Essential Concepts

- Why OO?
- What is OO?
- What are the benefits?
- What are the KEY concepts?
- Basis for all the lectures

Object-Orientation

- Is a paradigm not a technology
- Reflects, simulates the real world
- Thinks in terms of organization
- Tries to
  - Handle complexity
  - Enhance reusability
  - Minimize maintenance cost

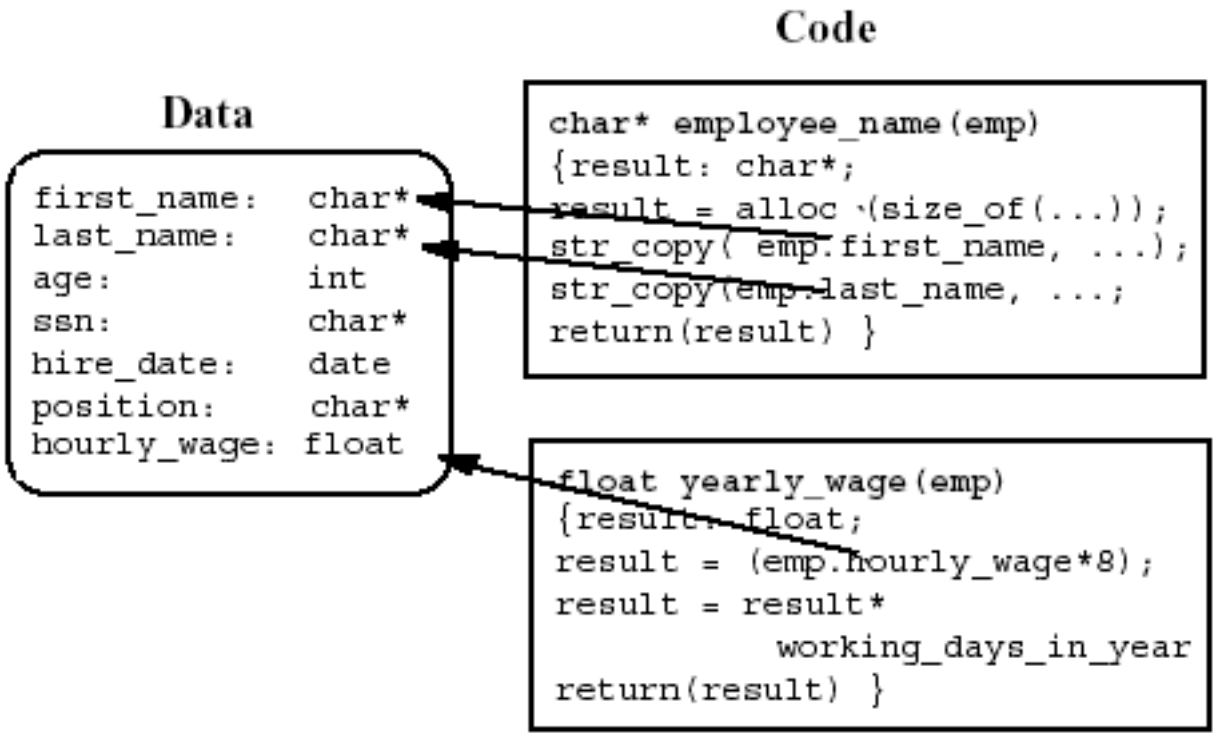
Evolution

- Procedures
- Structured Programming
- Fourth Generation Languages
- Object-Oriented Programming
- ???

Traditional ViewPoint

- Focuses upon procedures
- Functionality is vested in procedures
- Data exists solely to be operated upon by procedures
- Procedures know about the structure of data
- Requires large number of procedures and procedure names

Data and Procedures



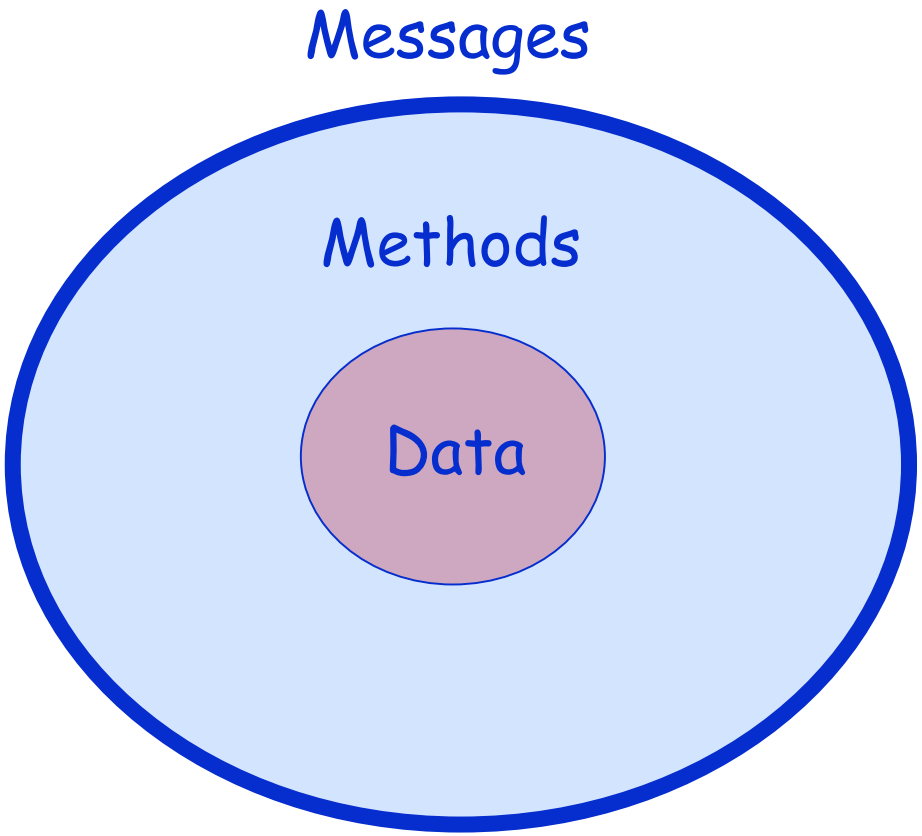
What is OOP?

- An application is a collection of interacting entities (objects)
- Objects are characterised by behaviour and state
- Inter-object behaviour needs to be coordinated
- Inter-object communication is the key to coordination

Object-Oriented Viewpoint

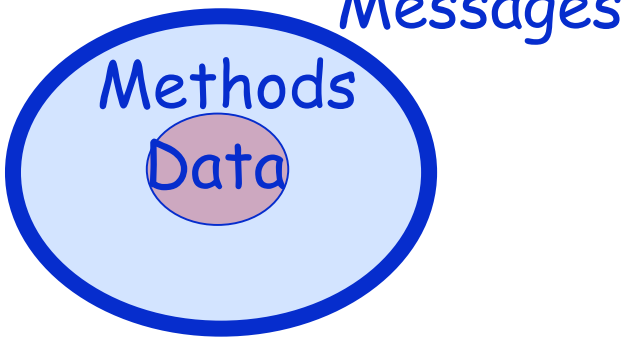
- An **application** is a set of **objects** interacting by sending **messages**
- The functionality of an object is described by its **methods**, it's data is stored in private **variables**
- An object's functionality can be invoked by sending a **message**
- **Everything** is an object

Data/Messages/Methods



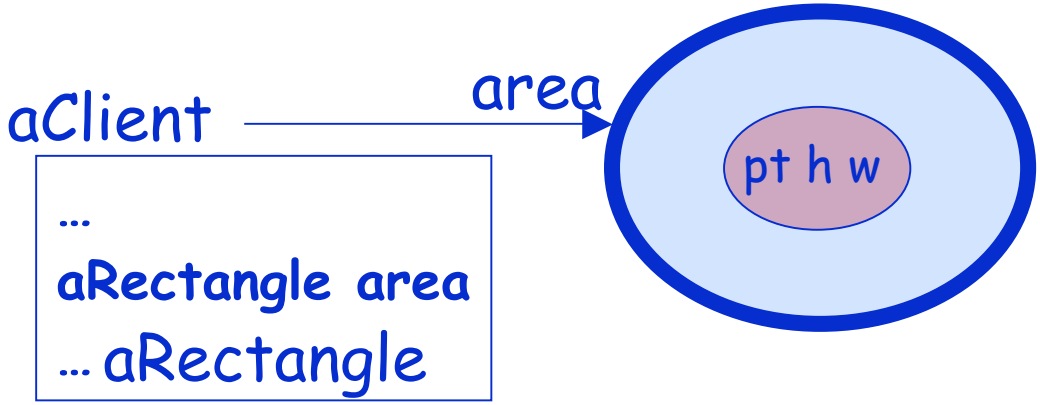
What vs How

- What: Messages
  - Specify what behavior 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 (contain or be passed) data
  - Need detailed knowledge of data
  - Can manipulate data directly



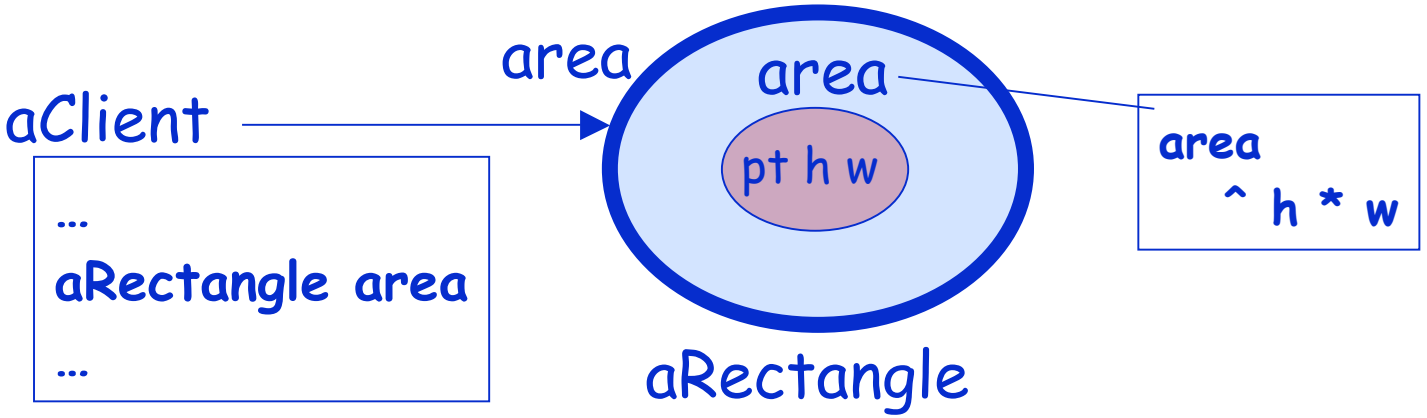
Message

- Sent to receiver object: receiver-object message
- A message may include parameters necessary for performing the action
- In Smalltalk, a message-send always returns 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

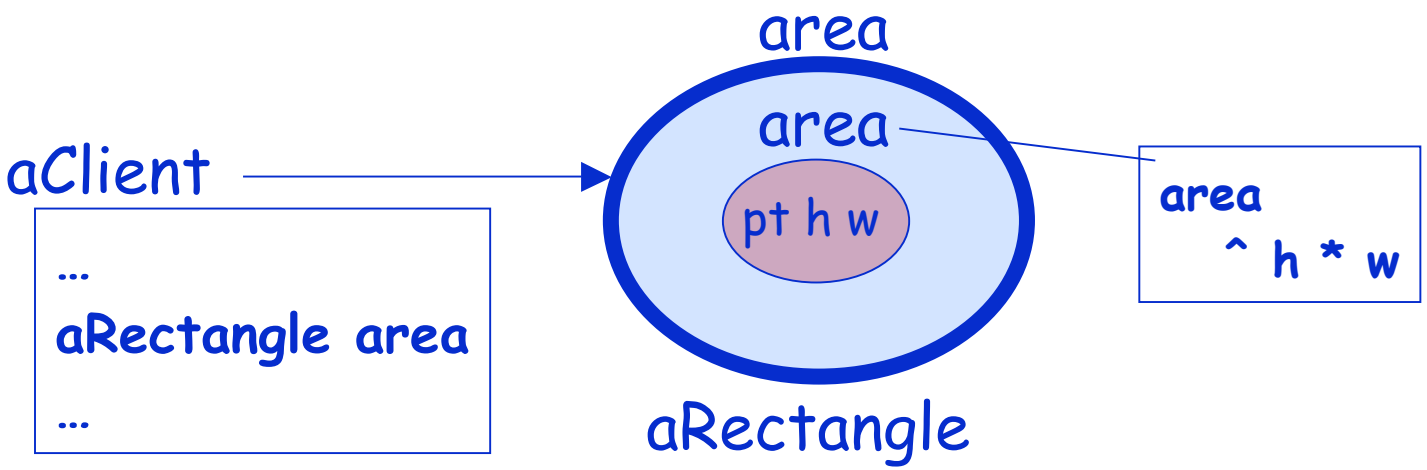


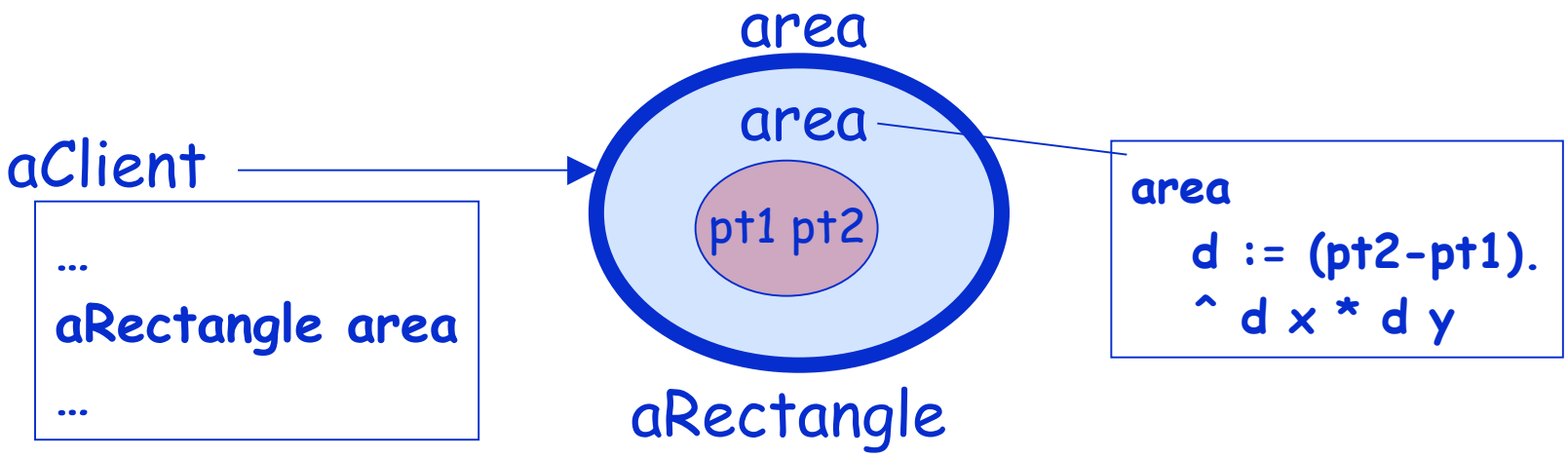
Object Encapsulation

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

External perspective	vs.	Internal perspective
What	vs.	How
Message	vs.	Method

Encapsulation at Work



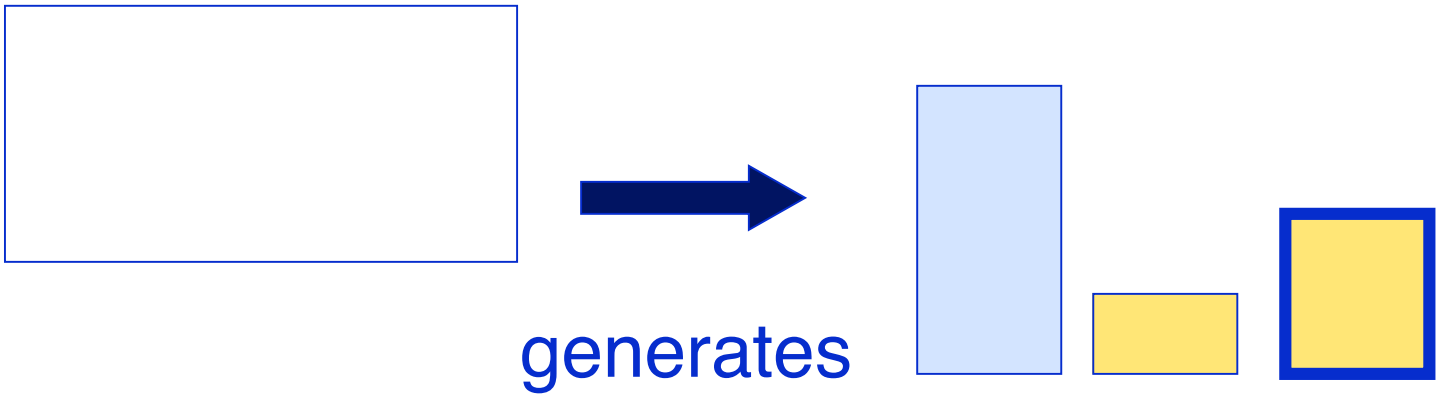


Objects

- Unique identity
- Private state
- Shared behavior among other similar objects

Class: Factory of Objects

- Reuse behavior
  => Factor into class
- Class: "Factory" object for creating new objects of the same kind
- Template for objects that share common characteristics



Class: Mold of Objects

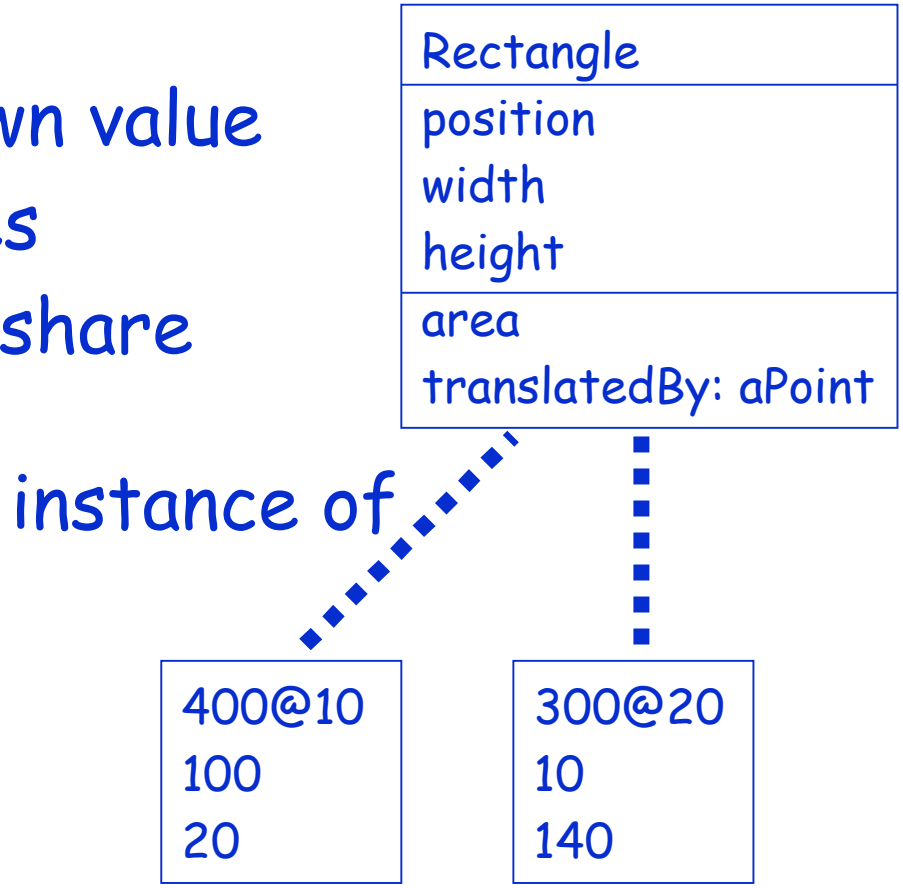
- **Describe** state but not value of all the instances of the class
  - Position, width and height for rectangles
- **Define** behavior of all instances of the class

```
area
    ^ width * height
```

Rectangle
position width height
area translatedBy: aPoint

Instances

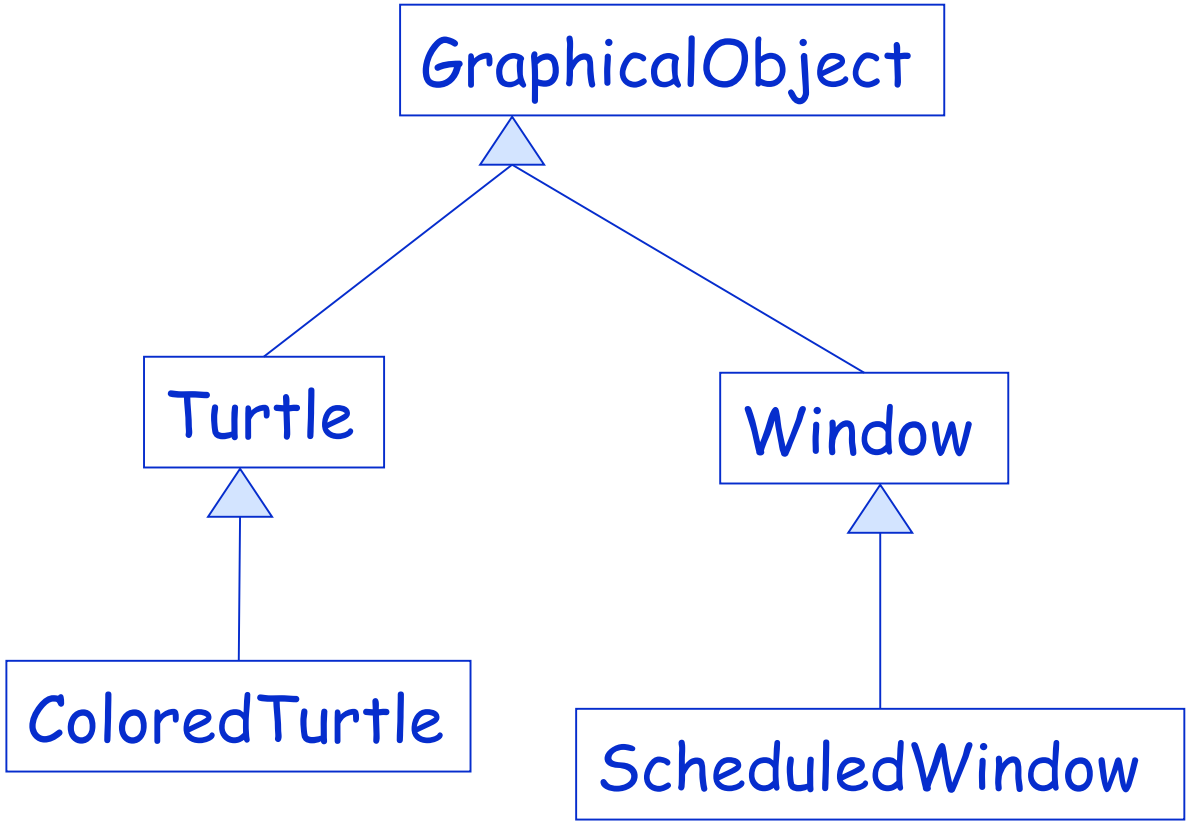
- A particular occurrence of an object defined by a class
- Each instance has its own value for the instance variables
- All instances of a class share the same methods



How to Share Specification?

- Do not want to rewrite everything!
- Often times want small changes
- Class hierarchies for sharing of definitions
- Each class defines or refines the definition of its ancestors
- => inheritance

Example

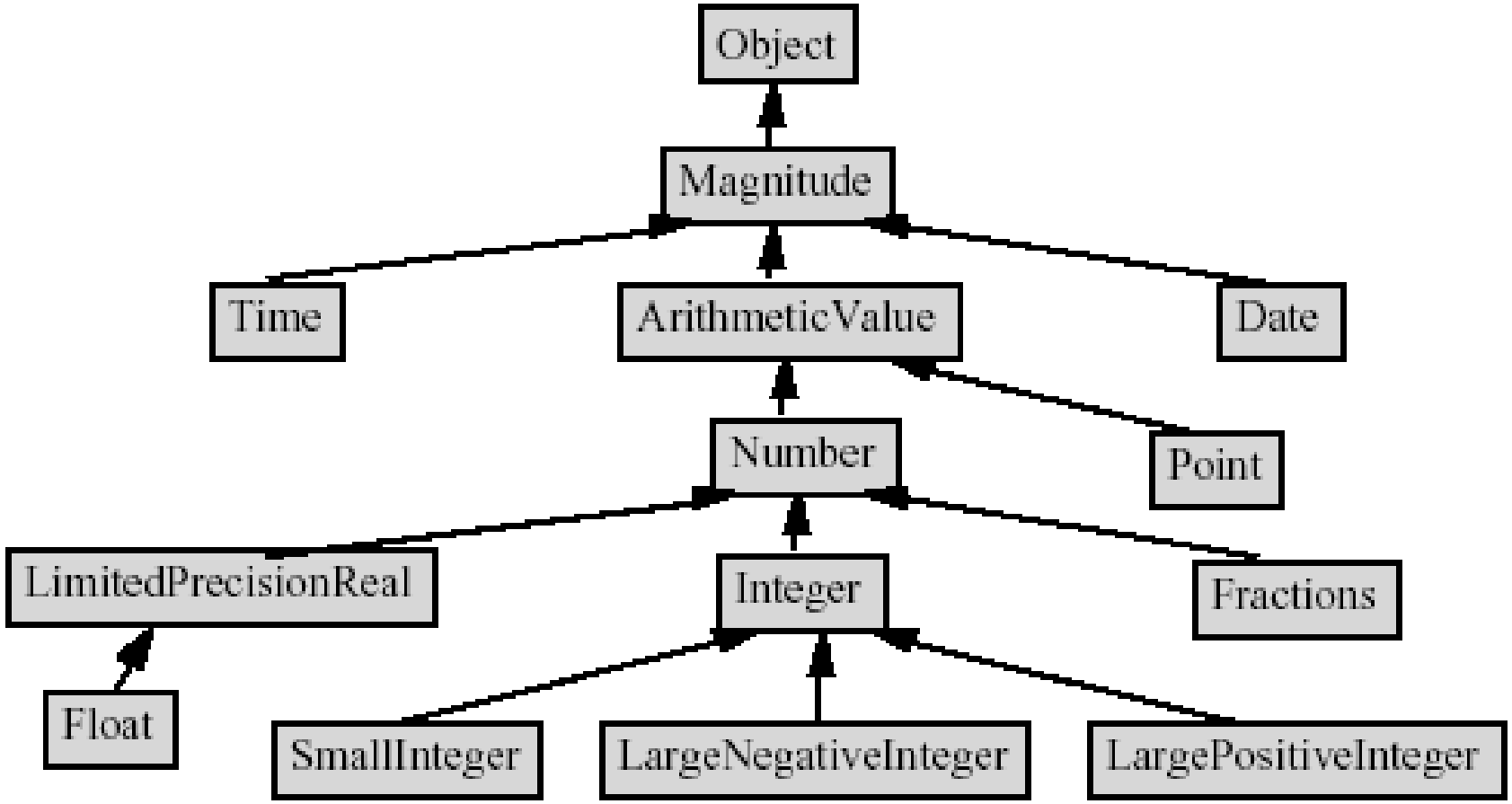

GraphicalObject
Turtle
Window
ColoredTurtle
ScheduledWindow

Inheritance

- New classes
  - Can add state and behavior
  - Can specialize ancestor behavior
  - Can use ancestor's behavior and state
  - Can hide ancestor's behavior

- Direct ancestor = superclass
- Direct descendant = subclass

Comparable Quantity Hierarchy



Polymorphism

- Same message can be sent to different objects
- Different receivers react differently (different methods)

  - aWindow open
  - aScheduledWindow open
  - aColoredWindow open

  - aRectangle area
  - aCircle area

Late binding: "Let's the Receiver decides"

- Mapping of messages to methods deferred until run-time (dynamic binding)
- Allows for rapid incremental development without the need to recompile (in Smalltalk)
- Most traditionl languages do this at compile time (static binding)

Procedural Solution for a List of Graphical Objects

```
tArea
  element = Circle
        then tArea := tArea + element.circleArea.
  element = Rectangle
        then tArea := tArea + element.rectangleArea
  ...
Intersect, color, rotate translate....
```

In Java for example

```
public static long sumShapes(Shape shapes[]) {
long sum = 0;
for (int i=0; i<shapes.length; i++) {
  switch (shapes[i].kind()) {
  // a class constant
  case Shape.CIRCLE:
       sum += shapes[i].circleArea();
       break;
  case Shape.RECTANGLE:
       sum += shapes[i].rectangleArea();
       break;
       ... // more cases
       }
  }
  return sum;
}
```

Problems of the Procedural Solution

Adding a kind of graphical element

=> Change all the methods area, intersect, rotate, translate…

=> Always have to check what is the data I manipulate

Object-Oriented Solution

```
Circle>>area
  ^ Float pi * r * r
Rectangle>>area
  ^ width * height
XXX>>area
   elements do:
        [:each | tArea := tArea + each area]
```

Advantage

- Adding a new graphical object does not require to change the list operations
- I do not have know the kind of objects I'm manipulating as soon as they all share a common interface

Recap

- OOP see the world as interacting objects

- Objects
  - have their own state
  - Share the behavior among similar objects
- Classes: Factory of objects
  - Define behavior of objects
  - Describe the structure of objects
  - Share specification via hierarchies

Recap

- OOP is based on
  - Encapsulating data and procedures
  - Inheritance
  - Polymorphism
  - Late Binding
- OOP promotes
  - Modularity
  - Reuse