

# Some Basic Points on Classes

---

- Class definition
- Method definition
- Basic class instantiation

# Class Definition: The Class Packet

---

A template is proposed by the browser:

```
NameOfSuperclass subclass: #NameOfClass
    instanceVariableNames: 'instVarName1 instVarName2'
    classVariableNames: 'ClassVarName1 ClassVarName2'
    poolDictionaries: ''
    category: 'CategoryName'
```

Just fill this Template in:

```
Object subclass: #Packet
    instanceVariableNames: 'contents addressee originator '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'LAN-Simulation'
```

Automatically a class named "Packet class" is created. Packet is the unique instance of Packet class. To see it, click on the class button in the browser

# Named Instance Variables

---

NameOfSuperclass subclass: #NameOfClass

instanceVariableNames: 'instVarName1 instVarName2'

...

Object subclass: #Packet

instanceVariableNames: 'contents addressee originator '

...

- Begins with a lowercase letter
- Explicitly declared: a list of instance variables
- Name should be unique in the inheritance chain
- Default value of instance variable is nil
- Private to the instance: instance based (vs. C++ class-based)
- Can be accessed by all the methods of the class and its subclasses (instance methods)
- Instance variables cannot be accessed by class methods.
- A client cannot directly access instance variables.
- The clients must use accessor methods to access an instance variable.

# Method Definition

---

Fill in the template. For example:

```
Packet>>defaultContents
```

```
    "returns the default contents of a Packet"
```

```
    ^ 'contents no specified'
```

```
Workstation>>originate: aPacket
```

```
    aPacket originator: self.
```

```
    self send: aPacket
```

How to invoke a method on the same object? Send the message to self

```
Packet>>isAddressedTo: aNode
```

```
    "returns true if I'm addressed to the node aNode"
```

```
    ^ self addressee = aNode name
```

# Accessing Instance Variables

---

Using direct access for the methods of the class

```
Packet>>isSentBy: aNode  
    ^ originator = aNode
```

is equivalent to use accessors

```
Packet>>originator  
    ^ originator
```

```
Packet>>isSentBy: aNode  
    ^ self originator = aNode
```

Design Hint: Do not directly access instance variables of a superclass from the subclass methods. This way classes will not be strongly linked at the structure level.

# Methods always return a Value

---

- Message = effect + return value
- By default, a method returns self
- In a method body, the `^` expression returns the value of the expression as the result of the method execution.

```
Node>>accept: thePacket
```

```
"Having received the packet, send it on. This is the default behavior."
```

```
self send: thePacket
```

- This is equivalent to:

```
Node>>accept: thePacket
```

```
"Having received the packet, send it on. This is the default behavior."
```

```
self send: thePacket.
```

```
^self
```

# Methods always return a value

---

- If we want to return the value returned by `#send`:

```
Node>>accept: thePacket
```

```
"Having received the packet, send it on. This is the default  
behavior."
```

```
  ^self send: thePacket.
```

- Use `^ self` to notify the reader that something abnormal is arriving

```
MyClass>>foo
```

```
  ... ^ self
```

# Some Naming Conventions

---

- Shared variables begin with an upper case letter
- Private variables begin with a lower case letter
- For accessors, use the same name as the instance variable accessed:

```
Packet>>addressee  
      ^ addressee  
Packet>>addressee: aSymbol  
      addressee := aSymbol
```



# Some Naming Conventions

---

- Use imperative verbs for methods performing an action like `#openOn:`, `#close`, `#sleep`
- For predicate methods (returning a boolean) prefix the method with `is` or `has`
  - `isNil`, `isAddressedTo:`, `isSentBy:`
- For converting methods prefix the method with `as`
  - `asString`

# Object Instantiation

---

- Objects can be created by:
  - Direct Instance creation:  
(basic)new/new:
  - Messages to instances that create other objects
  - Class specific instantiation messages

# Instance Creation

---

- aClass new/basicNew returns a newly and UNINITIALIZED instance

OrderedCollection new -> OrderedCollection ()

Packet new -> aPacket

- Instance variable values = nil
- Messages to Instances that create Objects

1 to: 6	(an interval)
1@2	(a point)
(0@0) extent: (100@100)	(a rectangle)
#lulu asString	(a string)
1 printString	(a string)
3 asFloat	(a float)
#{23 2 3 4} asSortedCollection	(a sortedCollection)

# Opening the Box

---

1 to: 6 -> an Interval

Number>>to: stop

"Answer an Interval from the receiver up to the argument, stop, with each next element computed by incrementing the previous one by 1."

^Interval from: self to: stop by: 1

1 printString -> aString

Object>>printString

"Answer a String whose characters are a description of the receiver."

| aStream |

aStream := WriteStream on: (String new: 16).

self printOn: aStream.

^aStream contents

# Instance Creation

---

1@2 -> aPoint

Number>>@ y

"Answer a new Point whose x value is the receiver  
and whose y value is the argument."

<primitive: 18>

^Point x: self y: y

# Class-specific Instantiation Messages

---

- Array with: 1 with: 'lulu'
- OrderedCollection with: 1 with: 2 with: 3
- Rectangle fromUser -> 179@95 corner: 409@219
- Browser browseAllImplementorsOf: #at:put:
- Packet send: 'Hello mac' to: #mac
- Workstation withName: #mac

# New and basicNew

---

- `#new:/basicNew:` is used to specify the size of the created instance

Array new: 4 ->  `#(nil nil nil nil)`

- `#new/#new:` can be specialized to define customized creation
- `#basicNew/#basicNew:` should never be overridden
- `#new/basicNew` and `new:/basicNew:` are class methods