SchoolNova

IT101

Inheritance, Encapsulation, Polymorphism and Constructors

OOP Advantages and Concepts

- What are OOP's claims to fame?
  - Better suited for team development
  - Facilitates utilizing and creating reusable software components
  - Easier program maintenance

- Main OOP Concepts
  - Inheritance
  - Encapsulation
  - Polymorphism

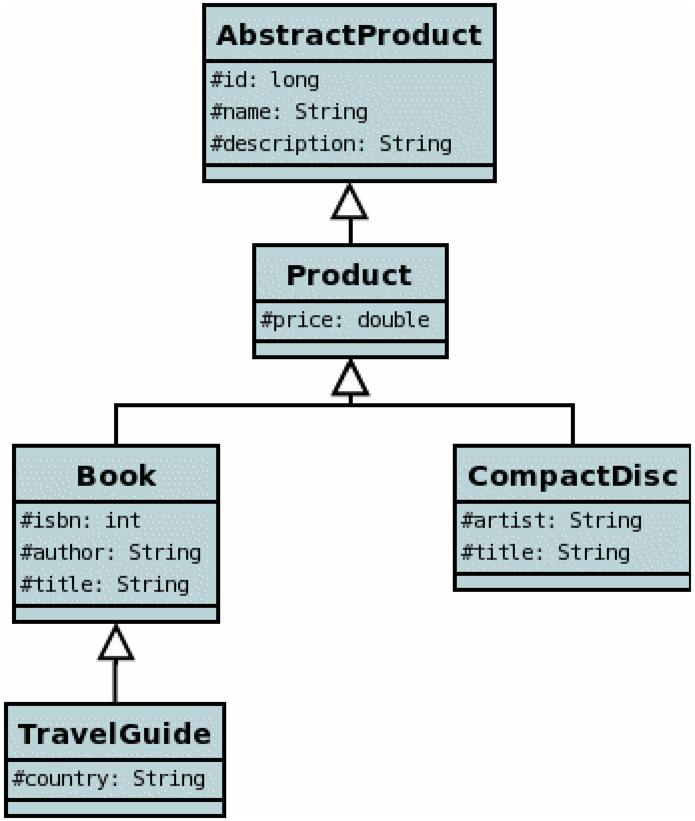
http://www.cs.columbia.edu/~gmw/teaching/1001-s04/Lec17.ppt

Inheritance

- A class can extend another class, inheriting all its data elements and methods while redefining some of them and/or adding its own. Example:
  - `class Student extends Person`
- A class can implement an interface, implementing all the specified methods. Example:
  - `class Poker extends Game implements Gambling`
- Inheritance implements the “is a” relationship between objects.
- In Java, a subclass can extend only one superclass.
- In Java, a subinterface can extend one superinterface
- In Java, a class can implement several interfaces — this is Java’s form of multiple inheritance.
- An abstract class can have code for some of its methods; other methods are declared abstract and left with no code.
- An interface only lists methods but does not have any code.
- A concrete class may extend an abstract class and/or implement one or several interfaces, supplying the code for all the methods.
- Inheritance plays a dual role:
  - A subclass reuses the code from the superclass.
  - A subclass (or a class that implements an interface) inherits the data type of the superclass (or the interface) as its own secondary type.

Inheritance

- Inheritance leads to a hierarchy of classes and/or interfaces in an application:
- An object of a class at the bottom of a hierarchy inherits all the methods of all the classes above. It also inherits the data types of all the classes and interfaces above.
- Inheritance is also used to extend hierarchies of library classes, reusing the library code and inheriting library data types.
- Inheritance implements the “is a” relationship. Not to be confused with embedding (an object has another object as a part), which represents the “has a” relationship.
  - Sailboat **IS A** Boat
  - Sailboat **HAS A** Sail



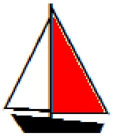
AbstractProduct
#id: long
#name: String
#description: String

Product
#price: double

Book
#isbn: int
#author: String
#title: String

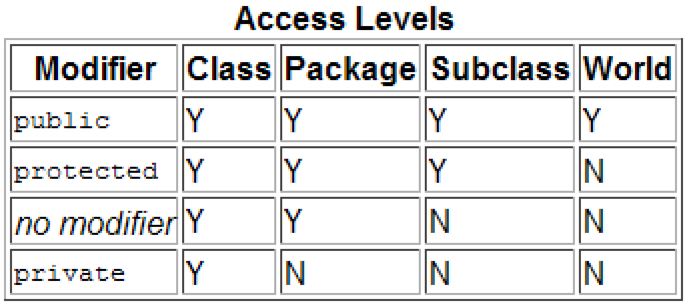
CompactDisc
#artist: String
#title: String

TravelGuide
#country: String



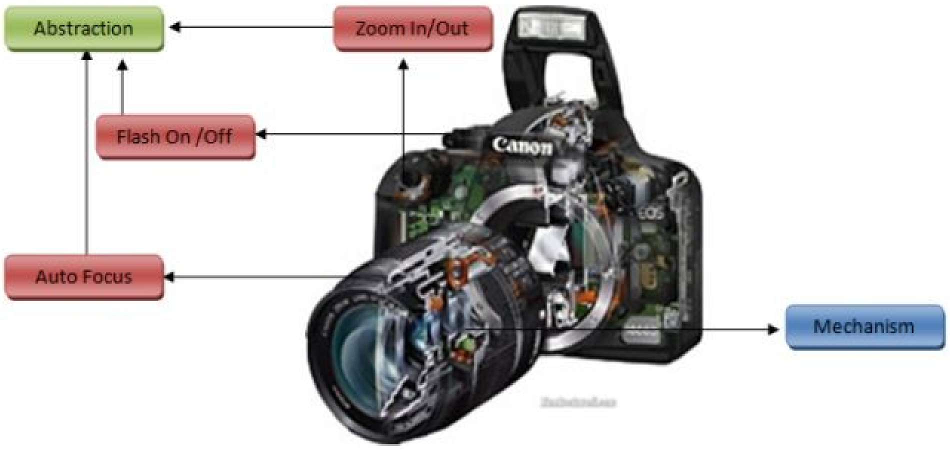
Encapsulation

- Encapsulation means that all data members (variables) of a class are declared private. Methods may be private, too. Example:
  - `private String ssn;`
- Private methods or data are visible to and accessible only by that object. In other words, private data or methods can not be accessed from outside the object.
- The class interacts with other classes mainly through the class's constructors and public methods. When data or methods are declared public, other objects of your program can access it.



Access Levels

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

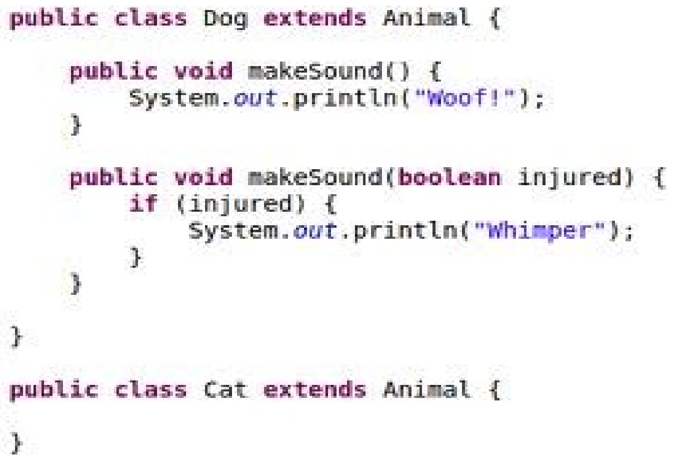


Polymorphism

- We often want to refer to an object by its primary, most specific, data type.
- This is necessary when we call methods specific to this particular type of object
- Polymorphism (meaning “many forms” in Greek) ensures that the appropriate method is called for an object of a specific type when the object is disguised as a more generic type
- Polymorphism is implemented using a technique called late (or dynamic) method binding: which exact method to call is determined at run time.

by Sinipull for codecall.net

http://www.c-sharpcorner.com/UploadFile/433c33/polymorphism-in-java/



```
public class Dog extends Animal {

    public void makeSound() {
        System.out.println("Woof!");
    }

    public void makeSound(boolean injured) {
        if (injured) {
            System.out.println("Whimper");
        }
    }

}

public class Cat extends Animal {

}
```

Constructors

- A constructor initializes an object when it is created. It has the same name (case sensitive) as its class and is syntactically similar to a method.
- Constructors have no return type, but they can accept parameters.
- Use constructors to set initial variable values. Example:

```
class Candle {

    private String color;

    Candle (String aColor) {

        this.color = aColor;

    }

}
```

- A subclass can (and typically should) call the constructor of the superclass using the special super method.
- The super method should be the first statement of the subclass constructor.
- Example:

```
class ScentedCandle extends Candle {

    private float height;

    ScentedCandle(String c, float h) {

        super(c);

        this.height = h;

    }

}
```

Homework

- Prior to developing this program, read the requirements and draw a class diagram of the classes you intend to build (see slide 4 for example of a class diagram).
- Mick’s Wicks makes candles in various sizes.
- Create a class named **Candle** that contains data fields for color, height and price.
- Create get methods for all three fields, e.g.:

```
public String getColor() {
    return this.color;
}
```

- Create set methods for color and height, but not for price. Instead, when height is set, determine the price as $2 per inch. Example:

```
public void setHeight(float newHeight) {
    float pricePerInch = 2;
    this.height = newHeight;
    this.price = this.height * pricePerInch;
}
```

- Create a child class named **ScentedCandle** that contains an additional data field named scent and methods to get and set it. In the child class, override the parent’s setHeight() method to set the price of a ScentedCandle object at $3 per inch.
- Write a **MickWicks** class that instantiates an object of each type of candle, sets color and height, and displays the details (name, height and price).
- Run the MickWicks program to test the results.