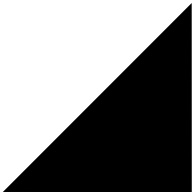


# (Object-)Interaction Diagrams



# Specifying (constraints on) Dynamics

- Class diagrams describe (constraints on) the structure of instances.
- (Object-)Interaction diagrams describe (constraints on) the behaviour of an application.
  - ◆ Not structure (static), but behaviour (dynamic)
  - ◆ Composed run-time entities : objects (no classes at run-time)
- Two kinds of object-interaction diagrams
  - ◆ Communication Diagrams
  - ◆ Sequence Diagrams

- Object-Interaction diagrams depict dynamic, run-time behaviour (between objects, not internal view!)
  - ◆ communication between objects via **messages**
  - ◆ **sequence** of transactions in a dialog between a user and a system, or between objects in a system
  - ◆ one **trace** of behaviour ideally corresponds to one use case
- With the object-interaction diagram, we introduce the notion of **time** (may be abstraction: **progress**).

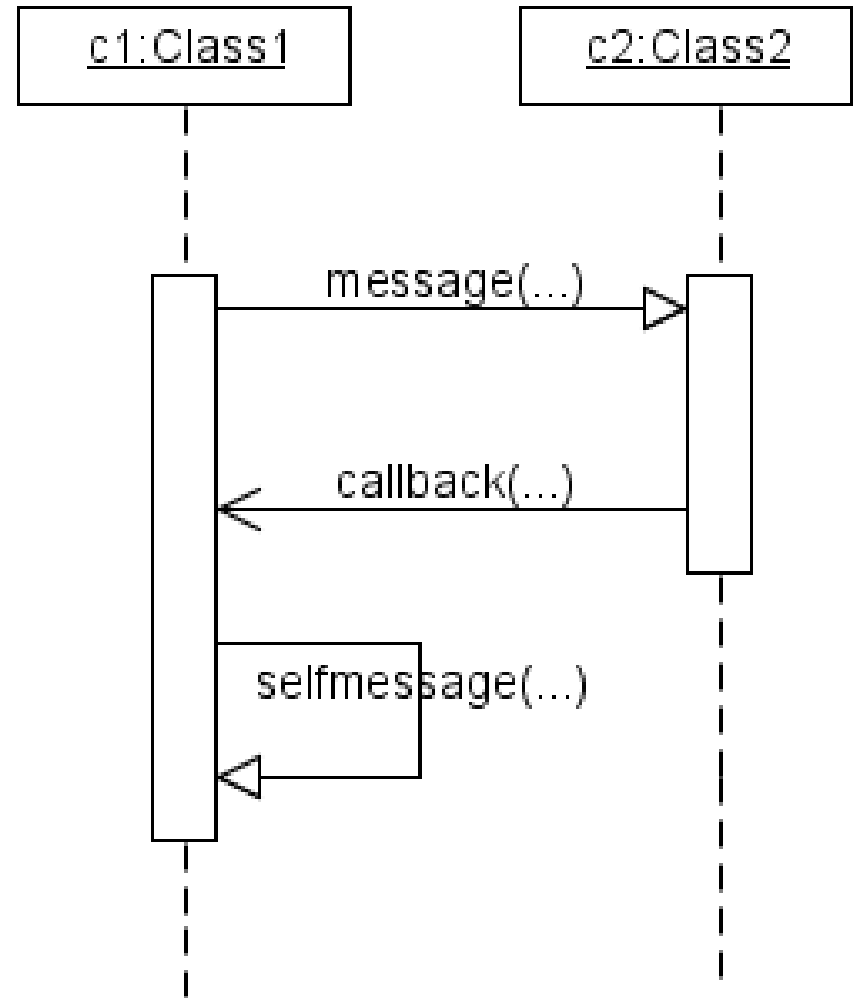
# Communication Diagrams

- Communication diagrams represent **objects** in a system and their **links**.
- They are composed of three elements:
  - ◆ Objects
  - ◆ Links (“instance” of Association)
  - ◆ Messages



# Sequence Diagrams

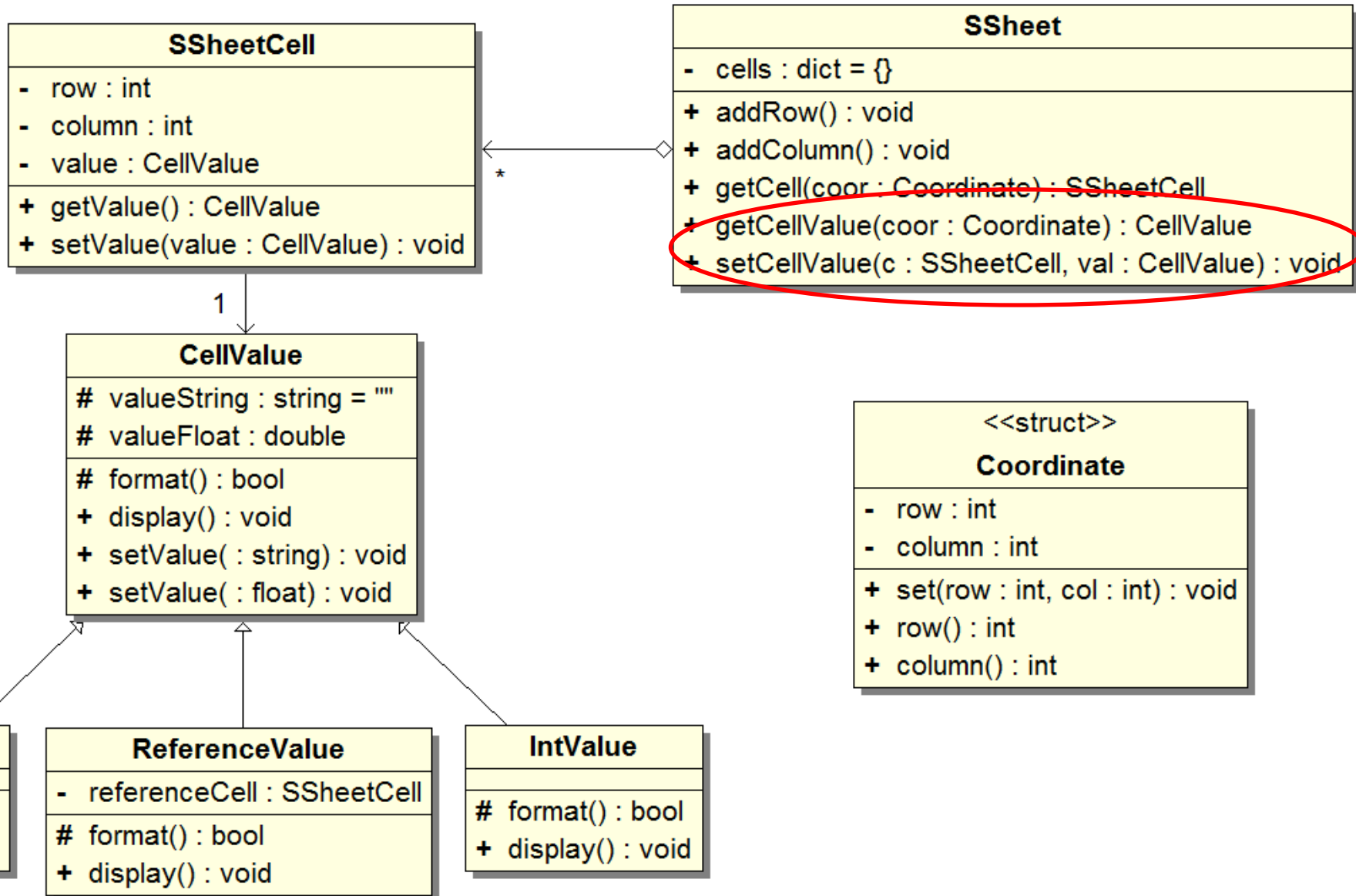
- Sequence diagrams illustrate the sequence of actions that occur in a system.
- They are composed of 2 elements
  - ◆ Object
  - ◆ Messages



# Sequence vs. Communication

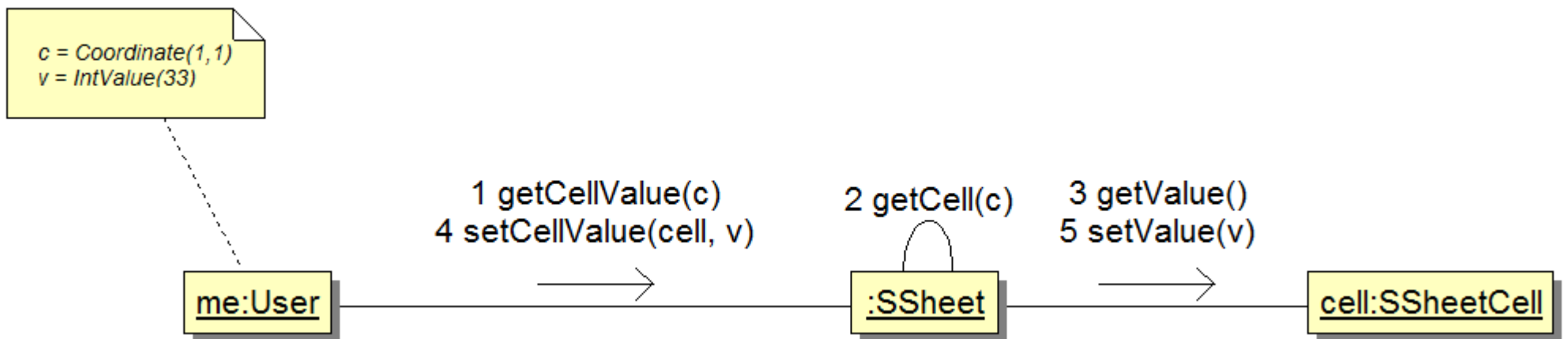
- Both diagrams specify/constrain object interaction.
  - ◆ Sequence is used to illustrate temporal interactions.
  - ◆ Collaboration is better suited to display the association between the objects.
- In principle, a sequence diagram can be converted into a collaboration diagrams (and vice-versa). Need to contain equal amount of information.

# Spread Sheet example



# Communication Diagram

- Remember, we are depicting the interaction between instances, not classes.
- ac1 has a reference to a Flap named leftFlap.
- In the code of the method land(), there is a call leftFlap.setangle(int)
- Add sequence numbers !





# A few things to note

- To depict a message, we draw a small arrow from the sender object to the target object. This shows the direction of communication
- With the arrow is the operation name we desire to execute, along with all arguments
- The arrow is parallel to a line, which depicts there is a link between the objects (usually an instance of an association, but not necessarily)

# No association?

- If objects aren't linked by association, then how could they be linked?
- Suppose o1 sent o2 a reference to itself. So, o2 may refer to o1 (via the reference) even though there is no association between the classes of o1 and o2.
- This is known as a **dynamic** reference.
- This reference allows a target object to “callback” a sender object.

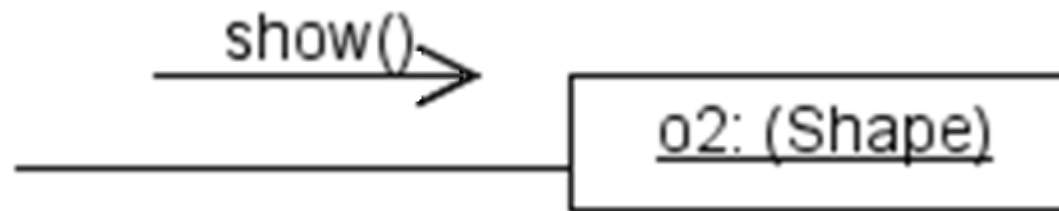
- The more formal description of callback is executable code that is passed as an argument to other code.
- However, the term callback is also used when a reference is passed to achieve the same thing.
- Callbacks are often used in asynchronous messaging.
- A piece of code or a reference is assigned to do something when a specific event occurs.
  - ◆ i.e. Swing and an ActionListener

# Polymorphism

- Triangle, Rectangle and Square are **subclasses** of Shape.
- Suppose we want to send the **message** show() to a Shape object.
  - ◆ At run-time, this object could be an instance of Triangle, Rectangle or Square.
- We know **source and destination** of the message:  
o1 sends a message to o2

# Polymorphism (cont.)

- Make the target object's class the **lowest** class in the inheritance hierarchy that is a **superclass** of **all** the classes to which the target object may belong to.
- Put the superclass name in **parenthesis** to show that it will be **evaluated at run-time**.
- This is a form of **substitutability**.

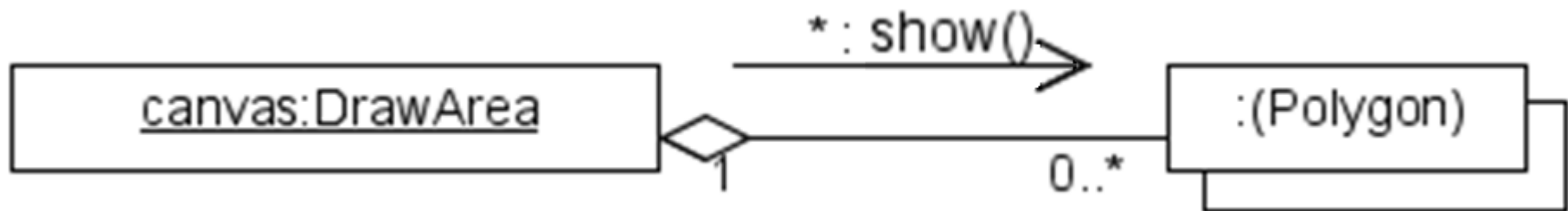


# Iterated Messages

- Suppose we have an object canvas:DrawArea which contains an array of Polygons (Triangles, Rectangles and Squares).
- We want to send the message show() to **all** the contained objects (Polygons) of the aggregate object canvas.
- The Iterator Pattern (a design pattern) can be used as a traversal method.

# Iterated Messages (cont.)

- Notice the **aggregate connector** (in Class Diagram, really).
- show() message is called many times (the **\***)
- DrawArea may have 0 or more Polygons in its collection named shapes.
- The target object is unnamed and **double boxed** to show multiplicity.



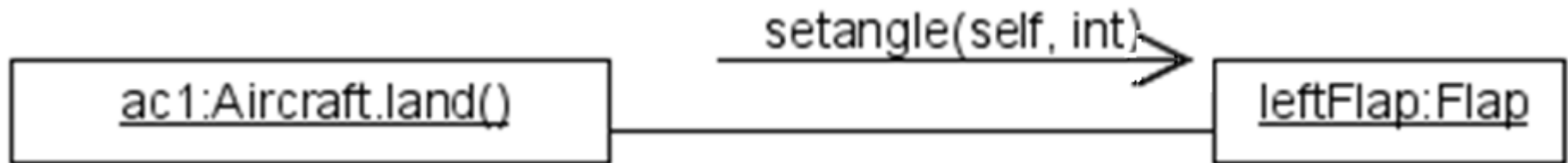
# Referring to Self

- When an object refers to **self**, it is referring to its own object handle.
  - ◆ In Python, we also use the keyword *self*
  - ◆ In Java, C++ and PHP, we use the keyword *this*
  - ◆ In Visual Basic, we use the keyword *me*
- This is useful to
  - ◆ Pass the target object a reference to the sender object (for callbacks)
  - Send a message to itself



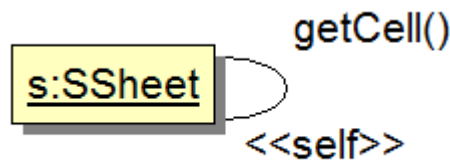
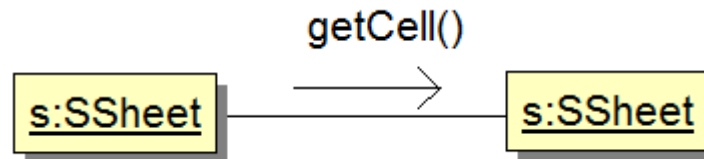
# Passing a reference to self

- In message, just add self as an argument.



# Sending a message to self

- There are two ways to depict this (note: **aliases** in UML)



# Why send a message to self?

- Implementation / information hiding.
  - ◆ We don't want to show how a variable is stored or manipulated.
    - get/set (accessor/mutator) methods
- We might want to hide implementation details from methods within the same class (especially if those methods are public).