Creative Programming

Object Orientation

Overview of this lecture

- A historical note
- Thinking Object Oriented:
  - What is Object-Oriented Programming?
  - Key Elements
- Example: the investment manager
- Coding of Key Elements:
  - Classes
  - Methods and Messages
  - Inheritance
- Common Design Flaws

A historical note

- Smalltalk: first object-oriented language
- Originated within XEROX Parc
  - Very important non-academic research center
- Developed by Alan Kay, Adele Goldberg, Dan Ingalls

What is Object-Oriented Programming?

- OOP is a revolutionary idea, totally unlike anything that has come before in programming
- OOP is an evolutionary step, following naturally on the heels of earlier programming abstractions
- It shows resonant similarity to techniques of thinking about problems in other domains (e.g. architecture)
- It is a new programming paradigm

A Paradigm

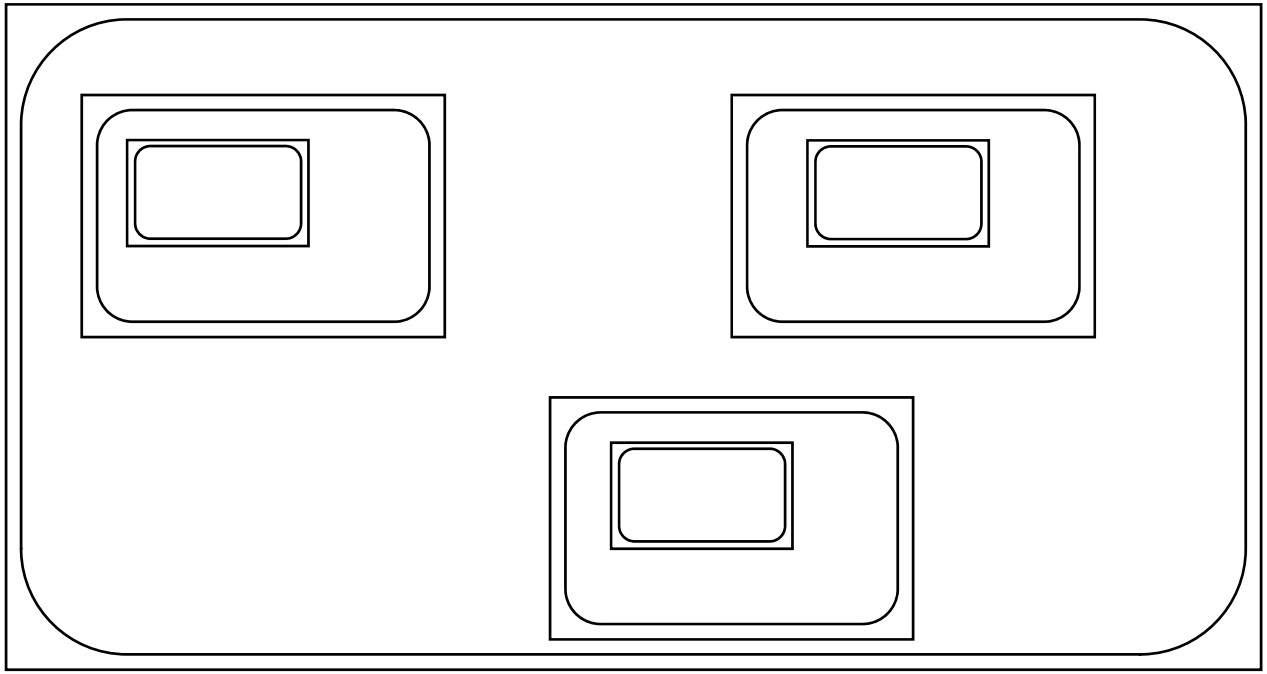
- A set of theories, standards, and methods that together represent a way of organising knowledge – that is, a way of viewing the world.

What is Object-Oriented Programming?

- OOP is based on the principle of *recursive design*
  1. Everything is an object
  2. Objects perform computation by making requests of each other through the medium of messages
  3. Every object has it's own memory, which consists of other objects
  4. Every object is an instance of a class. A class groups similar objects
  5. The class is the repository for behaviour associated with an object
  6. Classes are organised into tree structures, called inheritance hierarchies.

Recursive design

- The structure of the part mirrors the structure of the larger unit



Elements of OOP

1. *Everything is an object*

Actions in OOP are performed by agents, called instances or objects

Elements of OOP-Messages

2. *Objects perform computation by making requests of each other through the medium of messages*

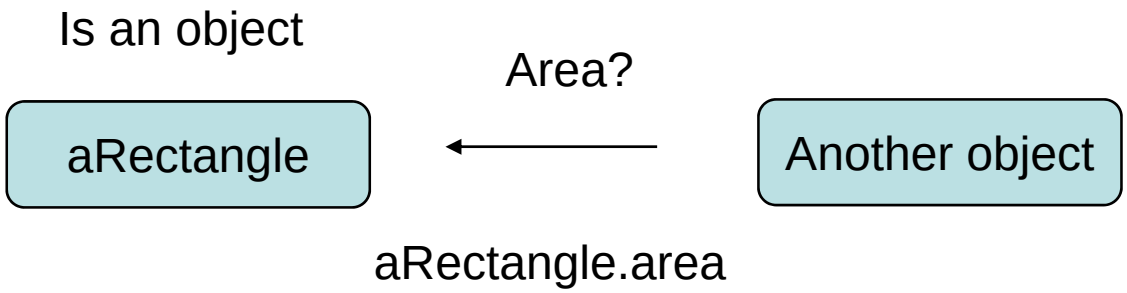
Actions in OOP are produced in response to requests for actions, called messages. An instance may accept a message and in return it will perform an action and return a result.

What vs How

- What: messages
  - Specify what behaviour objects are to perform
  - Details of how are left up to the receiver
  - State information only accessed via messages
- How: Methods
  - Specify how operation is to be performed
  - Must have access to data
  - Need detailed knowledge of data
  - Can manipulate data directly

Message

- Sent to receiver object: receiver-object message
- A message may include parameters necessary for performing the action
- Message-send always return a result (an object)
- Only way to communicate with an object and have it perform actions



Method

- Defines how to respond to a message
- Selected via method lookup technique
- Has name that is the same as message name
- Is a sequence of executable statements
- Returns an object as its result of execution

```
Float Rectangle area
Return side1*side2
```

Information hiding

- As a user of a service being provided by an object, I need only to know the set of messages that the object will accept. I need not to have any idea of how the methods are performed.

- Having accepted a message, an object is responsible for carrying it out.

Object Encapsulation

- Objects encapsulate state as a collection of instance variables
- Objects encapsulate behaviour via methods invoked by messages

External perspective	vs	Internal perspective
What	vs	How
Message	vs	Method

Object encapsulation ctd.

- Technique for
  - Creating for objects with encapsulated state/behaviour
  - Hiding implementation details
  - Protecting the state information of objects
- Puts objects in control
- Facilitates modularity, code reuse and maintenance

Rectangle
side1:Integer side2:Integer
circumference area moveTo:aPoint

Elements of OOP-Receivers

- Messages differ from traditional functions:
  - In a message there is a designated receiver that accepts the message
  - The interpretation of the message may be different, depending upon the receiver

Example

- Vars:

  Florist Flo;

  Secretary Beth;

  Dentist Ken;

- Code:

  Flo.sendFlowersTo(myFriend);

  Beth.sendFlowersTo(myFriend);

  Ken.sendFlowersTo(myFriend); (will probably not work)

- Although different objects might receive the same message, the behaviour they perform will likely be different

Elements of OOP-Recursive Design

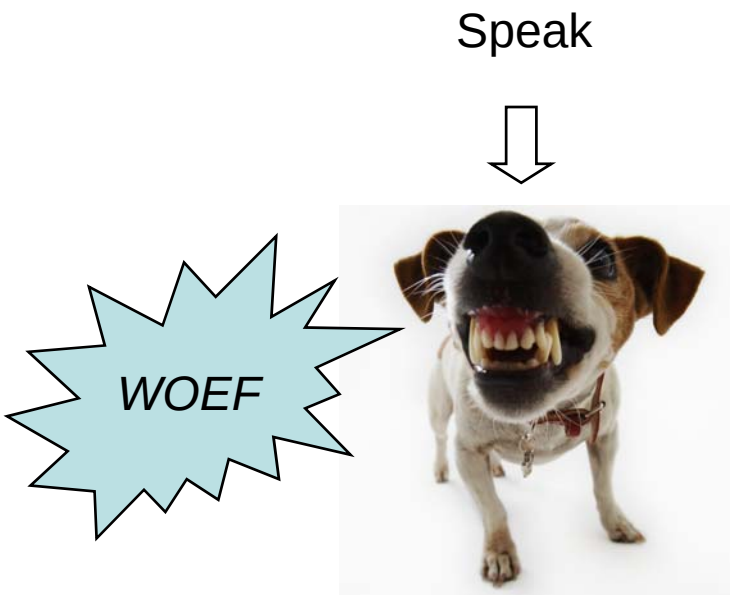
3. Every object has it's own memory, which consists of other objects

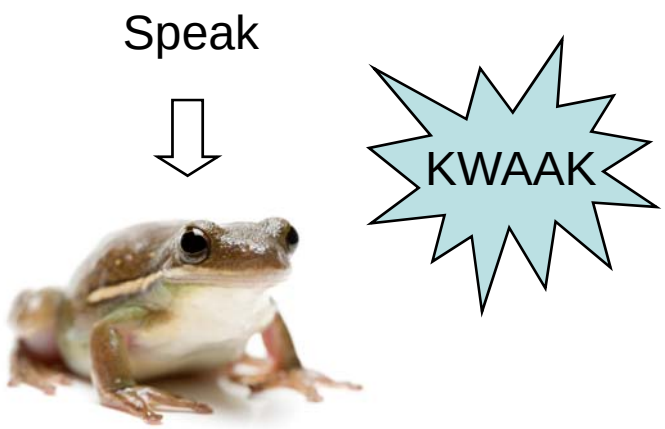
Each object is like a miniature computer itself – a specialised processor performing a specific task

"Ask not what you can do *to* your datastructures, but what your datastructures can do *for* you"

Polymorphism

- Same message may be sent to different objects





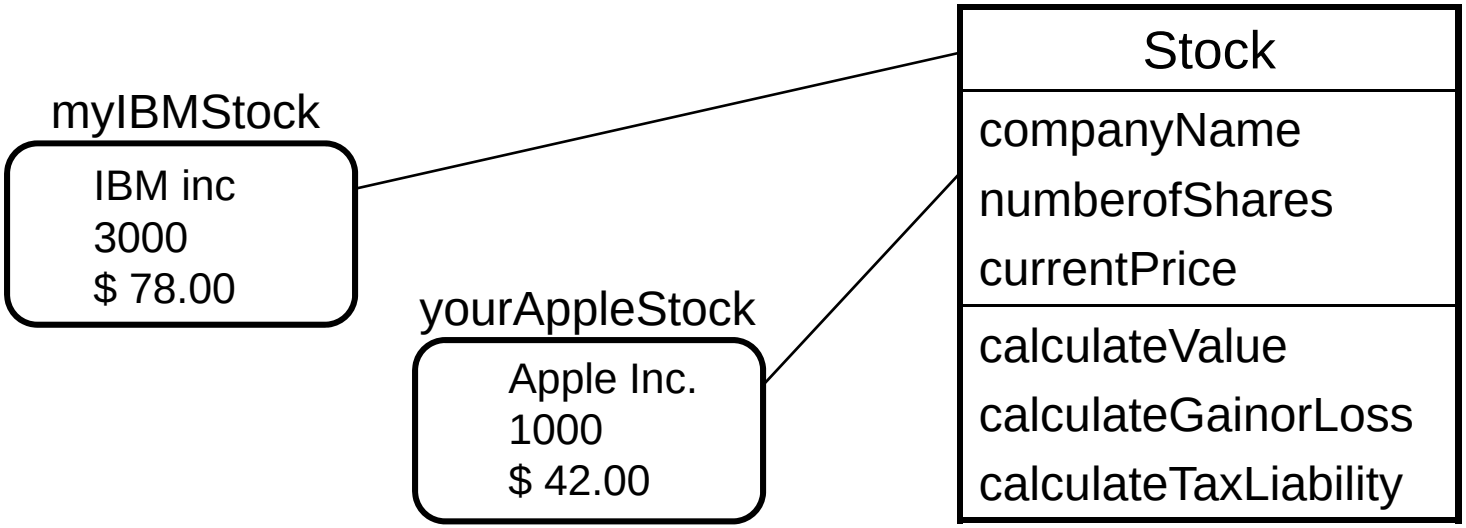
Elements of OOP - Classes

4. Every object is an instance of a class. A class groups similar objects
5. The class is the repository for behaviour associated with an object

- The behaviour I expect from Flo is determined from a general idea I have of the behaviour of florists
- We say Flo is an instance of the class Florist
- Behaviour is associated with classes, not with individual instances. All objects of a given class use the same method in response to similar messages

Instance

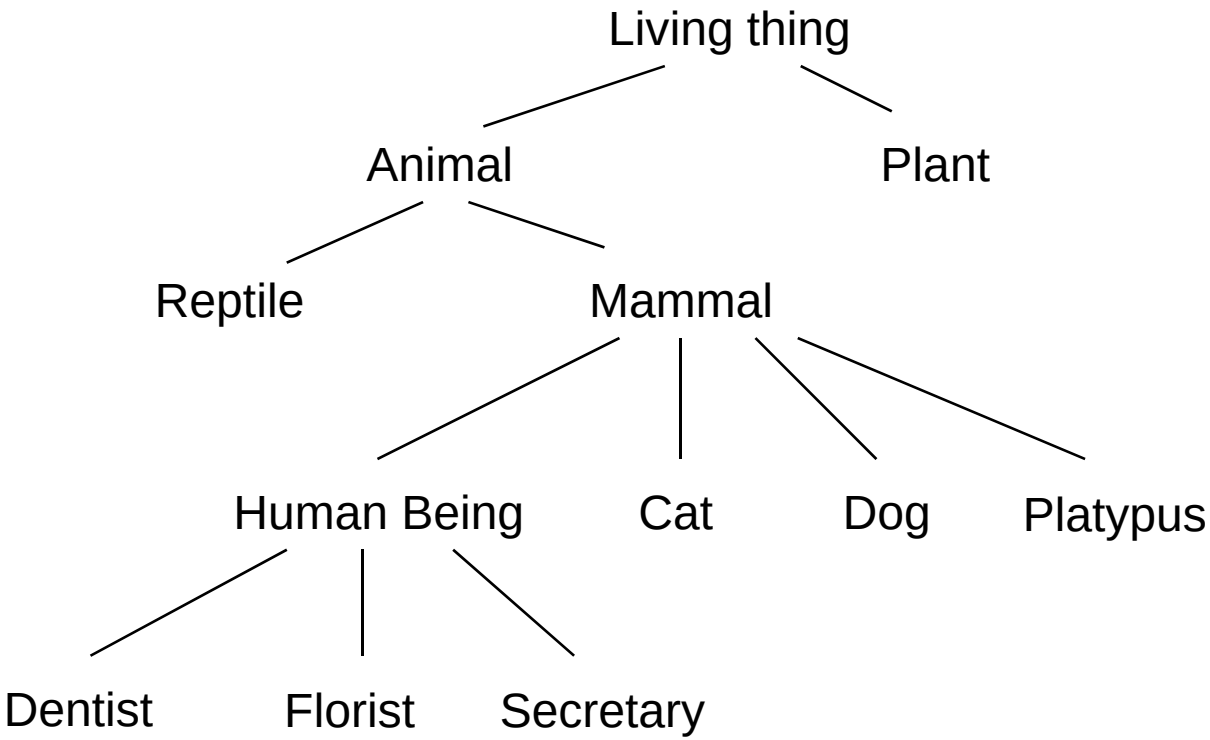
- Instance: a particular occurrence of an object defined by a class
- Each instance has its own value for each instance variable
- All instances of a class share the same methods



Hierarchies of categories

- But there is more I know about Flo then just that he is a Florist. He is a ShopKeeper, and a Human, and a Mammal, and so on.
- At each level of information a certain amount of info is recorded. That info is applicable to all lower (more specialised) levels.

Class Hierarchies



Elements of OOP - Inheritance

6. Classes are organised into tree structures, called inheritance trees

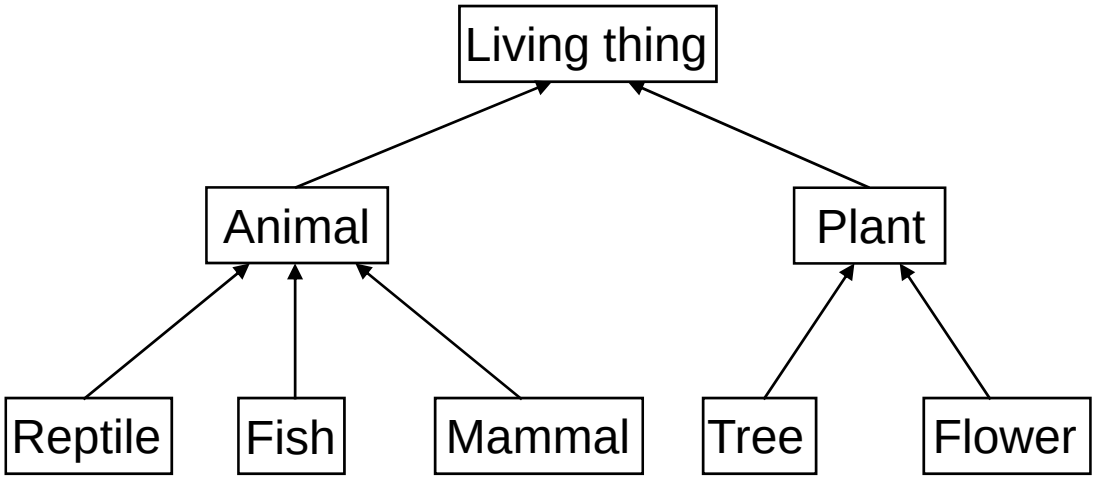
- Information (data and/or behaviour) I associate with one level in a class hierarchy is automatically applicable to lower level of the hierarchy
- Class hierarchies thus allow sharing of definition
- Each class refines/specializes the definition of its ancestor

Elements of OOP - Overriding

- Subclasses can alter or override information inherited from parent classes:
  - All mammals give birth to live young
  - A platypus is an egg-laying mammal

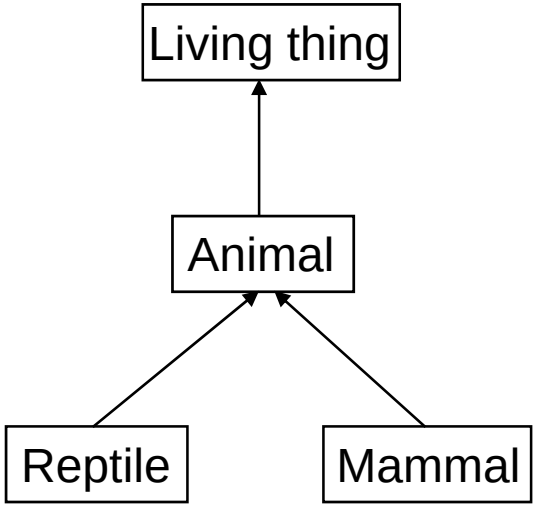
Superclass/subclass

- Classes form a hierarchy
- The focus point determines the relationship
- Superclass is the parent and subclass is a child
- Subclasses specialize their superclass



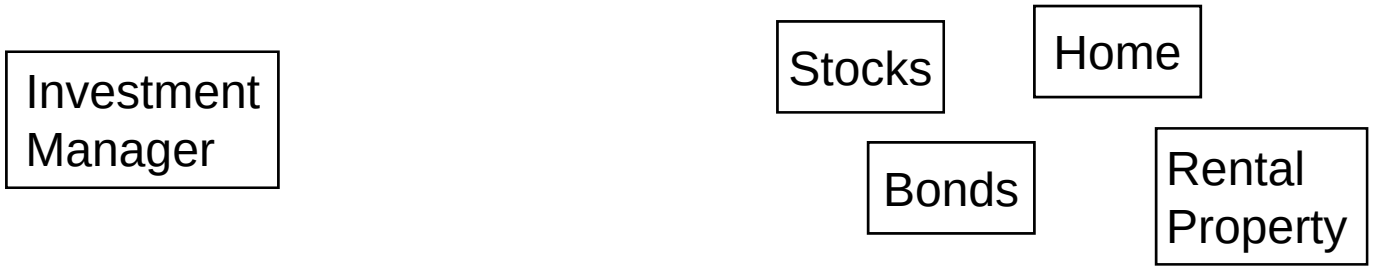
Concrete vs. Abstract Classes

- Abstract Class
  - Holds on to common characteristics shared by other classes
  - Not expected to have instances
- Concrete Class
  - Contains complete characterisation of actual objects
  - Expected to have instances



Example:The Investment Manager

- Many activities for each investment, e.g.,
  - Calculate current value
  - Calculate tax liability
- Nature of activity depends on:
  - Kind of investment
  - How long investment has been held
- What has happened during a particular period



The OO Solution

- Have a unique class description for each kind of investment
- Each investment object will have its own instance variables
- Each investment has a calculateTaxLiability method

Stock	Bond	Rental Property
companyName numberofShares currentPrice	issuerName interestRate purchasePrice	location rentalRate purchasePrice
calculateValue calculateGainorLoss calculateTaxLiability	calculateValue calculateGainorLoss calculateTaxLiability	calculateValue calculateGainorLoss calculateTaxLiability

The OO Solution, ctd.

Suppose I have a list of current investments: *investmentList myList*

Code to generate a tax report might look like:

(message) myList.calculateAllTaxes

(method) investmentList

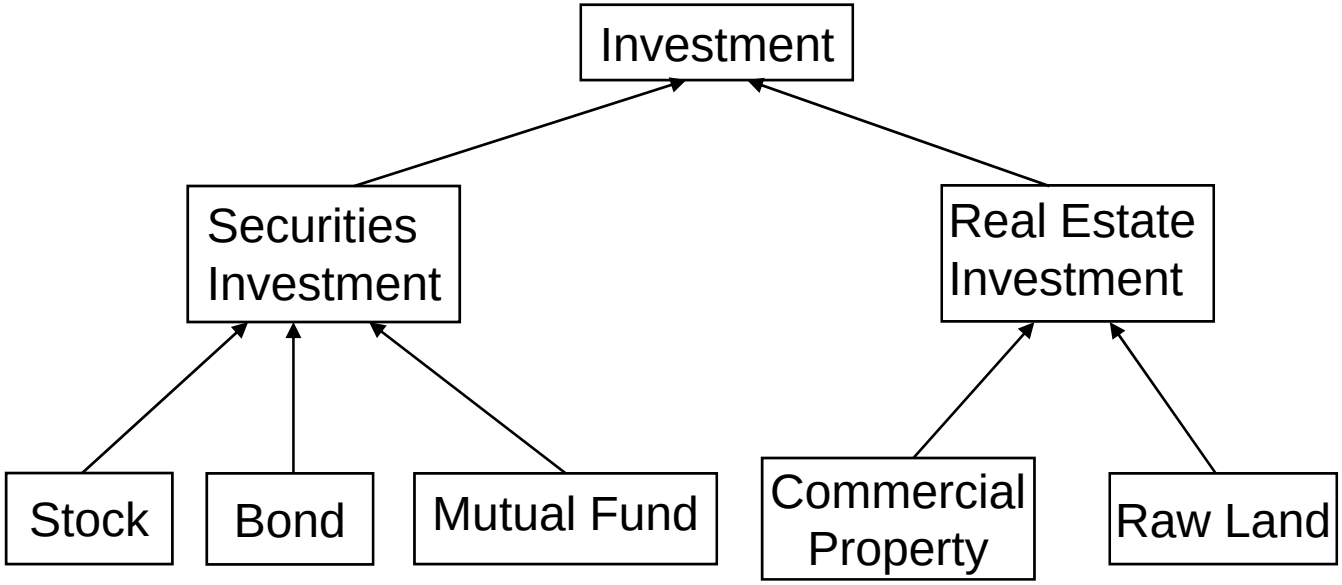
for loop: eachInvestment.calculateTaxLiability

The OO Solution, ctd.

- Suppose we wish to extend with Mutual funds and Commercial Land
- What do you do?

- Create new class description for each new type of investment

The OO Solution: a hierarchy



A method like calculateTaxLiability will move up in the hierarchy to the investment Class and will be overridden in the subclasses.

Coding of Key Elements: Classes

- Start with generic class definition
- Elucidate with examples:
  - We start with defining Cards in a card game
  - Burito recipe example explained in book
  - Shape example for *inheritance*

Generic class definition

```
Class ClassName {
    //properties
    int property1;
    float property2;

    //constructors
    ClassName(){}
    ClassName(int prop1,float prop2){
        property1 = prop1;
        property2 = prop2;
    }
```

Generic class definition,ctd

```
    //methods
    void setProperty1(int prop1){
        property1 = prop1;
    }
    int getProperty1(){
        return property1;
    }
    …

} //class ends
```

Card Example

```
Class Card {
    //static values for colors and suits
    final public int red = 0;
    final public int black = 1;
    final public int spade = 0;
    final public int heart = 1;
    final public int diamond = 2;
    final public int club = 3;

    // data fields
    private boolean faceup
    private int r;
    private int s;
```

Public and Private view

- Public view: those features (data or behaviour) that other objects can see and use.
- Private view: those features (data or behaviour) that are only used within the object.
- In **java** or **processing** keywords *public* and *private* are applied individually to every instance value or method
- If you add the keyword ***final***, it becomes a constant.

Card example, ctd.

- Constructors tie together creation and initialisation, method with same name as class name.

```
// constructor
Card(int sv,int rv) {s=sv; r=rv; faceup=false;}
// access attributes of card
public int getSuit(){return s;}
public int getRank(){return r;}

public int getColor(){
    if(suit()==heart||suit++diamond)
        return red;
    return black;}

public boolean faceUp(){return faceup;}
```

Card example, ctd.

- Classes can have multiple constructors

```
// constructor
Card(int sv,int rv) {s=sv; r=rv; faceup=false;}
// an alternative
Card() {s=0;r=3;faceup=false;}
```

- Depends on the number and type of parameters

Card example: instance creation

- How do I create an object?

Card aCard = new Card(0,4);

- The variable aCard is assigned a reference to the newly created Card object

Card standardCard = new Card();

- Uses the second constructor

Card example: “this”

```
private int r;
private int s;

Card(int sv,int rv) {this.sv=sv; this.rv=rv; faceup=false;}
```

- “this” refers to the class itself

Card example: methods

- After constructor(s) list of methods follow

```
public int getSuit(){return s;}
public int getRank(){return r;}

public int getColor(){
    if(suit()==heart||suit++diamond)
        return red;
    return black;}
```

Card example: methods, ctd

- Suppose we have magic cards

```
public void setSuit(int sv){this.sv=sv;}
public void setRank(int rv){this.rv=rv;}
```

Why *void*?

The Shape example: inheritance

- A note on inheritance in Java:
  - A single root class: *Object*
  - All classes inherit from some class, default *Object*

Shape example, ctd.

```
class Shape {
	//class properties
	int x;
	int y;
	int w;
	int h;
	//constructors
	Shape(){}
	Shape(int x, int y, int w, int h){
		this.x=x;
		this.y=y;
		this.w=w;
		this.h=h;
	}
}
```

Shape example, ctd

```
class Polygon extends Shape{
   int pts;
   //constructor
   Polygon(int x, int y, int w, int h, int pts){
        super(x, y, w, h);
        this.pts = pts;
   }
   // method to draw poly, see book
```

Shape example, ctd.

- Keyword *extends* to create subclass
- Keyword *super* refers to superclass

Shape example, ctd.

- Create a program
- Add following setyp function along with the two classes

```
void setup(){
    size(400,400);
    background(50);
    smooth();
    Polygon p = new Polygon(0, 0, 175, 175, 8);
    translate(width/2, height/2);
    p.create();
}
```

Common Design Flaws

- **Direct modification:** classes that make direct modification of data values in other classes are a direct violation of ***encapsulation.***
- **Too much responsibility:** Classes with too much responsibility are difficult to understand and use. Responsibility should be split into smaller meaningful packages.
- **No responsibility:** Classes with no responsibility serve no purpose. Often arise when designers equate physical existence with logical design existence. “Money is no object”.

Common Design Flaws, ctd.

- **Classes with unused responsibility:** Usually the results of designing software components without thinking about how they will be used.
- **Misleading names:** Names should be short and unambiguously indicate what the responsibilities of the class involve.
- **Inappropriate inheritance:** Occurs when subclassing is used in situations where the concepts do not share an "*is-a*" relationship.