
클래스와 객체란 무엇인가?

HL만도 소프트웨어 트랙 3기 박종현

객체지향 프로그래밍의 탄생배경

- 소프트웨어 복잡성 증가, 코드 재사용의 제약과 한계, 현실 세계 모델링 필요성



```
Class Rectangle (Width, Height): Real Width, Height;
! Class with two parameters:
Begin
  Real Area, Perimeter; ! Attributes:

  Procedure Update; ! Methods (Can be Virtual):
  Begin
    Area := Width * Height;
    Perimeter := 2*(Width + Height)
  End of Update;

  Boolean Procedure IsSquare;
  IsSquare := Width=Height;

  Update; ! Life of rectangle started at creation:
  OutText("Rectangle created: "); OutFix(Width,2,6);
  OutFix(Height,2,6); OutImage End of Rectangle;
End of Rectangle
```

시뮬라(Simula):

1960년대 후반, ALGOL 60을 기반으로 구축된 객체 지향의 초기 형태가 나타나기 시작했습니다. 시뮬라 언어가 초기 객체지향 개념을 제공했습니다(최소한의 프로그램은 빈 블록으로 간단히 표현)



1970년대 후반, 객체 지향 프로그래밍의 개념을 대중화한 프로그래밍 언어인 스몰토크 (Smalltalk)를 개발했습니다. Smalltalk 언어가 등장하여 현대적인 객체지향 프로그래밍의 모습을 갖추게 되었습니다.

```
| x y className methodName
className := 'Poo'.
methodName := 'hello'.
x := (Compiler evaluate: className).
(x isKindOfClass: Class) ifTrue: [
  y := x new.
  (y respondsTo: methodName asSymbol) ifTrue: [
    y perform: methodName asSymbol
  ]
]
```



C++, Java, C# 등의 언어가 등장하며 객체지향의 발전을 이끌었습니다.

객체지향 프로그래밍과 procedural programming의 차이



●Object Oriented Programming

- 코드의 높은 재활용성
- 유지보수가 쉽다
- 개발 속도 및 실행 속도가 procedural programming 보다 느리다

●procedural programming

- 유지보수가 어렵다.
- 함수가 너무 많아져서 프로그램의 이해가 어려움
- 디버깅이 힘들

Struct와 Class의 차이점

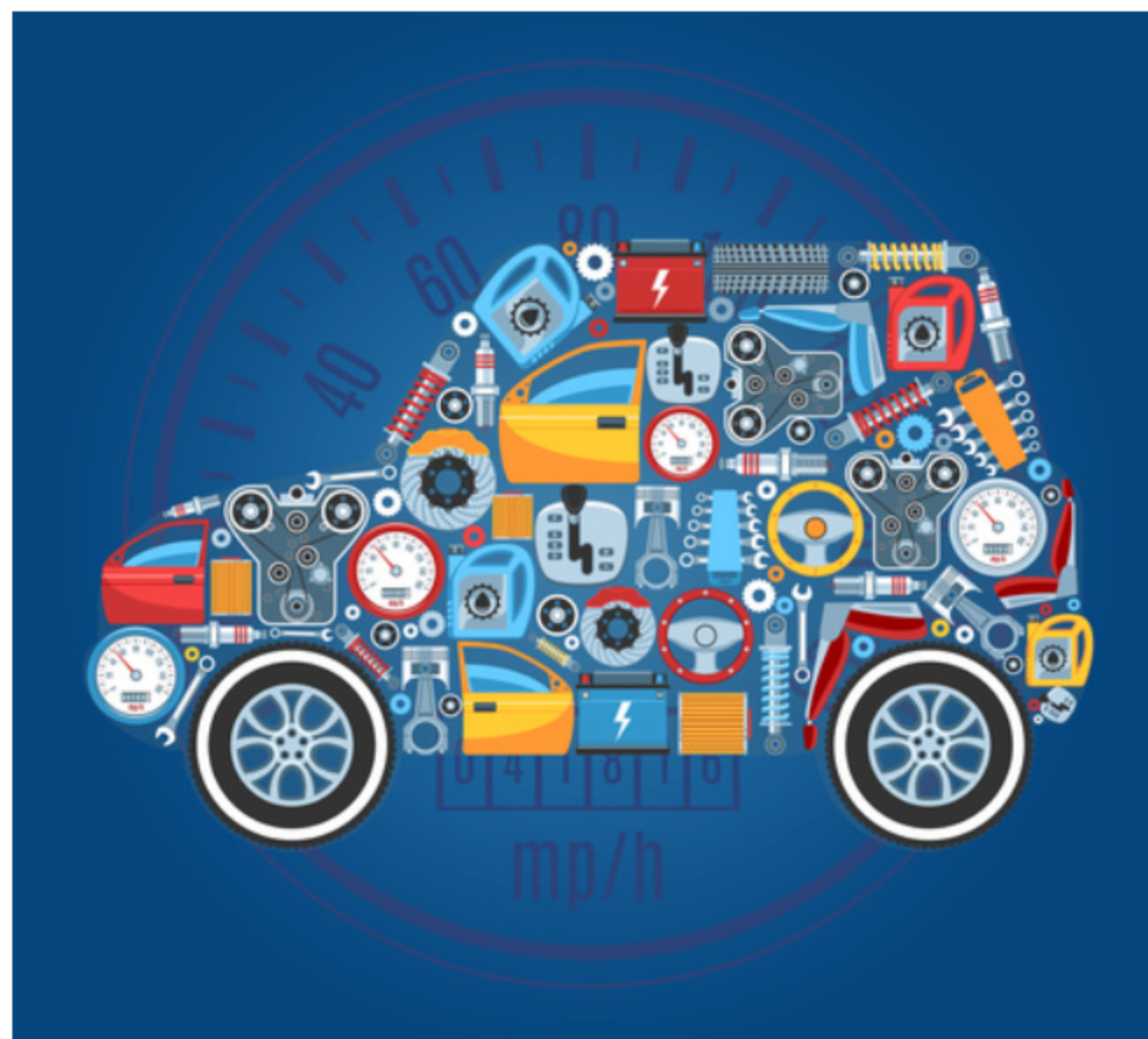
```
public struct struct_test
{
    public int a;
    public int b;

    public struct_test(int a_a, int a_b) //파라미터가 반드시 있어야함
    {
        a = a_a;
        b = a_b;
    }
    public void show()
    {
        Console.WriteLine("a={0} , b={1}",a,b);
    }
}
class Program
{
    static void Main(string[] args)
    {
        struct_test struct_; //new가 필요없다.
        struct_.a = 10;
        struct_.b = 20;
        struct_.show();
    }
}
```

- 구조체는 생성자를 선언할 수 있으나 반드시 파라미터가 있어야 한다.
- 클래스는 상속이 가능하지만, 구조체는 상속이 불가능하다.
- 구조체는 기본 접근 지정자가 Public이고 클래스는 Private임
- 타입(값, 참조)에 따른 메모리 할당 방식의 차이(Struct는 값 타입 (ValueType)이지만 Class는 참조(Reference Type))

객체지향 프로그래밍이란?

- 객체 지향 프로그래밍(OOP)은 컴퓨터 프로그래밍 패러다임 중 하나다.
- 객체(Object)들이 모여 협력하면서 데이터를 처리하는 방식의 프로그래밍 설계 방법이다.
- 프로그램을 묶음 단위로 쪼개서, 나중에 가져다 쓰기 편하게 만들어 놓은 프로그래밍 방식



Thread 문제점

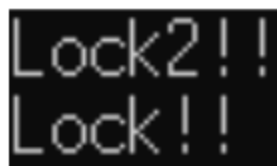
```
class Program
{
    private static object _lock = new object();
    private static object _lock2 = new object();

    public static void Run()
    {
        for (int i = 0; i < 5; i++)
        {
            new Thread(LockingTest).Start();
            new Thread(LockingTest2).Start();
        }
    }

    private static void LockingTest()
    {
        lock (_lock)
        {
            Thread.Sleep(1000);
            Console.WriteLine("Lock!!");
            LockingTest2();
        }
    }

    private static void LockingTest2()
    {
        lock (_lock2)
        {
            Thread.Sleep(1000);
            Console.WriteLine("Lock2!!");
            LockingTest();
        }
    }

    static void Main(string[] args)
    {
        Run();
    }
}
```



```
Lock2!!
Lock!!
```

- 멀티 스레드 이용시 여러 개의 스레드가 동시에 실행 되기 때문에 실행결과가 달라질 수 있음
- 멀티 스레드가 , 각각이 서로 다른 자원을 사용하고자 할 때 데드락이 발생
- 두개 이상의 작업이 서로 상대방의 작업이 끝나기만을 기다릴때 데드락 발생
- 서로 다른 자원을 독점적으로 점유하려고 할 때 데드락이 발생할 수 있습니다.

Class, 객체

클래스(Class)란

- 연관되어 있는 변수와 메서드의 집합
- 객체를 만들어 내기 위한 설계도 혹은 틀
- 여러 프로그램에서 코드를 재사용하고 충돌을 방지하기 위한 것입니다.

객체(Object)란

- 소프트웨어 세계에 구현할 대상
- 클래스에 선언된 모양 그대로 생성된 실체

특징:

- 클래스의 인스턴스(instance) 라고도 부른다.
- oop의 관점에서 클래스의 타입으로 선언되었을 때 ‘객체’라고 부른다.

클래스(Class) VS 객체(Object)

- 클래스는 ‘설계도’, 객체는 ‘설계도로 구현한 모든 대상’을 의미한다.

클래스를 사용하는 방법

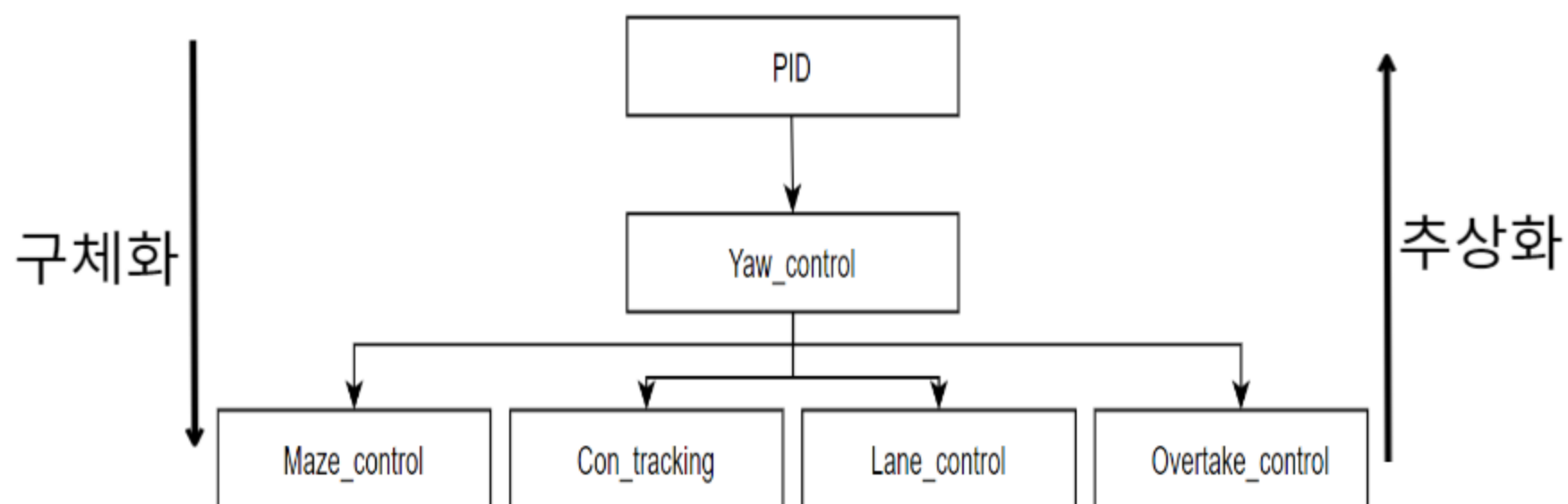
```
class Cart {  
    string name;  
    string color;  
    int speed;  
  
    void start() {}  
    void stop() {}  
    void drift() {}  
};
```

속성(attribute)

메서드(method)

- 클래스의 속성(attribute)과 메서드(method)는 변수와 함수로 표현된다.
- 속성은 어떤 값을 저장하는 역할을 하기 때문에 변수로 표현한다.
- 메서드는 기능적인 측면을 표현하는 것이므로 함수로 표현되는 것이다.

객체지향 프로그래밍의 4가지 특징



추상화 (Abstraction)

추상 : 여러 가지 사물이나 개념에서 공통되는 특성이나 속성 따위를 추출해 파악하는 작용

- 클래스들의 공통적인 특성(변수, 메소드)들을 묶어 표현하는 것
- 상속이 자식 클래스를 만드는데 부모 클래스를 사용하는 것이라면, 추상화는 이와 반대로 기존 클래스의 공통 부분을 뽑아내 부모 클래스를 만드는 것이라고 볼 수 있다.

객체지향 프로그래밍의 4가지 특징

```
class Person {  
    private :  
        string name;  
        int height;  
        int weight;  
  
    public :  
        void Print() {  
            cout << name << " is " << height << "cm tall and weighs "  
            << weight << "kg." << endl;  
        }  
        void Change(int a, int b) { height = a; weight = b; }  
};
```

Person 클래스의
객체의 멤버변수에
접근할 수 없음

Print() 함수나 Change()
함수에는 접근할 수 있다.

캡슐화(Encapsulation)

- 캡슐화는 데이터, 데이터를 활용하는 함수를 캡슐 안에 두는 것을 의미(데이터와 함수를 클래스 안에 넣는 것)



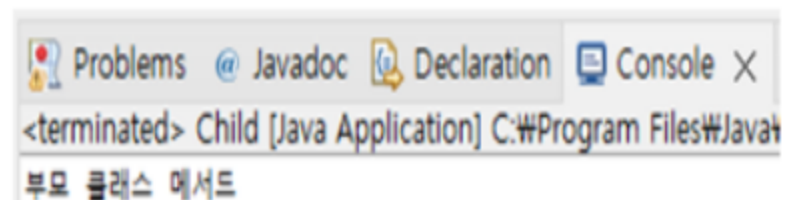
Class

- 캡슐화를 사용하여 표시할 클래스의 속성과 숨길 속성을 선택할 수 있음
- 캡슐화는 어떻게 클래스 정보에 접근 혹은 수정하는지 결하는 권한을 제공

객체지향 프로그래밍의 4가지 특징

- Example) `extends` 라는 키워드 이용

```
public class Parent {  
    public void inheritance() {  
        System.out.println("부모 클래스 메서드");  
    }  
}  
  
class Child extends Parent {  
    public static void main(String[] args) {  
        Child child = new Child();  
        child.inheritance();  
    }  
}
```



상속성(Inheritance)

- 부모 클래스에 정의된 변수 및 메서드를 자식 클래스에서 상속받아 사용하는 것
- 상속 덕분에 코드를 더 작은 단위로. Class 로 쪼개고, 더 작은 단위로 나누고, 재사용 할 수 있음.
- 코드를 재사용하고 공용 클래스와 인터페이스를 통해 원본 소프트웨어를 독립적으로 확장할 수 있다.

객체지향 프로그래밍의 4가지 특징

다형성(Polymorphism)

- 메시지에 의해 객체가 연산을 수행하게 될 때, 하나의 메시지에 대해 각 객체가 가지고 있는 고유한 방법으로 응답할 수 있는 능력
- 다형성의 특징:
다형성을 활용하면 기능을 확장하거나, 객체를 변경해야할 때 타입 변경 없이 객체 주입만으로 수정이 일어나게 할 수 있다.
- 다형성을 구현하는 방법:
오버로딩 (Overloading):동일한 이름의 메서드를 여러 개 정의하는 것,결국 같은 기능을 하도록 만들기 위한 작업

오버라이딩 (Overriding):상위 클래스의 메서드를 하위 클래스에서 재정의하는 것을 말한다.

객체지향 프로그래밍의 4가지 특징

오버로딩 (Overloading)

```
class Calculator {  
  add(a: number, b: number): number {  
    return a + b;  
  }  
  
  add(a: string, b: string): string {  
    return a + b;  
  }  
}  
  
const calculator = new Calculator();  
  
console.log(calculator.add(1, 2)); // 3  
console.log(calculator.add("Hello, ", "world!")); // Hello, world!
```

매개변수 타입에
따라 다르게 동작

오버라이딩 (Overriding)

```
class Animal {  
  makeSound() {  
    console.log("Animal is making a sound.");  
  }  
}  
  
class Dog extends Animal {  
  makeSound() {  
    console.log("Dog is barking.");  
  }  
}  
  
class Cat extends Animal {  
  makeSound() {  
    console.log("Cat is meowing.");  
  }  
}  
  
const animal = new Animal();  
const dog = new Dog();  
const cat = new Cat();  
  
animal.makeSound(); // Animal is making a sound.  
dog.makeSound(); // Dog is barking.  
cat.makeSound(); // Cat is meowing.
```

makeSound 메서드를 각각 오버라이딩하여 동작을 다르게 수행할 수 있습니다.(클래스만의 메서드를 재정의)

