

    myDog.bark(); // เรียกใช้เมธอด bark ของอ็อบเจกต์ myDog
    myDog.eat();  // เรียกใช้เมธอด eat ของอ็อบเจกต์ myDog

    return 0;
}
```

3.3 คอนสตรัคเตอร์ (Constructors)

คอนสตรัคเตอร์ (Constructor) คือเมธอดพิเศษที่ใช้ในการเริ่มต้น (Initialize) อ็อบเจกต์เมื่อถูกสร้างขึ้น คอนสตรัคเตอร์จะมีชื่อเดียวกับคลาสและไม่มีชนิดข้อมูลส่งกลับ (return type)

3.3.1 คอนสตรัคเตอร์เริ่มต้น (Default Constructor)

หากเราไม่ได้กำหนดคอนสตรัคเตอร์ใดๆ คอมไพเลอร์จะสร้างคอนสตรัคเตอร์เริ่มต้น (Default Constructor) ให้โดยอัตโนมัติ ซึ่งจะไม่มีพารามิเตอร์และไม่มีการทำงานใดๆ

3.3.2 คอนสตรัคเตอร์ที่มีพารามิเตอร์ (Parameterized Constructor)

เราสามารถกำหนดคอนสตรัคเตอร์ที่มีพารามิเตอร์เพื่อรับค่ามาเริ่มต้นคุณสมบัติของอ็อบเจกต์ได้:

Java:

```java
public class Dog {
    String name;
    String breed;
    int age;

    // Parameterized Constructor
    public Dog(String name, String breed, int age) {
        this.name = name;
        this.breed = breed;
        this.age = age;
    }

    public void bark() {
        System.out.println(name + " barks!");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog("Buddy", "Golden Retriever", 3); // สร้างอ็อบเจกต์พร้อมกำหนดค่าเริ่มต้น
        myDog.bark();
    }
}
```

C++:

```cpp
#include <string>
#include <iostream>

class Dog {
public:
    std::string name;
    std::string breed;
    int age;

    // Parameterized Constructor
    Dog(std::string name, std::string breed, int age) {
        this->name = name;
        this->breed = breed;
        this->age = age;
    }

    void bark() {
        std::cout << name << " barks!" << std::endl;
    }
};

int main() {
    Dog myDog("Buddy", "Golden Retriever", 3); // สร้างอ็อบเจกต์พร้อมกำหนดค่าเริ่มต้น
    myDog.bark();

    return 0;
}
```

- `this` (Java) หรือ `this->` (C++): ใช้เพื่ออ้างถึงคุณสมบัติของอ็อบเจกต์ปัจจุบัน เพื่อแยกความแตกต่างจากพารามิเตอร์ที่มีชื่อเดียวกัน

3.4 เมธอด (Methods)

เมธอด (Method) คือบล็อกของโค้ดที่ทำงานเฉพาะอย่าง เมธอดเป็นพฤติกรรมของวัตถุที่กำหนดไว้ในคลาส

3.4.1 การประกาศเมธอด (Declaring Methods)

เมธอดประกอบด้วย:

- **ชนิดข้อมูลส่งกลับ (Return Type):** ชนิดของข้อมูลที่เมธอดจะส่งกลับ (ถ้าไม่มี ให้ใช้ `void`)
- **ชื่อเมธอด (Method Name):** ชื่อที่ใช้เรียกเมธอด
- **พารามิเตอร์ (Parameters):** ข้อมูลที่ส่งเข้าไปในเมธอด (ถ้ามี)

Java:

```java
public class Calculator {
    public int add(int a, int b) { // เมธอด add รับค่า int สองตัวและคืนค่า int
        return a + b;
    }

    public void displayMessage(String message) { // เมธอด displayMessage ไม่คืนค่า
        System.out.println(message);
    }
}
```

C++:

```cpp
#include <iostream>
#include <string>

class Calculator {
public:
    int add(int a, int b) { // เมธอด add รับค่า int สองตัวและคืนค่า int
        return a + b;
    }

    void displayMessage(std::string message) { // เมธอด displayMessage ไม่คืนค่า
        std::cout << message << std::endl;
    }
};
```

3.4.2 การเรียกใช้เมธอด (Calling Methods)

เมธอดจะถูกเรียกใช้ผ่านอ็อบเจกต์ของคลาสนั้นๆ:

Java:

```java
public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        int sum = calc.add(5, 3); // เรียกใช้เมธอด add
        System.out.println("Sum: " + sum); // Output: Sum: 8

        calc.displayMessage("Hello from Calculator!"); // เรียกใช้เมธอด
displayMessage
    }
}
```

C++:

```cpp
#include <iostream>
#include <string>

class Calculator {
public:
    int add(int a, int b) {
        return a + b;
    }

    void displayMessage(std::string message) {
        std::cout << message << std::endl;
    }
};

int main() {
    Calculator calc;
    int sum = calc.add(5, 3); // เรียกใช้เมธอด add
    std::cout << "Sum: " << sum << std::endl; // Output: Sum: 8

    calc.displayMessage("Hello from Calculator!"); // เรียกใช้เมธอด
displayMessage

    return 0;
}
```

บทที่ 4: การห่อหุ้มและระดับการเข้าถึง

4.1 แนวคิดของการห่อหุ้ม (Encapsulation)

การห่อหุ้ม (Encapsulation) เป็นหนึ่งในหลักการพื้นฐานของ Object-Oriented Programming (OOP) ที่หมายถึงการรวมข้อมูล (data) และเมธอด (methods) ที่ทำงานกับข้อมูลนั้นๆ เข้าไว้ด้วยกันในหน่วยเดียวที่เรียกว่า “คลาส” (Class) พร้อมทั้งซ่อนรายละเอียดการทำงานภายในจากภายนอก เพื่อป้องกันการเข้าถึงข้อมูลโดยตรงที่ไม่เหมาะสม

ประโยชน์ของการห่อหุ้ม:

1. **การซ่อนข้อมูล (Data Hiding):** ป้องกันไม่ให้ข้อมูลภายในอ็อบเจกต์ถูกแก้ไขโดยตรงจากภายนอก ทำให้ข้อมูลมีความปลอดภัยและสอดคล้องกัน (consistent)
2. **การควบคุมการเข้าถึง (Controlled Access):** การเข้าถึงและแก้ไขข้อมูลจะทำได้ผ่านเมธอดที่กำหนดไว้เท่านั้น (เช่น getter และ setter methods) ซึ่งช่วยให้สามารถตรวจสอบความถูกต้องของข้อมูลก่อนการเปลี่ยนแปลงได้
3. **ความยืดหยุ่นในการเปลี่ยนแปลง (Flexibility and Maintainability):** หากมีการเปลี่ยนแปลงโครงสร้างข้อมูลภายในคลาส จะไม่ส่งผลกระทบต่อโค้ดภายนอกที่เรียกใช้งาน เนื่องจากโค้ดภายนอกยังคงเรียกใช้เมธอดเดิม

4.2 ระดับการเข้าถึง (Access Modifiers)

ระดับการเข้าถึง (Access Modifiers) เป็นคีย์เวิร์ดที่ใช้กำหนดขอบเขตการเข้าถึงคุณสมบัติ (attributes) และเมธอด (methods) ของคลาสในภาษาโปรแกรมเชิงวัตถุ ภาษา Java และ C++ มีระดับการเข้าถึงที่คล้ายคลึงกัน แต่มีคีย์เวิร์ดและพฤติกรรมบางส่วนที่แตกต่างกันเล็กน้อย

4.2.1 ระดับการเข้าถึงใน Java

Java มี 4 ระดับการเข้าถึงหลัก:

1. `public`: สามารถเข้าถึงได้จากทุกที่ ไม่ว่าจะเป็นภายในคลาสเดียวกัน คลาสอื่นในแพ็กเกจเดียวกัน หรือคลาสในแพ็กเกจอื่น
2. `private`: สามารถเข้าถึงได้เฉพาะภายในคลาสที่ประกาศเท่านั้น ไม่สามารถเข้าถึงได้จากภายนอกคลาส

3. **`protected` :** สามารถเข้าถึงได้ภายในคลาสเดียวกัน คลาสลูก (subclass) ไม่ว่าจะอยู่ในแพ็กเกจเดียวกันหรือต่างแพ็กเกจ และคลาสอื่นๆ ในแพ็กเกจเดียวกัน
4. **`default` (ไม่มีคีย์เวิร์ด):** หากไม่ระบุ access modifier จะถือว่าเป็น `default` หรือ `package-private` ซึ่งสามารถเข้าถึงได้เฉพาะภายในคลาสเดียวกันและคลาสอื่นๆ ในแพ็กเกจเดียวกันเท่านั้น

ตัวอย่าง Java:

```java
package com.example.model;

public class Person {
    public String name; // เข้าถึงได้จากทุกที่
    private int age;    // เข้าถึงได้เฉพาะภายในคลาส Person เท่านั้น
    protected String address; // เข้าถึงได้ภายในแพ็กเกจเดียวกันและคลาสลูก
    String email; // default access (package-private), เข้าถึงได้ภายในแพ็กเกจเดียวกัน

    public Person(String name, int age, String address, String email) {
        this.name = name;
        this.age = age;
        this.address = address;
        this.email = email;
    }

    public int getAge() { // Getter method สำหรับ age
        return age;
    }

    public void setAge(int age) { // Setter method สำหรับ age
        if (age > 0) {
            this.age = age;
        } else {
            System.out.println("Age cannot be negative.");
        }
    }

    public void displayInfo() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Address: " + address);
        System.out.println("Email: " + email);
    }
}
```

4.2.2 ระดับการเข้าถึงใน C++

C++ มี 3 ระดับการเข้าถึงหลัก:

1. **`public`** : สมาชิก (member) ที่ประกาศเป็น `public` สามารถเข้าถึงได้จากทุกที่ ทั้งภายในและภายนอกคลาส

2. **`private`:** สมาชิกที่ประกาศเป็น `private` สามารถเข้าถึงได้เฉพาะภายในคลาสที่ประกาศเท่านั้น ไม่สามารถเข้าถึงได้จากภายนอกคลาสโดยตรง
3. **`protected`:** สมาชิกที่ประกาศเป็น `protected` สามารถเข้าถึงได้ภายในคลาสที่ประกาศและคลาสที่สืบทอด (derived classes) เท่านั้น

ตัวอย่าง C++:

```cpp
#include <string>
#include <iostream>

class Person {
public:
    std::string name; // เข้าถึงได้จากทุกที่

    Person(std::string name, int age, std::string address) {
        this->name = name;
        this->age = age;
        this->address = address;
    }

    int getAge() { // Getter method สำหรับ age
        return age;
    }

    void setAge(int age) { // Setter method สำหรับ age
        if (age > 0) {
            this->age = age;
        } else {
            std::cout << "Age cannot be negative." << std::endl;
        }
    }

    void displayInfo() {
        std::cout << "Name: " << name << std::endl;
        std::cout << "Age: " << age << std::endl;
        std::cout << "Address: " << address << std::endl;
    }

protected:
    std::string address; // เข้าถึงได้ภายในคลาสและคลาสลูก

private:
    int age;    // เข้าถึงได้เฉพาะภายในคลาส Person เท่านั้น
};
```

4.3 Getter และ Setter Methods

เพื่อรักษาหลักการห่อหุ้ม เรามักจะใช้ **Getter Methods** และ **Setter Methods** ในการเข้าถึงและแก้ไขข้อมูล `private` หรือ `protected` ของอ็อบเจกต์

- **Getter Method (Accessor Method):** เป็นเมธอดที่ใช้สำหรับดึงค่าของคุณสมบัติ (attribute) ออกมา โดยทั่วไปจะมีชื่อขึ้นต้นด้วย `get` ตามด้วยชื่อคุณสมบัติ เช่น `getAge()`, `getName()`
- **Setter Method (Mutator Method):** เป็นเมธอดที่ใช้สำหรับกำหนดหรือแก้ไขค่าของคุณสมบัติ โดยทั่วไปจะมีชื่อขึ้นต้นด้วย `set` ตามด้วยชื่อคุณสมบัติ และรับพารามิเตอร์เป็นค่าใหม่ที่จะกำหนด เช่น `setAge(int newAge)`, `setName(String newName)`

ตัวอย่างการใช้งาน Getter และ Setter:

Java:

```
public class Main {
    public static void main(String[] args) {
        Person person1 = new Person("Alice", 30, "123 Main St",
"alice@example.com");

        // เข้าถึงข้อมูลผ่าน Getter
        System.out.println("Person's age: " + person1.getAge()); // Output:
Person's age: 30

        // แก้ไขข้อมูลผ่าน Setter
        person1.setAge(31);
        System.out.println("New age: " + person1.getAge()); // Output: New
age: 31

        person1.setAge(-5); // พยายามกำหนดค่าที่ไม่ถูกต้อง
        System.out.println("Age after invalid set: " + person1.getAge()); //
Output: Age after invalid set: 31 (ค่าไม่เปลี่ยน)
    }
}
```

C++:

```cpp
int main() {
    Person person1("Alice", 30, "123 Main St");

    // เข้าถึงข้อมูลผ่าน Getter
    std::cout << "Person's age: " << person1.getAge() << std::endl; //
Output: Person's age: 30

    // แก้ไขข้อมูลผ่าน Setter
    person1.setAge(31);
    std::cout << "New age: " << person1.getAge() << std::endl; // Output:
New age: 31

    person1.setAge(-5); // พยายามกำหนดค่าที่ไม่ถูกต้อง
    std::cout << "Age after invalid set: " << person1.getAge() << std::endl;
// Output: Age after invalid set: 31 (ค่าไม่เปลี่ยน)

    return 0;
}
```

บทที่ 5: การสืบทอดคุณสมบัติ

5.1 แนวคิดของการสืบทอดคุณสมบัติ (Inheritance)

การสืบทอดคุณสมบัติ (Inheritance) เป็นหนึ่งในหลักการสำคัญของ Object-Oriented Programming (OOP) ที่ช่วยให้คลาสหนึ่ง (เรียกว่า **Subclass** หรือ **Child Class**) สามารถรับคุณสมบัติ (attributes) และพฤติกรรม (methods) ของอีกคลาสหนึ่ง (เรียกว่า **Superclass** หรือ **Parent Class**) มาใช้งานได้ ซึ่งส่งเสริมแนวคิดการนำโค้ดกลับมาใช้ใหม่ (Code Reusability) และสร้างความสัมพันธ์แบบ “เป็น” (is-a relationship) ระหว่างคลาสต่างๆ

ประโยชน์ของการสืบทอดคุณสมบัติ:

1. **การนำโค้ดกลับมาใช้ใหม่ (Code Reusability):** คลาสลูกไม่จำเป็นต้องเขียนโค้ดซ้ำสำหรับคุณสมบัติและเมธอดที่มีอยู่ในคลาสแม่
2. **การขยายความสามารถ (Extensibility):** สามารถเพิ่มคุณสมบัติหรือเมธอดใหม่ๆ ในคลาสลูกได้ โดยไม่กระทบต่อคลาสแม่

3. **การสร้างลำดับชั้นของคลาส (Class Hierarchy):** ช่วยในการจัดระเบียบและสร้างความสัมพันธ์ระหว่างคลาส ทำให้โครงสร้างโปรแกรมมีความชัดเจนและจัดการได้ง่ายขึ้น

5.2 การประกาศคลาสแม่และคลาสลูก

5.2.1 การสืบทอดคุณสมบัติใน Java

ใน Java ใช้คีย์เวิร์ด `extends` ในการระบุว่าคลาสลูกสืบทอดมาจากคลาสแม่

ตัวอย่าง Java:

```java
// Superclass (คลาสแม่)
class Animal {
    String name;

    public Animal(String name) {
        this.name = name;
    }

    public void eat() {
        System.out.println(name + " is eating.");
    }
}

// Subclass (คลาสลูก) สืบทอดจาก Animal
class Dog extends Animal {
    String breed;

    public Dog(String name, String breed) {
        super(name); // เรียกใช้ constructor ของคลาสแม่
        this.breed = breed;
    }

    public void bark() {
        System.out.println(name + " barks!");
    }

    public void displayInfo() {
        System.out.println("Name: " + name + ", Breed: " + breed);
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog("Buddy", "Golden Retriever");
        myDog.eat();  // เมธอด eat สืบทอดมาจาก Animal
        myDog.bark(); // เมธอด bark เป็นของ Dog เอง
        myDog.displayInfo();
    }
}
```

- `class Dog extends Animal` : หมายความว่าคลาส `Dog` สืบทอดมาจากคลาส `Animal`
- `super(name)` : ใช้สำหรับเรียก constructor ของคลาสแม่

5.2.2 การสืบทอดคุณสมบัติใน C++

ใน C++ ใช้เครื่องหมาย `:` ตามด้วย `public` (หรือ `protected`, `private`) และชื่อคลาสแม่

ตัวอย่าง C++:

```cpp
#include <string>
#include <iostream>

// Superclass (คลาสแม่)
class Animal {
public:
    std::string name;

    Animal(std::string name) : name(name) {}

    void eat() {
        std::cout << name << " is eating." << std::endl;
    }
};

// Subclass (คลาสลูก) สืบทอดจาก Animal
class Dog : public Animal {
public:
    std::string breed;

    Dog(std::string name, std::string breed) : Animal(name), breed(breed) {}

    void bark() {
        std::cout << name << " barks!" << std::endl;
    }

    void displayInfo() {
        std::cout << "Name: " << name << ", Breed: " << breed << std::endl;
    }
};

int main() {
    Dog myDog("Buddy", "Golden Retriever");
    myDog.eat();  // เมธอด eat สืบทอดมาจาก Animal
    myDog.bark(); // เมธอด bark เป็นของ Dog เอง
    myDog.displayInfo();

    return 0;
}
```

- `class Dog : public Animal` : หมายความว่าคลาส `Dog` สืบทอดมาจากคลาส `Animal` แบบ `public`
- `Animal(name)` : ใช้สำหรับเรียก constructor ของคลาสแม่ใน initializer list

5.3 การเรียกใช้ Constructor ของคลาสแม่

เมื่อมีการสร้างอ็อบเจกต์ของคลาสลูก Constructor ของคลาสแม่จะถูกเรียกใช้ก่อนเสมอ เพื่อให้มั่นใจว่าส่วนของคลาสแม่ในอ็อบเจกต์นั้นได้รับการเริ่มต้นอย่างถูกต้อง

- **Java:** ใช้คีย์เวิร์ด `super()` เพื่อเรียก Constructor ของคลาสแม่ โดยต้องเป็นคำสั่งแรกใน Constructor ของคลาสลูก
- **C++:** เรียก Constructor ของคลาสแม่ใน initializer list ของ Constructor คลาสลูก

5.4 การ Overriding Methods

การ Overriding Method คือการที่คลาสลูกเขียนเมธอดที่มีชื่อ, ชนิดข้อมูลส่งกลับ และพารามิเตอร์เหมือนกับเมธอดในคลาสแม่ เพื่อให้คลาสลูกมีพฤติกรรมที่แตกต่างออกไปจากคลาสแม่สำหรับเมธอดนั้นๆ

5.4.1 การ Overriding ใน Java

ใน Java สามารถใช้ `@Override` annotation เพื่อช่วยตรวจสอบว่าเมธอดนั้นเป็นการ override เมธอดในคลาสแม่จริงหรือไม่

ตัวอย่าง Java:

```java
class Animal {
    String name;

    public Animal(String name) {
        this.name = name;
    }

    public void makeSound() {
        System.out.println(name + " makes a sound.");
    }
}

class Cat extends Animal {
    public Cat(String name) {
        super(name);
    }

    @Override
    public void makeSound() { // Overriding makeSound method
        System.out.println(name + " meows.");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Animal("Generic Animal");
        animal.makeSound(); // Output: Generic Animal makes a sound.

        Cat cat = new Cat("Whiskers");
        cat.makeSound();    // Output: Whiskers meows.
    }
}
```

5.4.2 การ Overriding ใน C++

ใน C++ สามารถใช้คีย์เวิร์ด `virtual` ในเมธอดของคลาสแม่ และ `override` ในเมธอดของคลาสลูก (C++11 ขึ้นไป) เพื่อระบุว่าเมธอดนั้นสามารถถูก override ได้ และเป็นการ override จริงๆ

ตัวอย่าง C++:

```cpp
#include <string>
#include <iostream>

class Animal {
public:
    std::string name;

    Animal(std::string name) : name(name) {}

    virtual void makeSound() { // ใช้ virtual เพื่อให้เมธอดนี้สามารถถูก override ได้
        std::cout << name << " makes a sound." << std::endl;
    }
};

class Cat : public Animal {
public:
    Cat(std::string name) : Animal(name) {}

    void makeSound() override { // ใช้ override เพื่อระบุว่าเป็นการ override เมธอดในคลาสแม่
        std::cout << name << " meows." << std::endl;
    }
};

int main() {
    Animal animal("Generic Animal");
    animal.makeSound(); // Output: Generic Animal makes a sound.

    Cat cat("Whiskers");
    cat.makeSound();    // Output: Whiskers meows.

    // การใช้งาน Polymorphism (จะกล่าวถึงในบทถัดไป)
    Animal* polyAnimal = new Cat("Tom");
    polyAnimal->makeSound(); // Output: Tom meows. (ถ้า makeSound เป็น virtual)
    delete polyAnimal;

    return 0;
}
```

- `virtual` : คีย์เวิร์ดนี้ใน C++ มีความสำคัญอย่างยิ่งสำหรับการทำ Polymorphism โดยเฉพาะเมื่อมีการเรียกเมธอดผ่าน Pointer หรือ Reference ของคลาสแม่

- `override` : เป็นตัวช่วยในการตรวจสอบ (compiler check) ว่าเมธอดนั้นเป็นการ override เมธอดในคลาสแม่จริงหรือไม่

บทที่ 6: การพหุสัณฐาน (Polymorphism)

6.1 แนวคิดของการพหุสัณฐาน (Polymorphism)

การพหุสัณฐาน (Polymorphism) เป็นหนึ่งในหลักการสำคัญของ Object-Oriented Programming (OOP) ที่หมายถึงความสามารถของวัตถุที่จะแสดงพฤติกรรมได้หลายรูปแบบ ขึ้นอยู่กับบริบทหรือประเภทของวัตถุนั้นๆ คำว่า “Polymorphism” มาจากภาษากรีก โดย “Poly” แปลว่า “หลาย” และ “morph” แปลว่า “รูปแบบ” หรือ “รูปร่าง” ดังนั้น Polymorphism จึงหมายถึง “หลายรูปแบบ”

Polymorphism ช่วยให้โค้ดมีความยืดหยุ่น สามารถขยายและบำรุงรักษาได้ง่ายขึ้น โดยมีสองรูปแบบหลักคือ:

1. **Compile-time Polymorphism (Static Polymorphism):** เกิดขึ้นในระหว่างการคอมไพล์ มักเกี่ยวข้องกับ Method Overloading และ Operator Overloading (ใน C++)
2. **Run-time Polymorphism (Dynamic Polymorphism):** เกิดขึ้นในระหว่างการรันโปรแกรม มักเกี่ยวข้องกับ Method Overriding และ Dynamic Method Dispatch

6.2 Compile-time Polymorphism: Method Overloading

Method Overloading คือการที่คลาสมีเมธอดหลายเมธอดที่มีชื่อเดียวกัน แต่มีรายการพารามิเตอร์ (Parameter List) ที่แตกต่างกัน ซึ่งอาจแตกต่างกันที่:

- จำนวนพารามิเตอร์
- ชนิดข้อมูลของพารามิเตอร์
- ลำดับของชนิดข้อมูลพารามิเตอร์

เมธอด Overloading ช่วยให้สามารถใช้ชื่อเมธอดที่สื่อความหมายเดียวกันในการทำงานที่คล้ายคลึงกันแต่รับอินพุตต่างกันได้

6.2.1 Method Overloading ใน Java

ตัวอย่าง Java:

```
class Calculator {
    // เมธอด add สำหรับบวกจำนวนเต็มสองตัว
    public int add(int a, int b) {
        return a + b;
    }

    // เมธอด add สำหรับบวกจำนวนเต็มสามตัว
    public int add(int a, int b, int c) {
        return a + b + c;
    }

    // เมธอด add สำหรับบวกจำนวนทศนิยมสองตัว
    public double add(double a, double b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println("Sum of two ints: " + calc.add(5, 10)); // เรียกใช้
add(int, int)
        System.out.println("Sum of three ints: " + calc.add(1, 2, 3)); //
เรียกใช้ add(int, int, int)
        System.out.println("Sum of two doubles: " + calc.add(2.5, 3.5)); //
เรียกใช้ add(double, double)
    }
}
```

6.2.2 Method Overloading ใน C++

ตัวอย่าง C++:

```cpp
#include <iostream>

class Calculator {
public:
    // เมธอด add สำหรับบวกจำนวนเต็มสองตัว
    int add(int a, int b) {
        return a + b;
    }

    // เมธอด add สำหรับบวกจำนวนเต็มสามตัว
    int add(int a, int b, int c) {
        return a + b + c;
    }

    // เมธอด add สำหรับบวกจำนวนทศนิยมสองตัว
    double add(double a, double b) {
        return a + b;
    }
};

int main() {
    Calculator calc;
    std::cout << "Sum of two ints: " << calc.add(5, 10) << std::endl; //
เรียกใช้ add(int, int)
    std::cout << "Sum of three ints: " << calc.add(1, 2, 3) << std::endl; //
เรียกใช้ add(int, int, int)
    std::cout << "Sum of two doubles: " << calc.add(2.5, 3.5) << std::endl;
// เรียกใช้ add(double, double)
    return 0;
}
```

6.3 Run-time Polymorphism: Method Overriding

Method Overriding คือการที่คลาสลูก (Subclass) มีเมธอดที่มีชื่อ, ชนิดข้อมูลส่งกลับ และรายการพารามิเตอร์เหมือนกับเมธอดในคลาสแม่ (Superclass) ทุกประการ โดยมีวัตถุประสงค์เพื่อเปลี่ยนพฤติกรรมของเมธอดนั้นในคลาสลูกให้แตกต่างไปจากคลาสแม่

6.3.1 Method Overriding ใน Java

ใน Java การ Overriding เกิดขึ้นโดยธรรมชาติเมื่อคลาสลูกมีเมธอดที่ตรงตามเงื่อนไข สามารถใช้ `@Override` annotation เพื่อช่วยตรวจสอบความถูกต้อง

ตัวอย่าง Java:

```java
class Animal {
    public void makeSound() {
        System.out.println("Animal makes a sound.");
    }
}

class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog barks.");
    }
}

class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Cat meows.");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Animal();
        myAnimal.makeSound(); // Output: Animal makes a sound.

        Dog myDog = new Dog();
        myDog.makeSound();    // Output: Dog barks.

        Cat myCat = new Cat();
        myCat.makeSound();    // Output: Cat meows.

        // Run-time Polymorphism: Dynamic Method Dispatch
        Animal animalRef1 = new Dog(); // อ้างอิงถึง Dog ด้วย Animal reference
        animalRef1.makeSound();        // Output: Dog barks. (เมธอดของ Dog ถูกเรียก)

        Animal animalRef2 = new Cat(); // อ้างอิงถึง Cat ด้วย Animal reference
        animalRef2.makeSound();        // Output: Cat meows. (เมธอดของ Cat ถูกเรียก)
    }
}
```

6.3.2 Method Overriding ใน C++

ใน C++ การทำ Method Overriding เพื่อให้เกิด Run-time Polymorphism จำเป็นต้องใช้คีย์เวิร์ด `virtual` กับเมธอดในคลาสแม่ และ `override` กับเมธอดในคลาสลูก (ตั้งแต่ C++11)

ตัวอย่าง C++:

```cpp
#include <iostream>

class Animal {
public:
    virtual void makeSound() { // ต้องใช้ virtual เพื่อให้เกิด dynamic dispatch
        std::cout << "Animal makes a sound." << std::endl;
    }
};

class Dog : public Animal {
public:
    void makeSound() override { // ใช้ override เพื่อระบุว่าเป็นการ override
        std::cout << "Dog barks." << std::endl;
    }
};

class Cat : public Animal {
public:
    void makeSound() override {
        std::cout << "Cat meows." << std::endl;
    }
};

int main() {
    Animal myAnimal;
    myAnimal.makeSound(); // Output: Animal makes a sound.

    Dog myDog;
    myDog.makeSound();    // Output: Dog barks.

    Cat myCat;
    myCat.makeSound();    // Output: Cat meows.

    // Run-time Polymorphism: Dynamic Method Dispatch
    Animal* animalPtr1 = new Dog(); // Pointer ของ Animal ชี้ไปที่ Object ของ
Dog
    animalPtr1->makeSound();        // Output: Dog barks. (เมธอดของ Dog ถูก
เรียก)
    delete animalPtr1;

    Animal* animalPtr2 = new Cat(); // Pointer ของ Animal ชี้ไปที่ Object ของ
Cat
    animalPtr2->makeSound();        // Output: Cat meows. (เมธอดของ Cat ถูก
เรียก)
    delete animalPtr2;
```

```
    return 0;
}
```

6.4 Dynamic Binding (Dynamic Method Dispatch)

Dynamic Binding หรือ **Dynamic Method Dispatch** คือกระบวนการที่ระบบตัดสินใจว่าจะเรียกใช้เมธอดเวอร์ชันใด (จากคลาสแม่หรือคลาสลูก) ในระหว่างการรันโปรแกรม (run-time) ไม่ใช่ตอนคอมไพล์ (compile-time) ซึ่งเป็นหัวใจสำคัญของ Run-time Polymorphism

เกิดขึ้นเมื่อ:

- มีการอ้างอิงถึงอ็อบเจกต์ของคลาสลูกด้วยตัวแปรอ้างอิง (reference variable) หรือพอยน์เตอร์ (pointer) ของคลาสแม่
- เมธอดที่ถูกเรียกเป็นเมธอดที่ถูก Overridden ในคลาสลูก (และเป็น `virtual` ใน C++)

ในตัวอย่าง Java และ C++ ข้างต้นที่แสดงการ Overriding จะเห็นได้ว่าเมื่อ `Animal animalRef1 = new Dog();` หรือ `Animal* animalPtr1 = new Dog();` แล้วเรียก `animalRef1.makeSound();` หรือ `animalPtr1->makeSound();` เมธอด `makeSound()` ของคลาส `Dog` จะถูกเรียกใช้ ไม่ใช่ของ `Animal` นี่คือตัวอย่างของการทำงานของ Dynamic Binding

---

บทที่ 5: การสืบทอดคุณสมบัติ

5.1 แนวคิดของการสืบทอดคุณสมบัติ (Inheritance)

การสืบทอดคุณสมบัติ (Inheritance) เป็นหนึ่งในหลักการสำคัญของ Object-Oriented Programming (OOP) ที่ช่วยให้คลาสหนึ่ง (เรียกว่า **Subclass** หรือ **Child Class**) สามารถรับคุณสมบัติ (attributes) และพฤติกรรม (methods) ของอีกคลาสหนึ่ง (เรียกว่า **Superclass** หรือ **Parent Class**) มาใช้งานได้ ซึ่งส่งเสริมแนวคิดการนำโค้ดกลับมาใช้ใหม่ (Code Reusability) และสร้างความสัมพันธ์แบบ “เป็น” (is-a relationship) ระหว่างคลาสต่างๆ

ประโยชน์ของการสืบทอดคุณสมบัติ:

1. **การนำโค้ดกลับมาใช้ใหม่ (Code Reusability):** คลาสลูกไม่จำเป็นต้องเขียนโค้ดซ้ำสำหรับคุณสมบัติและเมธอดที่มีอยู่ในคลาสแม่
2. **การขยายความสามารถ (Extensibility):** สามารถเพิ่มคุณสมบัติหรือเมธอดใหม่ๆ ในคลาสลูกได้ โดยไม่กระทบต่อคลาสแม่
3. **การสร้างลำดับชั้นของคลาส (Class Hierarchy):** ช่วยในการจัดระเบียบและสร้างความสัมพันธ์ระหว่างคลาส ทำให้โครงสร้างโปรแกรมมีความชัดเจนและจัดการได้ง่ายขึ้น

5.2 การประกาศคลาสแม่และคลาสลูก

5.2.1 การสืบทอดคุณสมบัติใน Java

ใน Java ใช้คีย์เวิร์ด `extends` ในการระบุว่าคลาสลูกสืบทอดมาจากคลาสแม่

ตัวอย่าง Java:

```java
// Superclass (คลาสแม่)
class Animal {
    String name;

    public Animal(String name) {
        this.name = name;
    }

    public void eat() {
        System.out.println(name + " is eating.");
    }
}

// Subclass (คลาสลูก) สืบทอดจาก Animal
class Dog extends Animal {
    String breed;

    public Dog(String name, String breed) {
        super(name); // เรียกใช้ constructor ของคลาสแม่
        this.breed = breed;
    }

    public void bark() {
        System.out.println(name + " barks!");
    }

    public void displayInfo() {
        System.out.println("Name: " + name + ", Breed: " + breed);
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog("Buddy", "Golden Retriever");
        myDog.eat();  // เมธอด eat สืบทอดมาจาก Animal
        myDog.bark(); // เมธอด bark เป็นของ Dog เอง
        myDog.displayInfo();
    }
}
```

- `class Dog extends Animal` : หมายความว่าคลาส `Dog` สืบทอดมาจากคลาส `Animal`
- `super(name)` : ใช้สำหรับเรียก constructor ของคลาสแม่

5.2.2 การสืบทอดคุณสมบัติใน C++

ใน C++ ใช้เครื่องหมาย `:` ตามด้วย `public` (หรือ `protected`, `private`) และชื่อคลาสแม่

ตัวอย่าง C++:

```cpp
#include <string>
#include <iostream>

// Superclass (คลาสแม่)
class Animal {
public:
    std::string name;

    Animal(std::string name) : name(name) {}

    void eat() {
        std::cout << name << " is eating." << std::endl;
    }
};

// Subclass (คลาสลูก) สืบทอดจาก Animal
class Dog : public Animal {
public:
    std::string breed;

    Dog(std::string name, std::string breed) : Animal(name), breed(breed) {}

    void bark() {
        std::cout << name << " barks!" << std::endl;
    }

    void displayInfo() {
        std::cout << "Name: " << name << ", Breed: " << breed << std::endl;
    }
};

int main() {
    Dog myDog("Buddy", "Golden Retriever");
    myDog.eat();  // เมธอด eat สืบทอดมาจาก Animal
    myDog.bark(); // เมธอด bark เป็นของ Dog เอง
    myDog.displayInfo();

    return 0;
}
```

- `class Dog : public Animal` : หมายความว่าคลาส `Dog` สืบทอดมาจากคลาส `Animal` แบบ `public`
- `Animal(name)` : ใช้สำหรับเรียก constructor ของคลาสแม่ใน initializer list

5.3 การเรียกใช้ Constructor ของคลาสแม่

เมื่อมีการสร้างอ็อบเจกต์ของคลาสลูก Constructor ของคลาสแม่จะถูกเรียกใช้ก่อนเสมอ เพื่อให้มั่นใจว่าส่วนของคลาสแม่ในอ็อบเจกต์นั้นได้รับการเริ่มต้นอย่างถูกต้อง

- **Java:** ใช้คีย์เวิร์ด `super()` เพื่อเรียก Constructor ของคลาสแม่ โดยต้องเป็นคำสั่งแรกใน Constructor ของคลาสลูก
- **C++:** เรียก Constructor ของคลาสแม่ใน initializer list ของ Constructor คลาสลูก

5.4 การ Overriding Methods

การ Overriding Method คือการที่คลาสลูกเขียนเมธอดที่มีชื่อ, ชนิดข้อมูลส่งกลับ และพารามิเตอร์เหมือนกับเมธอดในคลาสแม่ เพื่อให้คลาสลูกมีพฤติกรรมที่แตกต่างออกไปจากคลาสแม่สำหรับเมธอดนั้นๆ

5.4.1 การ Overriding ใน Java

ใน Java สามารถใช้ `@Override` annotation เพื่อช่วยตรวจสอบว่าเมธอดนั้นเป็นการ override เมธอดในคลาสแม่จริงหรือไม่

ตัวอย่าง Java:

```java
class Animal {
    String name;

    public Animal(String name) {
        this.name = name;
    }

    public void makeSound() {
        System.out.println(name + " makes a sound.");
    }
}

class Cat extends Animal {
    public Cat(String name) {
        super(name);
    }

    @Override
    public void makeSound() { // Overriding makeSound method
        System.out.println(name + " meows.");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Animal("Generic Animal");
        animal.makeSound(); // Output: Generic Animal makes a sound.

        Cat cat = new Cat("Whiskers");
        cat.makeSound();    // Output: Whiskers meows.
    }
}
```

5.4.2 การ Overriding ใน C++

ใน C++ สามารถใช้คีย์เวิร์ด `virtual` ในเมธอดของคลาสแม่ และ `override` ในเมธอดของคลาสลูก (C++11 ขึ้นไป) เพื่อระบุว่าเมธอดนั้นสามารถถูก override ได้ และเป็นการ override จริงๆ

ตัวอย่าง C++:

```cpp
#include <string>
#include <iostream>

class Animal {
public:
    std::string name;

    Animal(std::string name) : name(name) {}

    virtual void makeSound() { // ใช้ virtual เพื่อให้เมธอดนี้สามารถถูก override ได้
        std::cout << name << " makes a sound." << std::endl;
    }
};

class Cat : public Animal {
public:
    Cat(std::string name) : Animal(name) {}

    void makeSound() override { // ใช้ override เพื่อระบุว่าเป็นการ override เมธอดในคลาสแม่
        std::cout << name << " meows." << std::endl;
    }
};

int main() {
    Animal animal("Generic Animal");
    animal.makeSound(); // Output: Generic Animal makes a sound.

    Cat cat("Whiskers");
    cat.makeSound();    // Output: Whiskers meows.

    // การใช้งาน Polymorphism (จะกล่าวถึงในบทถัดไป)
    Animal* polyAnimal = new Cat("Tom");
    polyAnimal->makeSound(); // Output: Tom meows. (ถ้า makeSound เป็น virtual)
    delete polyAnimal;

    return 0;
}
```

- `virtual` : คีย์เวิร์ดนี้ใน C++ มีความสำคัญอย่างยิ่งสำหรับการทำ Polymorphism โดยเฉพาะเมื่อมีการเรียกเมธอดผ่าน Pointer หรือ Reference ของคลาสแม่

- `override` : เป็นตัวช่วยในการตรวจสอบ (compiler check) ว่าเมธอดนั้นเป็นการ override เมธอดในคลาสแม่จริงหรือไม่

บทที่ 7: คลาสเชิงนามธรรมและอินเทอร์เฟซ

7.1 แนวคิดของ Abstraction (การซ่อนรายละเอียด)

Abstraction (การซ่อนรายละเอียด) เป็นหนึ่งในหลักการสำคัญของ Object-Oriented Programming (OOP) ที่มุ่งเน้นการแสดงเฉพาะข้อมูลที่จำเป็นและซ่อนรายละเอียดที่ไม่จำเป็นออกไปจากผู้ใช้งาน ช่วยให้ผู้ใช้สามารถโต้ตอบกับวัตถุได้โดยไม่ต้องรู้ว่าภายในวัตถุนั้นทำงานอย่างไร หลักการนี้ช่วยลดความซับซ้อนของระบบและทำให้โค้ดเข้าใจง่ายขึ้น

เราสามารถทำ Abstraction ได้โดยใช้ **Abstract Classes** และ **Interfaces**

7.2 คลาสเชิงนามธรรม (Abstract Classes)

คลาสเชิงนามธรรม (Abstract Class) คือคลาสที่ไม่สามารถสร้างอ็อบเจกต์ได้โดยตรง (ไม่สามารถ `new` ได้) แต่ถูกออกแบบมาเพื่อเป็นคลาสแม่ให้คลาสอื่นสืบทอดไปใช้งาน คลาสเชิงนามธรรมสามารถมีทั้งเมธอดปกติ (concrete methods) และเมธอดเชิงนามธรรม (abstract methods)

เมธอดเชิงนามธรรม (Abstract Method) คือเมธอดที่ประกาศไว้แต่ไม่มีการ implement (ไม่มี body) คลาสลูกที่สืบทอดจากคลาสเชิงนามธรรมจะต้อง implement เมธอดเชิงนามธรรมทั้งหมดที่อยู่ในคลาสแม่ มิฉะนั้นคลาสลูกนั้นก็จะต้องถูกประกาศเป็นคลาสเชิงนามธรรมด้วย

7.2.1 การประกาศ Abstract Class และ Abstract Method ใน Java

ใน Java ใช้คีย์เวิร์ด `abstract` ในการประกาศคลาสและเมธอด

ตัวอย่าง Java:

```java
// Abstract Class
abstract class Shape {
    String color;

    public Shape(String color) {
        this.color = color;
    }

    // Abstract method (ไม่มี body)
    public abstract double getArea();

    // Concrete method
    public String getColor() {
        return color;
    }

    public void displayColor() {
        System.out.println("Color: " + color);
    }
}

// Concrete Class สืบทอดจาก Shape
class Circle extends Shape {
    double radius;

    public Circle(String color, double radius) {
        super(color);
        this.radius = radius;
    }

    @Override
    public double getArea() { // ต้อง implement abstract method getArea()
        return Math.PI * radius * radius;
    }
}

// Concrete Class สืบทอดจาก Shape
class Rectangle extends Shape {
    double width;
    double height;

    public Rectangle(String color, double width, double height) {
        super(color);
        this.width = width;
        this.height = height;
```

```
    }

    @Override
    public double getArea() { // ต้อง implement abstract method getArea()
        return width * height;
    }
}

public class Main {
    public static void main(String[] args) {
        // Shape s = new Shape("Red"); // Error: Cannot instantiate the type
Shape

        Circle circle = new Circle("Red", 5.0);
        System.out.println("Circle Area: " + circle.getArea());
        circle.displayColor();

        Rectangle rectangle = new Rectangle("Blue", 4.0, 6.0);
        System.out.println("Rectangle Area: " + rectangle.getArea());
        rectangle.displayColor();

        // Polymorphism with Abstract Class
        Shape s1 = new Circle("Green", 7.0);
        Shape s2 = new Rectangle("Yellow", 3.0, 5.0);

        System.out.println("Area of s1: " + s1.getArea());
        System.out.println("Area of s2: " + s2.getArea());
    }
}
```

7.2.2 การประกาศ Abstract Class และ Abstract Method ใน C++

ใน C++ เมธอดเชิงนามธรรมเรียกว่า **Pure Virtual Function** ซึ่งประกาศโดยการกำหนด `= 0` ท้ายการประกาศเมธอด คลาสที่มี Pure Virtual Function อย่างน้อยหนึ่งเมธอดจะถือว่าเป็น Abstract Class โดยอัตโนมัติ

ตัวอย่าง C++:

```cpp
#include <string>
#include <iostream>

// Abstract Class
class Shape {
protected:
    std::string color;

public:
    Shape(std::string color) : color(color) {}

    // Pure Virtual Function (Abstract method)
    virtual double getArea() = 0;

    // Concrete method
    std::string getColor() {
        return color;
    }

    void displayColor() {
        std::cout << "Color: " << color << std::endl;
    }

    // Virtual destructor is important for proper memory deallocation
    virtual ~Shape() = default;
};

// Concrete Class สืบทอดจาก Shape
class Circle : public Shape {
private:
    double radius;

public:
    Circle(std::string color, double radius) : Shape(color), radius(radius)
{}

    double getArea() override { // ต้อง implement pure virtual function
getArea()
        return 3.14159 * radius * radius;
    }
};

// Concrete Class สืบทอดจาก Shape
class Rectangle : public Shape {
private:
```

```
    double width;
    double height;

public:
    Rectangle(std::string color, double width, double height) :
Shape(color), width(width), height(height) {}

    double getArea() override { // ต้อง implement pure virtual function
getArea()
        return width * height;
    }
};

int main() {
    // Shape s = new Shape("Red"); // Error: Cannot instantiate abstract
class

    Circle circle("Red", 5.0);
    std::cout << "Circle Area: " << circle.getArea() << std::endl;
    circle.displayColor();

    Rectangle rectangle("Blue", 4.0, 6.0);
    std::cout << "Rectangle Area: " << rectangle.getArea() << std::endl;
    rectangle.displayColor();

    // Polymorphism with Abstract Class
    Shape* s1 = new Circle("Green", 7.0);
    Shape* s2 = new Rectangle("Yellow", 3.0, 5.0);

    std::cout << "Area of s1: " << s1->getArea() << std::endl;
    std::cout << "Area of s2: " << s2->getArea() << std::endl;

    delete s1;
    delete s2;

    return 0;
}
```

7.3 อินเทอร์เฟซ (Interfaces)

อินเทอร์เฟซ (Interface) คือพิมพ์เขียวของคลาสที่ประกอบด้วยเมธอดเชิงนามธรรมทั้งหมด (ใน Java ก่อน Java 8) หรือเมธอดที่เป็น `public abstract` โดยปริยาย อินเทอร์เฟซไม่สามารถมีตัวแปรอินสแตนซ์ (instance variables) ได้ (ยกเว้น `public static final` constants) และ

ไม่สามารถมี constructor ได้ อินเทอร์เฟซใช้เพื่อกำหนดสัญญา (contract) ว่าคลาสใดๆ ที่ implement อินเทอร์เฟซนี้จะต้องมีเมธอดเหล่านั้น

7.3.1 การประกาศ Interface ใน Java

ใน Java ใช้คีย์เวิร์ด `interface` ในการประกาศ อินเทอร์เฟซสามารถมีเมธอด `default` และ `static` ได้ตั้งแต่ Java 8 เป็นต้นไป

ตัวอย่าง Java:

```java
// Interface
interface Flyable {
    void fly(); // โดยปริยายเป็น public abstract

    // Default method (ตั้งแต่ Java 8)
    default void land() {
        System.out.println("Landing...");
    }

    // Static method (ตั้งแต่ Java 8)
    static void describe() {
        System.out.println("This interface defines flying behavior.");
    }
}

// Class implement Interface
class Bird implements Flyable {
    String name;

    public Bird(String name) {
        this.name = name;
    }

    @Override
    public void fly() {
        System.out.println(name + " is flying with wings.");
    }
}

class Airplane implements Flyable {
    String model;

    public Airplane(String model) {
        this.model = model;
    }

    @Override
    public void fly() {
        System.out.println(model + " is flying with engines.");
    }
}

public class Main {
    public static void main(String[] args) {
        Bird eagle = new Bird("Eagle");
```

```
        eagle.fly();
        eagle.land(); // เรียกใช้ default method

        Airplane boeing = new Airplane("Boeing 747");
        boeing.fly();
        boeing.land();

        Flyable.describe(); // เรียกใช้ static method

        // Polymorphism with Interface
        Flyable f1 = new Bird("Sparrow");
        Flyable f2 = new Airplane("Airbus A380");

        f1.fly();
        f2.fly();
    }
}
```

7.3.2 การประกาศ Interface ใน C++ (Abstract Class with Pure Virtual Functions)

C++ ไม่มีแนวคิด `interface` โดยตรงเหมือน Java แต่สามารถจำลองพฤติกรรมของอินเทอร์เฟซได้โดยใช้ Abstract Class ที่มี Pure Virtual Function ทั้งหมด และไม่มีข้อมูลสมาชิก (data members) หรือมีแต่ `static const` เท่านั้น

ตัวอย่าง C++ (จำลอง Interface):

```cpp
#include <string>
#include <iostream>

// จำลอง Interface ด้วย Abstract Class ที่มีแต่ Pure Virtual Functions
class IFlyable {
public:
    virtual void fly() = 0;
    virtual void land() = 0;
    virtual ~IFlyable() = default; // Virtual destructor is important
};

// Class implement Interface
class Bird : public IFlyable {
private:
    std::string name;

public:
    Bird(std::string name) : name(name) {}

    void fly() override {
        std::cout << name << " is flying with wings." << std::endl;
    }

    void land() override {
        std::cout << name << " is landing on a branch." << std::endl;
    }
};

class Airplane : public IFlyable {
private:
    std::string model;

public:
    Airplane(std::string model) : model(model) {}

    void fly() override {
        std::cout << model << " is flying with engines." << std::endl;
    }

    void land() override {
        std::cout << model << " is landing on a runway." << std::endl;
    }
};

int main() {
```

```
    Bird eagle("Eagle");
    eagle.fly();
    eagle.land();

    Airplane boeing("Boeing 747");
    boeing.fly();
    boeing.land();

    // Polymorphism with simulated Interface
    IFlyable* f1 = new Bird("Sparrow");
    IFlyable* f2 = new Airplane("Airbus A380");

    f1->fly();
    f2->fly();

    delete f1;
    delete f2;

    return 0;
}
```

7.4 ความแตกต่างระหว่าง Abstract Class และ Interface

คุณสมบัติ	Abstract Class	Interface (Java)
การสร้างอ็อบเจกต์	ไม่สามารถสร้างอ็อบเจกต์ได้โดยตรง	ไม่สามารถสร้างอ็อบเจกต์ได้โดยตรง
เมธอด	มีทั้ง abstract และ concrete methods	มีแต่ abstract methods (ก่อน Java 8), มี default และ static methods ได้ (ตั้งแต่ Java 8)
ตัวแปร	มีตัวแปรอินสแตนซ์ได้	มีแต่ `public static final` constants
Constructor	มี constructor ได้	ไม่มี constructor
การสืบทอด/Implement	คลาสลูก `extends` คลาสแม่ได้เพียงคลาสเดียว	คลาส `implements` ได้หลาย interface
วัตถุประสงค์	ใช้สำหรับสร้างโครงสร้างพื้นฐานของคลาสที่มีความสัมพันธ์แบบ “เป็น” (is-a relationship) และต้องการให้คลาสลูกมีพฤติกรรมร่วมกันบางส่วน	ใช้สำหรับกำหนดสัญญา (contract) ของพฤติกรรมที่คลาสต่างๆ สามารถมีได้ โดยไม่สนใจว่าคลาสเหล่านั้นมีความสัมพันธ์กันอย่างไร

บทที่ 8: การจัดการข้อผิดพลาด (Exception Handling)

8.1 แนวคิดของการจัดการข้อผิดพลาด

การจัดการข้อผิดพลาด (Exception Handling) เป็นกลไกในภาษาโปรแกรมที่ช่วยให้โปรแกรมสามารถจัดการกับสถานการณ์ที่ไม่คาดคิดหรือข้อผิดพลาดที่เกิดขึ้นระหว่างการรันโปรแกรม (runtime errors) ได้อย่างสง่างาม แทนที่จะหยุดทำงานลงทันที (crash) การจัดการข้อผิดพลาดช่วยให้โปรแกรมมีความทนทาน (robust) และน่าเชื่อถือมากขึ้น

ข้อผิดพลาด (Error) คือปัญหาที่ร้ายแรงและมักจะแก้ไขไม่ได้ในระหว่างการรันโปรแกรม เช่น หน่วยความจำไม่พอ (OutOfMemoryError)

ข้อยกเว้น (Exception) คือเหตุการณ์ที่ไม่ปกติที่เกิดขึ้นระหว่างการรันโปรแกรม ซึ่งสามารถจัดการและกู้คืนได้ เช่น การหารด้วยศูนย์ (ArithmeticException), การเข้าถึงอาร์เรย์นอกขอบเขต (ArrayIndexOutOfBoundsException), การเปิดไฟล์ที่ไม่พบ (FileNotFoundException)

8.2 ประเภทของข้อยกเว้น (Types of Exceptions)

ใน Java และ C++ มีการแบ่งประเภทของข้อยกเว้นที่แตกต่างกันเล็กน้อย

8.2.1 ประเภทของข้อยกเว้นใน Java

Java แบ่งข้อยกเว้นออกเป็น 3 ประเภทหลักภายใต้คลาส `Throwable`:

1. **Checked Exceptions:** ข้อยกเว้นที่คอมไพเลอร์บังคับให้ต้องจัดการ (checked at compile-time) หากไม่จัดการจะเกิดข้อผิดพลาดในการคอมไพล์ ตัวอย่างเช่น `IOException`, `SQLException` มักเกิดจากปัจจัยภายนอกที่โปรแกรมควบคุมไม่ได้
2. **Unchecked Exceptions (Runtime Exceptions):** ข้อยกเว้นที่ไม่บังคับให้ต้องจัดการ (not checked at compile-time) แต่ควรหลีกเลี่ยงหรือแก้ไขที่ต้นเหตุ ตัวอย่างเช่น `ArithmeticException`, `NullPointerException`, `ArrayIndexOutOfBoundsException` มักเกิดจากข้อผิดพลาดในการเขียนโปรแกรม
3. **Errors:** ปัญหาที่ร้ายแรงและมักจะแก้ไขไม่ได้ในระหว่างการรันโปรแกรม เช่น `OutOfMemoryError`, `StackOverflowError`

8.2.2 ประเภทของข้อยกเว้นใน C++

C++ ไม่ได้แบ่งประเภทข้อยกเว้นอย่างเป็นทางการเหมือน Java แต่โดยทั่วไปจะใช้คลาส `std::exception` เป็นคลาสแม่ของข้อยกเว้นมาตรฐานต่างๆ และสามารถสร้างข้อยกเว้นของตัวเองได้

8.3 การจัดการข้อยกเว้นด้วย `try-catch-finally`

กลไกหลักในการจัดการข้อยกเว้นคือบล็อก `try-catch-finally`

- **`try` block:** เป็นส่วนของโค้ดที่อาจก่อให้เกิดข้อยกเว้น
- **`catch` block:** เป็นส่วนของโค้ดที่จะทำงานเมื่อเกิดข้อยกเว้นที่ระบุใน `try` block

- `finally` **block:** เป็นส่วนของโค้ดที่จะทำงานเสมอ ไม่ว่าจะเกิดข้อยกเว้นหรือไม่ก็ตาม มักใช้สำหรับคืนทรัพยากร (เช่น ปิดไฟล์, ปิดการเชื่อมต่อฐานข้อมูล)

8.3.1 การจัดการข้อยกเว้นใน Java

ตัวอย่าง Java:

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class ExceptionDemo {
    public static void main(String[] args) {
        // ตัวอย่าง 1: ArithmeticException (Unchecked Exception)
        try {
            int result = 10 / 0; // จะเกิด ArithmeticException
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Error: Division by zero is not allowed.");
            System.out.println("Exception message: " + e.getMessage());
        } finally {
            System.out.println("Finally block for division example.");
        }

        System.out.println("-----------------------------------");

        // ตัวอย่าง 2: FileNotFoundException (Checked Exception)
        FileReader fr = null;
        try {
            File file = new File("nonexistent.txt");
            fr = new FileReader(file);
            // อ่านไฟล์...
            System.out.println("File read successfully.");
        } catch (FileNotFoundException e) {
            System.out.println("Error: File not found.");
            System.out.println("Exception message: " + e.getMessage());
        } catch (IOException e) { // สามารถมีหลาย catch block ได้
            System.out.println("Error: An I/O error occurred.");
            System.out.println("Exception message: " + e.getMessage());
        } finally {
            System.out.println("Finally block for file example.");
            if (fr != null) {
                try {
                    fr.close();
                    System.out.println("File reader closed.");
                } catch (IOException e) {
                    System.out.println("Error closing file reader: " +
e.getMessage());
                }
            }
        }
```

```
        System.out.println("Program continues after exception handling.");
    }
}
```

8.3.2 การจัดการข้อยกเว้นใน C++

ตัวอย่าง C++:

```cpp
#include <iostream>
#include <string>
#include <stdexcept> // สำหรับ std::exception และคลาสลูก
#include <fstream>   // สำหรับ ifstream

void divide(int a, int b) {
    if (b == 0) {
        throw std::runtime_error("Division by zero is not allowed."); // โยนข้อยกเว้น
    }
    std::cout << "Result of division: " << a / b << std::endl;
}

void readFile(const std::string& filename) {
    std::ifstream file(filename);
    if (!file.is_open()) {
        throw std::runtime_error("Could not open file: " + filename);
    }
    std::string line;
    std::getline(file, line);
    std::cout << "First line of file: " << line << std::endl;
    file.close();
}

int main() {
    // ตัวอย่าง 1: การหารด้วยศูนย์
    try {
        divide(10, 0);
    } catch (const std::runtime_error& e) { // รับข้อยกเว้นประเภท runtime_error
        std::cerr << "Caught exception: " << e.what() << std::endl;
    } catch (const std::exception& e) { // สามารถรับข้อยกเว้นประเภทอื่นๆ ได้
        std::cerr << "Caught generic exception: " << e.what() << std::endl;
    } finally { // C++ ไม่มี finally block โดยตรง แต่สามารถจำลองได้ด้วย RAII หรือการจัดการด้วยมือ
        std::cout << "Simulated finally block for division example." << std::endl;
    }

    std::cout << "-----------------------------------" << std::endl;

    // ตัวอย่าง 2: การเปิดไฟล์ที่ไม่พบ
    try {
        readFile("nonexistent.txt");
    } catch (const std::runtime_error& e) {
        std::cerr << "Caught exception: " << e.what() << std::endl;
```

```
    } catch (const std::exception& e) {
        std::cerr << "Caught generic exception: " << e.what() << std::endl;
    }

    std::cout << "Program continues after exception handling." << std::endl;

    return 0;
}
```

- ใน C++ ไม่มี `finally` block โดยตรงเหมือน Java แต่สามารถจำลองพฤติกรรมที่คล้ายกันได้โดยใช้เทคนิค RAII (Resource Acquisition Is Initialization) หรือการจัดการทรัพยากรด้วยมือในทุกเส้นทางของโค้ด
- `e.what()` ใช้สำหรับดึงข้อความอธิบายข้อยกเว้น

8.4 การโยนข้อยกเว้น (Throwing Exceptions)

บางครั้งโปรแกรมจำเป็นต้องสร้างและโยนข้อยกเว้นเองเมื่อตรวจพบสถานการณ์ที่ไม่ถูกต้อง

8.4.1 การโยนข้อยกเว้นใน Java

ใช้คีย์เวิร์ด `throw` เพื่อโยนอ็อบเจกต์ข้อยกเว้น และ `throws` ในการประกาศเมธอดว่าอาจจะโยนข้อยกเว้นประเภทใดบ้าง (สำหรับ Checked Exceptions)

ตัวอย่าง Java:

```java
public class CustomException extends Exception {
    public CustomException(String message) {
        super(message);
    }
}

public class AgeValidator {
    public void validateAge(int age) throws CustomException {
        if (age < 0 || age > 120) {
            throw new CustomException("Invalid age: " + age + ". Age must be
between 0 and 120.");
        }
        System.out.println("Age " + age + " is valid.");
    }

    public static void main(String[] args) {
        AgeValidator validator = new AgeValidator();
        try {
            validator.validateAge(25);
            validator.validateAge(-10);
        } catch (CustomException e) {
            System.out.println("Caught custom exception: " +
e.getMessage());
        }
    }
}
```

8.4.2 การโยนข้อยกเว้นใน C++

ใช้คีย์เวิร์ด `throw` เพื่อโยนอ็อบเจกต์ข้อยกเว้น

ตัวอย่าง C++:

```cpp
#include <iostream>
#include <string>
#include <stdexcept>

// สร้าง Custom Exception โดยสืบทอดจาก std::runtime_error
class InvalidAgeException : public std::runtime_error {
public:
    InvalidAgeException(const std::string& message) :
std::runtime_error(message) {}
};

class AgeValidator {
public:
    void validateAge(int age) {
        if (age < 0 || age > 120) {
            throw InvalidAgeException("Invalid age: " + std::to_string(age)
+ ". Age must be between 0 and 120.");
        }
        std::cout << "Age " << age << " is valid." << std::endl;
    }
};

int main() {
    AgeValidator validator;
    try {
        validator.validateAge(25);
        validator.validateAge(-10);
    } catch (const InvalidAgeException& e) {
        std::cerr << "Caught custom exception: " << e.what() << std::endl;
    } catch (const std::exception& e) {
        std::cerr << "Caught generic exception: " << e.what() << std::endl;
    }

    return 0;
}
```

8.5 Custom Exceptions

การสร้าง Custom Exceptions ช่วยให้โปรแกรมสามารถระบุประเภทของข้อผิดพลาดที่เฉพาะเจาะจงกับโดเมนของปัญหาได้ ทำให้การจัดการข้อผิดพลาดมีความชัดเจนและมีประสิทธิภาพมากขึ้น

- **Java:** สร้างคลาสใหม่ที่สืบทอดจาก `Exception` (สำหรับ Checked Exception) หรือ `RuntimeException` (สำหรับ Unchecked Exception)
- **C++:** สร้างคลาสใหม่ที่สืบทอดจาก `std::exception` หรือคลาสลูกของ `std::exception` เช่น `std::runtime_error` หรือ `std::logic_error`

บทที่ 9: คอลเลกชันและโครงสร้างข้อมูลพื้นฐาน

9.1 แนวคิดของคอลเลกชัน (Collections)

คอลเลกชัน (Collections) คือเฟรมเวิร์กที่ช่วยให้เราสามารถจัดเก็บ จัดการ และเข้าถึงกลุ่มของอ็อบเจกต์ได้อย่างมีประสิทธิภาพ โดยมีโครงสร้างข้อมูล (Data Structures) ที่หลากหลายให้เลือกใช้ตามความเหมาะสมของงาน คอลเลกชันช่วยให้การทำงานกับข้อมูลจำนวนมากเป็นไปได้ง่ายขึ้นและลดความซับซ้อนในการเขียนโค้ด

ใน Java มี **Java Collections Framework** ซึ่งเป็นชุดของอินเทอร์เฟซและคลาสที่ใช้จัดการกลุ่มของอ็อบเจกต์ ส่วนใน C++ จะมี **Standard Template Library (STL)** ที่มีคอนเทนเนอร์ (Containers) และอัลกอริทึม (Algorithms) ที่คล้ายคลึงกัน

9.2 อินเทอร์เฟซหลักใน Java Collections Framework

Java Collections Framework มีอินเทอร์เฟซหลักๆ ดังนี้:

1. **`List`:** เป็นคอลเลกชันที่เก็บข้อมูลแบบมีลำดับ (ordered collection) และอนุญาตให้มีข้อมูลซ้ำกันได้ (duplicates) การเข้าถึงข้อมูลทำได้โดยใช้ index
    - **Implementations:** `ArrayList`, `LinkedList`, `Vector`
2. **`Set`:** เป็นคอลเลกชันที่เก็บข้อมูลแบบไม่ซ้ำกัน (unique elements) และไม่มีลำดับที่แน่นอน
    - **Implementations:** `HashSet`, `LinkedHashSet`, `TreeSet`
3. **`Map`:** เป็นคอลเลกชันที่เก็บข้อมูลในรูปแบบคู่ของคีย์-ค่า (key-value pairs) โดยที่คีย์จะต้องไม่ซ้ำกัน
    - **Implementations:** `HashMap`, `LinkedHashMap`, `TreeMap`, `Hashtable`

4. **`Queue` :** เป็นคอลเลกชันที่เก็บข้อมูลแบบเข้าก่อนออกก่อน (First-In, First-Out - FIFO)
    - **Implementations:** `LinkedList` , `PriorityQueue`

9.3 โครงสร้างข้อมูลพื้นฐานใน Java

9.3.1 `List` (ArrayList และ LinkedList)

`List` เป็นอินเทอร์เฟซที่ใช้เก็บข้อมูลแบบมีลำดับ

- **`ArrayList` :** ใช้โครงสร้างแบบ Dynamic Array เหมาะสำหรับการเข้าถึงข้อมูลแบบสุ่ม (random access) ด้วย index แต่การเพิ่ม/ลบข้อมูลตรงกลางอาจช้า
- **`LinkedList` :** ใช้โครงสร้างแบบ Doubly Linked List เหมาะสำหรับการเพิ่ม/ลบข้อมูลตรงกลาง แต่การเข้าถึงข้อมูลแบบสุ่มจะช้ากว่า `ArrayList`

ตัวอย่าง Java `ArrayList` :

```java
import java.util.ArrayList;
import java.util.List;

public class ListDemo {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");
        names.add("Bob"); // อนุญาตให้มีข้อมูลซ้ำ

        System.out.println("Names: " + names); // Output: Names: [Alice,
Bob, Charlie, Bob]
        System.out.println("Size: " + names.size()); // Output: Size: 4
        System.out.println("Element at index 1: " + names.get(1)); //
Output: Element at index 1: Bob

        names.remove("Bob"); // ลบ Bob ตัวแรกที่เจอ
        System.out.println("Names after removing Bob: " + names); // Output:
Names after removing Bob: [Alice, Charlie, Bob]

        names.remove(0); // ลบ element ที่ index 0
        System.out.println("Names after removing index 0: " + names); //
Output: Names after removing index 0: [Charlie, Bob]

        for (String name : names) {
            System.out.println("Name: " + name);
        }
    }
}
```

9.3.2 `Set` (HashSet และ TreeSet)

`Set` เป็นอินเทอร์เฟซที่ใช้เก็บข้อมูลแบบไม่ซ้ำกัน

- **`HashSet`** : เก็บข้อมูลแบบไม่เรียงลำดับ (unordered) และไม่รับประกันลำดับการเก็บข้อมูล เหมาะสำหรับกรณีที่ต้องการความเร็วในการเพิ่ม/ลบ/ค้นหาข้อมูล
- **`TreeSet`** : เก็บข้อมูลแบบเรียงลำดับ (sorted) ตามธรรมชาติของข้อมูล หรือตาม Comparator ที่กำหนด เหมาะสำหรับกรณีที่ต้องการข้อมูลที่เรียงลำดับ

ตัวอย่าง Java `HashSet` :

```java
import java.util.HashSet;
import java.util.Set;

public class SetDemo {
    public static void main(String[] args) {
        Set<String> uniqueNames = new HashSet<>();
        uniqueNames.add("Alice");
        uniqueNames.add("Bob");
        uniqueNames.add("Charlie");
        uniqueNames.add("Bob"); // จะไม่ถูกเพิ่ม เพราะซ้ำ

        System.out.println("Unique Names: " + uniqueNames); // Output:
Unique Names: [Alice, Bob, Charlie] (ลำดับอาจแตกต่างกัน)
        System.out.println("Size: " + uniqueNames.size()); // Output: Size:
3
        System.out.println("Contains Bob? " + uniqueNames.contains("Bob"));
// Output: Contains Bob? true

        uniqueNames.remove("Alice");
        System.out.println("Unique Names after removing Alice: " +
uniqueNames);
    }
}
```

9.3.3 `Map` (HashMap และ TreeMap)

`Map` เป็นอินเทอร์เฟซที่ใช้เก็บข้อมูลแบบคีย์-ค่า

- **`HashMap`:** เก็บข้อมูลแบบไม่เรียงลำดับ (unordered) และไม่รับประกันลำดับการเก็บข้อมูล เหมาะสำหรับกรณีที่ต้องการความเร็วในการเพิ่ม/ลบ/ค้นหาข้อมูลด้วยคีย์
- **`TreeMap`:** เก็บข้อมูลแบบเรียงลำดับคีย์ (sorted by key) ตามธรรมชาติของคีย์ หรือตาม Comparator ที่กำหนด เหมาะสำหรับกรณีที่ต้องการข้อมูลที่เรียงลำดับตามคีย์

ตัวอย่าง Java `HashMap`:

```java
import java.util.HashMap;
import java.util.Map;

public class MapDemo {
    public static void main(String[] args) {
        Map<String, Integer> ages = new HashMap<>();
        ages.put("Alice", 30);
        ages.put("Bob", 25);
        ages.put("Charlie", 35);
        ages.put("Bob", 26); // ค่าของ Bob จะถูกอัปเดตเป็น 26

        System.out.println("Ages: " + ages); // Output: Ages: {Alice=30,
Bob=26, Charlie=35} (ลำดับอาจแตกต่างกัน)
        System.out.println("Alice's age: " + ages.get("Alice")); // Output:
Alice's age: 30
        System.out.println("Contains key Charlie? " +
ages.containsKey("Charlie")); // Output: Contains key Charlie? true

        ages.remove("Bob");
        System.out.println("Ages after removing Bob: " + ages);

        for (Map.Entry<String, Integer> entry : ages.entrySet()) {
            System.out.println("Name: " + entry.getKey() + ", Age: " +
entry.getValue());
        }
    }
}
```

9.4 โครงสร้างข้อมูลพื้นฐานใน C++ Standard Template Library (STL)

C++ STL มีคอนเทนเนอร์ที่เทียบเท่ากับ Java Collections Framework:

1. **`std::vector`**: เทียบเท่า `ArrayList` เป็น Dynamic Array ที่สามารถปรับขนาดได้
2. **`std::list`**: เทียบเท่า `LinkedList` เป็น Doubly Linked List
3. **`std::set`**: เทียบเท่า `HashSet` (แต่ `std::set` จะเก็บข้อมูลแบบเรียงลำดับเสมอ)
4. **`std::unordered_set`**: เทียบเท่า `HashSet` (เก็บข้อมูลแบบไม่เรียงลำดับ)
5. **`std::map`**: เทียบเท่า `TreeMap` (เก็บข้อมูลแบบคีย์-ค่า และเรียงลำดับตามคีย์)

6. **`std::unordered_map`** : เทียบเท่า `HashMap` (เก็บข้อมูลแบบคีย์-ค่า และไม่เรียงลำดับ)
7. **`std::queue`** : เทียบเท่า `Queue` (FIFO)
8. **`std::stack`** : โครงสร้างข้อมูลแบบเข้าหลังออกก่อน (Last-In, First-Out - LIFO)

9.4.1 `std::vector`

ตัวอย่าง C++ `std::vector` :

```cpp
#include <iostream>
#include <vector>
#include <string>

int main() {
    std::vector<std::string> names;
    names.push_back("Alice");
    names.push_back("Bob");
    names.push_back("Charlie");
    names.push_back("Bob"); // อนุญาตให้มีข้อมูลซ้ำ

    std::cout << "Names: ";
    for (const std::string& name : names) {
        std::cout << name << " ";
    }
    std::cout << std::endl; // Output: Names: Alice Bob Charlie Bob

    std::cout << "Size: " << names.size() << std::endl; // Output: Size: 4
    std::cout << "Element at index 1: " << names[1] << std::endl; // Output:
Element at index 1: Bob

    names.erase(names.begin() + 1); // ลบ element ที่ index 1 (Bob)
    std::cout << "Names after removing index 1: ";
    for (const std::string& name : names) {
        std::cout << name << " ";
    }
    std::cout << std::endl; // Output: Names after removing index 1: Alice
Charlie Bob

    return 0;
}
```

9.4.2 `std::map`

ตัวอย่าง C++ `std::map`:

```cpp
#include <iostream>
#include <map>
#include <string>

int main() {
    std::map<std::string, int> ages;
    ages["Alice"] = 30;
    ages["Bob"] = 25;
    ages["Charlie"] = 35;
    ages["Bob"] = 26; // ค่าของ Bob จะถูกอัปเดตเป็น 26

    std::cout << "Ages: ";
    for (const auto& pair : ages) {
        std::cout << pair.first << "=" << pair.second << " ";
    }
    std::cout << std::endl; // Output: Ages: Alice=30 Bob=26 Charlie=35

    std::cout << "Alice's age: " << ages["Alice"] << std::endl; // Output:
Alice's age: 30

    if (ages.count("Charlie")) {
        std::cout << "Contains key Charlie? Yes" << std::endl; // Output:
Contains key Charlie? Yes
    }

    ages.erase("Bob");
    std::cout << "Ages after removing Bob: ";
    for (const auto& pair : ages) {
        std::cout << pair.first << "=" << pair.second << " ";
    }
    std::cout << std::endl; // Output: Ages after removing Bob: Alice=30
Charlie=35

    return 0;
}
```

บทที่ 10: การจัดการไฟล์และอินพุต/เอาต์พุต (File I/O)

10.1 แนวคิดของการจัดการไฟล์

การจัดการไฟล์ (File I/O - Input/Output) คือกระบวนการที่โปรแกรมอ่านข้อมูลจากไฟล์ (Input) หรือเขียนข้อมูลลงในไฟล์ (Output) การจัดการไฟล์เป็นสิ่งสำคัญในการพัฒนาโปรแกรมที่ต้องการเก็บข้อมูลแบบถาวร (persistent storage) ซึ่งข้อมูลจะยังคงอยู่แม้โปรแกรมจะปิดไปแล้ว

ไฟล์สามารถเป็นได้หลายประเภท เช่น ไฟล์ข้อความ (text files), ไฟล์ไบนารี (binary files), ไฟล์รูปภาพ, ไฟล์เสียง เป็นต้น ในบทนี้จะเน้นการจัดการไฟล์ข้อความเป็นหลัก

10.2 การอ่านและเขียนไฟล์ข้อความใน Java

Java มีคลาสและอินเทอร์เฟซจำนวนมากในแพ็กเกจ `java.io` สำหรับการจัดการไฟล์ โดยคลาสที่นิยมใช้สำหรับการอ่านและเขียนไฟล์ข้อความคือ `FileReader`, `FileWriter`, `BufferedReader`, และ `BufferedWriter`

10.2.1 การเขียนไฟล์ข้อความ (Writing to a Text File)

ใช้ `FileWriter` สำหรับเขียนอักขระลงในไฟล์ และ `BufferedWriter` เพื่อเพิ่มประสิทธิภาพในการเขียน (buffering)

ตัวอย่าง Java:

```java
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class FileWriterDemo {
    public static void main(String[] args) {
        String filename = "output.txt";
        try (BufferedWriter writer = new BufferedWriter(new
FileWriter(filename))) {
            writer.write("Hello, this is the first line.\n");
            writer.write("This is the second line.\n");
            writer.newLine(); // ขึ้นบรรทัดใหม่
            writer.write("And this is the third line.");
            System.out.println("Successfully wrote to the file: " +
filename);
        } catch (IOException e) {
            System.err.println("Error writing to file: " + e.getMessage());
        }
    }
}
```

- `FileWriter(filename)` : สร้างอ็อบเจกต์ `FileWriter` เพื่อเขียนไฟล์ หากไฟล์ไม่มีอยู่จะสร้างขึ้นใหม่ หากมีอยู่จะเขียนทับ (overwrite) หากต้องการเขียนต่อท้าย (append) ให้ใช้ `new FileWriter(filename, true)`
- `BufferedWriter` : ช่วยให้การเขียนมีประสิทธิภาพมากขึ้นโดยการเก็บข้อมูลไว้ในบัฟเฟอร์ก่อนที่จะเขียนลงไฟล์จริง
- `try-with-resources` : เป็นคุณสมบัติของ Java ที่ช่วยให้มั่นใจว่าทรัพยากร (เช่น `writer`) จะถูกปิดโดยอัตโนมัติเมื่อออกจากบล็อก `try` ไม่ว่าจะเกิดข้อยกเว้นหรือไม่ก็ตาม

10.2.2 การอ่านไฟล์ข้อความ (Reading from a Text File)

ใช้ `FileReader` สำหรับอ่านอักขระจากไฟล์ และ `BufferedReader` เพื่อเพิ่มประสิทธิภาพในการอ่าน (buffering) และสามารถอ่านได้ทีละบรรทัด

ตัวอย่าง Java:

```java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class FileReaderDemo {
    public static void main(String[] args) {
        String filename = "output.txt"; // ไฟล์ที่สร้างจากตัวอย่างก่อนหน้า
        try (BufferedReader reader = new BufferedReader(new
FileReader(filename))) {
            String line;
            System.out.println("Reading from file: " + filename);
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
            System.out.println("Successfully read from the file.");
        } catch (IOException e) {
            System.err.println("Error reading from file: " +
e.getMessage());
        }
    }
}
```

- `FileReader(filename)` : สร้างอ็อบเจกต์ `FileReader` เพื่ออ่านไฟล์
- `BufferedReader` : ช่วยให้การอ่านมีประสิทธิภาพมากขึ้นและมีเมธอด `readLine()` สำหรับอ่านข้อมูลทีละบรรทัด

10.3 การอ่านและเขียนไฟล์ข้อความใน C++

ใน C++ ใช้คลาส `fstream` ซึ่งอยู่ในไลบรารี `<fstream>` สำหรับการจัดการไฟล์ โดยมี `ofstream` สำหรับเขียนไฟล์ และ `ifstream` สำหรับอ่านไฟล์

10.3.1 การเขียนไฟล์ข้อความ (Writing to a Text File)

ใช้ `ofstream` (output file stream) เพื่อเขียนข้อมูลลงในไฟล์

ตัวอย่าง C++:

```cpp
#include <fstream> // สำหรับ ofstream
#include <iostream> // สำหรับ cout, cerr
#include <string>   // สำหรับ string

int main() {
    std::string filename = "output_cpp.txt";
    std::ofstream outputFile(filename); // สร้างอ็อบเจกต์ ofstream และเปิดไฟล์

    if (outputFile.is_open()) { // ตรวจสอบว่าเปิดไฟล์สำเร็จหรือไม่
        outputFile << "Hello from C++! This is line 1.\n";
        outputFile << "This is line 2.\n";
        outputFile << "And line 3.\n";
        outputFile.close(); // ปิดไฟล์
        std::cout << "Successfully wrote to the file: " << filename <<
std::endl;
    } else {
        std::cerr << "Error: Could not open file for writing: " << filename
<< std::endl;
    }

    return 0;
}
```

- `std::ofstream outputFile(filename);` : สร้างอ็อบเจกต์ `ofstream` และพยายามเปิดไฟล์ `filename` หากไฟล์ไม่มีอยู่จะสร้างขึ้นใหม่ หากมีอยู่จะเขียนทับ (overwrite) หากต้องการเขียนต่อท้าย (append) ให้ใช้ `std::ofstream outputFile(filename, std::ios::app);`
- `outputFile << ...` : ใช้ operator `<<` ในการเขียนข้อมูลลงไฟล์ คล้ายกับการใช้ `std::cout`
- `outputFile.close();` : ปิดไฟล์ เป็นสิ่งสำคัญเพื่อบันทึกข้อมูลและคืนทรัพยากร

10.3.2 การอ่านไฟล์ข้อความ (Reading from a Text File)

ใช้ `ifstream` (input file stream) เพื่ออ่านข้อมูลจากไฟล์

ตัวอย่าง C++:

```cpp
#include <fstream> // สำหรับ ifstream
#include <iostream> // สำหรับ cout, cerr
#include <string>   // สำหรับ string, getline

int main() {
    std::string filename = "output_cpp.txt"; // ไฟล์ที่สร้างจากตัวอย่างก่อนหน้า
    std::ifstream inputFile(filename); // สร้างอ็อบเจกต์ ifstream และเปิดไฟล์

    if (inputFile.is_open()) { // ตรวจสอบว่าเปิดไฟล์สำเร็จหรือไม่
        std::string line;
        std::cout << "Reading from file: " << filename << std::endl;
        while (std::getline(inputFile, line)) { // อ่านทีละบรรทัดจนกว่าจะหมดไฟล์
            std::cout << line << std::endl;
        }
        inputFile.close(); // ปิดไฟล์
        std::cout << "Successfully read from the file." << std::endl;
    } else {
        std::cerr << "Error: Could not open file for reading: " << filename
<< std::endl;
    }

    return 0;
}
```

- `std::ifstream inputFile(filename);`: สร้างอ็อบเจกต์ `ifstream` และพยายามเปิดไฟล์ `filename`
- `std::getline(inputFile, line)`: อ่านข้อมูลจาก `inputFile` ทีละบรรทัดและเก็บไว้ในตัวแปร `line` จะคืนค่า `true` ตราบใดที่ยังอ่านได้ และ `false` เมื่อถึงจุดสิ้นสุดไฟล์หรือเกิดข้อผิดพลาด

10.4 การจัดการไฟล์ไบนารี (Binary File I/O)

นอกจากการอ่านและเขียนไฟล์ข้อความแล้ว บางครั้งเราอาจต้องจัดการกับไฟล์ไบนารี ซึ่งเป็นการอ่านและเขียนข้อมูลในรูปแบบไบต์โดยตรง โดยไม่ผ่านการแปลงเป็นอักขระ

10.4.1 การจัดการไฟล์ไบนารีใน Java

ใช้ `FileInputStream` และ `FileOutputStream` สำหรับการอ่านและเขียนไบต์ และ `DataInputStream`, `DataOutputStream` สำหรับอ่าน/เขียนข้อมูลชนิดพื้นฐาน (primitive

types) ในรูปแบบไบนารี

ตัวอย่าง Java (เขียน/อ่านไฟล์ไบนารี):

```java
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class BinaryFileDemo {
    public static void main(String[] args) {
        String filename = "data.bin";

        // เขียนข้อมูลลงไฟล์ไบนารี
        try (DataOutputStream dos = new DataOutputStream(new
FileOutputStream(filename))) {
            dos.writeInt(123);
            dos.writeDouble(3.14);
            dos.writeBoolean(true);
            dos.writeUTF("Hello Binary!");
            System.out.println("Successfully wrote binary data to " +
filename);
        } catch (IOException e) {
            System.err.println("Error writing binary file: " +
e.getMessage());
        }

        // อ่านข้อมูลจากไฟล์ไบนารี
        try (DataInputStream dis = new DataInputStream(new
FileInputStream(filename))) {
            int intValue = dis.readInt();
            double doubleValue = dis.readDouble();
            boolean booleanValue = dis.readBoolean();
            String stringValue = dis.readUTF();

            System.out.println("Successfully read binary data from " +
filename);
            System.out.println("Int: " + intValue);
            System.out.println("Double: " + doubleValue);
            System.out.println("Boolean: " + booleanValue);
            System.out.println("String: " + stringValue);
        } catch (IOException e) {
            System.err.println("Error reading binary file: " +
e.getMessage());
        }
    }
}
```

10.4.2 การจัดการไฟล์ไบนารีใน C++

ใช้ `fstream` โดยเปิดไฟล์ในโหมดไบนารี (`std::ios::binary`) และใช้เมธอด `read()` และ `write()`

ตัวอย่าง C++ (เขียน/อ่านไฟล์ไบนารี):

```cpp
#include <fstream>
#include <iostream>
#include <string>

struct MyData {
    int id;
    double value;
    char name[20];
};

int main() {
    std::string filename = "data_cpp.bin";

    // เขียนข้อมูลลงไฟล์ไบนารี
    std::ofstream outFile(filename, std::ios::binary); // เปิดไฟล์ในโหมดไบนารี
    if (outFile.is_open()) {
        MyData data1 = {1, 10.5, "First Data"};
        MyData data2 = {2, 20.7, "Second Data"};

        outFile.write(reinterpret_cast<char*>(&data1), sizeof(MyData));
        outFile.write(reinterpret_cast<char*>(&data2), sizeof(MyData));
        outFile.close();
        std::cout << "Successfully wrote binary data to " << filename <<
std::endl;
    } else {
        std::cerr << "Error writing binary file: " << filename << std::endl;
    }

    // อ่านข้อมูลจากไฟล์ไบนารี
    std::ifstream inFile(filename, std::ios::binary); // เปิดไฟล์ในโหมดไบนารี
    if (inFile.is_open()) {
        MyData readData;
        std::cout << "Successfully read binary data from " << filename <<
std::endl;
        while (inFile.read(reinterpret_cast<char*>(&readData),
sizeof(MyData))) {
            std::cout << "ID: " << readData.id
                      << ", Value: " << readData.value
                      << ", Name: " << readData.name << std::endl;
        }
        inFile.close();
    } else {
        std::cerr << "Error reading binary file: " << filename << std::endl;
    }
```

```
    return 0;
}
```

- `std::ios::binary`: แฟล็กที่ระบุว่าไฟล์เป็นไฟล์ไบนารี
- `outFile.write(reinterpret_cast<char*>(&data1), sizeof(MyData));`: เขียนข้อมูลขนาด `sizeof(MyData)` ไบต์จากที่อยู่ของ `data1` ลงในไฟล์
- `inFile.read(reinterpret_cast<char*>(&readData), sizeof(MyData));`: อ่านข้อมูลขนาด `sizeof(MyData)` ไบต์จากไฟล์และเก็บไว้ใน `readData`

บทที่ 11: การเขียนโปรแกรมติดต่อผู้ใช้แบบกราฟิก (GUI Programming)

11.1 บทนำสู่ GUI Programming

การเขียนโปรแกรมติดต่อผู้ใช้แบบกราฟิก (Graphical User Interface - GUI Programming) คือการสร้างส่วนติดต่อผู้ใช้ที่ประกอบด้วยองค์ประกอบกราฟิก เช่น ปุ่ม (buttons), ช่องข้อความ (text fields), เมนู (menus), และหน้าต่าง (windows) แทนที่จะเป็นส่วนติดต่อแบบข้อความ (Command Line Interface - CLI) GUI ช่วยให้ผู้ใช้สามารถโต้ตอบกับโปรแกรมได้ง่ายขึ้นและเป็นธรรมชาติมากขึ้น

ในภาษา Java มีเฟรมเวิร์กหลักๆ สำหรับการพัฒนา GUI ได้แก่ AWT (Abstract Window Toolkit), Swing และ JavaFX ส่วนใน C++ มีไลบรารี GUI ยอดนิยมหลายตัว เช่น Qt, GTK+, MFC (สำหรับ Windows)

ในบทนี้จะเน้นที่พื้นฐานของ Java Swing และแนวคิดทั่วไปของ GUI Programming

11.2 พื้นฐาน Java Swing

Java Swing เป็นชุดเครื่องมือ GUI ที่สร้างขึ้นบน AWT โดยมีคุณสมบัติที่เรียกว่า “lightweight” ซึ่งหมายความว่าคอมโพเนนต์ของ Swing ถูกวาดโดย Java เอง ไม่ได้ขึ้นอยู่กับคอมโพเนนต์ของระบบปฏิบัติการโดยตรง ทำให้แอปพลิเคชัน Swing มีรูปลักษณ์ที่สอดคล้องกันในทุกแพลตฟอร์ม

11.2.1 คอมโพเนนต์พื้นฐานของ Swing

- **`JFrame`:** เป็นหน้าต่างหลักของแอปพลิเคชัน GUI
- **`JPanel`:** เป็นคอนเทนเนอร์สำหรับจัดกลุ่มคอมโพเนนต์อื่นๆ
- **`JButton`:** ปุ่มที่ผู้ใช้สามารถคลิกได้
- **`JLabel`:** แสดงข้อความหรือรูปภาพแบบอ่านอย่างเดียว
- **`JTextField`:** ช่องสำหรับรับข้อมูลข้อความจากผู้ใช้
- **`JTextArea`:** ช่องสำหรับรับข้อมูลข้อความหลายบรรทัด
- **`JCheckBox`:** ช่องทำเครื่องหมายสำหรับเลือกตัวเลือก
- **`JRadioButton`:** ปุ่มตัวเลือกที่เลือกได้เพียงตัวเดียวในกลุ่ม
- **`JComboBox`:** กล่องคอมโบสำหรับเลือกรายการจากรายการแบบดรอปดาวน์

11.2.2 โครงสร้างโปรแกรม Swing เบื้องต้น

ตัวอย่าง Java Swing:

```java
import javax.swing.*; // นำเข้าคลาส Swing ทั้งหมด
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class SimpleSwingApp extends JFrame implements ActionListener {

    private JLabel welcomeLabel;
    private JButton clickButton;
    private JTextField nameTextField;

    public SimpleSwingApp() {
        // กำหนดคุณสมบัติของ JFrame
        setTitle("My First Swing App"); // กำหนดชื่อหน้าต่าง
        setSize(400, 200); // กำหนดขนาดหน้าต่าง (กว้าง, สูง)
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // กำหนดให้โปรแกรมปิดเมื่อปิดหน้าต่าง
        setLocationRelativeTo(null); // จัดหน้าต่างให้อยู่กึ่งกลางจอ
        setLayout(null); // กำหนด Layout Manager เป็น null เพื่อจัดตำแหน่งเอง

        // สร้างคอมโพเนนต์
        welcomeLabel = new JLabel("Enter your name:");
        welcomeLabel.setBounds(50, 30, 150, 25); // (x, y, width, height)
        add(welcomeLabel);

        nameTextField = new JTextField();
        nameTextField.setBounds(180, 30, 150, 25);
        add(nameTextField);

        clickButton = new JButton("Click Me!");
        clickButton.setBounds(150, 80, 100, 30);
        clickButton.addActionListener(this); // เพิ่ม ActionListener ให้กับปุ่ม
        add(clickButton);

        // ทำให้หน้าต่างมองเห็นได้
        setVisible(true);
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == clickButton) {
            String name = nameTextField.getText();
            if (name.isEmpty()) {
                JOptionPane.showMessageDialog(this, "Please enter your name.", "Warning", JOptionPane.WARNING_MESSAGE);
            } else {
```

```
                JOptionPane.showMessageDialog(this, "Hello, " + name + "!",
"Greeting", JOptionPane.INFORMATION_MESSAGE);
            }
        }
    }

    public static void main(String[] args) {
        // รัน GUI บน Event Dispatch Thread (EDT) เพื่อความปลอดภัยของเธรด
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new SimpleSwingApp();
            }
        });
    }
}
```

- `JFrame` : เป็นคลาสพื้นฐานสำหรับหน้าต่างหลัก
- `setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)` : เมื่อปิดหน้าต่าง โปรแกรมจะสิ้นสุดการทำงาน
- `setLayout(null)` : การตั้งค่า `null` layout manager หมายถึงเราจะจัดตำแหน่งและขนาดของคอมโพเนนต์ด้วยตัวเองโดยใช้เมธอด `setBounds()`
- `addActionListener(this)` : การเพิ่ม `ActionListener` ให้กับปุ่ม ทำให้เมธอด `actionPerformed` ถูกเรียกเมื่อปุ่มถูกคลิก
- `SwingUtilities.invokeLater()` : เป็นวิธีที่แนะนำในการสร้างและอัปเดตคอมโพเนนต์ Swing เพื่อให้แน่ใจว่าโค้ด GUI ทำงานบน Event Dispatch Thread (EDT) ซึ่งเป็นเธรดเดียวที่รับผิดชอบการจัดการเหตุการณ์ GUI

11.3 การจัดการเหตุการณ์ (Event Handling)

GUI Programming อาศัยแนวคิดของการจัดการเหตุการณ์ (Event Handling) ซึ่งหมายถึงการตอบสนองต่อการกระทำของผู้ใช้ เช่น การคลิกปุ่ม การพิมพ์ข้อความ การเลื่อนเมาส์

11.3.1 แหล่งกำเนิดเหตุการณ์ (Event Source) และตัวรับเหตุการณ์ (Event Listener)

- **แหล่งกำเนิดเหตุการณ์ (Event Source):** คอมโพเนนต์ GUI ที่สร้างเหตุการณ์ขึ้นมา เช่น `JButton` เมื่อถูกคลิก

- **ตัวรับเหตุการณ์ (Event Listener):** อ็อบเจกต์ที่รอรับและประมวลผลเหตุการณ์ที่เกิดขึ้น ตัวรับเหตุการณ์จะต้อง implement อินเทอร์เฟซ Listener ที่เหมาะสม (เช่น `ActionListener` สำหรับ `ActionEvent`)

11.3.2 การลงทะเบียน Listener

แหล่งกำเนิดเหตุการณ์จะ “ลงทะเบียน” ตัวรับเหตุการณ์ โดยเรียกใช้เมธอด `addXxxListener()` (เช่น `addActionListener()`, `addMouseListener()`, `addKeyListener()`)

ตัวอย่างการจัดการเหตุการณ์ (จากตัวอย่างข้างต้น):

```
// ใน constructor ของ SimpleSwingApp
clickButton.addActionListener(this); // 'this' คืออ็อบเจกต์ SimpleSwingApp ซึ่ง
implement ActionListener

// เมธอดที่ถูกเรียกเมื่อเกิดเหตุการณ์
@Override
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == clickButton) {
        // โค้ดที่จะทำงานเมื่อปุ่ม clickButton ถูกคลิก
        String name = nameTextField.getText();
        if (name.isEmpty()) {
            JOptionPane.showMessageDialog(this, "Please enter your name.",
"Warning", JOptionPane.WARNING_MESSAGE);
        } else {
            JOptionPane.showMessageDialog(this, "Hello, " + name + "!",
"Greeting", JOptionPane.INFORMATION_MESSAGE);
        }
    }
}
```

11.4 Layout Managers

Layout Managers เป็นคลาสที่ช่วยในการจัดเรียงและจัดตำแหน่งคอมโพเนนต์ GUI ภายในคอนเทนเนอร์ เช่น `JPanel` หรือ `JFrame` การใช้ Layout Managers ช่วยให้ GUI สามารถปรับขนาดได้ดีเมื่อหน้าต่างถูกปรับขนาด หรือเมื่อรันบนหน้าจอที่มีความละเอียดต่างกัน

Layout Managers ที่นิยมใช้ใน Swing:

- **`BorderLayout`** : แบ่งพื้นที่ออกเป็น 5 ส่วน: North, South, East, West, Center
- **`FlowLayout`** : จัดเรียงคอมโพเนนต์จากซ้ายไปขวา และขึ้นบรรทัดใหม่เมื่อไม่มีพื้นที่พอ
- **`GridLayout`** : จัดเรียงคอมโพเนนต์ในรูปแบบตาราง (rows and columns)
- **`GridBagLayout`** : เป็น Layout Manager ที่ยืดหยุ่นและซับซ้อนที่สุด ช่วยให้สามารถควบคุมการจัดวางคอมโพเนนต์ได้อย่างละเอียด
- **`BoxLayout`** : จัดเรียงคอมโพเนนต์ในแนวตั้งหรือแนวนอน

ตัวอย่างการใช้ `BorderLayout` :

```
import javax.swing.*;
import java.awt.*; // สำหรับ BorderLayout

public class BorderLayoutDemo extends JFrame {
    public BorderLayoutDemo() {
        setTitle("BorderLayout Demo");
        setSize(400, 300);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);

        // กำหนด Layout Manager เป็น BorderLayout
        setLayout(new BorderLayout());

        add(new JButton("North"), BorderLayout.NORTH);
        add(new JButton("South"), BorderLayout.SOUTH);
        add(new JButton("East"), BorderLayout.EAST);
        add(new JButton("West"), BorderLayout.WEST);
        add(new JButton("Center"), BorderLayout.CENTER);

        setVisible(true);
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> new BorderLayoutDemo());
    }
}
```

11.5 พื้นฐาน JavaFX (ทางเลือก)

JavaFX เป็นแพลตฟอร์มรุ่นใหม่สำหรับการพัฒนา GUI ใน Java ที่มาแทนที่ Swing โดยมีคุณสมบัติที่ทันสมัยกว่า เช่น การใช้ CSS ในการจัดรูปแบบ, การรองรับกราฟิก 2D/3D, และการใช้ FXML สำหรับการออกแบบ UI แบบ declarative

11.5.1 โครงสร้างโปรแกรม JavaFX เบื้องต้น

ตัวอย่าง JavaFX:

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class SimpleJavaFXApp extends Application {

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("My First JavaFX App");

        Label welcomeLabel = new Label("Enter your name:");
        TextField nameTextField = new TextField();
        Button clickButton = new Button("Click Me!");

        clickButton.setOnAction(e -> {
            String name = nameTextField.getText();
            if (name.isEmpty()) {
                welcomeLabel.setText("Please enter your name.");
            } else {
                welcomeLabel.setText("Hello, " + name + "!");
            }
        });

        VBox root = new VBox(10); // Vertical Box with 10px spacing
        root.getChildren().addAll(welcomeLabel, nameTextField, clickButton);
        root.setStyle("-fx-padding: 20px;");

        Scene scene = new Scene(root, 300, 200);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

- `Application` class: เป็นคลาสพื้นฐานสำหรับแอปพลิเคชัน JavaFX
- `start(Stage primaryStage)`: เมธอดหลักที่ใช้ในการสร้างและแสดง UI

- `Stage` : แทนหน้าต่างหลักของแอปพลิเคชัน
- `Scene` : แทนเนื้อหาภายใน Stage
- `VBox` : เป็น Layout Pane ที่จัดเรียงคอมโพเนนต์ในแนวตั้ง
- `setOnAction(e -> { ... })` : การจัดการเหตุการณ์แบบ Lambda Expression ซึ่งเป็นวิธีที่กระชับกว่า `ActionListener` ใน Swing

---

บทที่ 12: การออกแบบซอฟต์แวร์เชิงวัตถุ (Object-Oriented Design)

12.1 บทนำสู่การออกแบบซอฟต์แวร์เชิงวัตถุ

การออกแบบซอฟต์แวร์เชิงวัตถุ (Object-Oriented Design - OOD) คือกระบวนการวางแผนและสร้างโครงสร้างของระบบซอฟต์แวร์โดยใช้แนวคิดเชิงวัตถุ OOD มุ่งเน้นการระบุวัตถุ (objects) และคลาส (classes) ที่เกี่ยวข้องกับปัญหา การกำหนดความสัมพันธ์ระหว่างวัตถุเหล่านั้น และการออกแบบพฤติกรรมของวัตถุ เพื่อให้ได้ระบบที่มีความยืดหยุ่น สามารถนำกลับมาใช้ใหม่ได้ และบำรุงรักษาได้ง่าย

OOD เป็นขั้นตอนสำคัญในวงจรการพัฒนาซอฟต์แวร์ (Software Development Life Cycle - SDLC) ที่อยู่ระหว่างการวิเคราะห์ความต้องการ (Requirements Analysis) และการนำไปใช้งาน (Implementation)

12.2 ภาษาแบบจำลองรวม (Unified Modeling Language - UML)

ภาษาแบบจำลองรวม (Unified Modeling Language - UML) เป็นภาษามาตรฐานที่ใช้ในการสร้างแบบจำลอง (modeling) ระบบซอฟต์แวร์เชิงวัตถุ UML ไม่ใช่ภาษาโปรแกรม แต่เป็นชุดของสัญลักษณ์และแผนภาพที่ช่วยให้นักพัฒนาสามารถสื่อสารแนวคิดการออกแบบระบบได้อย่างชัดเจนและเป็นมาตรฐาน

UML มีแผนภาพหลายประเภท แต่ที่นิยมใช้ในการออกแบบเชิงวัตถุ ได้แก่:

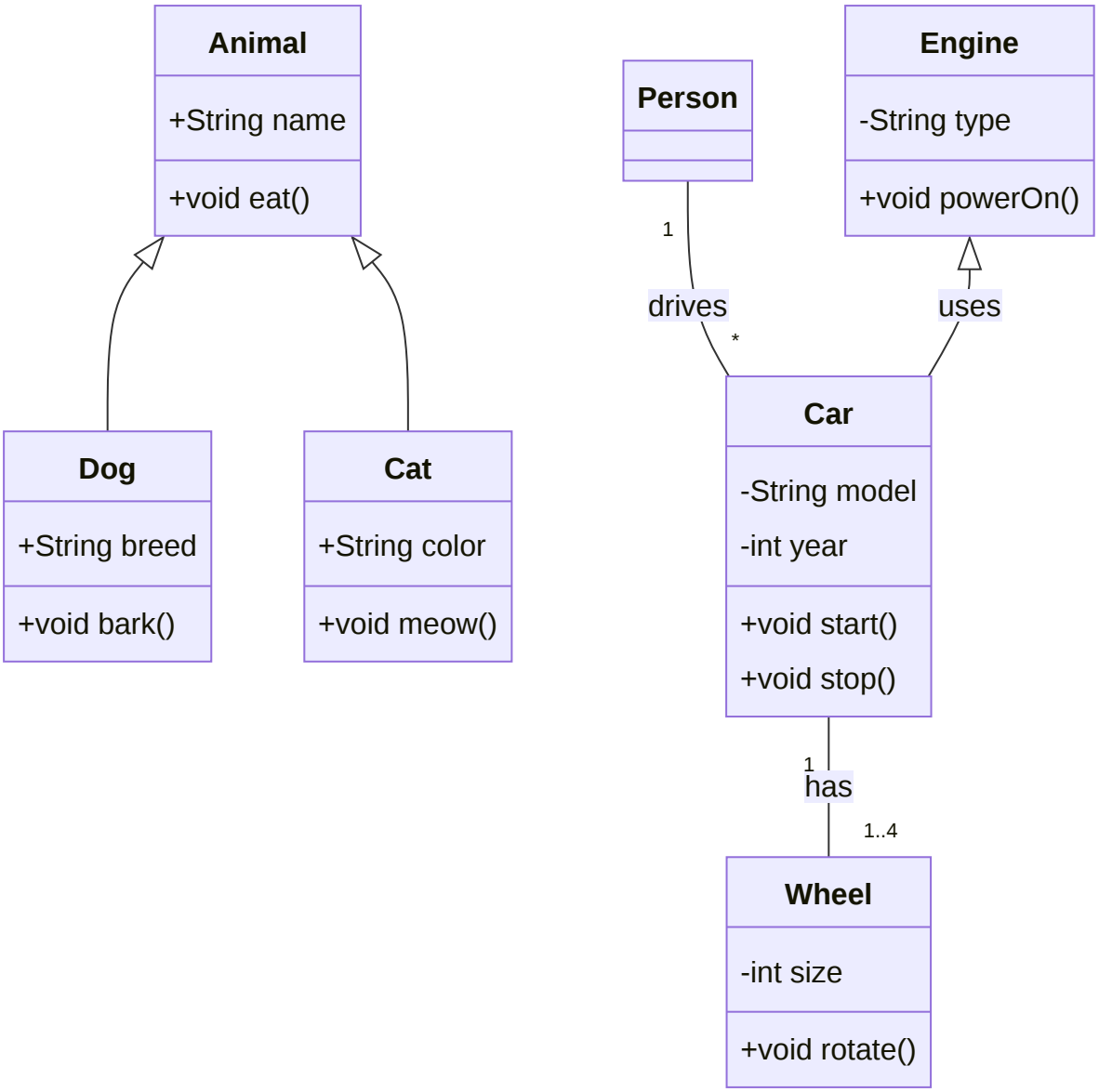
1. **Class Diagram:** แสดงโครงสร้างของระบบในแง่ของคลาส คุณสมบัติ เมธอด และความสัมพันธ์ระหว่างคลาส
2. **Object Diagram:** แสดงอินสแตนซ์ของคลาส (วัตถุ) และความสัมพันธ์ระหว่างวัตถุ ณ เวลาใดเวลาหนึ่ง
3. **Use Case Diagram:** แสดงปฏิสัมพันธ์ระหว่างผู้ใช้ (actors) กับระบบในแง่ของฟังก์ชันการทำงาน (use cases)
4. **Sequence Diagram:** แสดงลำดับการโต้ตอบระหว่างวัตถุต่างๆ ในระบบตามลำดับเวลา
5. **Activity Diagram:** แสดงขั้นตอนการทำงานหรือโฟลว์ของกิจกรรมในระบบ

12.2.1 Class Diagram เบื้องต้น

Class Diagram เป็นแผนภาพที่สำคัญที่สุดในการออกแบบเชิงวัตถุ โดยแสดงถึง:

- **คลาส (Class):** แสดงด้วยสี่เหลี่ยมผืนผ้า แบ่งเป็น 3 ส่วน: ชื่อคลาส, คุณสมบัติ (attributes), และเมธอด (methods)
- **คุณสมบัติ (Attributes):** แสดงด้วย `[visibility] name: type [= defaultValue]`
- **เมธอด (Methods):** แสดงด้วย `[visibility] name(parameters): returnType`
- **Visibility (การมองเห็น):**
  - `+` : public
  - `-` : private
  - `#` : protected
  - `~` : package/default (ใน Java)
- **ความสัมพันธ์ (Relationships):**
  - **Association:** ความสัมพันธ์ทั่วไประหว่างสองคลาส (เส้นตรง)
  - **Aggregation:** ความสัมพันธ์แบบ “has-a” ที่ส่วนประกอบสามารถอยู่ได้ด้วยตัวเอง (เพชรกลวง)
  - **Composition:** ความสัมพันธ์แบบ “has-a” ที่ส่วนประกอบไม่สามารถอยู่ได้ด้วยตัวเอง (เพชรทึบ)
  - **Generalization (Inheritance):** ความสัมพันธ์แบบ “is-a” (ลูกศรสามเหลี่ยมทึบชี้ไปคลาสแม่)
  - **Realization (Implementation):** คลาส implement interface (เส้นประลูกศรสามเหลี่ยมทึบชี้ไป interface)

ตัวอย่าง Class Diagram (Textual Representation):



12.3 รูปแบบการออกแบบ (Design Patterns) เบื้องต้น

รูปแบบการออกแบบ (Design Patterns) คือโซลูชันที่ผ่านการพิสูจน์แล้วสำหรับปัญหาการออกแบบซอฟต์แวร์ที่เกิดขึ้นซ้ำๆ ในบริบทที่แตกต่างกัน Design Patterns ไม่ใช่โค้ดสำเร็จรูป แต่เป็นแนวคิดหรือแม่แบบที่สามารถนำไปปรับใช้เพื่อแก้ปัญหาเฉพาะได้

Design Patterns ถูกแบ่งออกเป็น 3 ประเภทหลัก:

1. **Creational Patterns:** เกี่ยวข้องกับกลไกการสร้างอ็อบเจกต์ เพื่อให้โค้ดมีความยืดหยุ่นและสามารถนำกลับมาใช้ใหม่ได้

2. **Structural Patterns:** เกี่ยวข้องกับการจัดโครงสร้างของคลาสและอ็อบเจกต์ เพื่อสร้างโครงสร้างที่ใหญ่ขึ้นและมีความยืดหยุ่น
3. **Behavioral Patterns:** เกี่ยวข้องกับอัลกอริทึมและการกำหนดความรับผิดชอบระหว่างอ็อบเจกต์ เพื่อให้การสื่อสารระหว่างอ็อบเจกต์มีประสิทธิภาพ

12.3.1 Singleton Pattern (Creational Pattern)

Singleton Pattern เป็นรูปแบบการออกแบบที่รับประกันว่าคลาสจะมีเพียงหนึ่งอินสแตนซ์เท่านั้น และมีจุดเข้าถึงส่วนกลาง (global access point) ไปยังอินสแตนซ์นั้น มักใช้สำหรับทรัพยากรที่ต้องมีเพียงหนึ่งเดียวในระบบ เช่น การเชื่อมต่อฐานข้อมูล, ตัวจัดการการตั้งค่า (configuration manager), หรือตัวจัดการล็อก (logger)

ตัวอย่าง Singleton Pattern ใน Java:

```java
public class Singleton {
    // 1. สร้าง private static instance ของคลาส
    private static Singleton instance;

    // 2. กำหนด constructor เป็น private เพื่อป้องกันการสร้างอ็อบเจกต์จากภายนอก
    private Singleton() {
        // การเริ่มต้นค่าต่างๆ (ถ้ามี)
    }

    // 3. สร้าง public static method เพื่อให้เข้าถึง instance ได้
    public static Singleton getInstance() {
        if (instance == null) { // ตรวจสอบว่ายังไม่มี instance
            instance = new Singleton();
        }
        return instance;
    }

    public void showMessage() {
        System.out.println("Hello from Singleton!");
    }

    public static void main(String[] args) {
        // ไม่สามารถสร้างอ็อบเจกต์โดยตรงได้: Singleton obj = new Singleton(); //
Error

        Singleton s1 = Singleton.getInstance();
        Singleton s2 = Singleton.getInstance();

        System.out.println("Are s1 and s2 the same instance? " + (s1 ==
s2)); // Output: true
        s1.showMessage();
    }
}
```

12.3.2 Strategy Pattern (Behavioral Pattern)

Strategy Pattern เป็นรูปแบบการออกแบบที่ช่วยให้สามารถกำหนดชุดของอัลกอริทึม (strategies) ที่สามารถสับเปลี่ยนกันได้ในระหว่างการรันโปรแกรม โดยแต่ละอัลกอริทึมจะถูกห่อหุ้มอยู่ในคลาสของตัวเอง ทำให้สามารถเปลี่ยนพฤติกรรมของอ็อบเจกต์ได้โดยไม่ต้องแก้ไขโค้ดของอ็อบเจกต์นั้นๆ

ตัวอย่าง Strategy Pattern ใน Java:

```java
// Interface สำหรับ Strategy
interface PaymentStrategy {
    void pay(int amount);
}

// Concrete Strategy 1
class CreditCardPayment implements PaymentStrategy {
    private String cardNumber;
    private String name;

    public CreditCardPayment(String cardNumber, String name) {
        this.cardNumber = cardNumber;
        this.name = name;
    }

    @Override
    public void pay(int amount) {
        System.out.println(amount + " paid with credit card: " + cardNumber
+ " by " + name);
    }
}

// Concrete Strategy 2
class PaypalPayment implements PaymentStrategy {
    private String emailId;

    public PaypalPayment(String emailId) {
        this.emailId = emailId;
    }

    @Override
    public void pay(int amount) {
        System.out.println(amount + " paid using Paypal: " + emailId);
    }
}

// Context Class ที่ใช้ Strategy
class ShoppingCart {
    private PaymentStrategy paymentStrategy;

    public void setPaymentStrategy(PaymentStrategy paymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }

    public void checkout(int amount) {
```

```
        if (paymentStrategy == null) {
            System.out.println("No payment strategy selected.");
            return;
        }
        paymentStrategy.pay(amount);
    }

    public static void main(String[] args) {
        ShoppingCart cart = new ShoppingCart();

        // จ่ายด้วยบัตรเครดิต
        cart.setPaymentStrategy(new CreditCardPayment("1234-5678-9012-3456",
"John Doe"));
        cart.checkout(100);

        // เปลี่ยนไปจ่ายด้วย Paypal
        cart.setPaymentStrategy(new PaypalPayment("john.doe@example.com"));
        cart.checkout(250);
    }
}
```

12.4 หลักการ SOLID

หลักการ SOLID เป็นชุดของหลักการออกแบบเชิงวัตถุ 5 ประการที่ช่วยให้นักพัฒนาสร้างระบบซอฟต์แวร์ที่เข้าใจง่าย ยืดหยุ่น และบำรุงรักษาได้ง่ายขึ้น

- **S - Single Responsibility Principle (SRP):** คลาสควรมีเหตุผลเดียวในการเปลี่ยนแปลง นั่นคือคลาสควรมีหน้าที่รับผิดชอบเพียงอย่างเดียว
- **O - Open/Closed Principle (OCP):** เอนทิตีซอฟต์แวร์ (คลาส, โมดูล, ฟังก์ชัน) ควรเปิดสำหรับการขยาย (extension) แต่ปิดสำหรับการแก้ไข (modification)
- **L - Liskov Substitution Principle (LSP):** อ็อบเจกต์ของคลาสลูกควรสามารถใช้แทนอ็อบเจกต์ของคลาสแม่ได้โดยไม่ทำให้โปรแกรมทำงานผิดพลาด
- **I - Interface Segregation Principle (ISP):** ไคลเอนต์ไม่ควรถูกบังคับให้พึ่งพาอินเทอร์เฟซที่ไม่ได้ใช้ ควรมีอินเทอร์เฟซเฉพาะสำหรับแต่ละไคลเอนต์มากกว่าอินเทอร์เฟซขนาดใหญ่เพียงอันเดียว
- **D - Dependency Inversion Principle (DIP):** โมดูลระดับสูงไม่ควรขึ้นอยู่กับโมดูลระดับต่ำ แต่ควรขึ้นอยู่กับ abstraction (เช่น interface) ทั้งคู่ และ abstraction ไม่ควรขึ้นอยู่กับรายละเอียด แต่รายละเอียดควรขึ้นอยู่กับ abstraction

---