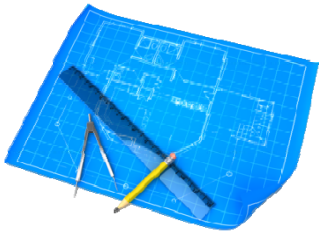
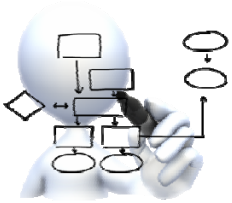


Decorator 패턴

NOTE 05



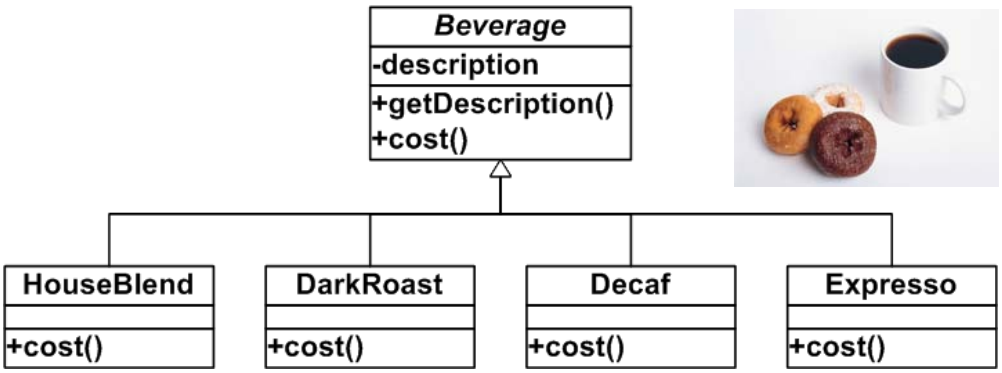
한국기술교육대학교 컴퓨터공학부 김상진

sangjin@koreatech.ac.kr
www.facebook.com/sangjin.kim.koreatech

교육목표

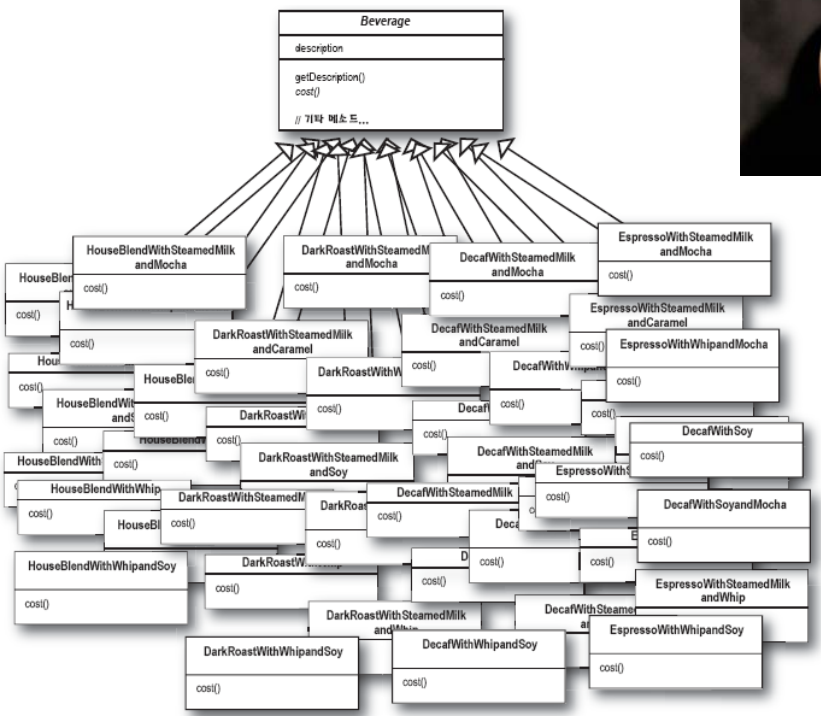
- 객체에 동적으로 새로운 행위를 추가할 수 있도록 해주는 장식 패턴 (decorator pattern) 익히기
  - 예) 케이크 장식: 케이크를 과일로 장식하여도 여전히 케이크임

Starbuzz Coffee App. (1/2)

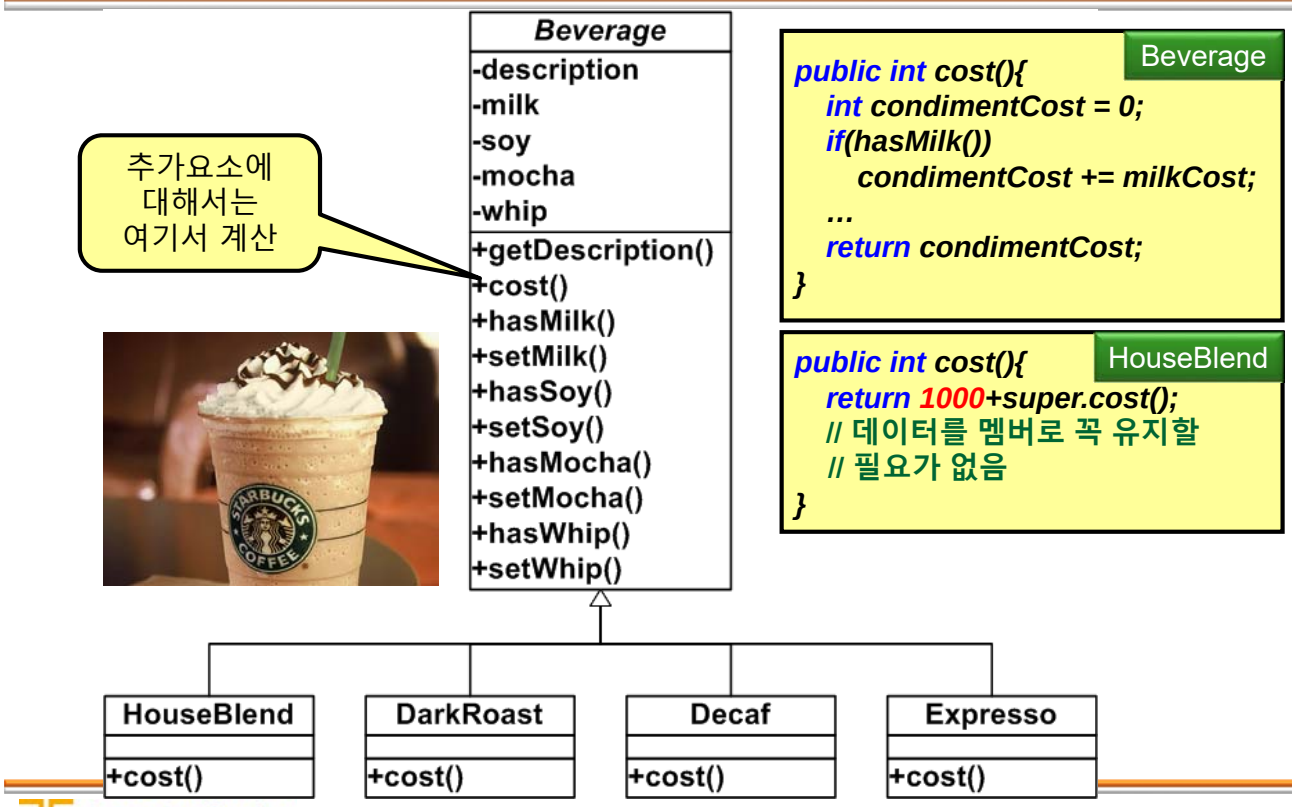


- 새로운 주문시스템을 만들고 싶음
  - 가격 계산이 주 목적
- 커피 주문시 추가 요구사항(steamed milk, 모카, 두유 등)이 있을 수 있음

Starbuzz Coffee App. (2/2)



Starbuzz Coffee App. 해결책 1. (1/2)



Starbuzz Coffee App. 해결책 1. (2/2)

- 첫 번째 해결책의 문제점
  - 추가되는 것의 가격이 변동하면 기존 코드를 변경해야 함
  - 새로운 추가요소가 생기면 새 메소드를 추가해야 하며, cost 메소드를 수정해야 함
  - 새로운 음료가 추가될 수 있고, 기존 추가요소들이 새 음료에 적합하지 않을 수 있음
    - 상속을 통해 행위를 상속받을 수 있지만 상속되는 행위는 컴파일시간에 고정되며, 모든 자식 클래스는 동일한 행위를 상속받아야 함
    - 반면에 연관 관계를 이용하면 객체에 행위를 실행시간에 추가할 수 있음
  - 고객이 더블모카를 원하면?

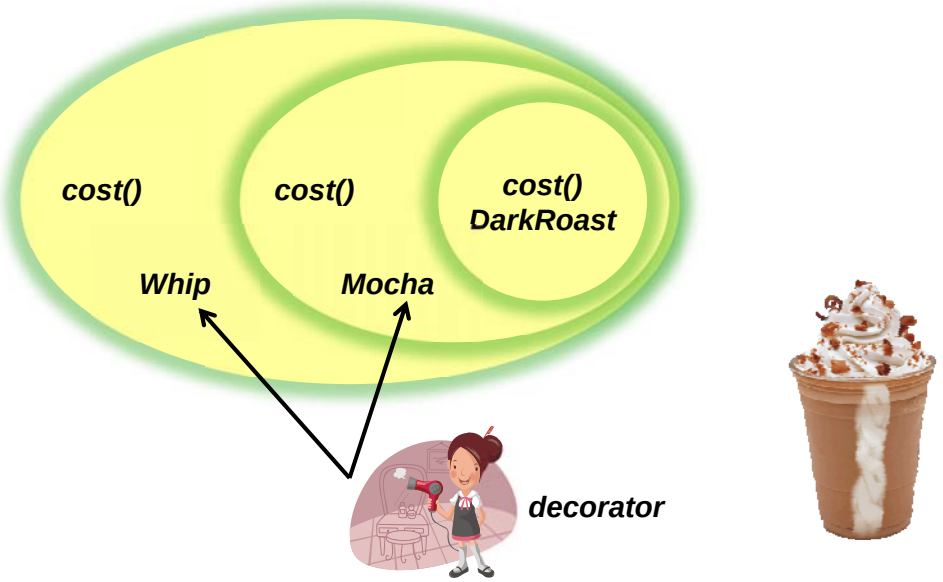
Design Principle 5 (Open-Closed Principle)
Classes should be open for extension, but closed for modification

질문

- 확장에 대해서는 개방적, 수정에 대해서는 폐쇄적? 어떻게
  - 모순되는 측면은 있지만 observer 패턴처럼 이것을 가능케 하는 설계 기술이 있음
- 일반적으로 open-closed 원리에 충실하게 설계할 수 있나?
  - 많은 패턴이 이것을 가능케 해줌. Decorator 패턴도 좋은 예임
- 하지만 모든 요소로 open-closed 원리에 충실하도록 설계할 수 없음. 따라서 설계 내용 중 변화 가능성이 많은 부분에 집중하여 이 원리가 적용되도록 해야 함

Decorator 패턴 (1/4)

- 예) 고객이 원하는 것이 "DarkRoast with Mocha and Whip"이면
  - DarkRoast 객체를 Mocha 객체로 장식하고
  - 장식된 객체를 다시 Whip 객체로 장식함



Decorator 패턴 (2/4)

- Decorator는 장식하는 원래 객체와 동일한 타입의 객체임
  - 장식된 객체를 원래 객체 대신에 사용 가능함
    - 이를 위해 기존 객체를 상속받아 정의하지만 행위의 추가는 상속을 통해 이루어지는 것은 아님 (has-a 활용)
- 한 객체를 여러 개의 decorator로 장식할 수 있음
- Decorator는 새 행위를 추가할 경우 다음과 같이 함

```
f(){
        add new computation;
        super.f();
}
```

```
f(){
        super.f();
        add new computation;
}
```

- 동적으로 실행시간에 객체를 장식할 수 있음
- 보통 장식된 순서가 중요하지 않음

Decorator 패턴 (3/4)

- Decorator 패턴: 객체에 동적으로 새로운 책임(행위, 상태)을 추가할 수 있음
  - 다른 말로: Wrapper
  - 하지만 새로운 책임보다는 기존 책임의 내용을 보완하는 경우에 많이 사용됨
  - 새로운 행위는 보통 내부적(private)으로만 사용됨
- 위임을 통한 문제 해결
  - 자바 Observable 클래스의 문제를 해결하기 위해 사용한 것과 동일하게 연관 관계를 이용하여 장식하고 있음

Decorator 패턴 (4/4)

StarBuzz와 Decorator

```
public abstract class Beverage {
    private String description;
    public Beverage(){
        description = "Unknown Beverage";
    }
    public Beverage(String d){
        description = d;
    }
    public String getDescription(){
        return description;
    }
    public abstract int cost();
}
```

Decorator들을 대표하는 추상 클래스를 정의하여 사용하면 얻어지는 이점

1) 장식자와 장식될 수 있는 타입의 구분이 명확해짐
2) 장식자가 반드시 정의하여야 하는 메소드를 지정할 수 있음

```
public abstract class CondimentDecorator extends Beverage {
    public abstract String getDescription();
}
```



```
public class DarkRoast extends Beverage {
    public DarkRoast(){
        super("Dark Roast Coffee");
    }
    @Override
    public int cost() {
        return 2100;
    }
}
```

```
public class Mocha extends CondimentDecorator {
    private Beverage beverage;
    public Mocha(Beverage b){
        beverage = b;
    }
    @Override
    public String getDescription() {
        return beverage.getDescription() + ", Mocha";
    }
    @Override
    public int cost() {
        return 200 + beverage.cost();
    }
}
```

```
public class StarBuzzCoffeeTest {
    public static void main(String[] args) {
        Beverage beverage = new DarkRoast();
        beverage = new Mocha(beverage);
        beverage = new Mocha(beverage);
        beverage = new Whip(beverage);
        System.out.printf("%s: %,d 원%n",
            beverage.getDescription(), beverage.cost());
    }
}
```

같은 문제를 전략 패턴으로

- 추가요소가 하나 밖에 없으면 전략패턴으로 쉽게 구현가능

```
public int cost() {
    return 2100+condiment.cost();
}
```

- StarBuzz처럼(일반적인 decorator 패턴의 예) 여러 개의 요소를 추가할 수 있고, 동일한 요소를 여러 번 적용 가능해야 하면 전략패턴으로 구현은 힘듦
- 또 전략패턴은 특정 메소드의 기능만 변경하지만 장식자 패턴은 여러 메소드의 기능을 변경할 수 있음. 정말?

```
public abstract class Beverage {
    private ArrayList<Condiment> condiments
        = new ArrayList<>();
    public void addCondiment(Condiment condiment){
        condiments.add(condiment);
    }
    public int cost(){
        int total = 0;
        for(Condiment c: condiments) total += c.cost();
        return total;
    }
}
```

또 다른 대안

질문

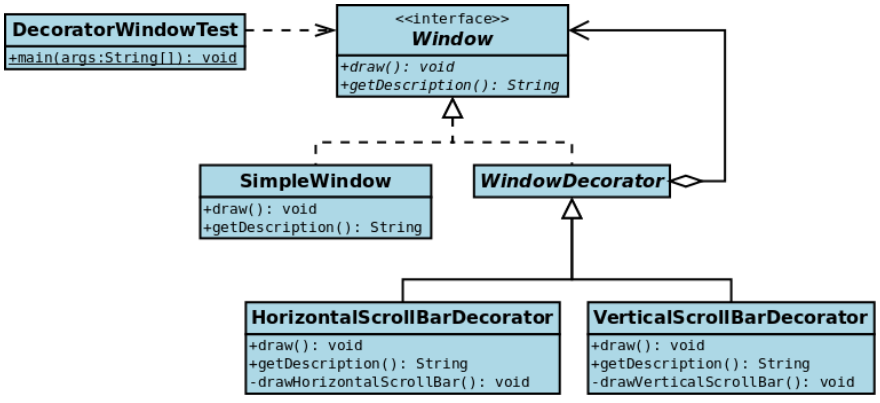
- 문제점
  - Concrete class에 구현된 코드는 decorator에 의해 제대로 동작하지 않을 수 있음
    - HouseBlend에 특별 할인되도록 구현하였을 경우 decorator를 사용하면 제대로 동작하지 않을 수 있음
  - 여러 겹으로 장식된 경우, 중간 겹에 대한 접근이 가능할 수 있음
    - 생성 패턴을 통해 극복 가능
  - 여러 겹으로 장식된 경우, 가장 바깥쪽 decorator가 모든 안쪽 decorator를 알 수 있나?
    - 알 수 있도록 만들 수 있지만 캡슐화 측면에서는 별로

Decorator 패턴의 예 (1/2)

- 자바 입출력

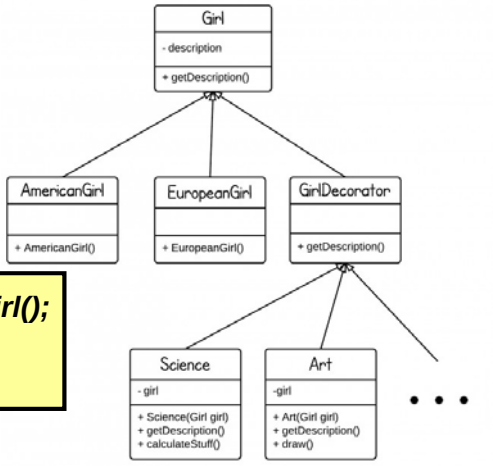
```
Reader reader = new FileReader("data.txt");
reader = new BufferedReader(reader);
```

- GUI



Decorator 패턴의 예 (2/2)

- 커플 매칭 서비스에서 여성의 모델링
  - 다양한 취미, 성격 모델링
  - 이 모델의 문제점?

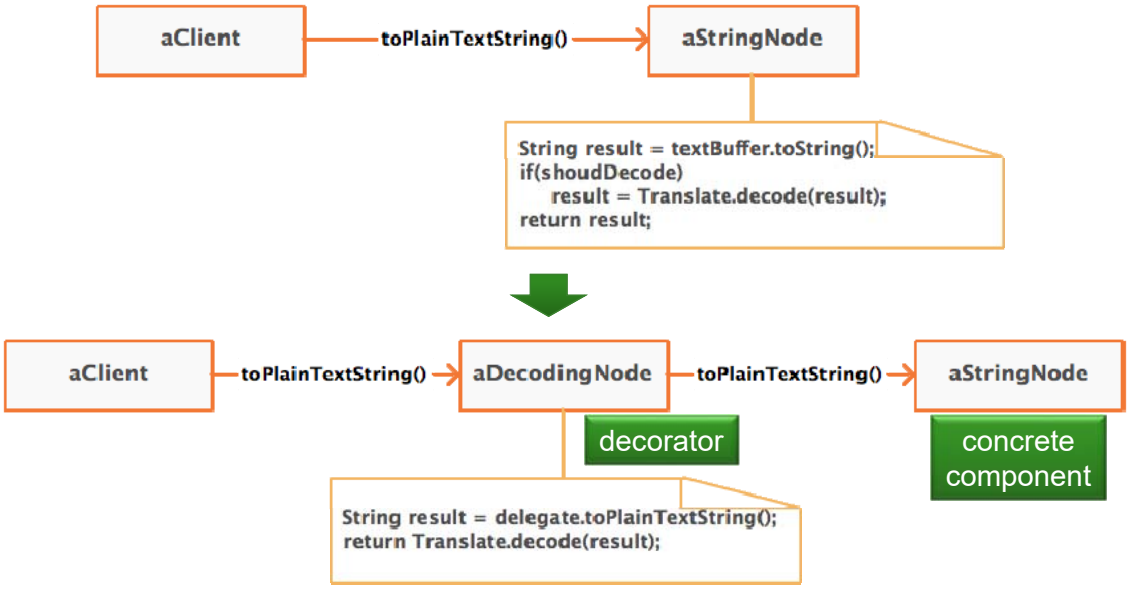



```
Girl g = new AmericanGirl();
g = new Science(g);
g = new Art(g);
```

http://www.programcreek.com/2012/05/java-design-pattern-decorator-decorate-your-girlfriend/

관련 리펙토링

- Move Embellishment(핵심기능이 아닌 것) to Decorator



Decorator 패턴 (1/2)

- 패턴의 특성: Structural
- 패턴의 수준: Component
- Applicability
  - 상속에 의한 제약사항에 구속 받지 않고 동적으로 객체에 기능 또는 상태를 추가하고 싶을 경우
  - 시스템이 동작하면서 객체에 기능을 추가하고 제거하고 싶을 경우
  - 동적으로 객체에 추가할 수 있는 서로 독립적인 다양한 기능들이 존재할 때
- 장단점
  - 코딩이 단순해지며, 클래스의 응집력이 높아질 수 있음
  - 상속보다 유연하게 객체에 기능 또는 상태를 추가할 수 있으며, 나중에 제거도 가능함. 또 어떤 책임을 두 번 적용하기도 쉬움
    - 여기서 제거란 빈 메소드 또는 예외를 발생하도록 decorator를 만드는 것을 말함

Decorator 패턴 (2/2)

- 클래스가 비교적 많이 정의될 수 있고, 디버깅이 힘들어질 수 있음
  - 참고. 실제는 클래스 수를 줄이기 위해 이 패턴을 사용하였음
- 관련 클래스들에 중복되어 있는 장식 요소를 한 곳에 정의할 수 있음
- 구현 고려사항
  - 최상위 component 클래스가 lightweight이어야 함. 그래야 그것을 상속해야 하는 나머지 클래스들도 너무 무거워지지 않을 수 있음
  - 하나의 책임만 추가해야 되면 꼭 abstract decorator를 정의할 필요가 없음
  - Equality testing: 가장 바깥쪽만 비교할 수 없음
  - Removing Layer: forward/backward reference

```
class Beverage {
private: string description;
public:
   Beverage():
      description("Unknown Beverage"){}
   Beverage(const string& description):
      description(move(description)){}
   virtual ~Beverage(){}
   virtual string getDescription(){
      return description; }
   virtual int cost() = 0;
};
class DarkRoast: public Beverage {
public:
   DarkRoast(): Beverage("Dark Roast"){}
   virtual ~DarkRoast(){}
   int cost() override { return 2000; }
};
```

```
unique_ptr<Beverage> beverage
   = make_unique<DarkRoast>();
beverage
   = make_unique<Mocha>(beverage);
beverage
   = make_unique<Mocha>(beverage);
cout << beverage->getDescription()
   << ":" << beverage->cost() << endl;
```

```
class CondimentDecorator:
   public Beverage {
public:
   CondimentDecorator(){}
   virtual ~CondimentDecorator(){}
   virtual string getDescription()
      override = 0;
};

class Mocha: public CondimentDecorator {
private:
   unique_ptr<Beverage> beverage;
public:
   Mocha(unique_ptr<Beverage>& beverage):
      beverage(move(beverage)){}
   virtual ~Mocha(){}
   string getDescription() override {
      return beverage->getDescription()+
         ", Mocha";
   }
   int cost() override {
      return 200 + beverage->cost();
   }
};
```



```
class Beverage(object):
   __metaclass__ = ABCMeta

   def __init__(self,
      description="Unknown Beverage"):
      self.description = description
   def getDescription(self):
      return self.description
   @abstractmethod
   def cost(self):
      pass

class CondimentDecorator(Beverage):
   __metaclass__ = ABCMeta

   @abstractmethod
   def getDescription(self):
      pass

class DarkRoast(Beverage):
   def __init__(self):
      Beverage.__init__(self,
      "Dark Roast Coffee")
   def cost(self):
      return 2100;
```

```
class Mocha(CondimentDecorator):
   def __init__(self, beverage):
      if not isinstance(beverage, Beverage):
         raise TypeError("must use Beverage")
      self.beverage = beverage
   def getDescription(self):
      return self.beverage.getDescription()
         +", Mocha"
   def cost(self):
      return self.beverage.cost()+200

beverage = DarkRoast()
beverage = Mocha(beverage)
beverage = Mocha(beverage)
print beverage.getDescription()
print beverage.cost()
```

```
var inherits = function(child, parent) {
    child.prototype
        = Object.create(parent.prototype);
    child.prototype.constructor = child;
};

var Beverage = function(){
    this.description = "Unknown Beverage";
};
Beverage.prototype.cost = function(){};
Beverage.prototype.getDescription
    = function(){
        return this.description;
};
var CondimentDecorator = function(){
    Beverage.call(this);
};
inherits(CondimentDecorator, Beverage);

var DarkRoast = function(){
    Beverage.call(this);
    this.description = "Dark Roast Coffee";
};
inherits(DarkRoast, Beverage);
DarkRoast.prototype.cost = function(){
    return 2100;
};
```

```
var Mocha = function(beverage){
    CondimentDecorator.call(this);
    this.beverage = beverage;
};
inherits(Mocha, CondimentDecorator);
Mocha.prototype.cost = function(){
    return this.beverage.cost()+200;
};
Mocha.prototype.getDescription = function(){
    return this.beverage.getDescription()+
        ", Mocha";
};

var beverage = new DarkRoast();
beverage = new Mocha(beverage);
beverage = new Mocha(beverage);
console.log(beverage.getDescription());
console.log(beverage.cost());
```