Creative Programming

Object Orientation

TU/e Technische Universiteit Eindhoven University of Technology

Where innovation starts

Overview of this lecture

Thinking Object Oriented:

  - What is Object-Oriented Programming?
  - Key Elements
- Example: Car
- Coding of Key Elements:
  - Classes
  - Methods and Messages
  - Inheritance
- Common Design Flaws

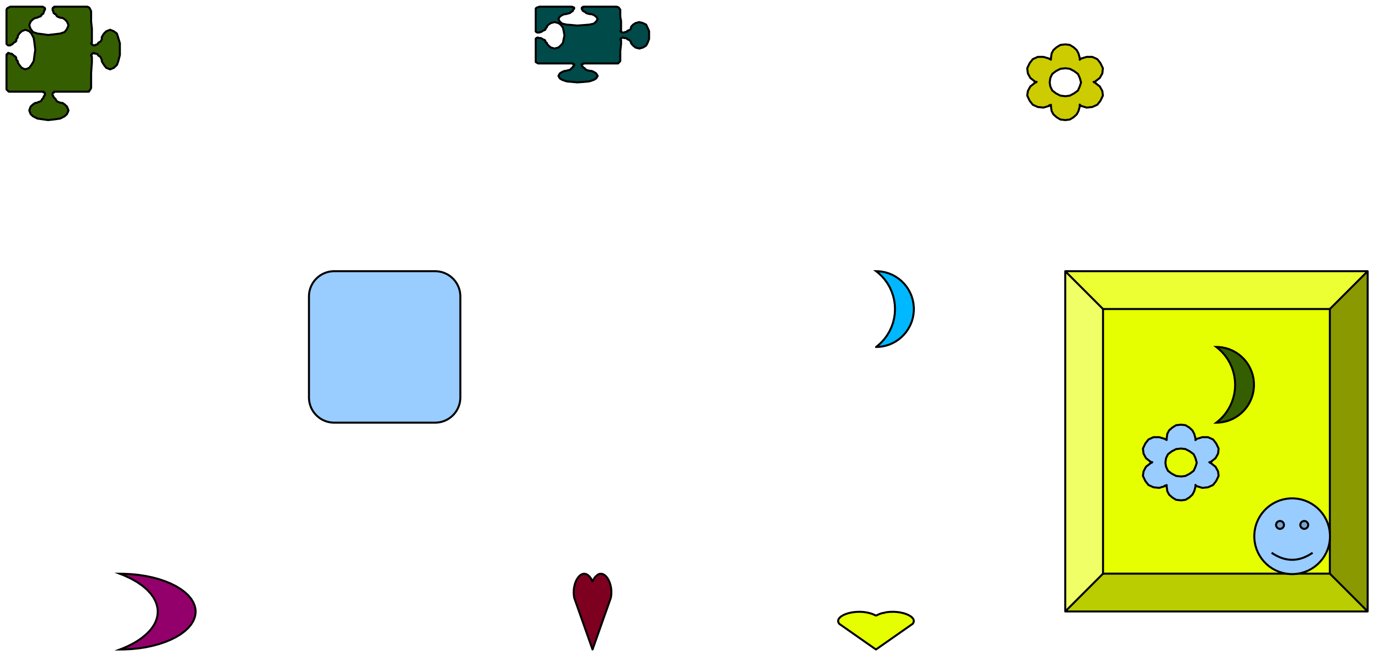
What is Object-Oriented Programming?

- OOP is a revolutionary extension of programming
- OOP extends earlier programming abstractions
- It shows resonant similarity to techniques of thinking about problems in other domains (e.g. Architecture)
  (a way of looking at situations .. to simplify dealing with those situations .... e.g. organizing information)
- It is the leading programming paradigm
- A paradigm is a set of theories, standards, and methods that together represent a way of organising knowledge – that is, a way of viewing the world.

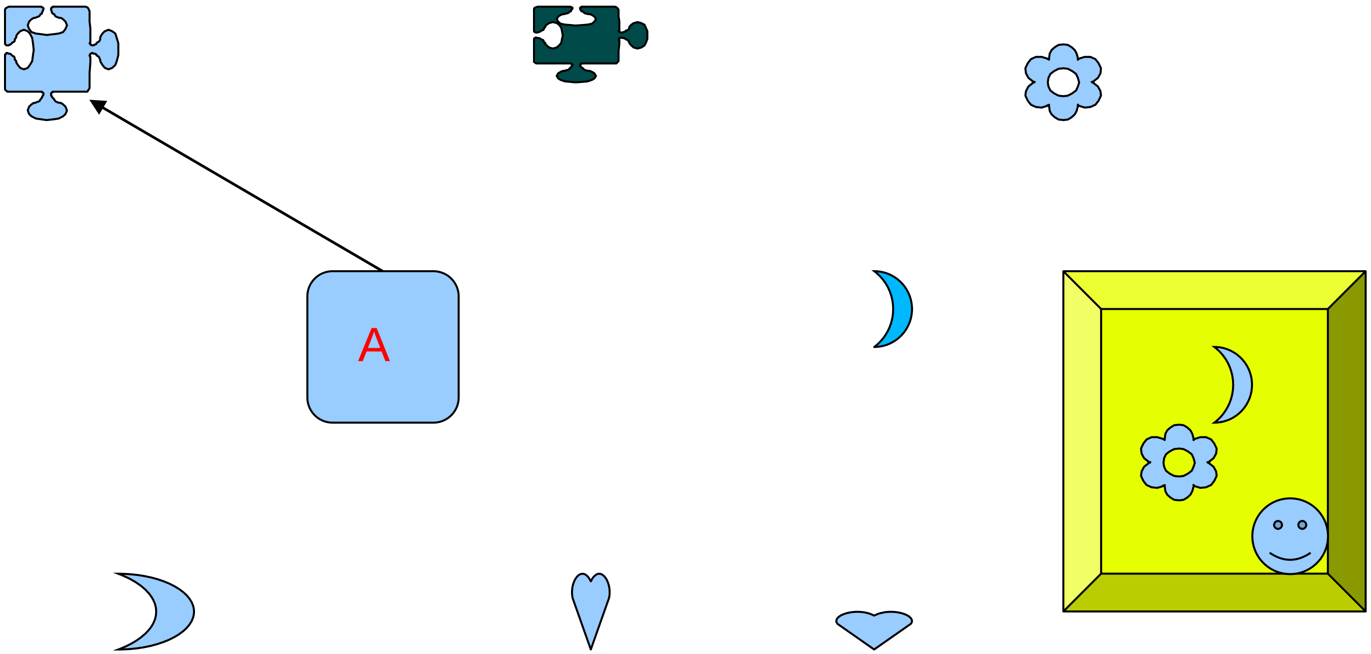
Basic ideas

- Program consist of many “things”. (**objects**)
- There are different kinds of “things”
- Objects are created as instances of **classes.**
- Objects can have an internal state and **components**.
- Objects exchange **messages.**
- If object A sends message to B then B does something and then returns a **result** to A.
- Results can be **int , float** or **string** or they can be an **object** themselves or there is no result. (**void**).
- There is some main object with a loop that starts everything off.

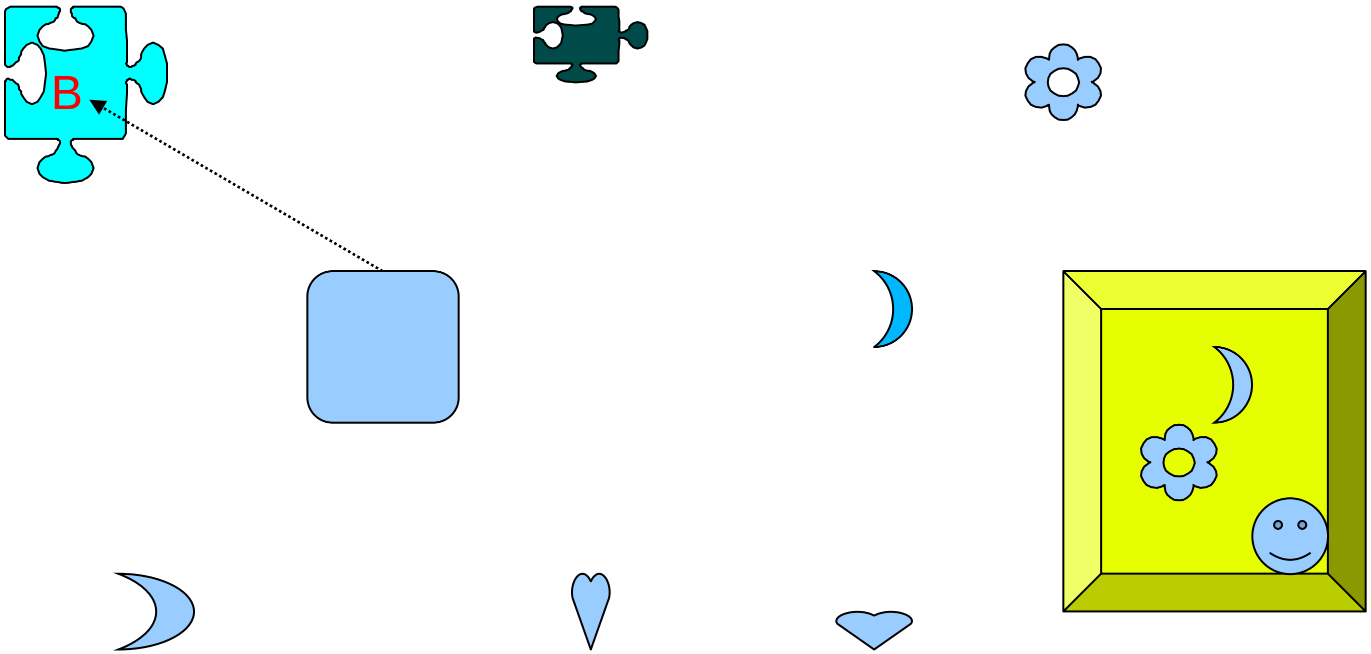
Program : A world of objects



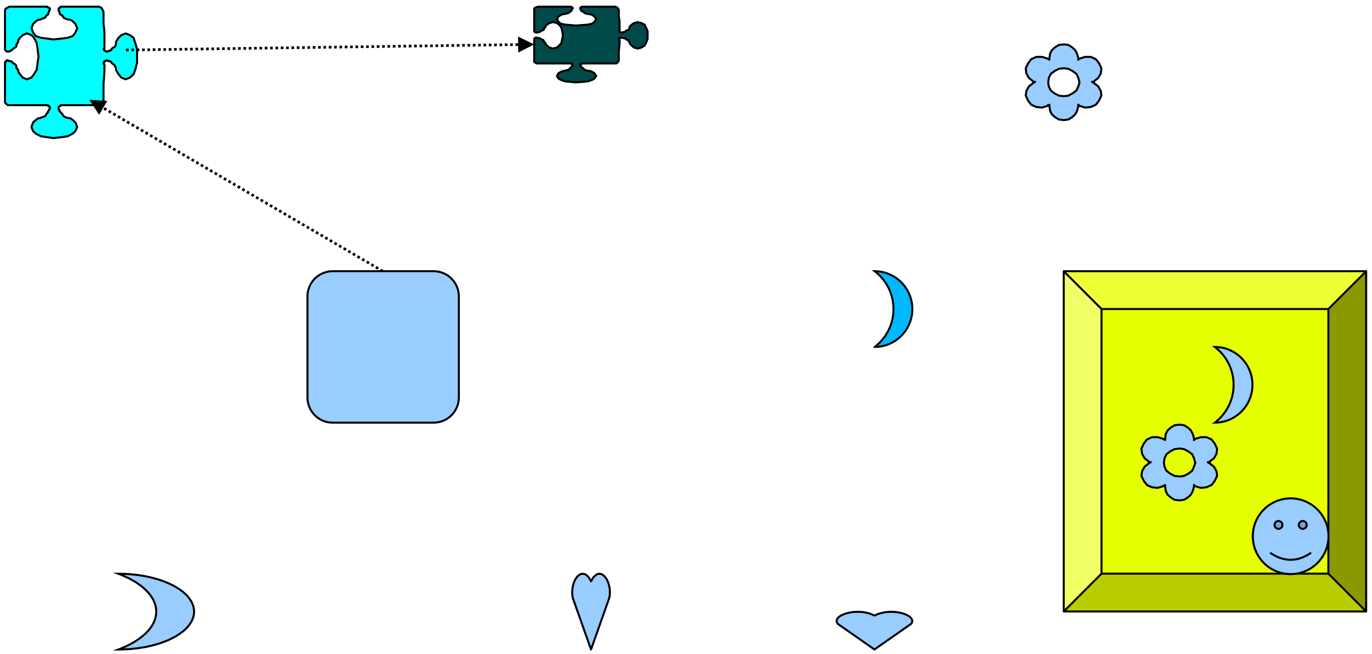
Active object A sends a message ..



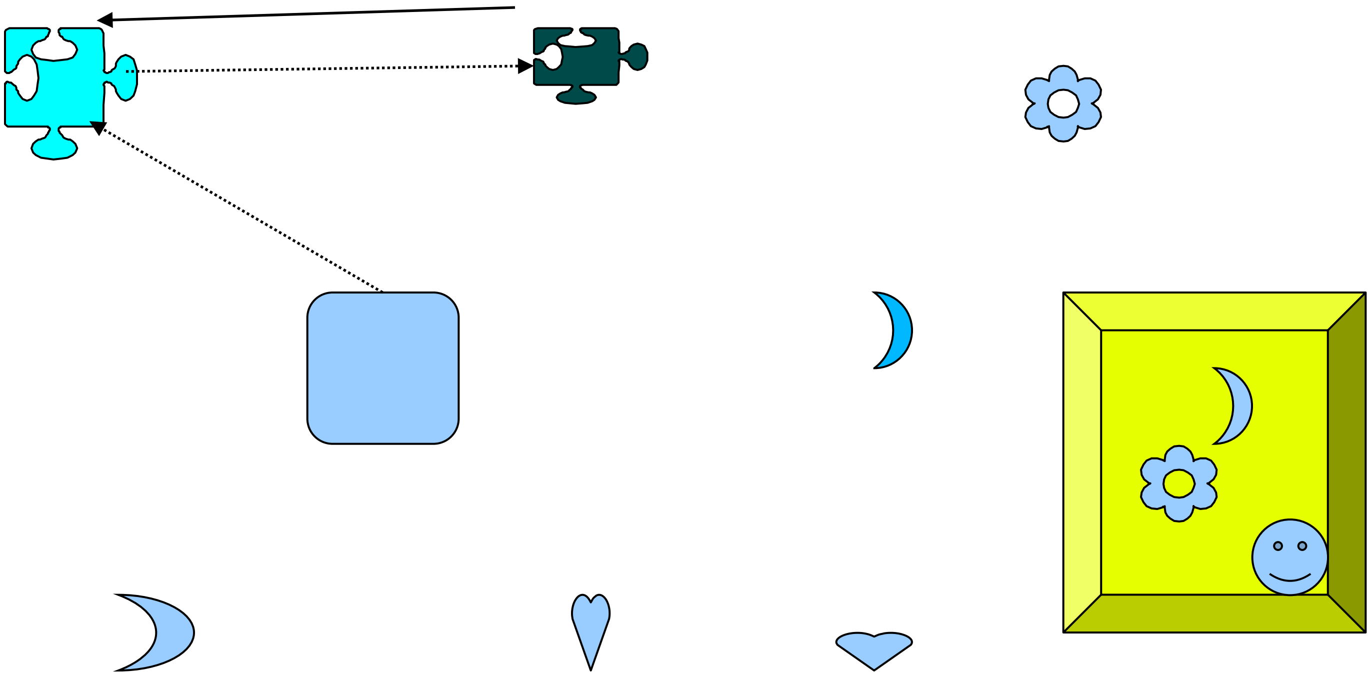
Receiver B is activated …



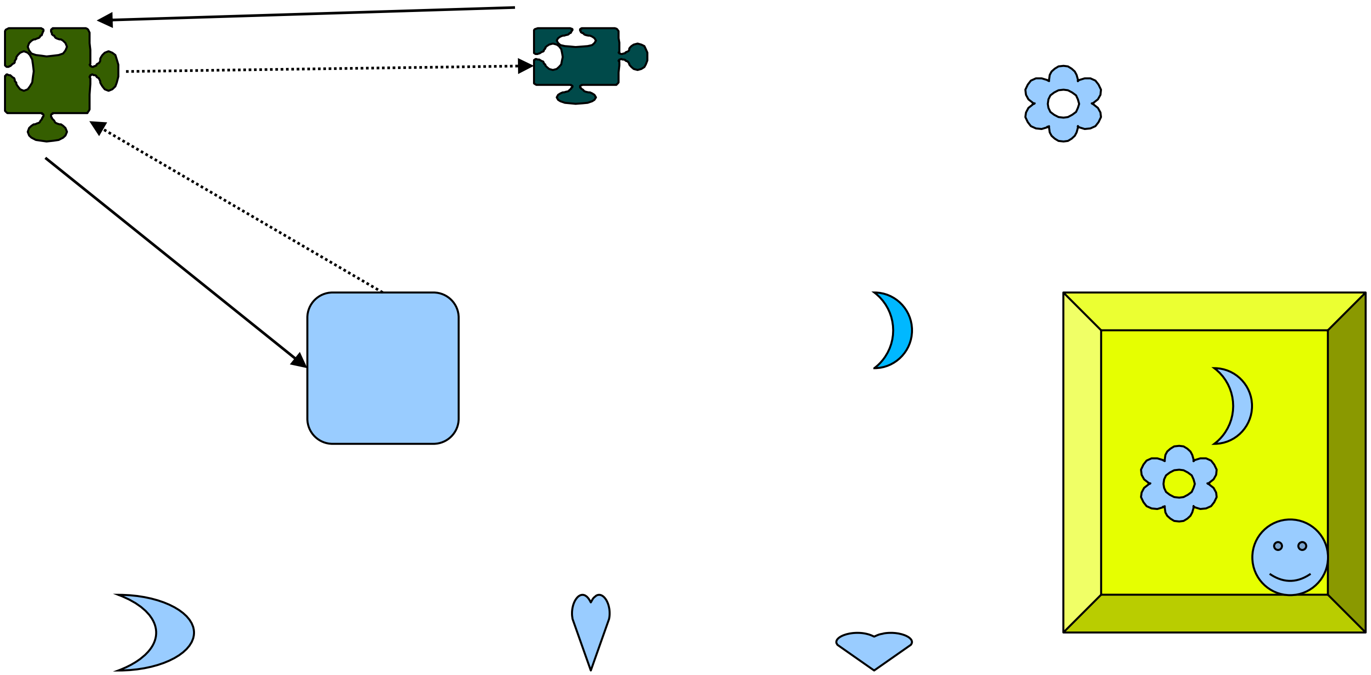
When active B can send a message …



B gets back result of message …

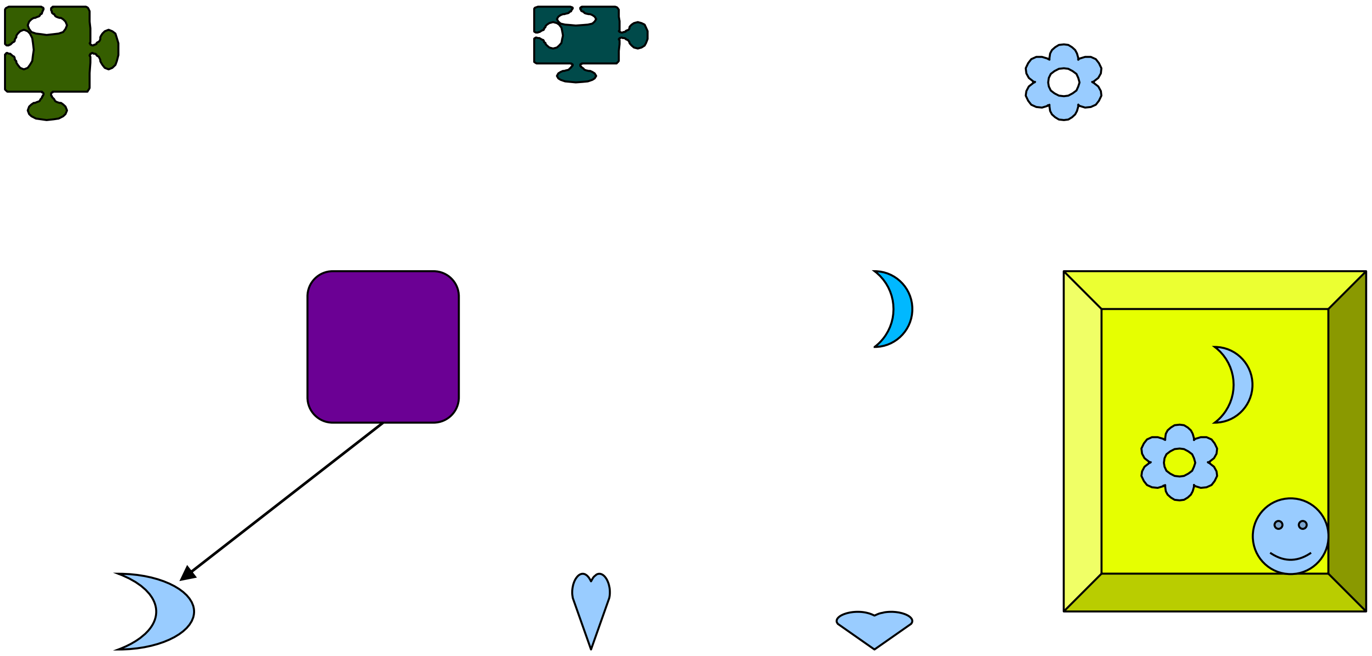


And eventually returns a result to A



Dept. of Industrial Design

Object A, now active again …may send new message …



All about objects and messages ....

- **OOP is based on the principle of *recursive design***
  - **Every thing is an object**
  - **Objects perform computation by making requests of each other through the medium of messages**
  - **Every object has it's own memory, which can consist of other objects**

Organised through classes

- Every object is an instance of a class. A class groups similar objects
- The class is the repository for behaviour associated with an object
- Classes are organised into tree structures, called inheritance hierarchies.

Elements of OOP

1. ***Every thing is an object***

Actions in OOP are performed by active objects.

(…. similar objects are found in the same class)

Elements of OOP-Messages

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

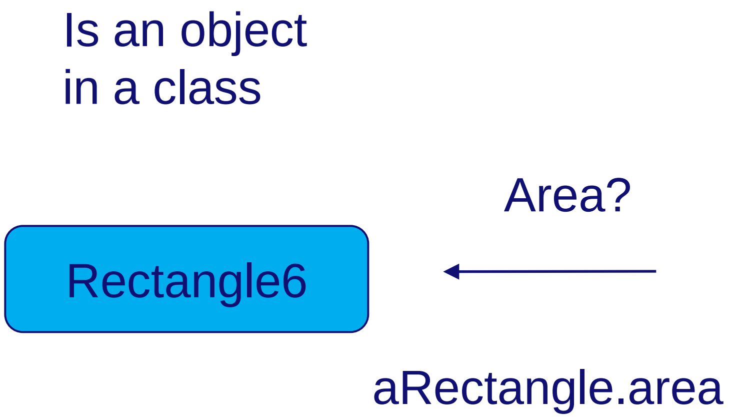
Actions in OOP are produced in response to requests for actions, called messages. An instance may accept a message and in return it will perform an action and return a result.

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 (found in objects that handle message)
  - 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
- Depends on class of receiver …
- Has name that is the same as message name
- Is a sequence of executable statements
- Returns a result of execution

Float area
Return side1*side2

Information hiding (behind curtain)

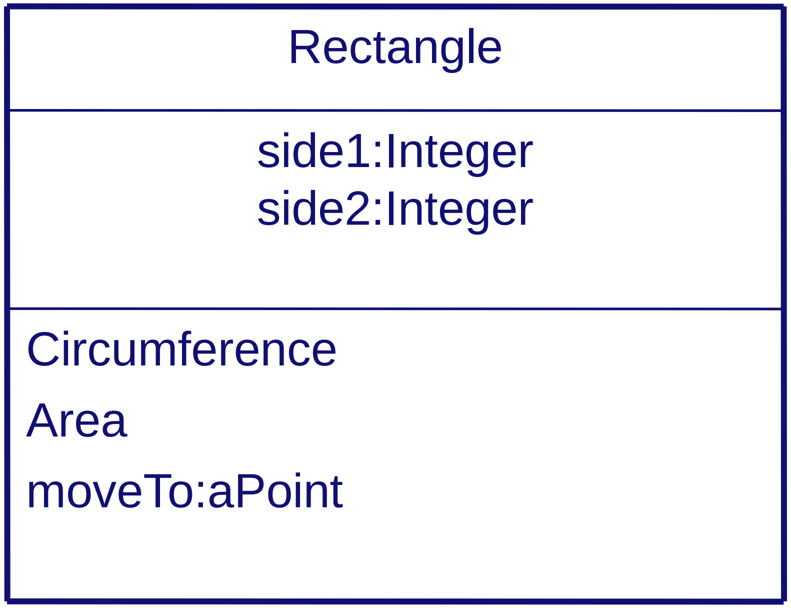
- As a user of a service being provided by an object, I need only to know the set of messages that the object will accept. I need not to have any idea of how the methods are performed.

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

External perspective	Internal perspective
What	How
Message	Method

Object Encapsulation

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



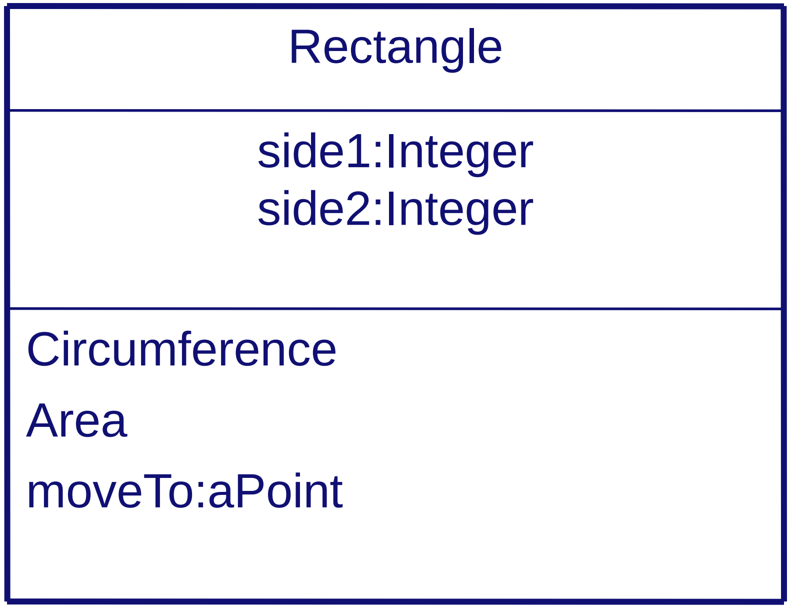
Rectangle

side1:Integer
side2:Integer

Circumference
Area
moveTo:aPoint

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

- objects:
  Florist Flo;
  Secretary Beth;
  Dentist Ken;
- messages:
  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 variables and 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”

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

Example :   class Car

- class Car { // Cars can drive and can be drawn on screen
- color c;
- int carlength ;
- float xpos;
- float ypos;
- float speed;
- int   tirewidth ;

- Car(){   //this is a constructor for a default Car
- c = color(223,34,45);
- xpos = 23;
- ypos = 34;
- carlength = 120;
- tirewidth = 12;
- }

Card example: instance creation

- How do I create an object with a constructor?

Car myfirstCar = new Car();

- The variable aCar is assigned a reference to the newly created Car object

- Uses the first constructor, there may be more compelx constructors ..

```
Car( color desiredColor  ){   //  constructor for coloured car
 c =  desiredColor
 xpos = 23;
 ypos = 34;
 Carlength = 120
tirewidth = 12;
}

 void Drawcar(){     // method to draw a car on screen
 ellipse(xpos,ypos, carlength,10);
 rect(xpos, ypos-10, 10,-tirewidth);
 rect(xpos,ypos+10, 10, tirewidth);
}
```

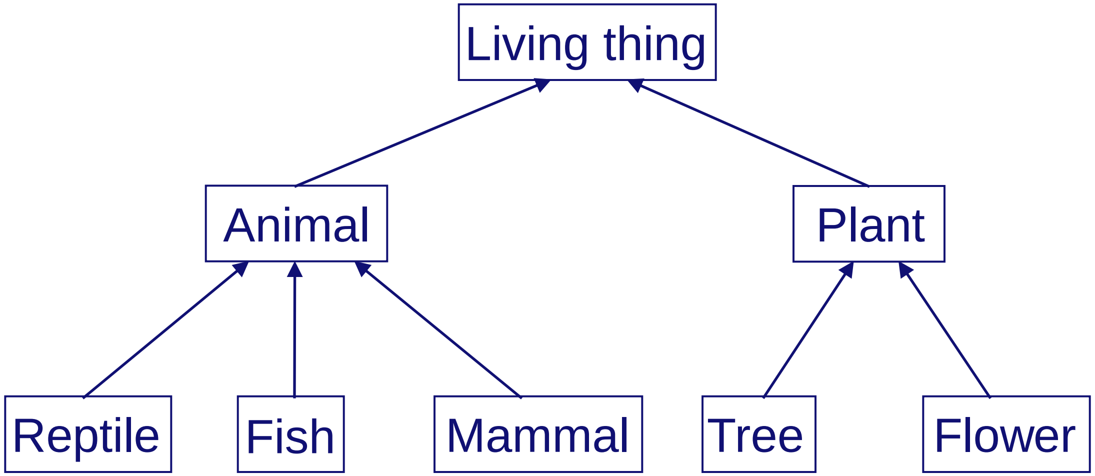
Coding of Key Elements: Classes

- **Elucidate with examples:**
  - **We start with defining Cars in a race game**
  - **Extend Car example to explain *inheritance***

  - ***generic class definition***

Superclass/subclass

- **Classes form a hierarchy**
- **Superclass is the parent and subclass is a child**
- **Subclasses “extend” (i.e. Specialize) their superclass**

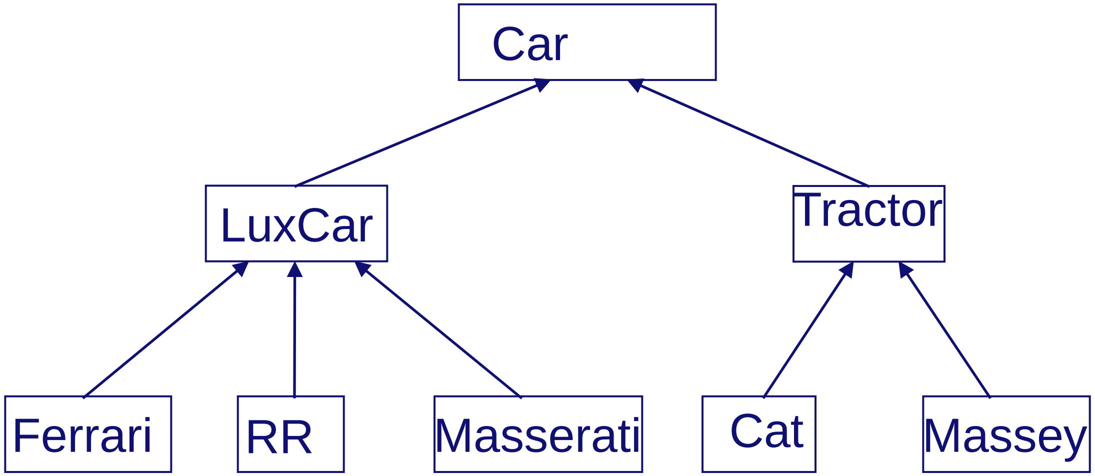


Elements of OOP - Overriding

- **Subclasses can alter or override information inherited from parent classes:**
  - **All mammals give birth to living young**
  - **All fish have gills**

Superclass/subclass

- Classes form a hierarchy
- Superclass is the parent and subclass is a child
- Subclasses “extend” (i.e. Specialize) their superclass



- void drive()   // method for Car to drive in x
- // direction
- { xpos = xpos + 3;}

- LuxCar extends Car() // Luxcars are cars with
- // somewhat better properties

- void drive() { xpos = xpos + 4;}

Generic class definition

```
class ClassName {
    //properties or components
    int property1;
    float property2;
    rectangle component3;

    //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;
    }
    …
other ... specific  methods

} //class ends
```

Inheritance in Java

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

Public and Private view

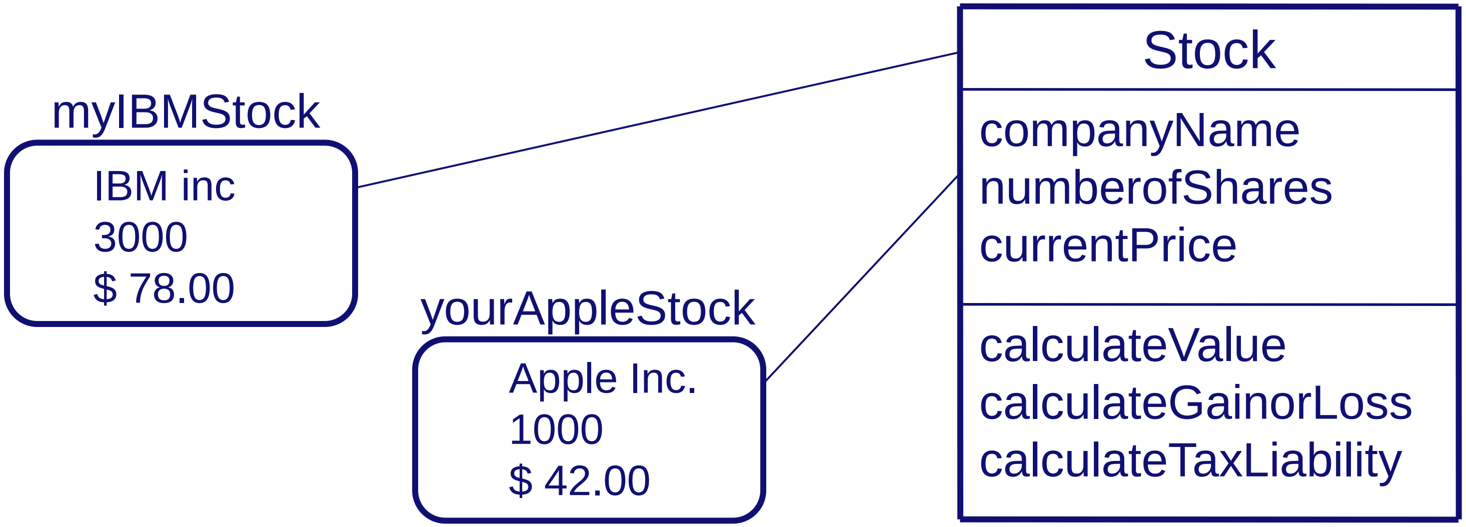
- **Public view: those features (data or behaviour) that other objects can see and use.**
- **Private view: those features (data or behaviour) that are only used within the object.**
- **In** java **or** processing **keywords *public* and *private* are applied individually to every component or method**

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".**
- 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.**

Instance

- **Instance: a particular occurrence of an object defined by a class**
- **Each instance may have its own state**
- **All instances of a class share the same methods**



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 definitions**
- **Each class refines/specializes the definition of its ancestor**

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

Investment Manager

Stocks

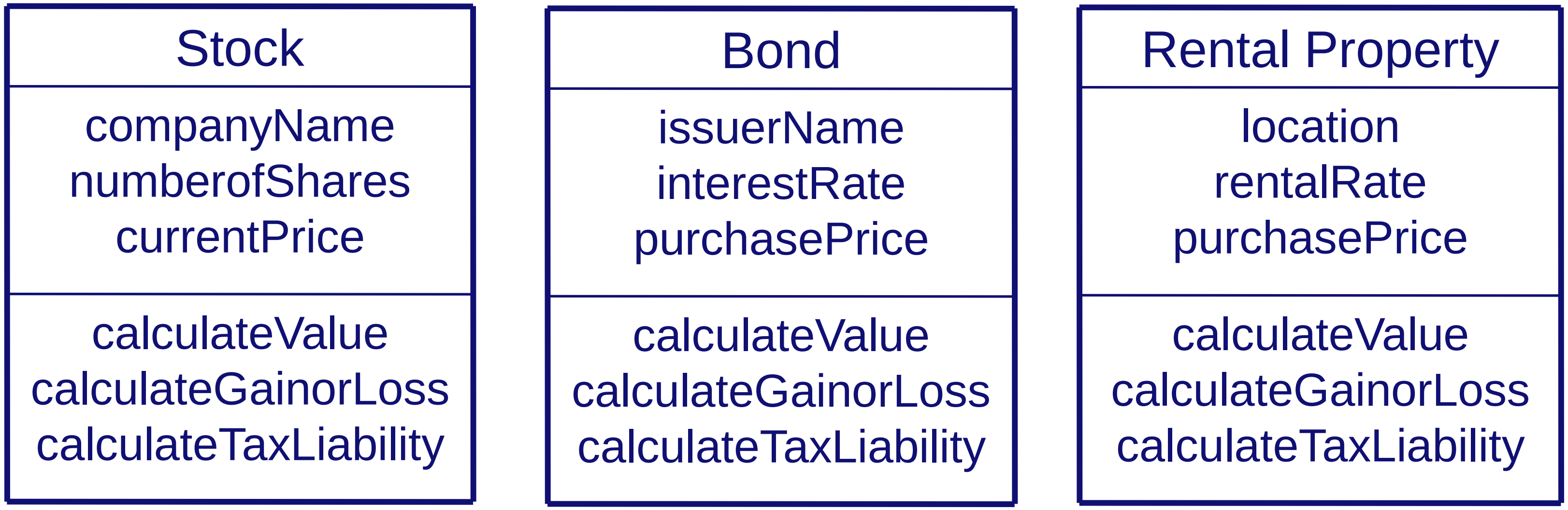
Home

Bonds

Rental Property

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

companyName
numberofShares
currentPrice

calculateValue
calculateGainorLoss
calculateTaxLiability

Bond

issuerName
interestRate
purchasePrice

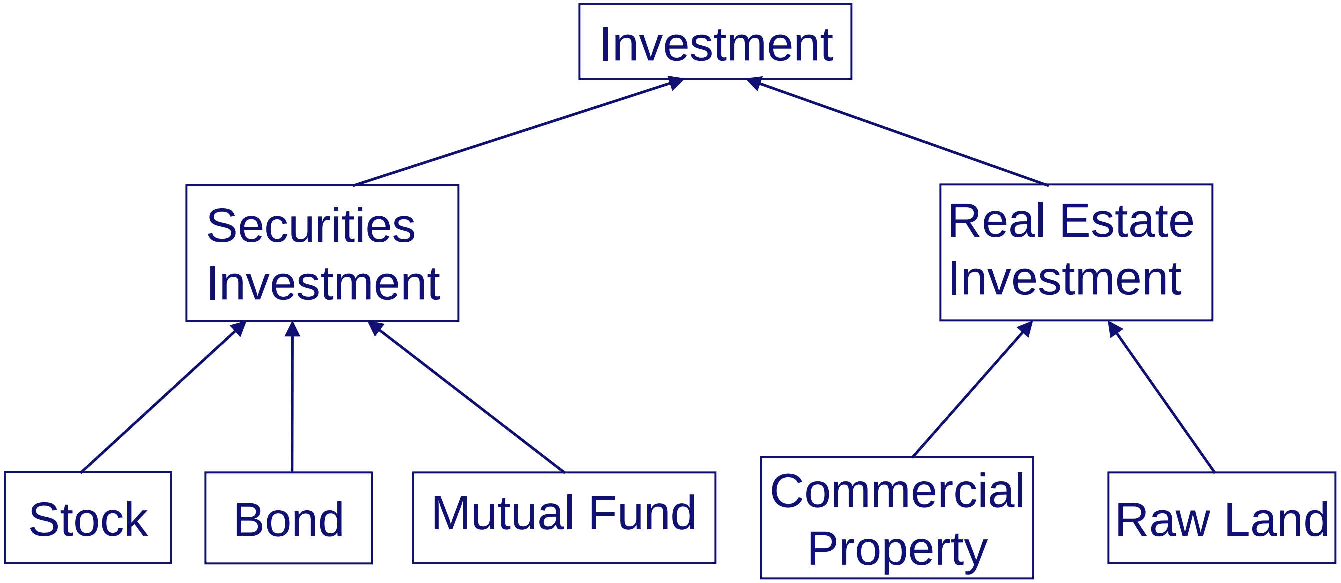
calculateValue
calculateGainorLoss
calculateTaxLiability

Rental Property

location
rentalRate
purchasePrice

calculateValue
calculateGainorLoss
calculateTaxLiability

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.

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
```

```
//method to draw poly
void create(){
   float px = 0, py = 0;
   float angle = 0;
   beginShape();
   for (int i=0; i<pts; i++){
    px = cos(radians(angle))*w;
    py = sin(radians(angle))*h;
    vertex(px, py);
    angle+=360.0/pts;
   }
   endShape(CLOSE);
}
```

Shape example, ctd.

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

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