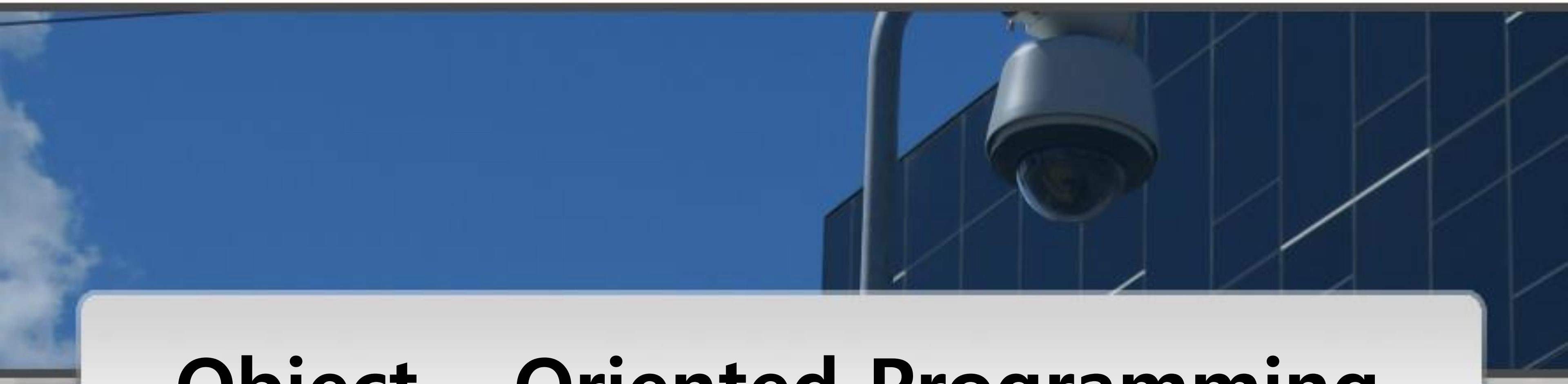


Object – Oriented Programming

Lab #8

Contents

- Polymorphism
  - Binding
  - Casting
- Abstract Classes

Introduction to Polymorphism

- There are three main programming mechanisms that constitute Object Oriented Programming (OOP)
  - Encapsulation: *{ combining data and actions into a single unit (Class) }*
  - Inheritance: *{ deriving information and functionality from base or super class}*
  - Polymorphism
    - What is Polymorphism?

- Comes from 2 Greek words
  - poly (many)
  - morph (forms, shapes)

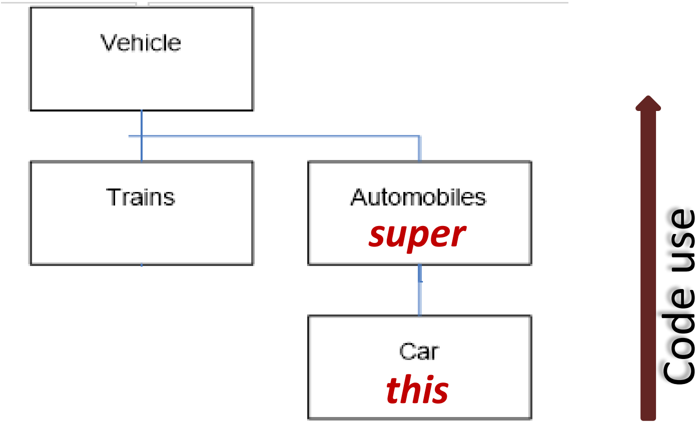
Introduction to Polymorphism (Contd)

- **Polymorphism is the ability to associate many meanings to one method name by means of a late binding mechanism**
  - It does this through a special mechanism known as ***late binding*** or ***dynamic binding***

Introduction to Polymorphism (Contd)

- **Inheritance allows a base class to be defined, and other classes derived from it**
  - Code for the base class can then be used for its own objects, as well as objects of any derived classes

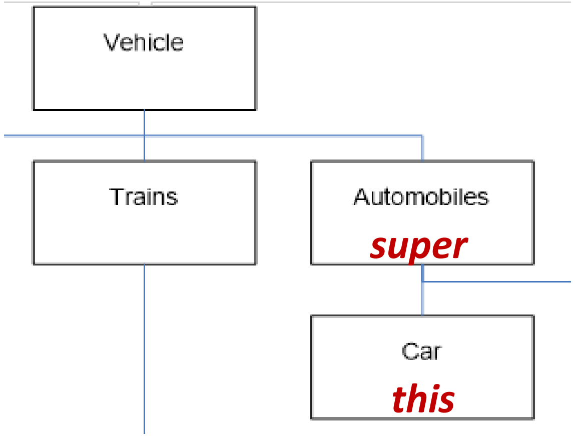
overriding : 같은 헤더를 같같은 메서드 들

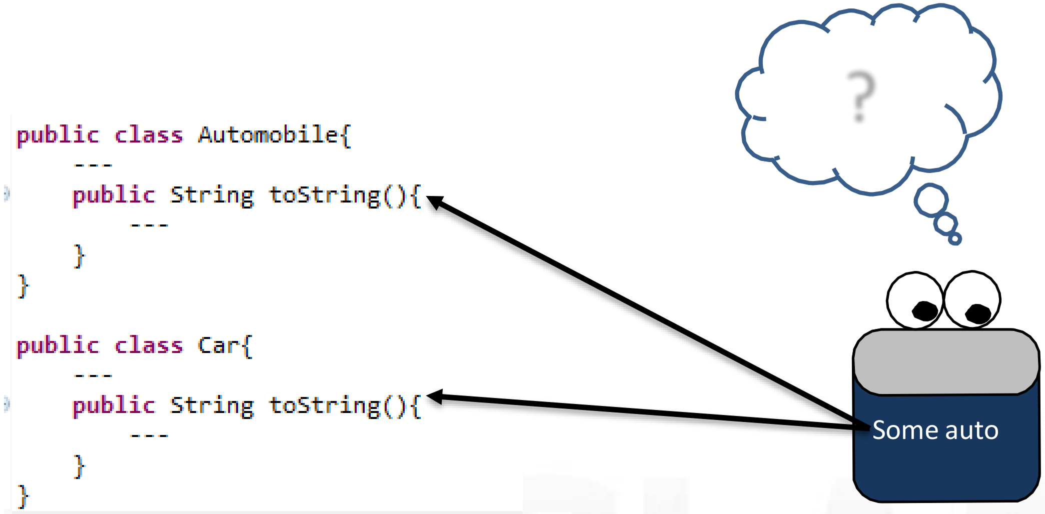


this.getX();
super.toString();

Introduction to Polymorphism (Contd)

- Polymorphism allows changes to be made to method definitions in the derived classes, and have those changes apply to the software written for the base class

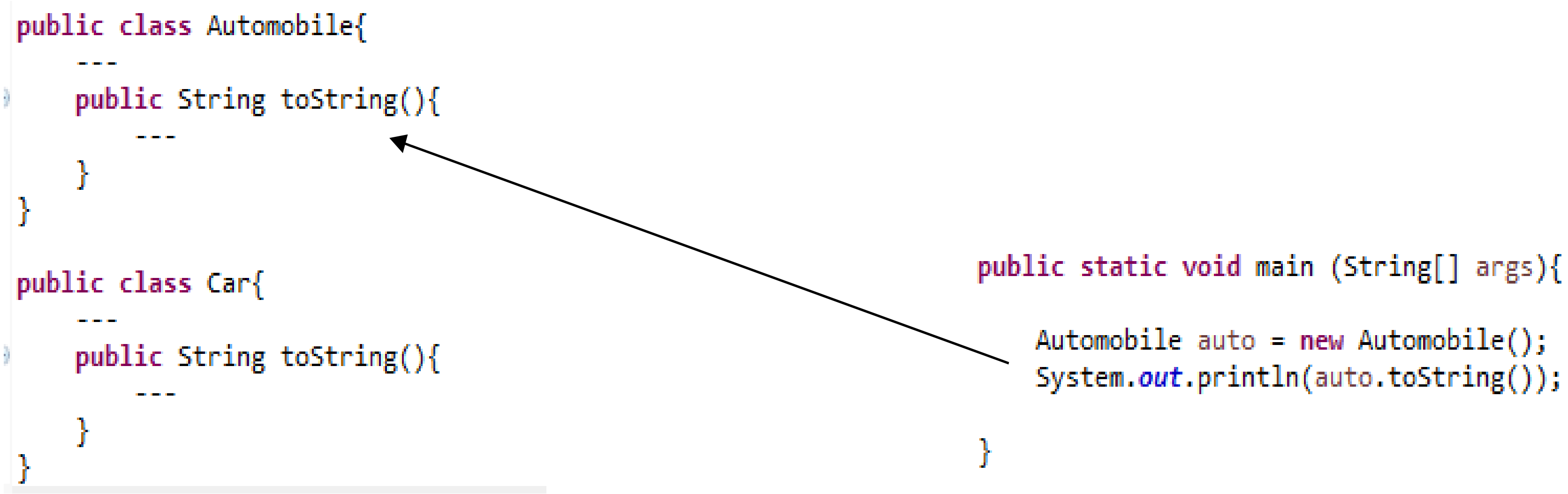




```
public class Automobile{
    ---
    public String toString(){
        ---
    }
}

public class Car{
    ---
    public String toString(){
        ---
    }
}
```

Binding

- The process of associating a method definition with a method invocation is called *binding*



```
public class Automobile{
    ---
    public String toString(){
        ---
    }
}

public class Car{
    ---
    public String toString(){
        ---
    }
}
```

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

    Automobile auto = new Automobile();
    System.out.println(auto.toString());

}
```

Binding

오버라이딩 된 값에서 어떤 곳에 연결을 시켜줄지 연결을 시켜주는 것

- **If the method definition is associated with its invocation when the code is compiled, that is called *early binding* or *static binding***

코딩을 할 때 ,즉 컴파일 될 때 이미 정의(연결)가 끝남

- **If the method definition is associated with its invocation when the method is invoked (at run-time), that is called *late binding* or *dynamic binding***

메서드들이 호출 될때서야 어디로 연결할지 결정하는 것

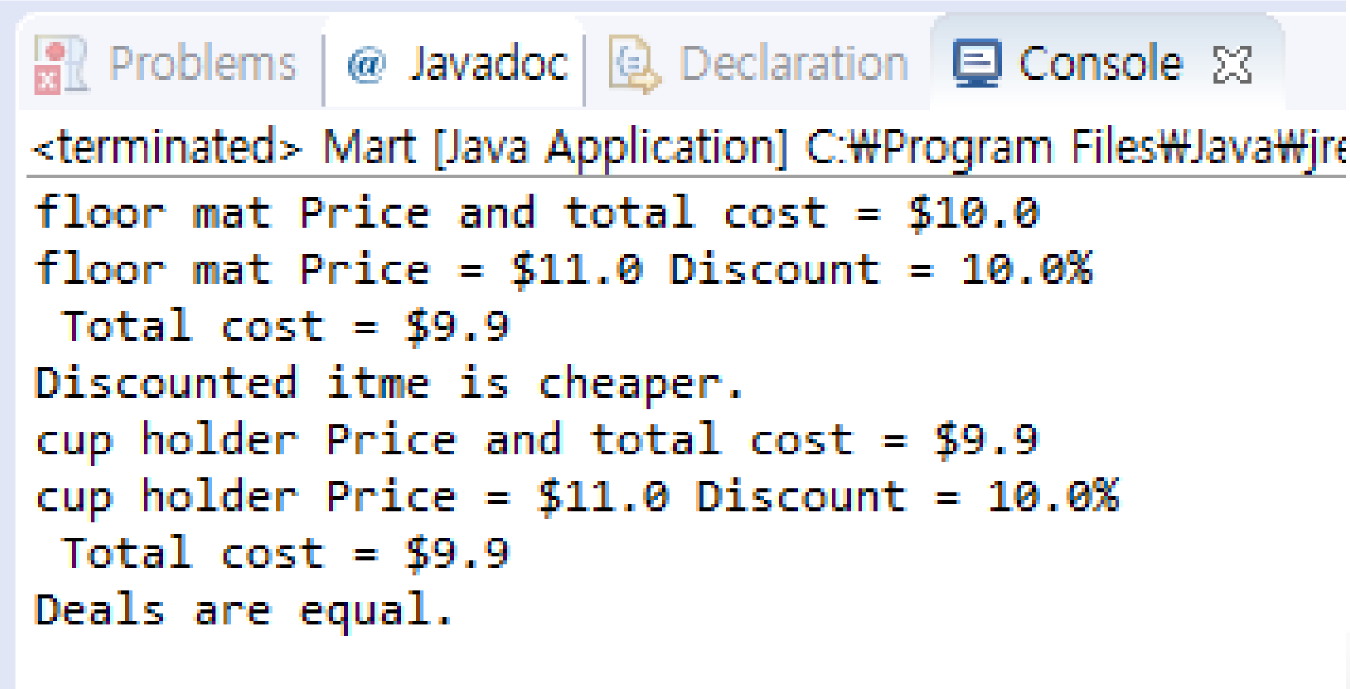
- **Java uses *late binding* for all methods**
  - Except for a few cases discussed later

Self-Test (1)

- **프로젝트 명: Project08_1**
  - git commit –m “Project08_1”
- **Sale 클래스에 다음 메소드를 작성할 것**
  - 두 객체의 name과 bill()이 동일할 경우 true를 반환하는 equalDeals(Sale otherSale) 메소드를 작성할 것
  - 호출한 객체의 bill()이 인자의 bill()보다 작을 경우 true를 반환하는 lessThan(Sale otherSale) 메소드를 작성할 것
- **DiscountSale 클래스에 다음 메소드를 작성할 것**
  - Sale 클래스의 bill() 메소드를 DiscountSale 클래스의 할인율(discount)이 적용된 값을 반환하는 bill() 메소드로 override 할 것
    - discount 값은 %(percentage)이다.
  - 두 객체의 name, bill, discount가 동일할 경우 true를 반환하는 equals(Object obj)를 작성할 것

Self-Test (1) (Contd)

- Mart 클래스의 main 메소드를 수행했을 때 다음과 같이 출력되어야 함



```
<terminated> Mart [Java Application] C:₩Program Files₩Java₩jre
floor mat Price and total cost = $10.0
floor mat Price = $11.0 Discount = 10.0%
 Total cost = $9.9
Discounted itme is cheaper.
cup holder Price and total cost = $9.9
cup holder Price = $11.0 Discount = 10.0%
 Total cost = $9.9
Deals are equal.
```

No Late Binding for Static Methods

- **When the decision of which definition of a method to use is made at compile time, that is called *static binding***
  - This decision is made based on the *type of the variable naming the object*
- **Java uses static, not late binding with *private*, *final*, and *static* methods**
  - In the case of *private* and *final* methods, late binding would serve no purpose
  - However, in the case of a static method invoked using a calling object, it does make a difference

The *final* Modifier

- *A method* marked *final* indicates that it cannot be overridden with a new definition in a derived class
  - If *final*, the compiler can use early binding with the method

```
public final void someMethod() { … }
```

- *A class* marked *final* indicates that it cannot be used as a base class from which to derive any other classes

Upcasting and Downcasting

- *Upcasting* is when an object of a derived class is assigned to a variable of a base class (or any ancestor class)

- *Downcasting* is when a type case is performed from a base class to a derived class (or from any ancestor class to any descendent class)
  - *Downcasting* has to be done very carefully
  - In many cases it doesn't make sense, or is illegal

Upcasting and Downcasting (Contd)

- *Downcasting* makes sense only if the object to be cast is an *instanceOf* the class type

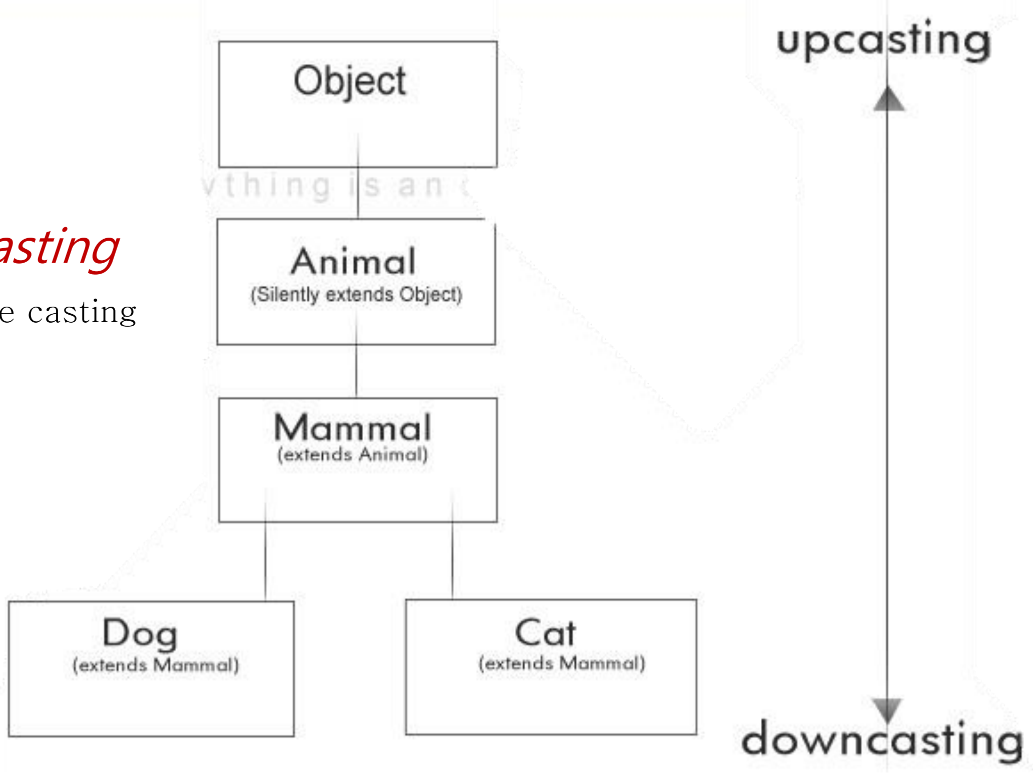
getclass는 자식의 class 타입은 포함하지않음

```
if(myObject instanceOf ClassType) {
        ClassType classObject = (ClassType) myObject;
        // now you can use classObject form here
}
```

Upcasting and Downcasting (Contd)

Cat c = new Cat();
Mammal m = c; // upcasting

upcasting이기 때문에 자동으로 type casting



Cat c1 = new Cat();
Animal a = c1; // automatic upcasting to Animal
Cat c2 = (Cat) a; // manual downcasting back to a Cat

A First Look at the *clone* Method

- **Every object inherits a method name *clone* from the class *Object***
  - The method *clone* has no parameters
  - It is supposed to return a deep copy of the calling object

- **However, the inherited version of the method was not designed to be used as is**
  - Instead, each class is expected to override it with a more appropriate version

Simple *clone* Method

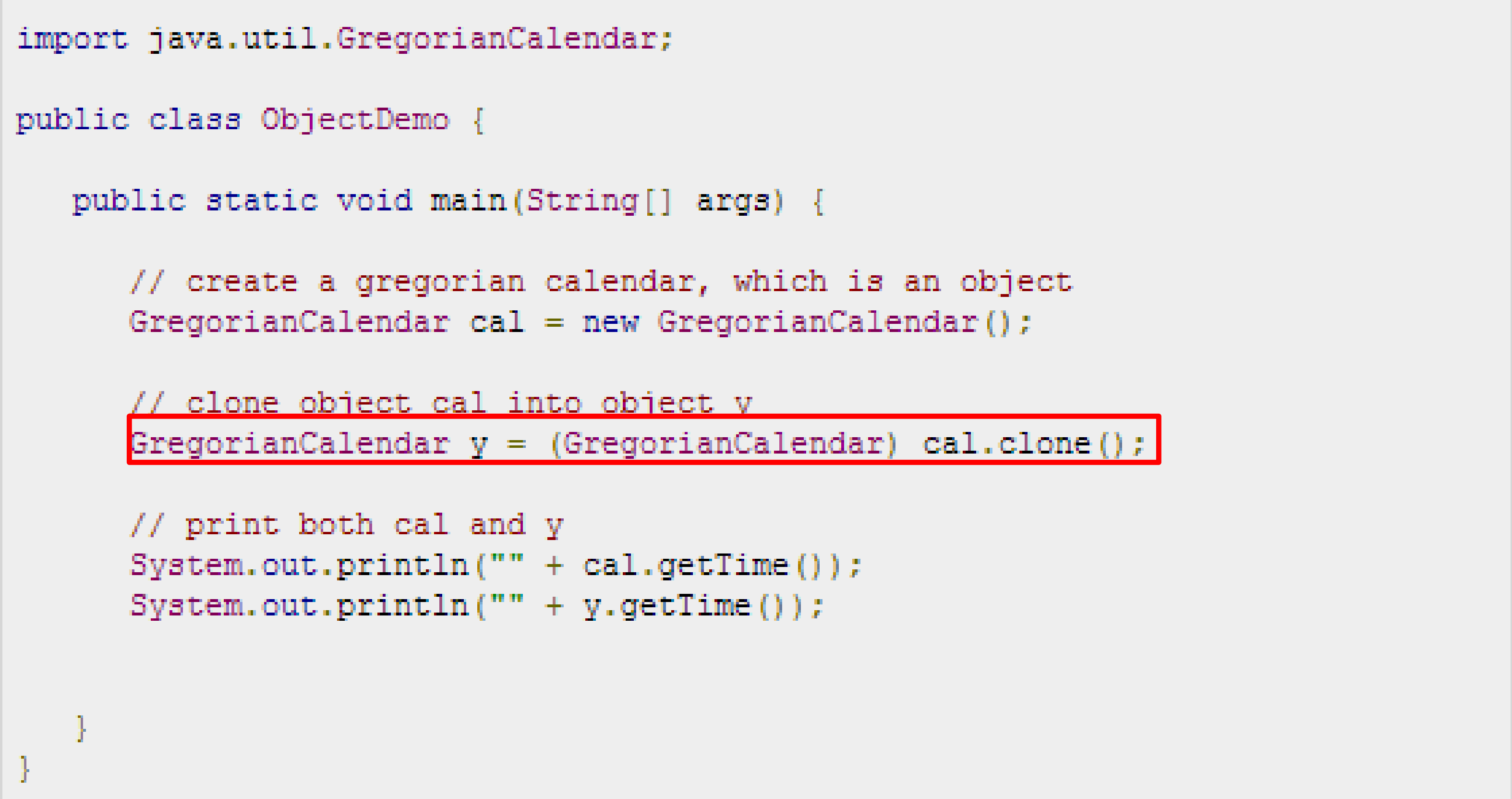
- We can define a simple clone method by using the copy constructor

```
public ClassType clone() {
        return new ClassType(this)
}
```

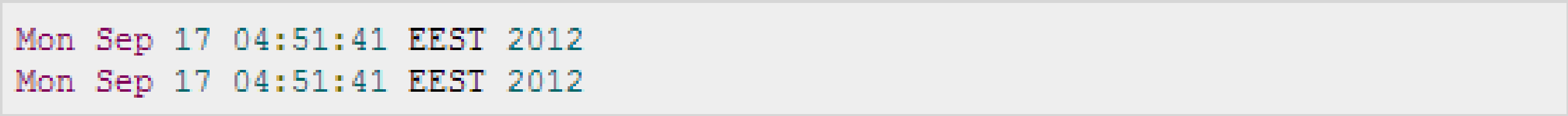
deep copy랑 다른 점은 privacy leak을 해결하기 위해선 deep copy는
정해진 타입을 써야한다. 하지만 부모 class type 배열에 원소들이
자식 class type이라면 제대로 된 deep copy를 하기 어렵다.(정해진 type으로만
deep copy를 하기 때문) 그러나 clone은 type형에 맞춰서 deep copy를
하기 때문에 deep copy의 태생적 문제를 해결할 수 있다.

- This is a very simple clone method, however more checks should be done before cloning
    - We do not cover this until Chapter 13

Example



```
import java.util.GregorianCalendar;

public class ObjectDemo {

   public static void main(String[] args) {

      // create a gregorian calendar, which is an object
      GregorianCalendar cal = new GregorianCalendar();

      // clone object cal into object y
      GregorianCalendar y = (GregorianCalendar) cal.clone();

      // print both cal and y
      System.out.println("" + cal.getTime());
      System.out.println("" + y.getTime());


   }
}
```



```
Mon Sep 17 04:51:41 EEST 2012
Mon Sep 17 04:51:41 EEST 2012
```

Abstract Classes

- Some classes may be defined with incomplete methods definitions (abstract methods).
- Such classes are said to be abstract
- Such classes cannot be instantiated but must be extended by a concrete class

method 들이 모두 정의 되어 있는 class

- The concrete class must implement all abstract methods
  - If all abstract methods cannot be implemented then the class must also be marked as abstract

Abstract Classes (Contd)

- **Definitions** concrete class 를 받아 정의가 되어 져야하는 class
  - An **abstract class** is a class that contains one or more abstract methods and therefore cannot be instantiated
  - Abstract methods are methods that without complete definitions. instead, they are simple placeholders
  - A **concrete class** is a class that contains no abstract methods and therefore can be instantiated

Tip: An Abstract Class Is a Type

- **Although an object of an abstract class cannot be created, it is perfectly fine to have a parameter of an abstract class type**
  - This makes it possible to plug in an object of any its descendent classes

추상 클래스는 파라미터로 사용되어 질 순 있음
객체 생성을 못하는데? -> 추상 클래스를 아규먼트로 전달하는 게 아닌 concrete class type을 전달한다.

- **It even make sense to have a variable of an abstract class type, although it can only name objects of its concrete descendent classes**

Defining Abstract Class

- Defining an abstract class is simple

```
public abstract class Myclass {
            // class constructors
            // accessors and mutators
            // other methods

            public abstract returnType myMethod();
}
```

Abstract class header
Common fields and methods
Abstract Methods

- When defining an abstract method only specify the header

```
public abstract returnType myMethod();
```

When to use

- Consider using abstract classes if any of these statements apply to your situation:
  - You want to **share code** among several **closely related classes**
  - You expect that classes that extend your abstract class have **many common methods** or **fields**, or require access modifiers other than public (such as *protected* and *private).*
  - You **want to declare non-static or non-final fields.** This enables you to define methods that can access and modify the state of the object to which they belong

Self-Test (2)

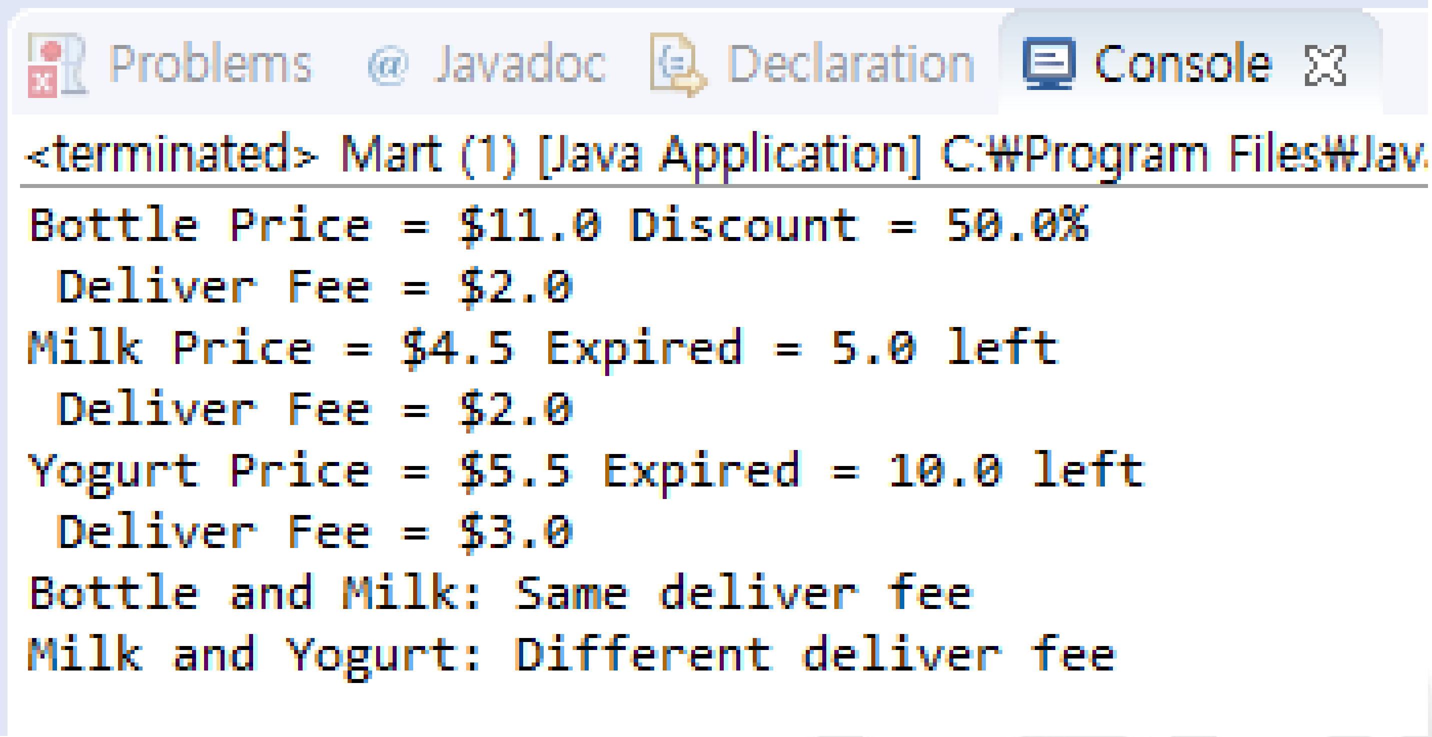
- 프로젝트 명: Project08_2
  - git commit –m “Project08_2”
- Sale 클래스에 다음 메소드를 작성할 것
  - 배송 비용을 반환하는 deliverFee() 메소드를 작성
  - deliverFee() 메소드는 할인율과 남은 유통기한에 따라 달라지므로 Sale 클래스에서는 abstract 메소드로 선언
  - 배송 비용이 같은 지 여부를 반환하는 equalDeliverFee() 메소드 작성 (deliverFee() abstract 메소드를 사용할 것)
- DiscountSale 클래스에 다음 메소드를 작성할 것
  - 매장에서는 제품의 할인율이 낮을 경우 무료 배송을 해주는 서비스를 진행하고 있다.
  - 할인율에 따라 달라지는 deliverFee() 메소드를 정의할 것
    - 할인율 >= 80%: 배송비용 3$
    - 30% <= 할인율 < 80%: 배송비용 2$
    - 할인율 < 30%: 배송비용 없음

Self-Test (2) (Contd)

- ExpiredSale 클래스에 다음 메소드를 작성할 것
  - 매장에서는 제품의 유통기한이 얼마 남지 않았을 경우 무료 배송을 해주는 서비스를 진행하고 있다.
  - 유통기한에 따라 달라지는 deliverFee() 메소드를 정의할 것
    - 유통기한 >= 10: 배송비용 3$
    - 3 <= 유통기한 < 10: 배송비용 2$
    - 1 < 유통기한 < 3: 배송비용 없음
  - 유통기한이 1일 이하일 경우 현장판매만 가능하므로 오류 처리

Self-Test (2) (Contd)

- Mart 클래스의 main 메소드를 수행했을 때 다음과 같이 출력되어야 함



```
<terminated> Mart (1) [Java Application] C:₩Program Files₩Jav
Bottle Price = $11.0 Discount = 50.0%
 Deliver Fee = $2.0
Milk Price = $4.5 Expired = 5.0 left
 Deliver Fee = $2.0
Yogurt Price = $5.5 Expired = 10.0 left
 Deliver Fee = $3.0
Bottle and Milk: Same deliver fee
Milk and Yogurt: Different deliver fee
```