CSEN401 – Computer Programming Lab

Topics:
Object Oriented Features: Abstraction and Polymorphism

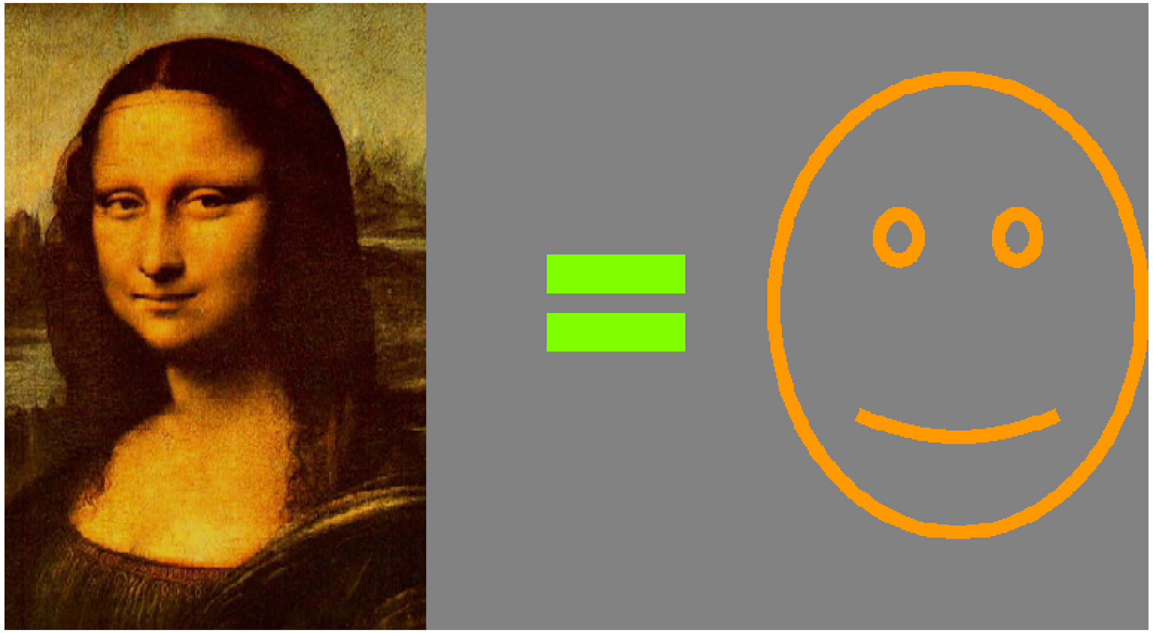
Prof. Dr. Slim Abdennadher

Object-Oriented Paradigm: Features

Easily remembered as **A-PIE**

- **A**bstraction
- **P**olymorphism
- **I**nheritance
- **E**ncapsulation

Abstraction

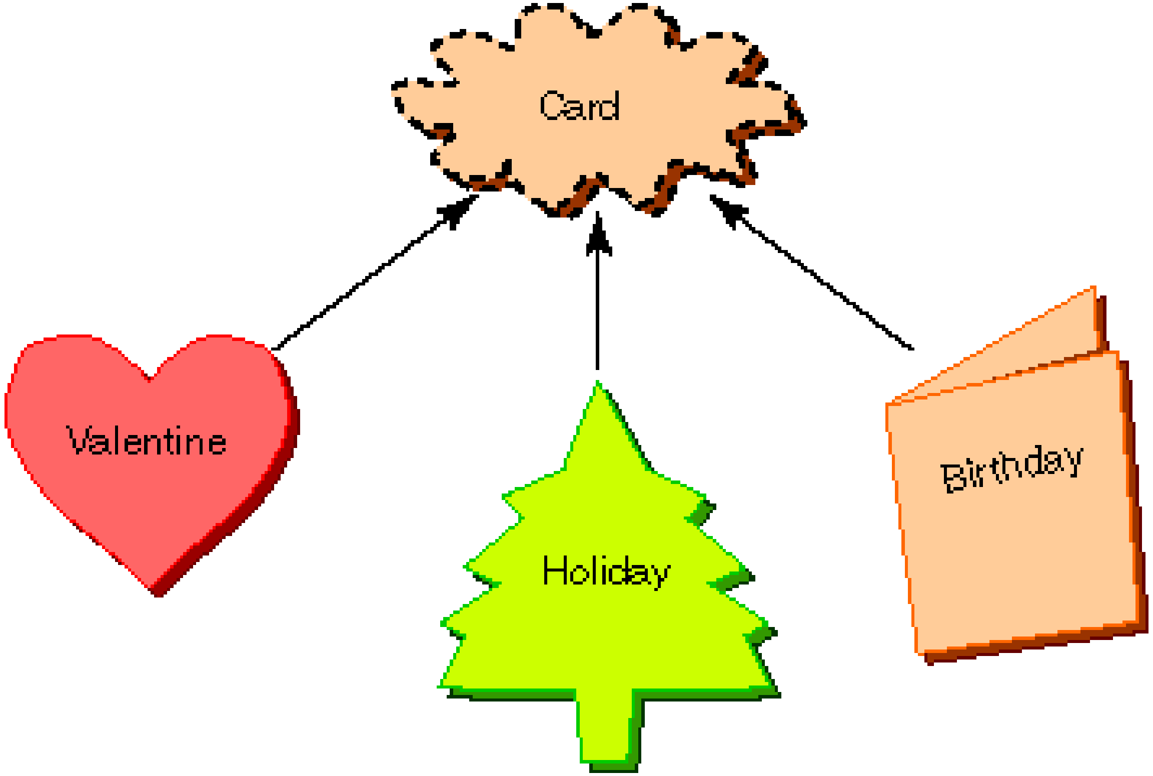


Data abstraction: The process of refining away the unimportant details of an object, so that only the useful characteristics that define it remain.

Abstract Classes

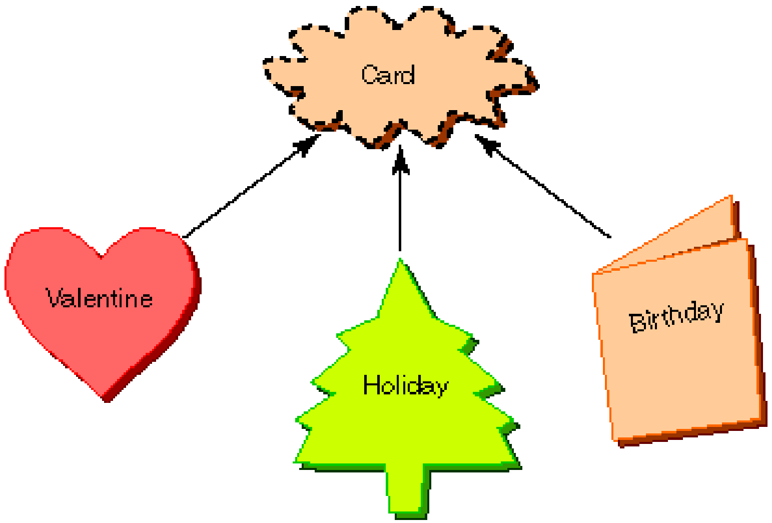
- **Abstract classes** are classes that represent an **Abstract Concept**.
- Creating an Object from the class makes no sense.

  Example: Greeting cards



- An abstract class cannot be **instantiated**.

Abstract Classes – Example (I)



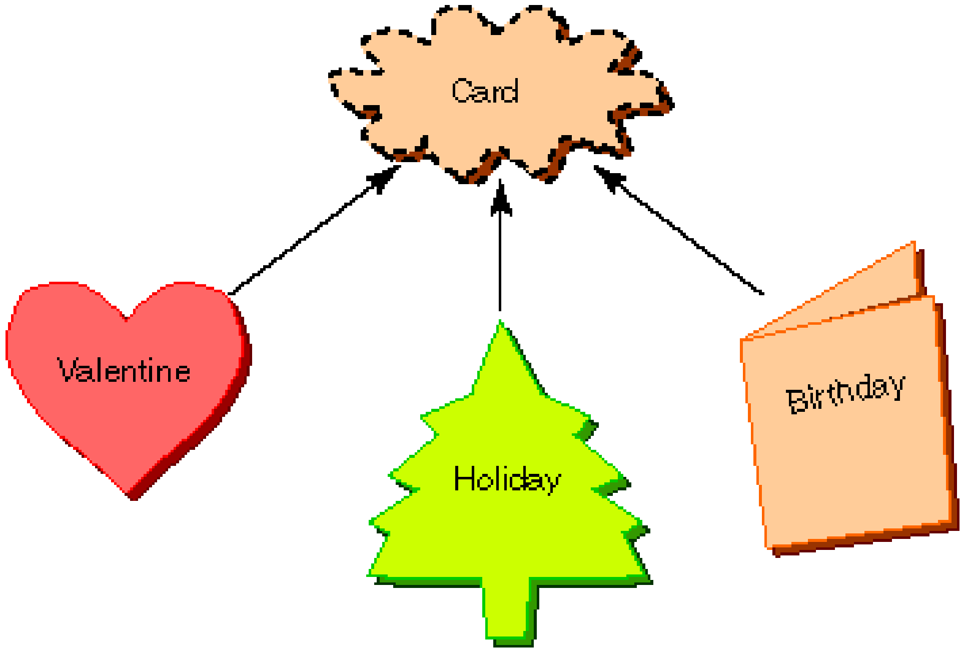
- The parent class is `Card`.

```
abstract class Card
{
   .... // definitions of methods and variables
}
```

- Children classes are `Valentine`, `Holiday`, and `Birthday`.

```
class Holiday extends Card
{
  ...
}
```

Abstract Classes – Example (II)



- An object must be an **instance** of one of the three child types: `Valentine`, `Holiday`, and `Birthday`.

- There will be no such thing as an object that is merely a “Card”.

- The **purpose** of an abstract class is to be a parent to several related classes. The child classes **inherit** from the abstract parent class.

Abstract Methods – Example

- A card object will have a `greeting()` method that writes out a greeting.
- Each type of card contains an appropriate greeting:
  - The `Holiday` card says `Season's Greetings`.
  - The `Birthday` card says `Happy Birthday`.
  - The `Valentine` card says `Love and Kisses`.
- Class definition of the abstract class `Card`

```
abstract class Card
{
  String recipient;                   // name of who gets the card
  public abstract void greeting();    // abstract greeting() method
}
```

- An **abstract method** has no body, i.e. it has no statements.
- An abstract method declares an **access modifier**, **return type**, and **method signature** followed by a semicolon.

Abstract and Non-Abstract Methods

- A non-abstract child class inherits the abstract method and **must** define a non-abstract method that matches the abstract method.

```
class Holiday extends Card
{
  public void greeting()
  {
    System.out.println("Dear " + recipient + ",\n");
    System.out.println("Season’s Greetings!\n\n");
  }
}
```

- An abstract class can contain **non-abstract methods**, which will be **inherited** by the children.
- An abstract child of an abstract parent does not have to define non-abstract methods for the abstract signatures it inherits.
- If a class contains even one abstract method, then the class itself has to be declared to be abstract.

Advantage of Abstract Classes

- The advantage of using an abstract class is that you can **group** several related classes together as siblings.
- Grouping classes together is important in keeping a program **organized** and **understandable**.
- You can get the same thing done without using this way to organize.
- This is a matter of **program design**, which is not easy at all.

Abstraction and the Programmer

- The task of the programmer, given a problem, is to determine what data needs to be **extracted** (i.e. abstracted) in order to adequately design and ultimately code a solution.
- Normally, with a good problem description and/or good **customer communication** this process is relatively painless.
- It is difficult to precisely describe how a programmer can determine which data items are important.
- In part it involves having a clear understanding of the problem.
- Equally, it involves being able to see ahead to how the problem might be modeled/solved.
- Given such understanding, it is normally possible to determine which data items will be needed to **model** the problem.
- However, please note, the ability to abstract relevant data is a **skill** that is largely not taught, but rather one that is refined through experience.

Problems with Abstraction

Two possible forms:

- **Something unnecessary was abstracted**, i.e. some data is redundant. This may or may not be a problem, but it is bad design.
- **Something needed was not abstracted**. This is a more serious problem, usually discovered later in the design or coding stages, and entails that the design needs to be changed to incorporate the missing data item, code changed, etc.

Interface for Stacks

```
public interface Stack {
   public boolean isEmpty();
   public boolean isFull();
   public Object peek();
   public Object pop();
   public void push(Object item);
   public int size();
}
```

Interfaces

- An **interface** contains method declarations. All methods declared in an interface are implicitly `public`, so the public modifier can be omitted.
- An **interface** can contain constant declarations in addition to method declarations. All constant values defined in an interface are implicitly `public`, `static`, and `final`.
- A class can implement more than one interface. This class should implement all methods declared in the interfaces.
- It is possible, however, to define a class that does not implement all of the interface methods, provided that the class is declared to be `abstract`.
- An interface cannot be **instantiated**.
- However, you can use an interface as a type.

Interfaces versus Abstract Classes

- An **abstract class** is an **incomplete class** that requires further specification (before instances of that class can exist).
- An **interface** is solely a **specification** or prescription of behavior.
- An interface does not have any suggestion of a **hierarchical** relationship that is central to inheritance.
- A class can implement **several different interfaces**, however it can only inherit from one class (single inheritance).
- Interfaces should be used to capture **similarities** between unrelated classes without artificially enforcing a class relationship.

Polymorphism

Polymorphism

- **Polymorphism** comes from a Greek word meaning **many forms**.
- Two types of polymorphism:
  - **Static Polymorphism**
  - **Dynamic Polymorphism**

Static Polymorphism

- In Java, a class can have **multiple methods** with the same name.
- When two or more methods in a class have the same name, the method is said **overloaded**.
- The methods must be **different** somehow, or else the compiler would not associate a call to a particular method definition.

Method Signature

- The compiler identifies a method by more than its name.
- A method is uniquely identified by its **signature**.
- A method **signature** consists of
  - the **method's name** and
  - its **parameter list**
- If the parameter types do not match exactly, both in number and position, then the method signatures are different.
- **Example:**

```
System.out.println(int)       -- prints number
System.out.println(String)    -- prints text
```

Example Overloading

1. `static void f() {/* ... */}`

   This version has no parameters, so its signature differs from all the others which each have at least one parameter.

2. `static void f(int x) {/* ... */}`

   This version differs from version 3, since its single parameter is an `int`, not a `double`.

3. `static void f(double x) {/* ... */}`

   This version differs from version 2, since its single parameter is a `double`, not an `int`.

4. `static void f(int x, double y) {/* ... */}`

   This version differs form version 5 because, even though versions 4 and 5 have the same number of parameters with the same types, the order of the types is different.

5. `static void f(double x, int y) {/* ... */}`

Dynamic Polymorphism – Late Binding

```
class Human {
    public String getGender() {return "Neutral";}
}

class Man extends Human {
   public String getGender() {return "Man";}
}

class Woman extends Human {
   public String getGender() {return "Woman";}
}
```

Overriding

- What is the output of the following code?

  ```
  Human human = new Human();
  System.out.println("Gender: " + human.getGender());
  Man man = new Man();
  System.out.println("Gender: " + man.getGender());
  Woman woman = new Woman();
  System.out.println("Gender: " + woman.getGender());
  ```

-
  ```
  Neutral
  Man
  Woman
  ```

Dynamic Binding

- What is the output of the following code?

  ```
  Human human = new Human();
  System.out.println("Gender: " + human.getGender());
  Human man = new Man();
  System.out.println("Gender: " + man.getGender());
  ```

-
  ```
  Neutral
  Man
  ```

- The method that got invoked, is the version that is present in the object type and NOT the reference type.

Dynamic Binding – Example

Assume that three subclasses (`Cow`, `Dog` and `Snake`) have been created based on the `Animal` super class, each having their own `speak()` method.

```
public class AnimalReference {
  public static void main(String args[])
  Animal ref                     // set up var for an Animal
  Cow aCow = new Cow("Bossy");   // makes specific objects
  Dog aDog = new Dog("Rover");
  Snake aSnake = new Snake("Earnie");

  // now reference each as an Animal
  ref = aCow;
  ref.speak();
  ref = aDog;
  ref.speak();
  ref = aSnake;
  ref.speak(); }
```

- No `Animal` object exists.
- The program is able to resolve the correct method related to the subclass object at **runtime**.

Downcasting and instanceof

```
class Animal {
    //...
}

class Dog extends Animal {
    public void woof() {
        System.out.println("Woof!");
    }
    //...
}

class Cat extends Animal {
    public void meow() {
        System.out.println("Meow!");
    }
    //...
}


class Hippopotamus extends Animal {
```

Downcasting and instanceof

```
    public void roar() {
        System.out.println("Roar!");
    }
    //...
}
```

Downcasting and instanceof

```
class Example {

    public static void main(String[] args) {

        makeItTalk(new Cat());
        makeItTalk(new Dog());
        makeItTalk(new Hippopotamus());
    }

    public static void makeItTalk(Animal animal) {

        if (animal instanceof Cat) {
            Cat cat = (Cat) animal;
            cat.meow();
        }
        else if (animal instanceof Dog) {
            Dog dog = (Dog) animal;
            dog.woof();
        }
        else if (animal instanceof Hippopotamus) {
```

Downcasting and `instanceof`

```
            Hippopotamus hippopotamus = (Hippopotamus) animal;
            hippopotamus.roar();
        }
    }
}
```

The OO Way

```
abstract class Animal {
    public abstract void talk();
    //...
}


class Dog extends Animal {
    public void talk() {
        System.out.println("Woof!");
    }
    //...
}


class Cat extends Animal {
    public void talk() {
        System.out.println("Meow!");
    }
```

The OO Way

```
    //...
}


class Hippopotamus extends Animal {
    public void talk() {
        System.out.println("Roar!");
    }
    //...
}

class Example2 {

    public static void main(String[] args) {

        makeItTalk(new Cat());
        makeItTalk(new Dog());
        makeItTalk(new Hippopotamus());
```

The OO Way

```
    }

    public static void makeItTalk(Animal animal) {

        animal.talk();
    }
}
```

Static Polymorphism versus Dynamic polymorphism

- Static polymorphism is associated with overloaded methods because it gives the impression that a single named method will accept a number of different argument types, e.g. `System.out.println` method.
- Each overloaded method is separate and the compiler can see the difference between them at compile time.
- Dynamic polymorphism is where a class overrides a superclass method.
- Any differences in the methods implementations are only seen at runtime, so they are considered dynamic.