한빛미디어 HanbitMedia, Inc.




Bsidesoft co.

since 2004


OBJECT ORIENTED JAVASCRIPT

1 2 3 4 5

Value Context
vs
Identifier Context

```
const a = {
    a:3,
    b:5
};
const b = {
    a:3,
    b:5
};

console.log(a === b);

console.log(JSON.stringify(a) === JSON.stringify(b));
```

```
const a = {
    a:3,
    b:5
};
const b = {
    a:3,
    b:5
};

console.log(a === b);

console.log(JSON.stringify(a) === JSON.stringify(b));
```

Identifier

```
const a = {
    a:3,
    b:5
};
const b = {
    a:3,
    b:5
};

console.log(a === b);
```

Identifier

```
console.log(JSON.stringify(a) === JSON.stringify(b));
```

Value

Value vs Identifier

1. 끝 없는 복사본
2. 상태 변화에 안전?
3. 연산을 기반으로 로직을 전개

Value vs Identifier

1. 끝 없는 복사본
2. 상태 변화에 안전?
3. 연산을 기반으로 로직을 전개

1. 하나의 원본
2. 상태 변화를 내부에서 책임짐
3. 메세지를 기반으로 로직을 전개

Polymorphism

```
const Worker = class{
  run(){
      console.log("working")
  }
  print(){
      this.run();
  }
};

const HardWorker = class extends Worker{
    run(){
        console.log("hardWorking")
    }
};

const worker = new HardWorker();
console.log(worker instanceof Worker);
worker.print();
```

```
const Worker = class{
  run(){
      console.log("working")
  }
  print(){
      this.run();
  }
};

const HardWorker = class extends Worker{
    run(){
        console.log("hardWorking")
    }
};

const worker = new HardWorker();
console.log(worker instanceof Worker);
worker.print();
```

```
const Worker = class{
  run(){
      console.log("working")
  }
  print(){
      this.run();
  }
};

const HardWorker = class extends Worker{
    run(){
        console.log("hardWorking")
    }
};

const worker = new HardWorker();
console.log(worker instanceof Worker);
worker.print();
```

```
const Worker = class{
  run(){
      console.log("working")
  }
  print(){
      this.run();
  }
};

const HardWorker = class extends Worker{
    run(){
        console.log("hardWorking")
    }
};

const worker = new HardWorker();
console.log(worker instanceof Worker);
worker.print();
```

substitution

```
const Worker = class{
  run(){
      console.log("working")
  }
  print(){
      this.run();
  }
};

const HardWorker = class extends Worker{
    run(){
        console.log("hardWorking")
    }
};

const worker = new HardWorker();
console.log(worker instanceof Worker);
worker.print();
```

substitution

internal identity

Substitution & Internal identity

확장된 객체는 원본으로 대체 가능
생성 시점의 타입이 내부에 일관성 있게 참조됨.

Polymorphism of Prototype

Polymorphism of Prototype

Polymorphism of Prototype

worker

HardWorker
class(Function)

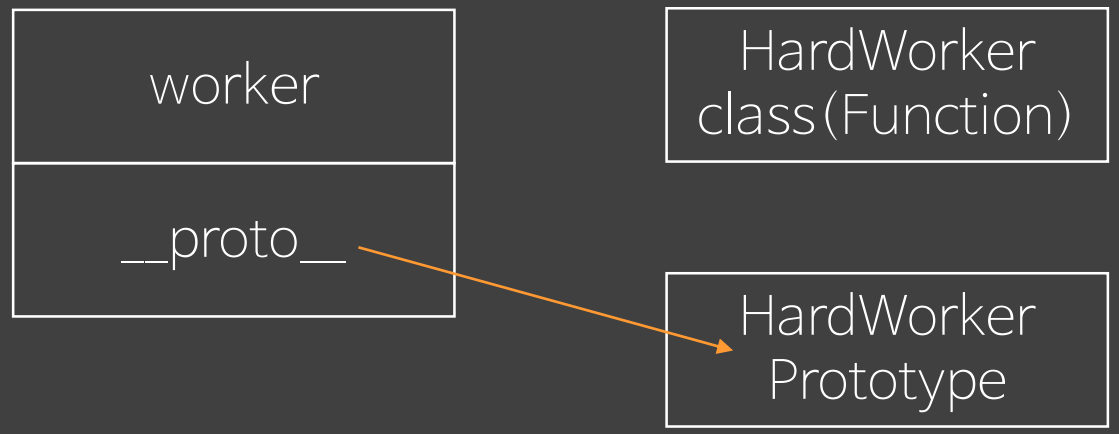
Polymorphism of Prototype

worker

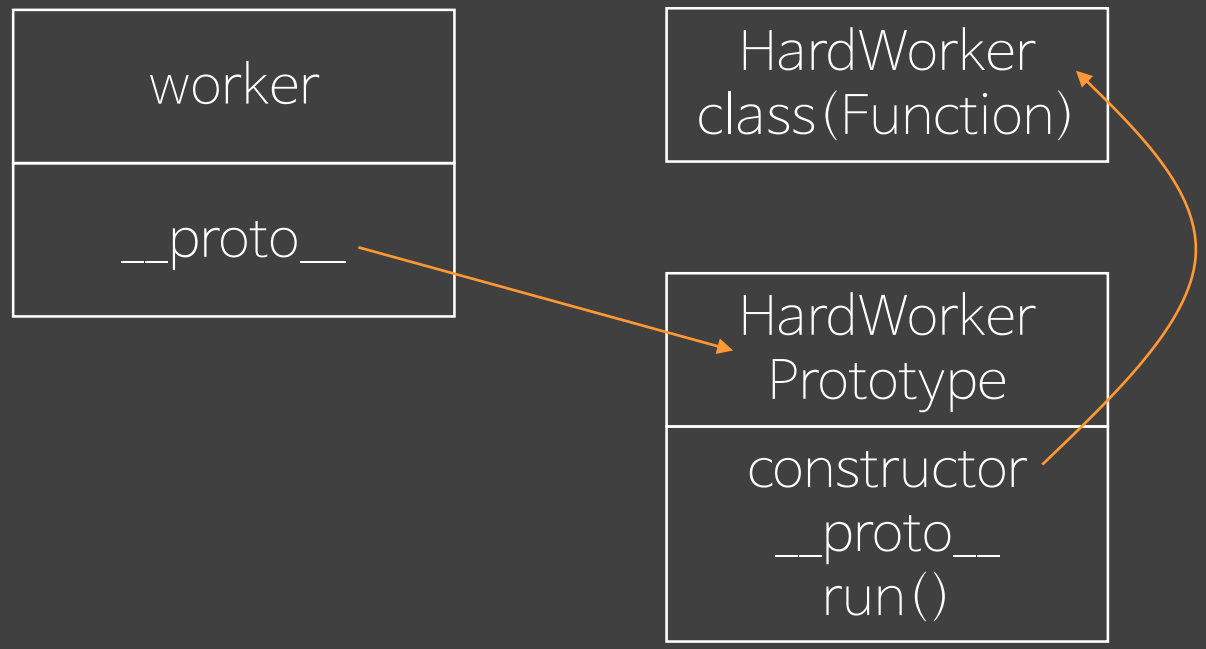
HardWorker
class(Function)

HardWorker
Prototype

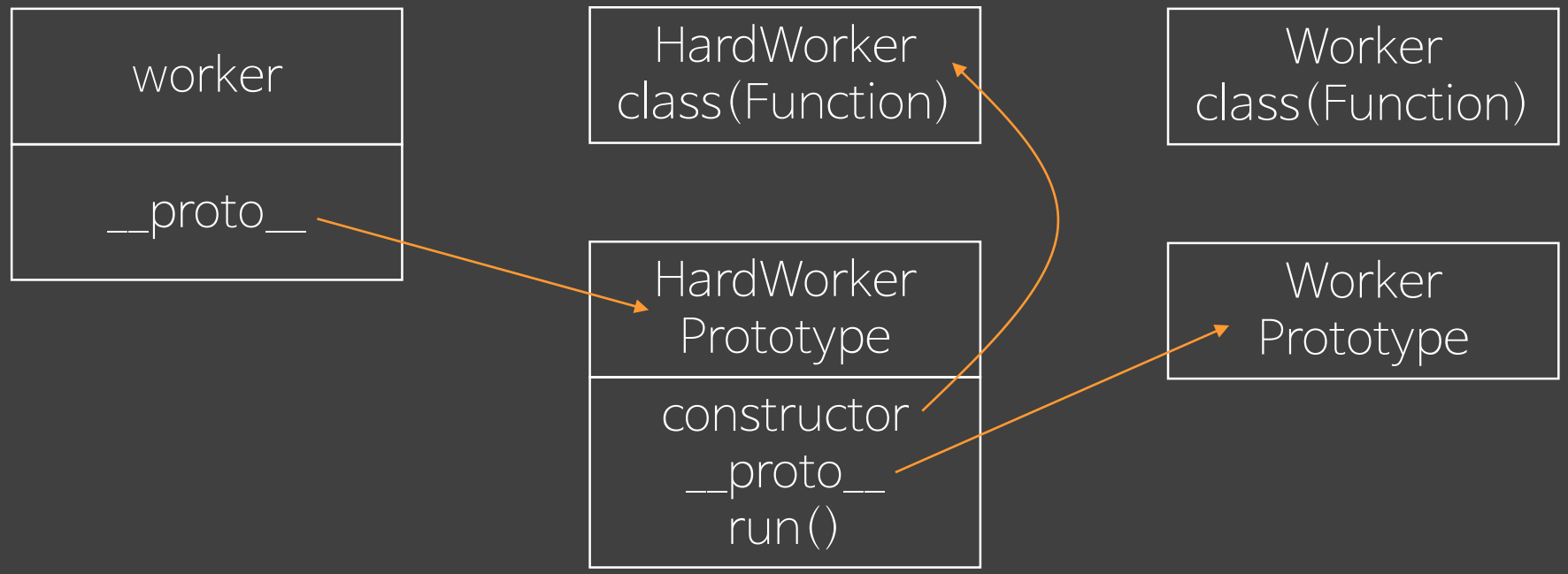
Polymorphism of Prototype



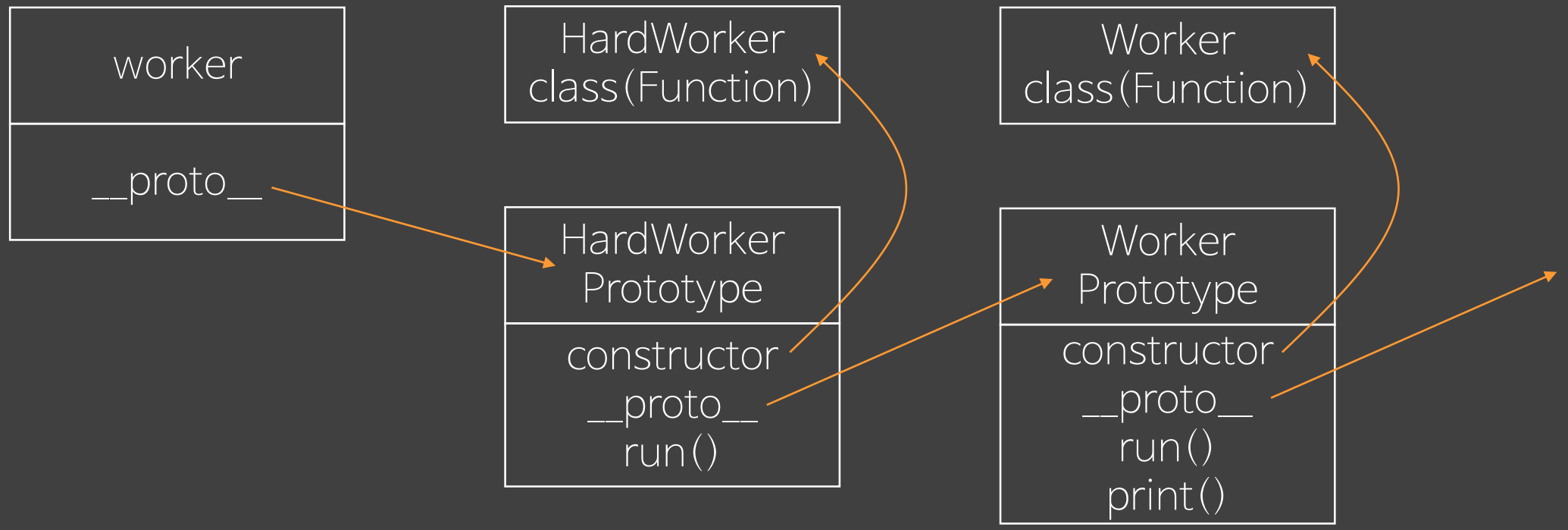
Polymorphism of Prototype



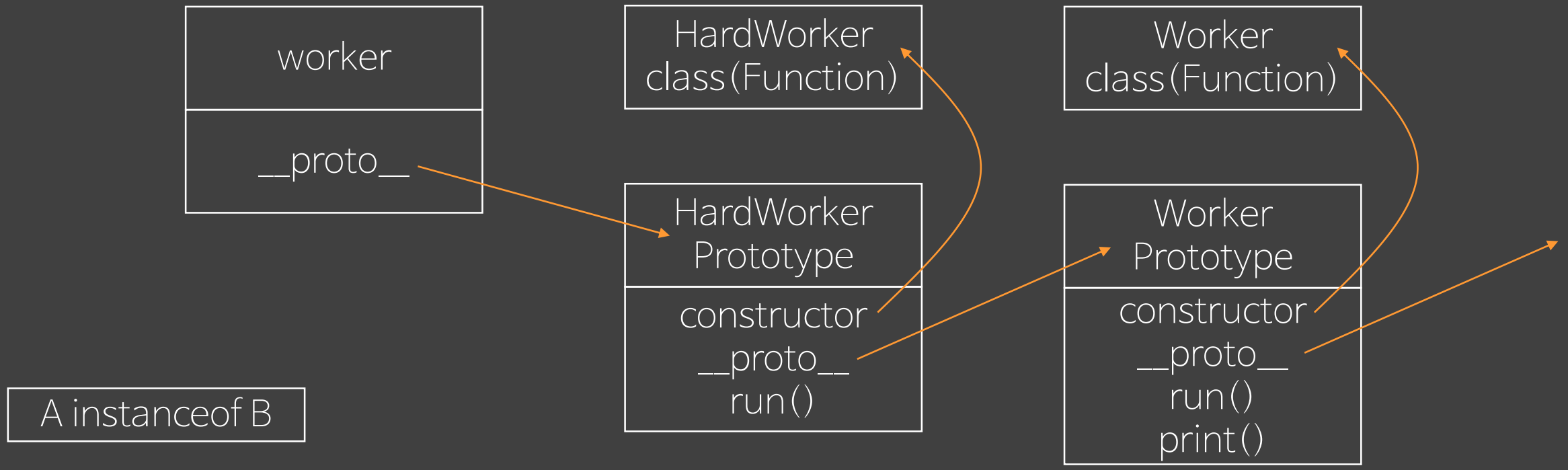
Polymorphism of Prototype



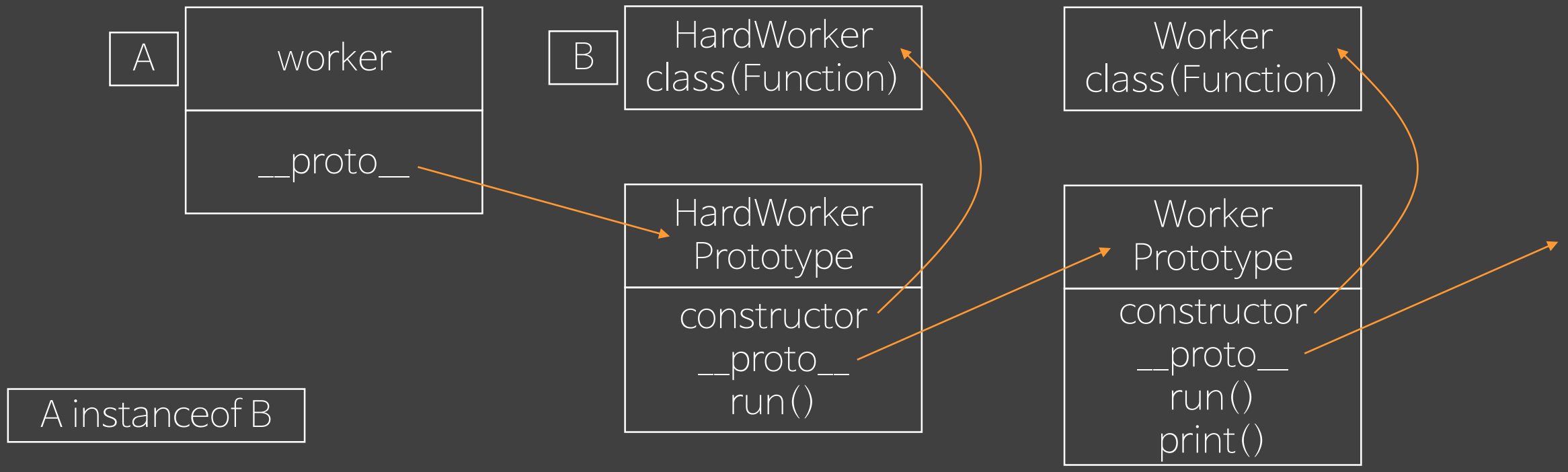
Polymorphism of Prototype



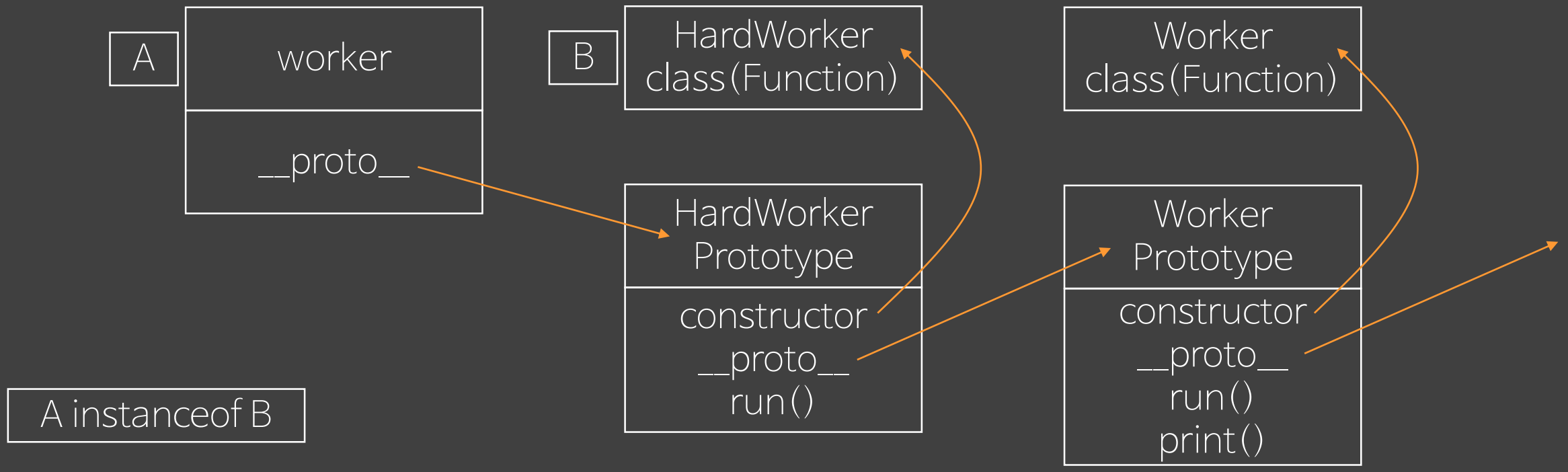
Polymorphism of Prototype



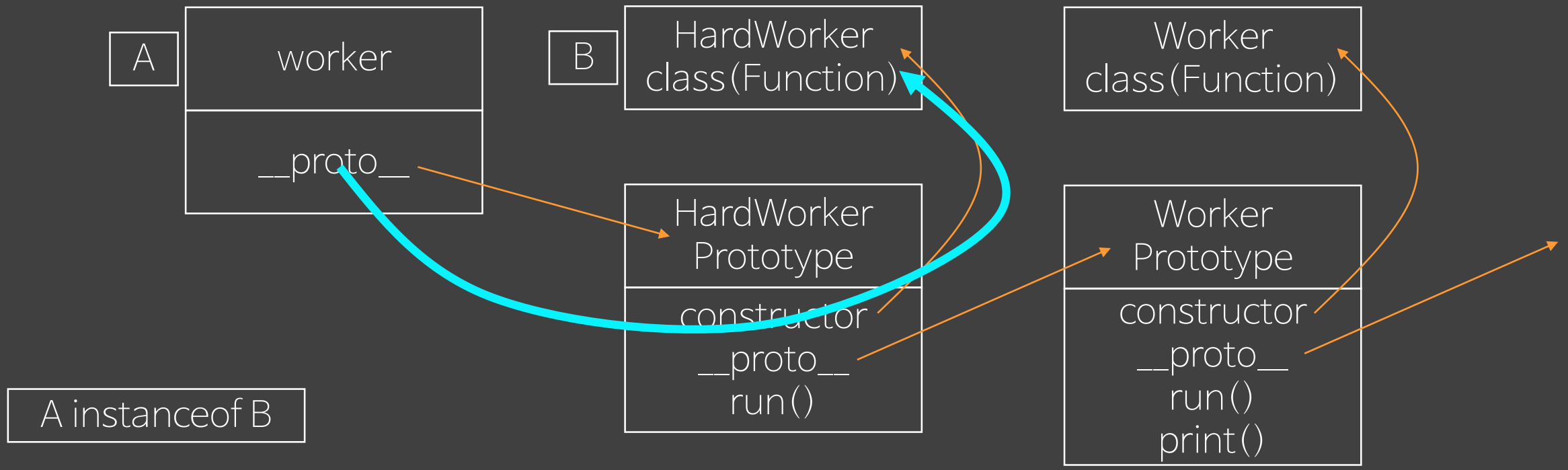
Polymorphism of Prototype



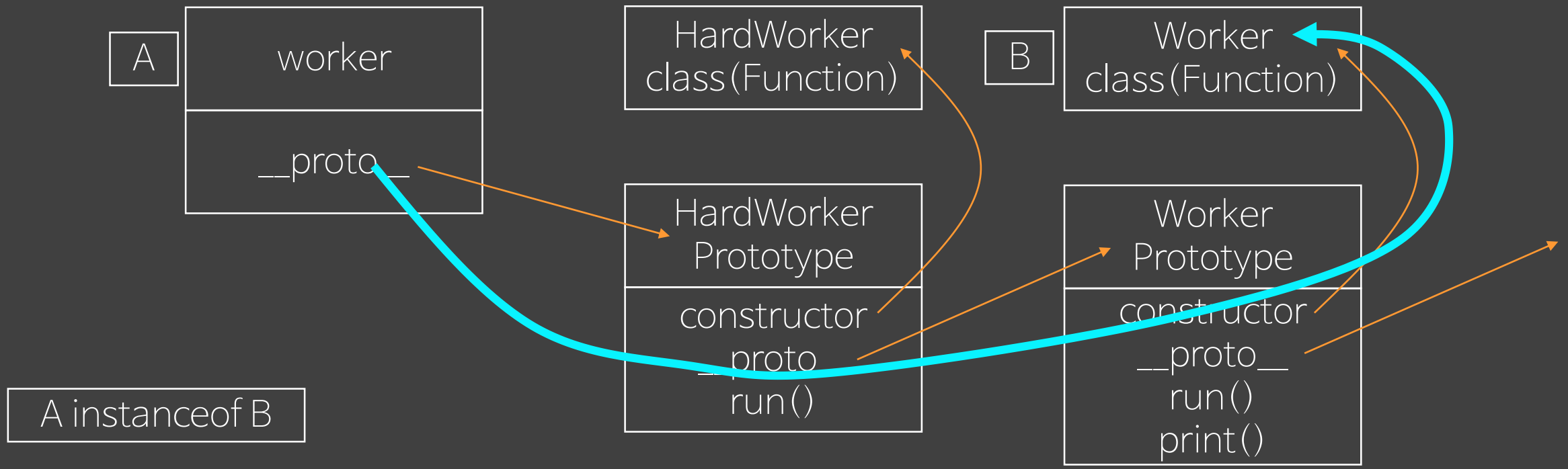
Polymorphism of Prototype



Polymorphism of Prototype



Polymorphism of Prototype



Object essentials

```
const EssentialObject = class{
    #name = "";
    #screen = null;
    constructor(name){
        this.#name = name;
    }
    camouflage(){
        this.#screen = (Math.random() * 10).toString(16).replace(".", "")
    }
    get name(){
        return this.#screen || this.#name;
    }
};
```

```
const EssentialObject = class{
    #name = "";
    #screen = null;
    constructor(name){
        this.#name = name;
    }
    camouflage(){
        this.#screen = (Math.random() * 10).toString(16).replace(".", "")
    }
    get name(){
        return this.#screen || this.#name;
    }
};
```

hide state

```
const EssentialObject = class{
    #name = "";
    #screen = null;
    constructor(name){
        this.#name = name;
    }
    camouflage(){
        this.#screen = (Math.random() * 10).toString(16).replace(".", "")
    }
    get name(){
        return this.#screen || this.#name;
    }
};
```

hide state

encapsulation

Object essentials

Encapsulation of Functionality
Maintenance of State

Object essentials

Encapsulation of Functionality
Maintenance of State

Isolation of change

알려진 기본 설계요령

SOLID원칙

SRP Single Responsibility 단일책임

SOLID원칙

SRP Single Responsibility 단일책임

산탄총 수술

shotgun surgery

SOLID원칙

SRP Single Responsibility 단일책임
OCP Open Closed 개방폐쇄

SOLID원칙

SRP Single Responsibility 단일책임
OCP Open Closed 개방폐쇄
LSP Liskov Substitusion 업캐스팅 안전

SOLID원칙

SRP Single Responsibility 단일책임
OCP Open Closed 개방폐쇄
LSP Liskov Substitusion 업캐스팅 안전

추상층의 정의가 너무 구체적이면 구상층의 구현에서 모순이 발생함.

추상층 - 생물
숨을 쉰다, 다리로 이동한다.

SOLID원칙

SRP Single Responsibility 단일책임
OCP Open Closed 개방폐쇄
LSP Liskov Substitusion 업캐스팅 안전

추상층의 정의가 너무 구체적이면 구상층의 구현에서 모순이 발생함.

추상층 - 생물
숨을 쉰다, 다리로 이동한다.

구상층

SOLID원칙

SRP Single Responsibility 단일책임
OCP Open Closed 개방폐쇄
LSP Liskov Substitusion 업캐스팅 안전

추상층의 정의가 너무 구체적이면 구상층의 구현에서 모순이 발생함.

추상층 - 생물
숨을 쉰다, 다리로 이동한다.

구상층
사람 ok!
타조 ok!

SOLID원칙

SRP Single Responsibility 단일책임
OCP Open Closed 개방폐쇄
LSP Liskov Substitusion 업캐스팅 안전

추상층의 정의가 너무 구체적이면 구상층의 구현에서 모순이 발생함.

추상층 - 생물
숨을 쉰다, 다리로 이동한다.

구상층
사람 ok!
타조 ok!
아메바 no!
독수리 no!
고래 no!

SOLID원칙

SRP Single Responsibility 단일책임
OCP Open Closed 개방폐쇄
LSP Liskov Substitusion 업캐스팅 안전

추상층의 정의가 너무 구체적이면 구상층의 구현에서 모순이 발생함.

추상층 - 생물
숨을 쉰다, ~~다리로 이동한다.~~

구상층
사람 ok!
타조 ok!
아메바 no!
독수리 no!
고래 no!

SOLID원칙

SRP Single Responsibility 단일책임
OCP Open Closed 개방폐쇄
LSP Liskov Substitusion 업캐스팅 안전

추상층의 정의가 너무 구체적이면 구상층의 구현에서 모순이 발생함.

추상층 - 생물(숨을 쉰다), 다리이동(다리 로이동한다)

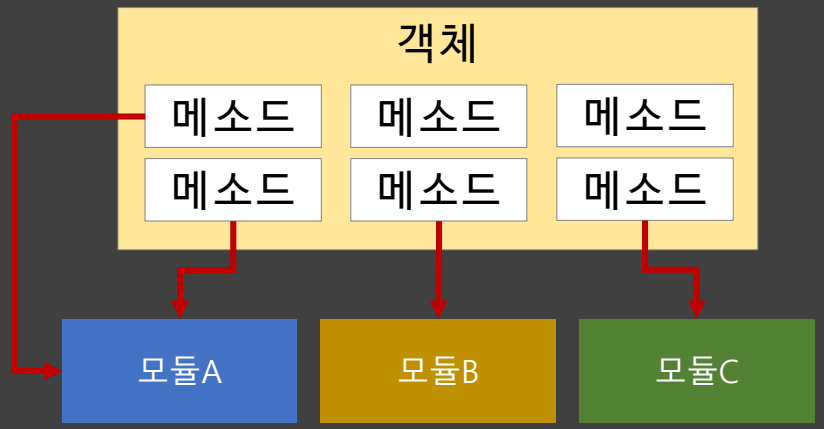
구상층
사람:생물,다리이동 ok!
타조:생물,다리이동 ok!
아메바:생물 ok!
독수리:생물 ok!
고래:생물 ok!

SOLID원칙

SRP Single Responsibility 단일책임
OCP Open Closed 개방폐쇄
LSP Liskov Substitusion 업캐스팅 안전
ISP Interface Segregation 인터페이스분리

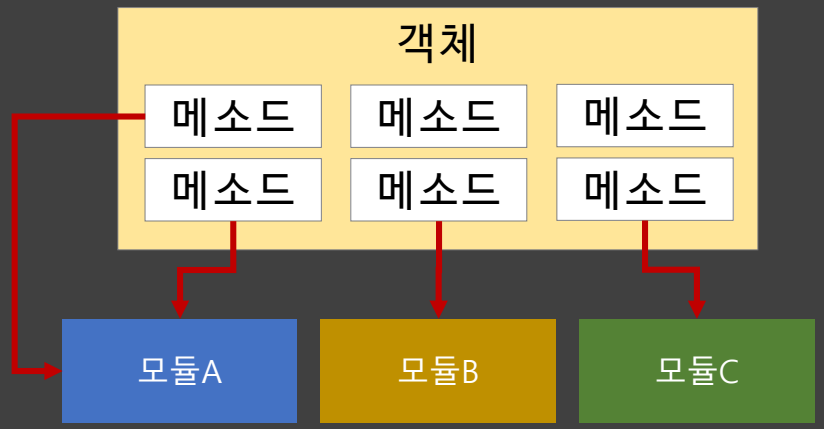
SOLID원칙

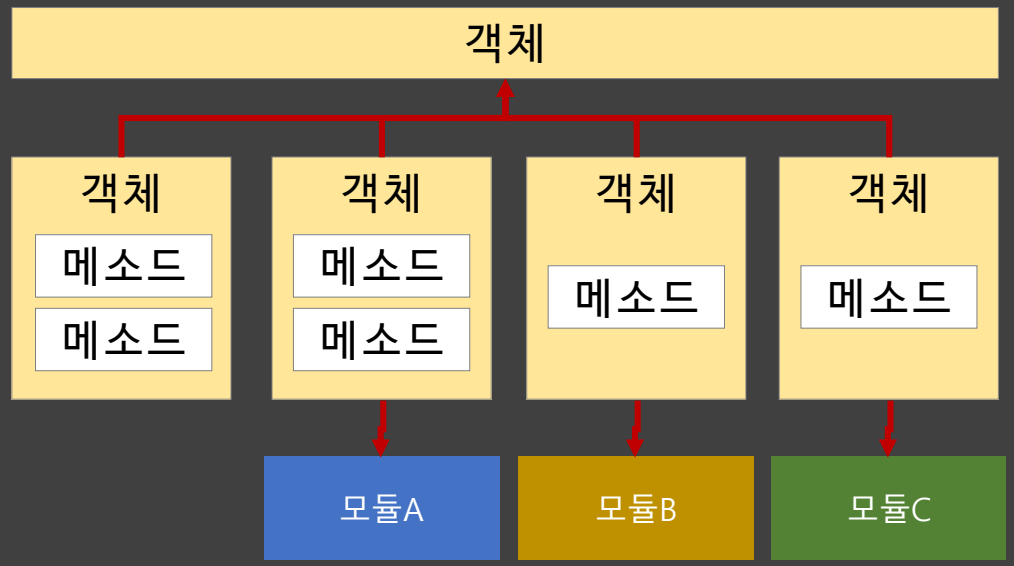
SRP Single Responsibility 단일책임
OCP Open Closed 개방폐쇄
LSP Liskov Substitusion 업캐스팅 안전
ISP Interface Segregation 인터페이스분리



SOLID원칙

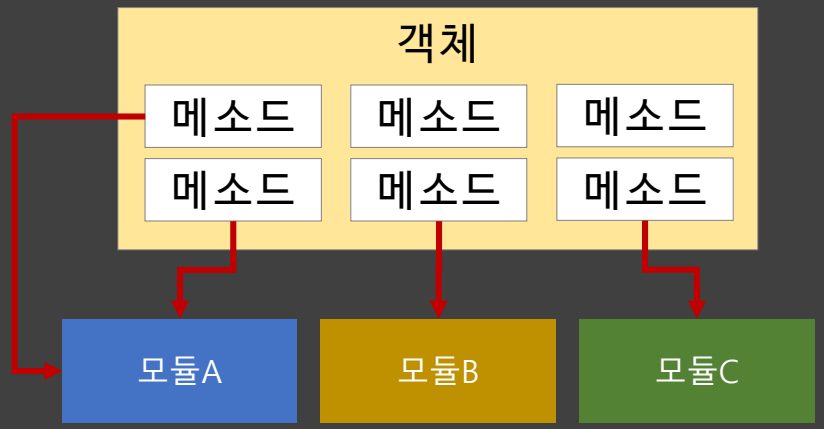
SRP Single Responsibility 단일책임
OCP Open Closed 개방폐쇄
LSP Liskov Substitusion 업캐스팅 안전
ISP Interface Segregation 인터페이스분리

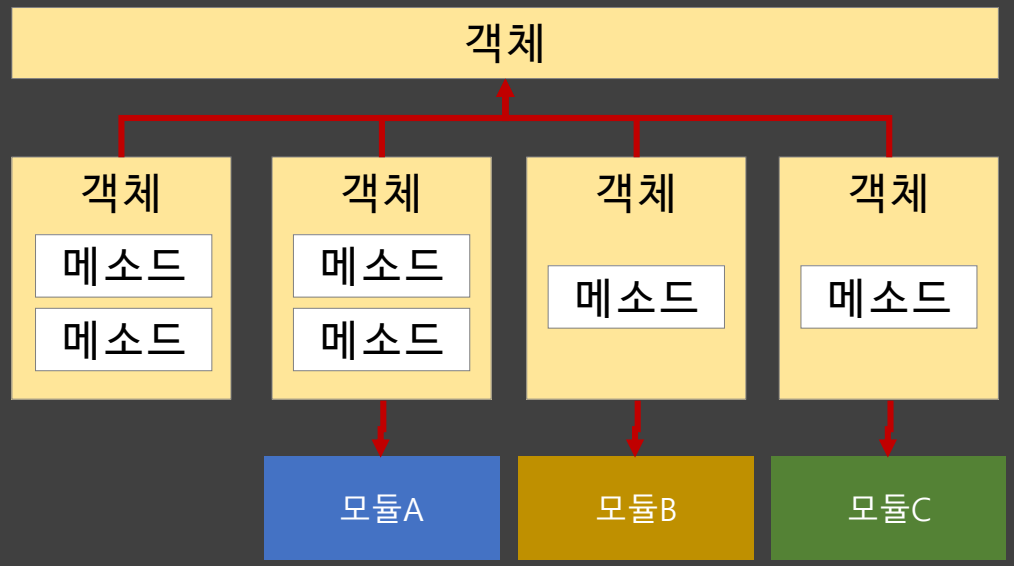




SOLID원칙

SRP Single Responsibility 단일책임
OCP Open Closed 개방폐쇄
LSP Liskov Substitusion 업캐스팅 안전
ISP Interface Segregation 인터페이스분리





SOLID원칙

SRP Single Responsibility 단일책임
OCP Open Closed 개방폐쇄
LSP Liskov Substitusion 업캐스팅 안전
ISP Interface Segregation 인터페이스분리
DIP Dependency Inversion 다운캐스팅금지

SOLID원칙

SRP Single Responsibility 단일책임
OCP Open Closed 개방폐쇄
LSP Liskov Substitusion 업캐스팅 안전
ISP Interface Segregation 인터페이스분리
DIP Dependency Inversion 다운캐스팅금지

고차원의 모듈은 저차원의 모듈에 의존하면 안된다.
이 두 모듈 모두 추상화된 것에 의존해야 한다.

SOLID원칙

SRP Single Responsibility 단일책임
OCP Open Closed 개방폐쇄
LSP Liskov Substitusion 업캐스팅 안전
ISP Interface Segregation 인터페이스분리
DIP Dependency Inversion 다운캐스팅금지

고차원의 모듈은 저차원의 모듈에 의존하면 안된다.
이 두 모듈 모두 추상화된 것에 의존해야 한다.

추상화 된 것은 구체적인 것에 의존하면 안 된다.
구체적인 것이 추상화된 것에 의존해야 한다.

SOLID원칙

SRP Single Responsibility 단일책임
OCP Open Closed 개방폐쇄
LSP Liskov Substitusion 업캐스팅 안전
ISP Interface Segregation 인터페이스분리
DIP Dependency Inversion 다운캐스팅금지

SOLID원칙

SRP Single Responsibility 단일책임
OCP Open Closed 개방폐쇄
LSP Liskov Substitusion 업캐스팅 안전
ISP Interface Segregation 인터페이스분리
DIP Dependency Inversion 다운캐스팅금지

DI Dependency Injection 의존성주입
(IoC Inversion of Control 제어역전)

SOLID원칙

SRP Single Responsibility 단일책임
OCP Open Closed 개방폐쇄
LSP Liskov Substitusion 업캐스팅 안전
ISP Interface Segregation 인터페이스분리
DIP Dependency Inversion 다운캐스팅금지

DI Dependency Injection 의존성주입
(IoC Inversion of Control 제어역전)

DRY Don't Repeat Yourself 중복방지

SOLID원칙

SRP Single Responsibility 단일책임
OCP Open Closed 개방폐쇄
LSP Liskov Substitusion 업캐스팅 안전
ISP Interface Segregation 인터페이스분리
DIP Dependency Inversion 다운캐스팅금지

DI Dependency Injection 의존성주입
(IoC Inversion of Control 제어역전)

DRY Don't Repeat Yourself 중복방지

Hollywood Principle 의존성 부패방지

SOLID원칙

SRP Single Responsibility 단일책임
OCP Open Closed 개방폐쇄
LSP Liskov Substitusion 업캐스팅 안전
ISP Interface Segregation 인터페이스분리
DIP Dependency Inversion 다운캐스팅금지

DI Dependency Injection 의존성주입
(IoC Inversion of Control 제어역전)

DRY Don't Repeat Yourself 중복방지

Hollywood Principle 의존성 부패방지

Law of demeter 최소 지식

SOLID원칙

SRP Single Responsibility 단일책임
OCP Open Closed 개방폐쇄
LSP Liskov Substitusion 업캐스팅 안전
ISP Interface Segregation 인터페이스분리
DIP Dependency Inversion 다운캐스팅금지

DI Dependency Injection 의존성주입
(IoC Inversion of Control 제어역전)

DRY Don't Repeat Yourself 중복방지

Hollywood Principle 의존성 부패방지

Law of demeter 최소 지식

classA.methodA의 최대지식한계
- classA의 필드 객체
- methodA가 생성한 객체
- methodA의 인자로 넘어온 객체

SOLID원칙

SRP Single Responsibility 단일책임
OCP Open Closed 개방폐쇄
LSP Liskov Substitusion 업캐스팅 안전
ISP Interface Segregation 인터페이스분리
DIP Dependency Inversion 다운캐스팅금지

DI Dependency Injection 의존성주입
(IoC Inversion of Control 제어역전)

DRY Don't Repeat Yourself 중복방지

Hollywood Principle 의존성 부패방지

Law of demeter 최소 지식

classA.methodA의 최대지식한계
- classA의 필드 객체
- methodA가 생성한 객체
- methodA의 인자로 넘어온 객체

열차전복
train wreck

Message

SRP를 준수하는 객체망이 문제를 해결

SRP를 준수하는 객체망이 문제를 해결

단일 책임 원칙을 준수하는 객체에게 책임 이상의 업무를 부여하면?

1. 만능 객체가 되려한다.
2. 다른 객체에게 의뢰한다.

다른 객체에게 의뢰하는 것 = 다른 객체에게 메세지를 보내는 것

1. 메세지 - 의뢰할 내용
2. 오퍼레이션 - 메세지를 수신할 객체가 제공하는 서비스
3. 메소드 - 오퍼레이션이 연결될 실제 처리기

SRP를 준수하는 객체망이 문제를 해결

단일 책임 원칙을 준수하는 객체에게 책임 이상의 업무를 부여하면?

1. 만능 객체가 되려한다.
2. 다른 객체에게 의뢰한다.

다른 객체에게 의뢰하는 것 = 다른 객체에게 메세지를 보내는 것

1. 메세지 - 의뢰할 내용
2. 오퍼레이션 - 메세지를 수신할 객체가 제공하는 서비스
3. 메소드 - 오퍼레이션이 연결될 실제 처리기

SRP를 준수하는 객체망이 문제를 해결

단일 책임 원칙을 준수하는 객체에게 책임 이상의 업무를 부여하면?

1. 만능 객체가 되려한다.
2. 다른 객체에게 의뢰한다.

다른 객체에게 의뢰하는 것 = 다른 객체에게 메세지를 보내는 것

1. 메세지 - 의뢰할 내용
2. 오퍼레이션 - 메세지를 수신할 객체가 제공하는 서비스
3. 메소드 - 오퍼레이션이 연결될 실제 처리기

SRP를 준수하는 객체망이 문제를 해결

단일 책임 원칙을 준수하는 객체에게 책임 이상의 업무를 부여하면?

1. 만능 객체가 되려한다.
2. 다른 객체에게 의뢰한다.

다른 객체에게 의뢰하는 것 = 다른 객체에게 메세지를 보내는 것

1. 메세지 - 의뢰할 내용
2. 오퍼레이션 - 메세지를 수신할 객체가 제공하는 서비스
3. 메소드 - 오퍼레이션이 연결될 실제 처리기

Dependency

의존성의 종류

객체의 생명 주기 전체에 걸친 의존성
- 상속(extends)
- 연관(association)

각 오퍼레이션 실행 시 임시적인 의존성
- 의존(dependency)

의존성의 종류

객체의 생명 주기 전체에 걸친 의존성
- 상속(extends)
- 연관(association)

각 오퍼레이션 실행 시 임시적인 의존성
- 의존(dependency)

의존성의 종류

객체의 생명 주기 전체에 걸친 의존성
- 상속(extends)
- 연관(association)

각 오퍼레이션 실행 시 임시적인 의존성
- 의존(dependency)

1. 수정 여파 규모증가
2. 수정하기 어려운 구조 생성
3. 순환 의존성

의존성의 종류

객체의 생명 주기 전체에 걸친 의존성
- 상속(extends)
- 연관(association)

각 오퍼레이션 실행 시 임시적인 의존성
- 의존(dependency)

1. 수정 여파 규모증가
2. 수정하기 어려운 구조 생성
3. 순환 의존성

의존성의 종류

객체의 생명 주기 전체에 걸친 의존성
- 상속(extends)
- 연관(association)

각 오퍼레이션 실행 시 임시적인 의존성
- 의존(dependency)

1. 수정 여파 규모증가
2. 수정하기 어려운 구조 생성
3. 순환 의존성

Dependency Inversion

어떠한 경우에도 다운캐스팅은 금지
폴리모피즘(추상인터페이스) 사용

```
const Worker = class{
  run(){
    console.log("working")
  }
  print(){
    this.run();
  }
};
const HardWorker = class extends Worker{
  run(){
    console.log("hardWorking")
  }
};
```

```
const Worker = class{
  run(){
    console.log("working")
  }
  print(){
    this.run();
  }
};
const HardWorker = class extends Worker{
  run(){
    console.log("hardWorking")
  }
};
```

```
const Manager = class{
  #workers;
  constructor(...workers) {
    if(workers.every(w=>w instanceof Worker)) this.#workers = workers;
    else throw "invalid workers";
  }
  doWork(){
    this.#workers.forEach(w=>w.run())
  }
};

const manager = new Manager(new Worker(), new HardWorker());
manager.doWork();
```

```
const Worker = class{
  run(){
    console.log("working")
  }
  print(){
    this.run();
  }
};
const HardWorker = class extends Worker{
  run(){
    console.log("hardWorking")
  }
};
```

```
const Manager = class{
  #workers;
  constructor(...workers) {
    if(workers.every(w=>w instanceof Worker)) this.#workers = workers;
    else throw "invalid workers";
  }
  doWork(){
    this.#workers.forEach(w=>w.run())
  }
};

const manager = new Manager(new Worker(), new HardWorker());
manager.doWork();
```

```
const Worker = class{
  run(){
    console.log("working")
  }
  print(){
    this.run();
  }
};
const HardWorker = class extends Worker{
  run(){
    console.log("hardWorking")
  }
};
```

```
const Manager = class{
  #workers;
  constructor(...workers) {
    if(workers.every(w=>w instanceof Worker)) this.#workers = workers;
    else throw "invalid workers";
  }
  doWork(){
    this.#workers.forEach(w=>w.run())
  }
};

const manager = new Manager(new Worker(), new HardWorker());
manager.doWork();
```

Inversion of Control

제어역전의 개념과 필요성

개념
1. Control = flow control(흐름제어)
2. 광의에서 흐름 제어 = 프로그램 실행 통제
3. 동기흐름제어, 비동기 흐름제어 등

제어역전의 개념과 필요성

개념
1. Control = flow control(흐름제어)
2. 광의에서 흐름 제어 = 프로그램 실행 통제
3. 동기흐름제어, 비동기 흐름제어 등

문제점
1. 흐름 제어는 상태와 결합하여 진행됨
2. 상태 통제와 흐름제어 = 알고리즘
3. 변화에 취약하고 구현하기도 어려움

제어역전의 개념과 필요성

개념
1. Control = flow control(흐름제어)
2. 광의에서 흐름 제어 = 프로그램 실행 통제
3. 동기흐름제어, 비동기 흐름제어 등

문제점
1. 흐름 제어는 상태와 결합하여 진행됨
2. 상태 통제와 흐름제어 = 알고리즘
3. 변화에 취약하고 구현하기도 어려움

대안
1. 제어를 추상화하고
2. 개별 제어의 차이점만 외부에서 주입받는다.

제어역전의 개념과 필요성

개념
1. Control = flow control(흐름제어)
2. 광의에서 흐름 제어 = 프로그램 실행 통제
3. 동기흐름제어, 비동기 흐름제어 등

문제점
1. 흐름 제어는 상태와 결합하여 진행됨
2. 상태 통제와 흐름제어 = 알고리즘
3. 변화에 취약하고 구현하기도 어려움

대안
1. 제어를 추상화하고
2. 개별 제어의 차이점만 외부에서 주입받는다.

제어역전의 개념과 필요성

개념
1. Control = flow control(흐름제어)
2. 광의에서 흐름 제어 = 프로그램 실행 통제
3. 동기흐름제어, 비동기 흐름제어 등

문제점
1. 흐름 제어는 상태와 결합하여 진행됨
2. 상태 통제와 흐름제어 = 알고리즘
3. 변화에 취약하고 구현하기도 어려움

대안
1. 제어를 추상화하고
2. 개별 제어의 차이점만 외부에서 주입받는다.

```
const Renderer = class{
  #view = null; #base = null;
  constructor(baseElement) {
    this.#base = baseElement;
  }
  set view(v){
    if(v instanceof View) this.#view = v;
    else throw `invalid view :${v}`;
  }
  render(data){
    const base = this.#base, view = this.#view;
    if(!base || !view) throw "no base or view";
    let target = base.firstElementChild;
    do base.removeChild(target); while(target = target.nextElementSibling);
    base.appendChild(view.getElement(data));
    view.initAni();
    view.startAni();
  }
};
```

```
const View = class{
  getElement(data){throw "override!"}
  initAni(){throw "override!"}
  startAni(){throw "override!"}
};
```

```
const Renderer = class{
  #view = null; #base = null;
  constructor(baseElement) {
    this.#base = baseElement;
  }
  set view(v){
    if(v instanceof View) this.#view = v;
    else throw `invalid view :${v}`;
  }
  render(data){
    const base = this.#base, view = this.#view;
    if(!base || !view) throw "no base or view";
    let target = base.firstElementChild;
    do base.removeChild(target); while(target = target.nextElementSibling);
    base.appendChild(view.getElement(data));
    view.initAni();
    view.startAni();
  }
};
```

```
const View = class{
  getElement(data){throw "override!"}
  initAni(){throw "override!"}
  startAni(){throw "override!"}
};
```

```
const Renderer = class{
  #view = null; #base = null;
  constructor(baseElement) {
    this.#base = baseElement;
  }
  set view(v){
    if(v instanceof View) this.#view = v;
    else throw `invalid view :${v}`;
  }
  render(data){
    const base = this.#base, view = this.#view;
    if(!base || !view) throw "no base or view";
    let target = base.firstElementChild;
    do base.removeChild(target); while(target = target.nextElementSibling);
    base.appendChild(view.getElement(data));
    view.initAni();
    view.startAni();
  }
};
```

```
const View = class{
  getElement(data){throw "override!"}
  initAni(){throw "override!"}
  startAni(){throw "override!"}
};
```

```
const Renderer = class{
  #view = null; #base = null;
  constructor(baseElement) {
    this.#base = baseElement;
  }
  set view(v){
    if(v instanceof View) this.#view = v;
    else throw `invalid view :${v}`;
  }
  render(data){
    const base = this.#base, view = this.#view;
    if(!base || !view) throw "no base or view";
    let target = base.firstElementChild;
    do base.removeChild(target); while(target = target.nextElementSibling);
    base.appendChild(view.getElement(data));
    view.initAni();
    view.startAni();
  }
};
```

```
const View = class{
  getElement(data){throw "override!"}
  initAni(){throw "override!"}
  startAni(){throw "override!"}
};
```

```
const Renderer = class{
  #view = null; #base = null;
  constructor(baseElement) {
    this.#base = baseElement;
  }
  set view(v){
    if(v instanceof View) this.#view = v;
    else throw `invalid view :${v}`;
  }
  render(data){
    const base = this.#base, view = this.#view;
    if(!base || !view) throw "no base or view";
    let target = base.firstElementChild;
    do base.removeChild(target); while(target = target.nextElementSibling);
    base.appendChild(view.getElement(data));
    view.initAni();
    view.startAni();
  }
};
```

```
const View = class{
  getElement(data){throw "override!"}
  initAni(){throw "override!"}
  startAni(){throw "override!"}
};
```

```
const Renderer = class{
  #view = null; #base = null;
  constructor(baseElement) {
    this.#base = baseElement;
  }
  set view(v){
    if(v instanceof View) this.#view = v;
    else throw `invalid view :${v}`;
  }
  render(data){
    const base = this.#base, view = this.#view;
    if(!base || !view) throw "no base or view";
    let target = base.firstElementChild;
    do base.removeChild(target); while(target = target.nextElementSibling);
    base.appendChild(view.getElement(data));
    view.initAni();
    view.startAni();
  }
};
```

```
const View = class{
  getElement(data){throw "override!"}
  initAni(){throw "override!"}
  startAni(){throw "override!"}
};
```

```
const renderer = new Renderer(document.body);
renderer.view = new class extends View{
  #el;
  getElement(data){
    this.#el = document.createElement("div");
    this.#el.innerHTML = `<h2>${data.title}</h2><p>${data.description}</p>`;
    this.#el.style.cssText = "`width:100%;background:${data.background}`;
    return this.#el;
  }
  initAni() {
    const style = this.#el.style;
    style.marginLeft = "100%";
    style.transition = "all 0.3s";
  }
  startAni() {
    requestAnimationFrame(_=>this.#el.style.marginLeft = 0)
  }
};
```

```
const View = class{
  getElement(data){throw "override!"}
  initAni(){throw "override!"}
  startAni(){throw "override!"}
};
```

```
const renderer = new Renderer(document.body);
renderer.view = new class extends View{
  #el;
  getElement(data){
    this.#el = document.createElement("div");
    this.#el.innerHTML = `<h2>${data.title}</h2><p>${data.description}</p>`;
    this.#el.style.cssText = "`width:100%;background:${data.background}`;
    return this.#el;
  }
  initAni() {
    const style = this.#el.style;
    style.marginLeft = "100%";
    style.transition = "all 0.3s";
  }
  startAni() {
    requestAnimationFrame(_=>this.#el.style.marginLeft = 0)
  }
};
```

```
const View = class{
  getElement(data){throw "override!"}
  initAni(){throw "override!"}
  startAni(){throw "override!"}
};
```

```
const renderer = new Renderer(document.body);
renderer.view = new class extends View{
  #el;
  getElement(data){
    this.#el = document.createElement("div");
    this.#el.innerHTML = `<h2>${data.title}</h2><p>${data.description}</p>`;
    this.#el.style.cssText = "`width:100%;background:${data.background}`;
    return this.#el;
  }
  initAni() {
    const style = this.#el.style;
    style.marginLeft = "100%";
    style.transition = "all 0.3s";
  }
  startAni() {
    requestAnimationFrame(_=>this.#el.style.marginLeft = 0)
  }
};
```

```
const View = class{
  getElement(data){throw "override!"}
  initAni(){throw "override!"}
  startAni(){throw "override!"}
};
```

```
const renderer = new Renderer(document.body);
renderer.view = new class extends View{
  #el;
  getElement(data){
    this.#el = document.createElement("div");
    this.#el.innerHTML = `<h2>${data.title}</h2><p>${data.description}</p>`;
    this.#el.style.cssText = "`width:100%;background:${data.background}`;
    return this.#el;
  }
  initAni() {
    const style = this.#el.style;
    style.marginLeft = "100%";
    style.transition = "all 0.3s";
  }
  startAni() {
    requestAnimationFrame(_=>this.#el.style.marginLeft = 0)
  }
};
```

```
const View = class{
  getElement(data){throw "override!"}
  initAni(){throw "override!"}
  startAni(){throw "override!"}
};
```

```
const renderer = new Renderer(document.body);
renderer.view = new class extends View{
  #el;
  getElement(data){
    this.#el = document.createElement("div");
    this.#el.innerHTML = `<h2>${data.title}</h2><p>${data.description}</p>`;
    this.#el.style.cssText = "`width:100%;background:${data.background}`;
    return this.#el;
  }
  initAni() {
    const style = this.#el.style;
    style.marginLeft = "100%";
    style.transition = "all 0.3s";
  }
  startAni() {
    requestAnimationFrame(_=>this.#el.style.marginLeft = 0)
  }
};
```

```
const View = class{
  getElement(data){throw "override!"}
  initAni(){throw "override!"}
  startAni(){throw "override!"}
};
```

```
const renderer = new Renderer(document.body);
renderer.view = new class extends View{
  #el;
  getElement(data){
    this.#el = document.createElement("div");
    this.#el.innerHTML = `<h2>${data.title}</h2><p>${data.description}</p>`;
    this.#el.style.cssText = "`width:100%;background:${data.background}`;
    return this.#el;
  }
  initAni() {
    const style = this.#el.style;
    style.marginLeft = "100%";
    style.transition = "all 0.3s";
  }
  startAni() {
    requestAnimationFrame(_=>this.#el.style.marginLeft = 0)
  }
};
```

```
renderer.render({title:"title test", description:"contents.......", background:"#ffffaa"});
```

제어역전 실제 구현

전략 패턴 & 템플릿 메소드 패턴 < 컴포지트 패턴 < 비지터 패턴
보다 넓은 범위의 제어 역전을 실현함

제어역전 실제 구현

전략 패턴 & 템플릿 메소드 패턴 < 컴포지트 패턴 < 비지터 패턴
보다 넓은 범위의 제어 역전을 실현함

추상팩토리메소드 패턴
왼쪽 패턴은 이미 만들어진 객체의 행위를 제어역전에 참여시킬 수 있지만 참여할 객체 자체를 생성할 수 없음.
참여할 객체를 상황에 맞게 생성하고 행위까지 위임하기 위해 추상팩토리 메소드를 사용함