Inheritance ด้วย C++

เทพพิทักษ์ การุญบุญญานันท์

เกริ่นนำ

- กุญแจหลักของ OOP คือ abstraction ซึ่งประกอบด้วยแนวคิด 3 ประการ
  1. Encapsulation
  2. Inheritance
  3. Polymorphism

Object-Oriented vs. Object-Based

- Encapsulation สร้างออบเจกต์ที่มีทั้งข้อมูลและการกระทำครบในตัว
- ภาษาที่เป็น Object-Based จะรองรับเพียง encapsulation เท่านั้น
  (JavaScript, Visual Basic รุ่นแรก ๆ)
- ภาษาที่เป็น Object-Oriented จะครอบคลุมทั้ง encapsulation, inheritance และ polymorphism
  (C++, Java, Objective-C, C#, Python, Ruby, Visual Basic .NET)
- Inheritance ถือเป็นแนวคิดที่นำเข้าสู่โลกของ OOP อย่างแท้จริง

กรณีตัวอย่าง

พนักงานโรงงาน

1. พนักงานประจำ (staff)
   ชื่อ, เลขประจำตัวประชาชน, อายุ, เงินเดือน, วันเข้าทำงานวันแรก, รายการวันลา
2. ลูกจ้างรายวัน (temporary worker)
   ชื่อ, เลขประจำตัวประชาชน, อายุ, ค่าตอบแทน, รายการวันเข้าทำงาน

กรณีตัวอย่าง

```
class Staff {
public:
  // common info
  const char* name();
  const char* citizenID();
  int         age();

  // staff specific
  int         salary();
  Date        entryDate();
  DateList    leaveDates();
};
```

กรณีตัวอย่าง

```
class TempWorker {
public:
  // common info
  const char* name();
  const char* citizenID();
  int         age();

  // temporary worker specific
  int         wage();
  DateList    workDates();
};
```

กรณีตัวอย่าง

ข้อสังเกต

- มีงานซ้ำซ้อนในส่วน common info ในคลาสต่าง ๆ
- จะเกิดอะไรขึ้นถ้ามีข้อมูลช่องใหม่ของพนักงาน (เช่น เพศ, การศึกษา)?
- จะเกิดอะไรขึ้นถ้ามีพนักงานชนิดที่สาม (เช่น พนักงาน contract)?
- จะเกิดอะไรขึ้นถ้าต้องการแยกประเภทพนักงานประจำ (เช่น พนักงานระดับบริหาร, พนักงานระดับปฏิบัติการ)?

→ *วิธีนี้ไม่ scale!*

Inheritance (การสืบทอดคุณสมบัติ)

กลไกการจัดแจงส่วนที่คล้ายกันระหว่างคลาสต่าง ๆ
โดยจัดลำดับชั้นของคลาสเป็นคลาสทั่วไปและคลาสเฉพาะ

- implement ส่วนที่ใช้ร่วมกันเพียงครั้งเดียว – ใน *base class*
- ต่อเติมเป็นกรณีเฉพาะเพิ่มได้ตามต้องการ – ใน *derived class*
- เป็นลำดับชั้นของความสัมพันธ์แบบ “is-a” หรือ “generalization/specialization” คล้ายอนุกรมวิธาน (taxonomy)

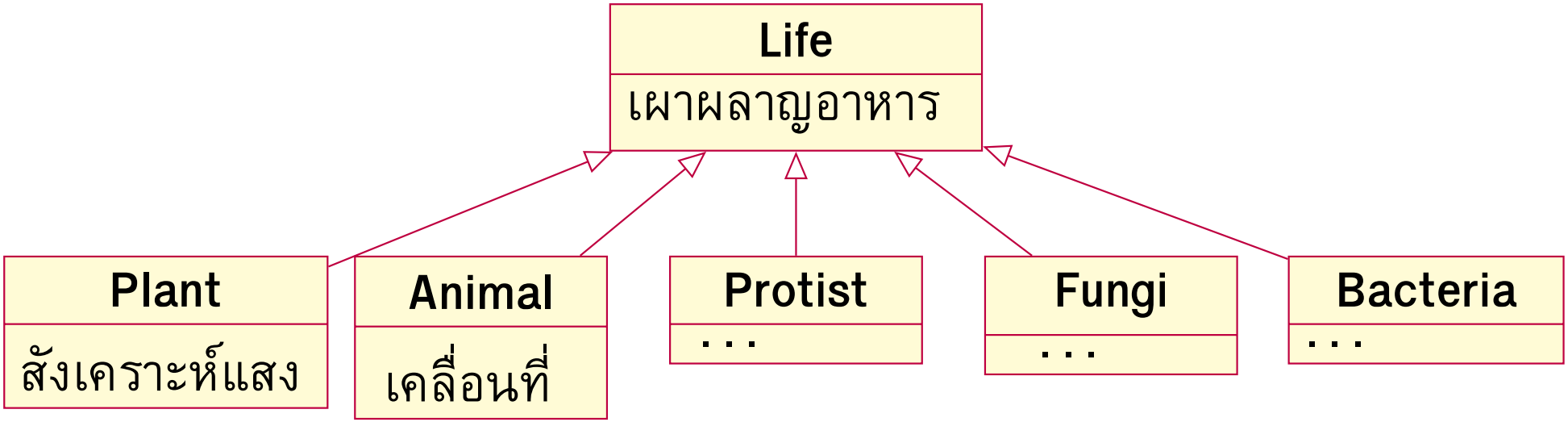
คำศัพท์

คำศัพท์ base class / derived class ในตำราต่าง ๆ

- **superclass / subclass**
  - superclass เป็นชนิดทั่วไป, subclass เป็นชนิดย่อย
- **parent class / child class**
  - parent class เป็นชนิดตั้งต้น, child class เป็นชนิดที่เกิดจาก parent

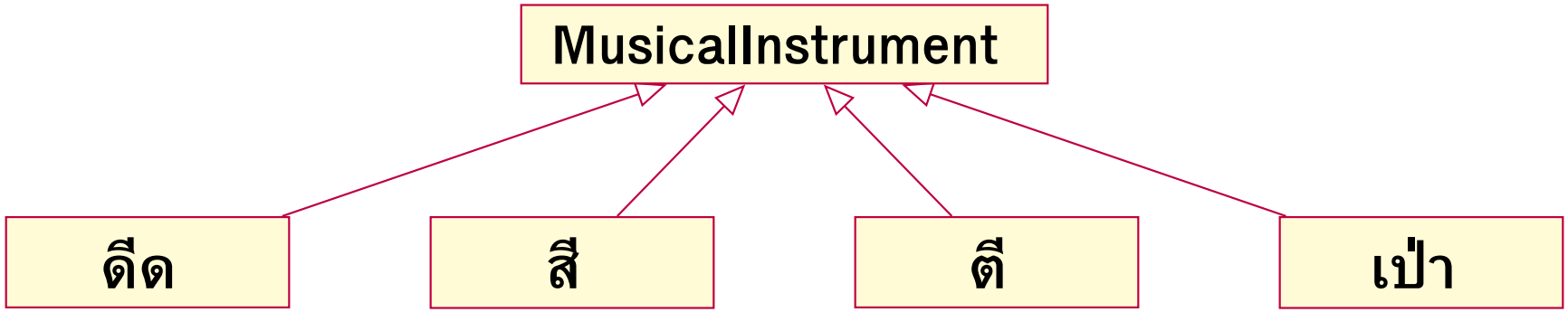
ในที่นี้จะใช้คำว่า base class / derived class

ตัวอย่างแนวคิด Inheritance



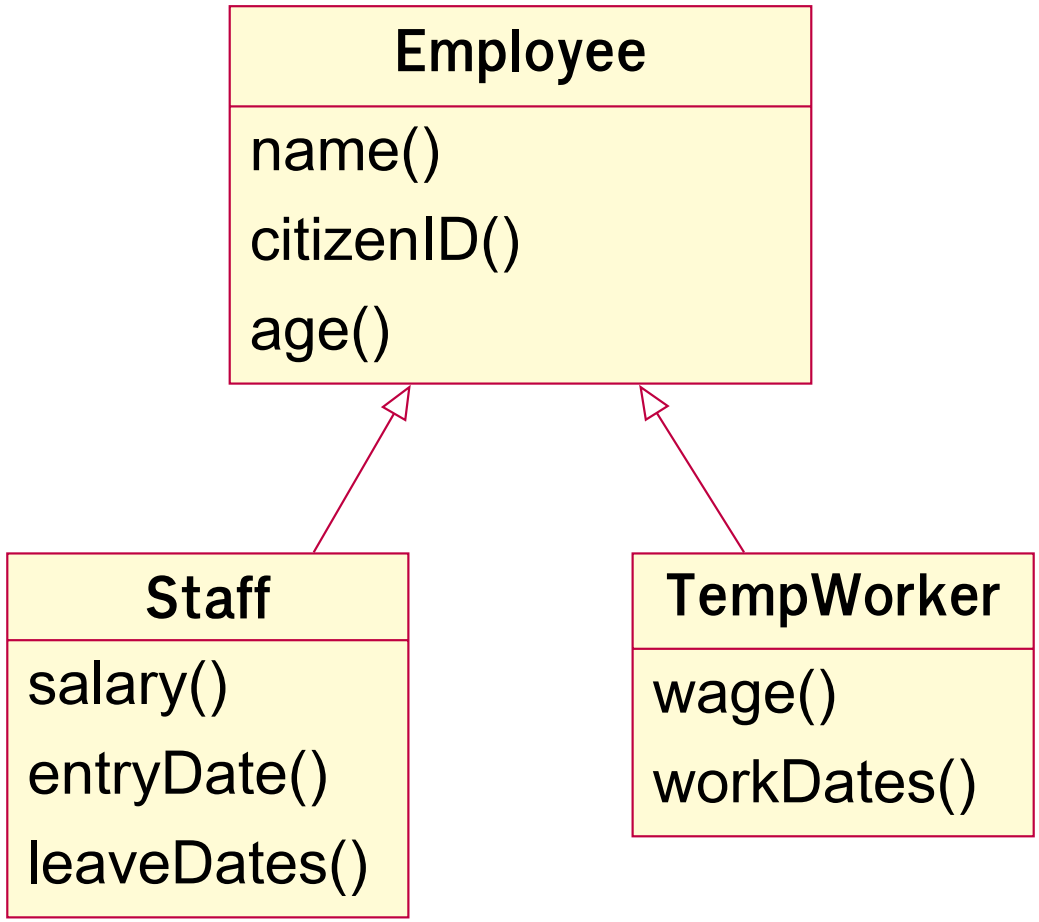
- อนุกรมวิธานของสิ่งมีชีวิต
  - การเข้าใจปฏิกิริยาการเผาผลาญอาหาร ทำให้เข้าใจการสร้างพลังงานของสิ่งมีชีวิตทุกชนิด
  - การเข้าใจปฏิกิริยาสังเคราะห์แสง ทำให้เข้าใจการสร้างอาหารของพืชทุกชนิด

ตัวอย่างแนวคิด Inheritance



- การจำแนกเครื่องดนตรี
  - เครื่องดนตรีทุกชนิดทำให้เกิดเสียง
  - กีตาร์, พิณ, ซึง, จะเข้ ใช้ทักษะการดีดและกดสายคล้ายกัน
  - ไวโอลิน, ซอ, สะล้อ ใช้ทักษะการสีและกดสายคล้ายกัน
  - ระนาด, กลอง, ฉิ่ง, ฉาบ ใช้การกระทบของอุปกรณ์คล้ายกัน
  - ขลุ่ย, ปี่, ฟลุต ใช้ทักษะการปิดเปิดรูบนท่อคล้ายกัน

Inheritance กับกรณีตัวอย่าง



Inheritance กับกรณีตัวอย่าง

```
class Employee {
public:
   const char* name();
   const char* citizenID();
   int         age();
};
```

Inheritance กับกรณีตัวอย่าง

```
class Staff : public Employee {
public:
   int       salary();
   Date      entryDate();
   DateList  leaveDates();
};

class TempWorker : public Employee {
public:
   int       wage();
   DateList  workDates();
};
```

Inheritance กับกรณีตัวอย่าง

ผล → ออบเจกต์ชนิด **Staff** และ **TempWorker** จะ

- สืบทอดคุณสมบัติของ **Employee** มาทุกประการ
- พร้อมกับมีคุณสมบัติเพิ่มเติมของตัวเอง

```
int main()
{
  Staff  s;
  cout << s.name() << ": " << s.salary() << endl;

  TempWorker  t;
  cout << t.name() << ": " << t.wage() << endl;

  return 0;
}
```

Inheritance กับกรณีตัวอย่าง

ผล → ออบเจกต์ชนิด **Staff** และ **TempWorker** จะ

- สามารถเป็นออบเจกต์ชนิด **Employee** ได้

```
void greet (Employee *e) { cout << "Hi, " << e->name(); }

int main()
{
  Staff  s;
  greet (&s);

  TempWorker  t;
  greet (&t);

  return 0;
}
```

Inheritance กับกรณีตัวอย่าง

- แต่ในทางกลับกันไม่ได้
  (**Staff** ทุกคนเป็น **Employee** แต่ **Employee** ไม่จำเป็นต้องเป็น **Staff** ทุกคน)

```
void greet (Staff *s) { cout << "Hi, " << s->name(); }

int main()
{
  Employee e;
  greet (&e); // error!

  TempWorker  t;
  greet ((Employee *)&t); // error!

  return 0;
```

ความยืดหยุ่นที่ได้

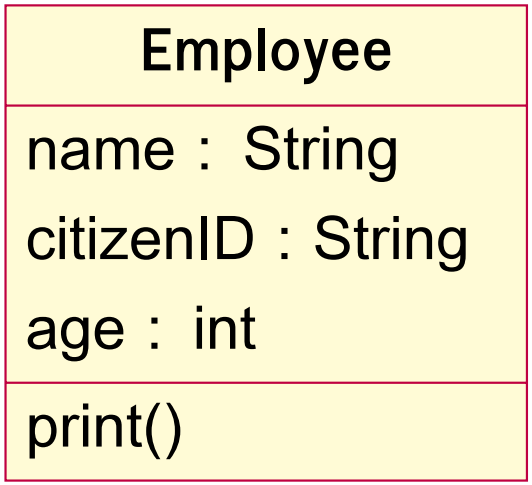
- งานส่วน common info รวมอยู่ในคลาส **`Employee`** ที่เดียว
- จะเกิดอะไรขึ้นถ้ามีข้อมูลช่องใหม่ของพนักงาน (เช่น เพศ, การศึกษา)?
  → เพิ่มใน **`Employee`** แห่งเดียว
- จะเกิดอะไรขึ้นถ้ามีพนักงานชนิดที่สาม (เช่น พนักงาน contract)?
  → สร้างคลาส **`ContractedEmployee`** โดยสืบทอดจากคลาส **`Employee`**
- จะเกิดอะไรขึ้นถ้าต้องการแยกประเภทพนักงานประจำ (เช่น พนักงานระดับบริหาร, พนักงานระดับปฏิบัติการ)?
  → สร้างคลาส **`ExecutiveStaff`** และ **`OperationStaff`** โดยสืบทอดจากคลาส **`Staff`**

UML Notation

UML (Unified Modelling Language)

- รูปแบบการเขียน diagram สำหรับการออกแบบซอฟต์แวร์แบบ Object-Oriented โดยไม่อิงกับภาษาที่ใช้เขียนโปรแกรม
- ประกอบด้วย diagram หลายชุด เพื่อบรรยายแง่มุมต่าง ๆ ของซอฟต์แวร์ เช่น
  - class diagram
  - object diagram
  - component diagram
  - package diagram
  - sequence diagram
  - state diagram
  - use case diagram
  - ...
- ในที่นี้จะสนใจเพียง class diagram เพื่อประโยชน์ในการแสดงโครงสร้าง

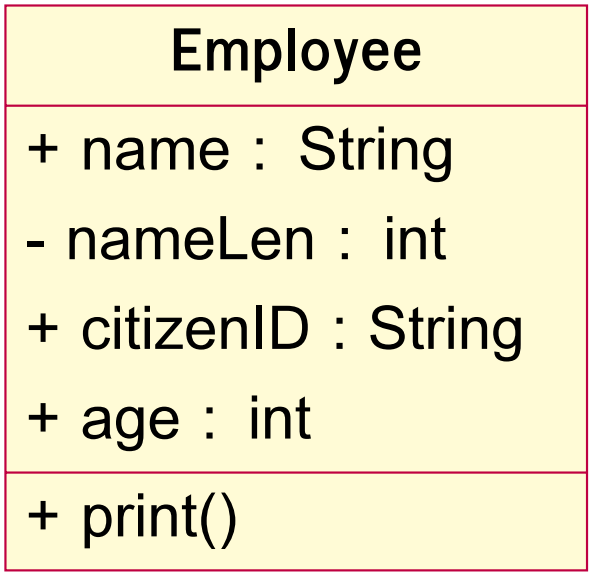
UML Class Diagram



ส่วนประกอบของคลาส

- ชื่อคลาส
- attribute
- operation

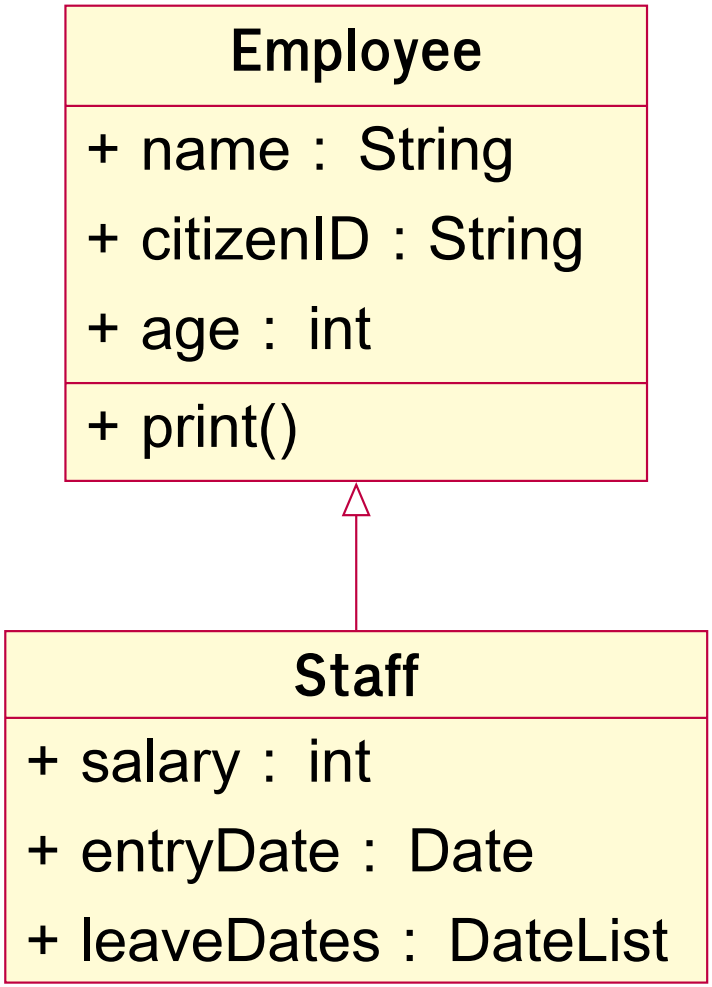
UML Class Diagram



visibility ของ attribute และ operation

- + public
- # protected
- – private

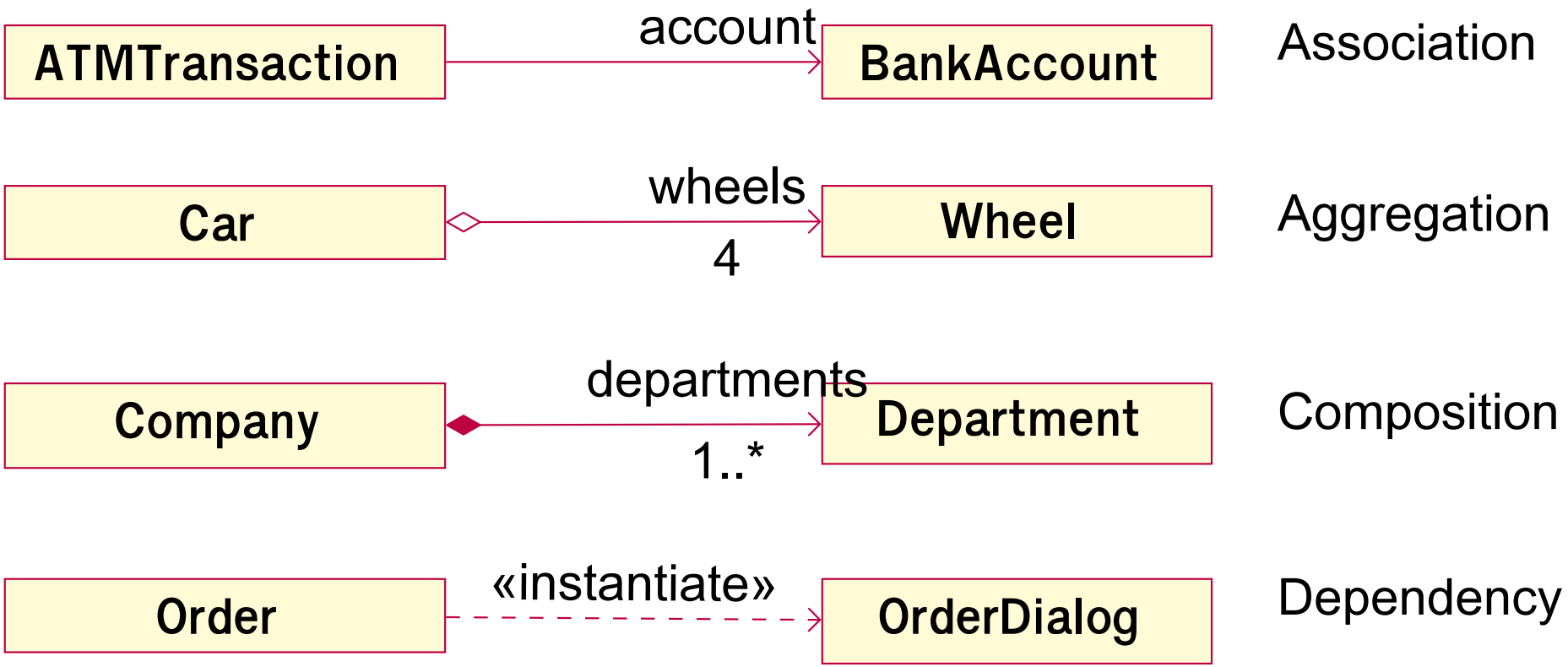
UML Class Diagram



Inheritance

- ใช้เส้นทึบ หัวลูกศรสามเหลี่ยมกลวง
  (เส้นและหัวลูกศรแบบอื่นมีความหมายอื่น)
- โยงลูกศรชี้จาก derived class ไปยัง base class
- ทิศทางของลูกศรแทนทิศของ dependency
  (derived class ต้องอาศัย base class ในการนิยามตัวเอง)

ตัวอย่างความสัมพันธ์แบบอื่น



การ Derive Class ใน C++

```
class Base {
public:
   int pubData;

private:
   int privData;
};
```

```
class Derived : public Base {
   // ...
};
```

การเข้าถึง	ผู้ใช้คลาส **Derived**	ตัวคลาส **Derived** เอง
Base::pubData	ได้	ได้
Base::privData	ไม่ได้	ไม่ได้

Protected Member

- สมาชิกที่เป็น **`protected`** ใน base class:
  - ตัว derived class ทั้งหลายสามารถเข้าถึงได้
  - ผู้ใช้ภายนอก**ไม่**สามารถเข้าถึงได้

Protected Member

```
class Base {
public:
   int pubData;
protected:
   int protData;
private:
   int privData;
};
```

```
class Derived : public Base {
   // ...
};
```

การเข้าถึง	ผู้ใช้คลาส **Derived**	ตัวคลาส **Derived** เอง
Base::pubData	ได้	ได้
Base::protData	ไม่ได้	ได้
Base::privData	ไม่ได้	ไม่ได้

Private & Protected Inheritance

```
class Derived : public Base {
   // ...
};
```

- access specifier ขณะประกาศ inheritance:
  - ประกาศให้สิทธิ์ผู้ใช้ derived class ในการเข้าถึงสมาชิกของ base class *ไม่เกิน* access specifier ที่กำหนด

member accessibility	access specifier ใน base class		
	public	protected	private
public inheritance	public	protected	private
protected interitance	protected	protected	private
private inheritance	private	private	private

Private & Protected Inheritance

```
class Base {
public:
   int pubData;
protected:
   int protData;
private:
   int privData;
};
```

```
class Derived : private Base {
   // ...
};
```

การเข้าถึง	ผู้ใช้คลาส **Derived**	ตัวคลาส **Derived** เอง
Base::pubData	ไม่ได้	ได้
Base::protData	ไม่ได้	ได้
Base::privData	ไม่ได้	ไม่ได้

Private & Protected Inheritance

```
class Base {
public:
   int pubData;
protected:
   int protData;
private:
   int privData;
};
```

```
class Derived : protected Base {
   // ...
};
```

การเข้าถึง	ผู้ใช้คลาส **Derived**	ตัวคลาส **Derived** เอง
Base::pubData	ไม่ได้	ได้
Base::protData	ไม่ได้	ได้
Base::privData	ไม่ได้	ไม่ได้

Private & Protected Inheritance

ข้อสังเกต Private & Protected Inheritance:

- accessibility สำหรับผู้ใช้คลาส **`Derived`** ไม่ต่างกัน
- ความแตกต่างอยู่ที่สิทธิ์ของคลาสที่ inherit จาก **`Derived`** อีกทอด

```
class Derived2 : public Derived {
  // ...
};
```

  - protected inheritance:
    - คลาส **`Derived2`** จะยังคงสามารถเข้าถึง **`Base::pubData`** และ **`Base::protData`** ได้
  - private inheritance:
    - คลาส **`Derived2`** จะไม่สามารถเข้าถึง **`Base::pubData`** และ **`Base::protData`** ได้

Private & Protected Inheritance

- inheritance ส่วนใหญ่ในโปรแกรมต่าง ๆ จะเป็น public inheritance
- private & protected inheritance ใช้ในบางกรณีที่ต้องการซ่อนรายละเอียดของ implementation จากผู้ใช้เท่านั้น

การเข้าถึงสมาชิกของ Base Class

member function ของ derived class สามารถเข้าถึง member ที่เป็น public และ protected ของ base class ได้เหมือนเป็น member ของตัวเอง:

```
class Staff : public Employee {
  // ...
public:
  void print() const;
};

void Staff::print() const
{
  // Calling Employee::name() and Staff::salary()
  cout << name() << ": " << salary() << endl;
}
```

การ Override Base Class

derived class สามารถกำหนด member function ทับของ base class ได้

```
class Employee {
  // ...
public:
  void print() const;
};

class Staff : public Employee {
  // ...
public:
  void print() const;
};
```

```
Employee e;
// Employee::print()
e.print();

Staff s;
// Staff::print()
s.print();
```

การ Override Base Class

ในฟังก์ชันของ derived class อาจเรียกฟังก์ชันของ base class ได้

```
void Staff::print()
{
  Employee::print();

  cout << ": " << salary();
}
```

Constructor & Destructor

ถ้า base class มี constructor ที่รับอาร์กิวเมนต์
constructor ของ derived class ก็สามารถเรียกใน initialization list ได้

```
class Employee {
public:
  Employee (const char* name, const char* id, int age);
  // ...
};

class Staff : public Employee {
public:
  Staff (const char* name, const char* id, int age,
         int salary)
    : Employee (name, id, age), salary (salary) {}
};
```

Constructor & Destructor

ขั้นตอนการ construct ออบเจกต์: จากล่างขึ้นบน

1. เรียก constructor ของ base class
2. เรียก constructor ของ member ต่าง ๆ ตามลำดับการประกาศ
3. เรียก constructor ของตัว derived class เอง

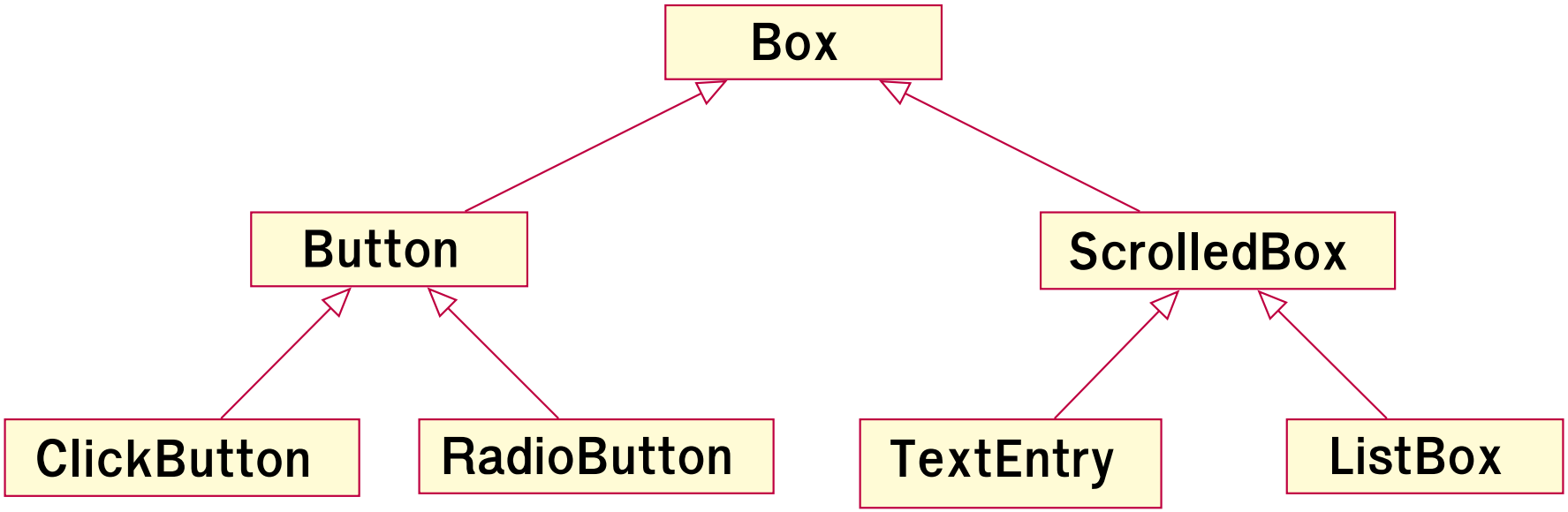
ขั้นตอนการ destruct ออบเจกต์: ย้อนคืน คือจากบนลงล่าง

1. เรียก destructor ของตัว derived class เอง
2. เรียก destructor ของ member ต่าง ๆ ย้อนลำดับการประกาศ
3. เรียก destructor ของ base class

Class Hierarchy

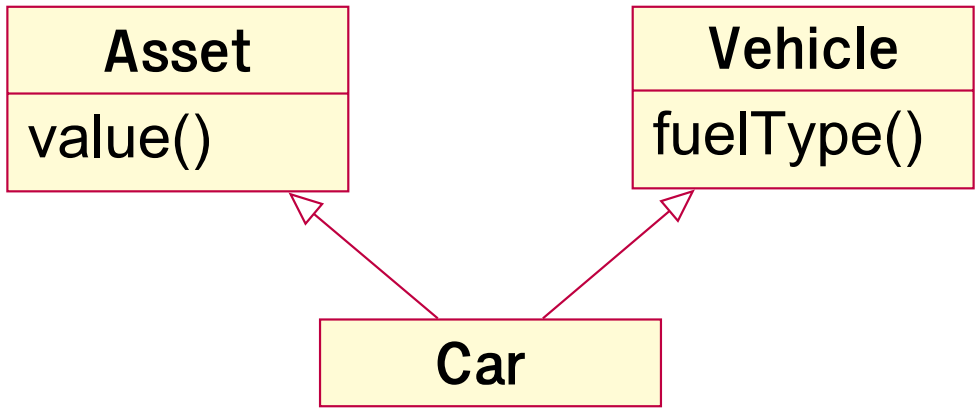
derived class สามารถเป็น base class ให้กับคลาสอื่นได้อีก

$\rightarrow$ กลายเป็น “class hierarchy”



Multiple Inheritance

คลาสในภาษา C++ สามารถ inherit จาก base class มากกว่าหนึ่งคลาสได้
$\rightarrow$ เรียกว่า “multiple inheritance”




```
class Car : public Asset, public Vehicle {
   // ...
};
```

ผล:

- ออบเจกต์ชนิด **Car** สืบทอด method มาจากทั้ง **Asset** และ **Vehicle**

```
Car c;
int val = c.value();        // Asset::value()
Fuel fuel = c.fuelType(); // Vehicle::fuelType()
```

- พอยน์เตอร์ **Car*** สามารถใช้เป็นพอยน์เตอร์ **Asset*** และ **Vehicle*** ได้

```
Car c;
Asset* pA = &c;
Vehicle* pV = &c;
```

ข้อควรระวังในการใช้ Multiple Inheritance

- เมื่อชื่อสมาชิกของ base class ชนกัน

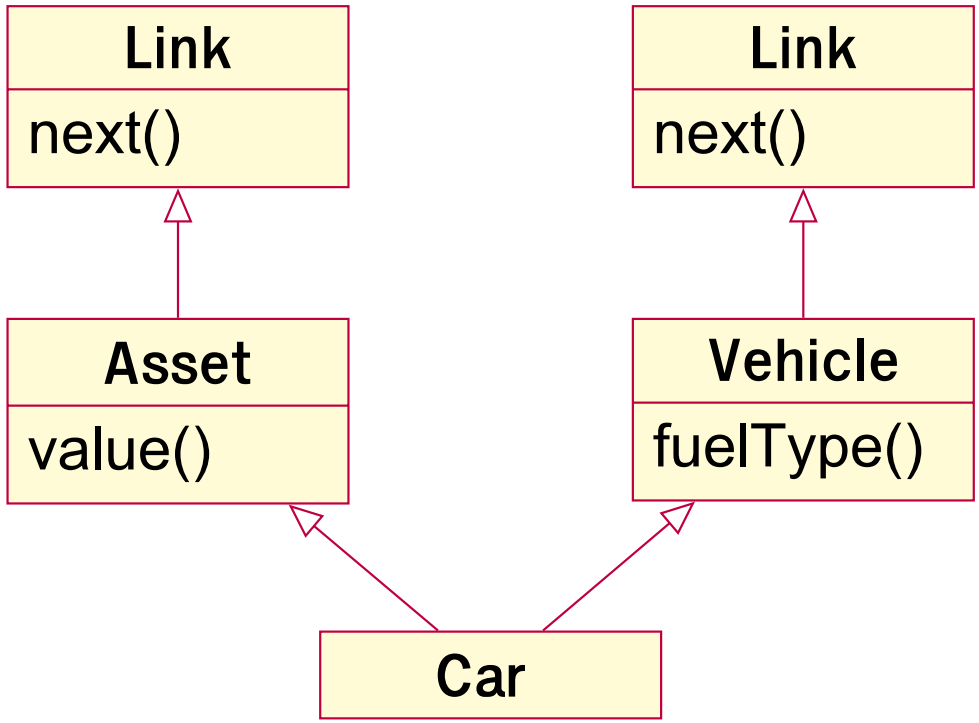
```
class Asset {
  // ...
protected:
  DebugInfo* getDebug();
};

class Vehicle {
  // ...
protected:
  DebugInfo* getDebug();
};
```

```
void f (Car* c)
{
  // error: ambiguous!
  DebugInfo* db = c->getDebug();
  // ok
  db = c->Asset::getDebug();
  // ok
  db = c->Vehicle::getDebug();
}
```

ข้อควรระวังในการใช้ Multiple Inheritance

- เมื่อ base class มีบรรพบุรุษเหมือนกัน



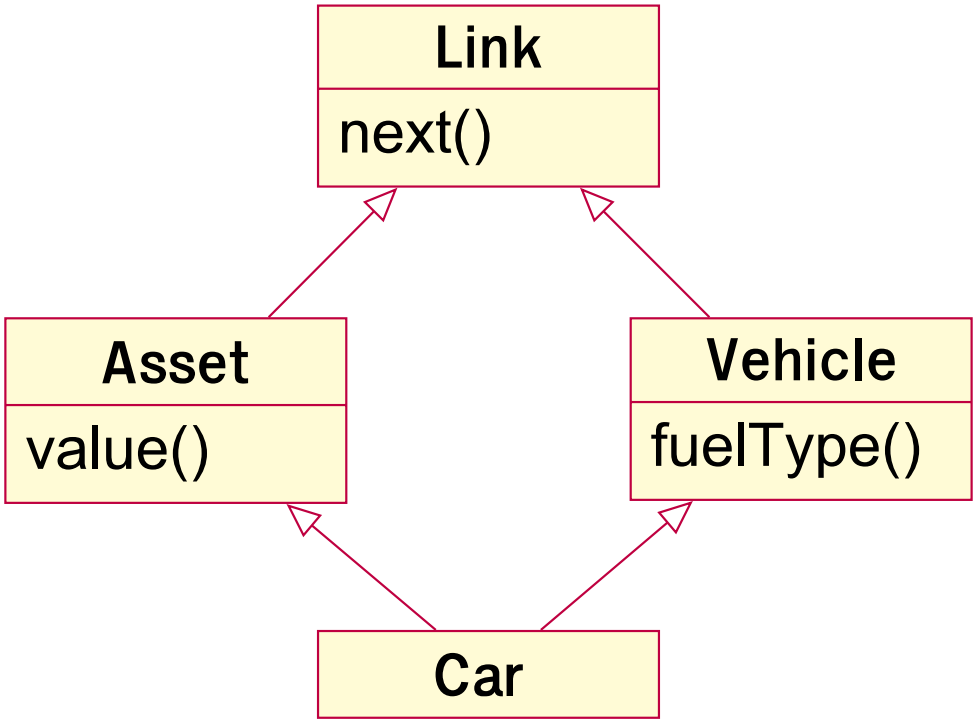
ข้อควรระวังในการใช้ Multiple Inheritance

- เมื่อ base class มีบรรพบุรุษเหมือนกัน

```
void f (Car* c)
{
  Link* next = 0;
  next = c->next();                    // error, which Link?
  next = c->Link::next();              // error, which Link?
  next = c->Asset::Link::next();       // ok
  next = c->Vehicle::Link::next();     // ok
}
```

Virtual Base Class

เมื่อ base class มีบรรพบุรุษเหมือนกัน สามารถทำให้เหลือชุดเดียวได้



Virtual Base Class

วิธีการ: ใช้ keyword **`virtual`** ในการ derive class

```
class Asset : public virtual Link {
   // ...
};

class Vehicle : public virtual Link {
   // ...
};

class Car : public Asset, public Vehicle {
   // ...
};
```

Virtual Base Class

ผล:

- ออบเจกต์ชนิด **Car** จะมีข้อมูลของ **Link** เพียงชุดเดียว
- ไม่เกิดความกำกวมในการเรียก member ของ **Link**

```
void f (Car* c)
{
  Link* next = c->next();   // ok
}
```