

---

## Object Identity

= VS. ==  
copying objects  
creating objects

## Object Protocol

---

Operations understood by all objects:

== anObject identity - same object  
= anObject equality - same value  
~~ anObject different objects  
~= anObject different values

## Equality vs. Identity

---

Equality is user defined.

**= anObject**

`^self == anObject`

Identity is system defined.

**== anObject**

`<primitive: 110>`

## Equality

---

`(Date newDay: 40 year: 1995) ==`

`(Date newDay: 40 year: 1995)`

is false, because they are physically two distinct objects. However, they represent the same value, so they are equal.

---

Each object has its own region of memory.

“Object ID” is essentially a pointer.

Variable contains object ID, not the space of an object.

“Passing an object as an argument” means passing the object ID.

## New Objects

---

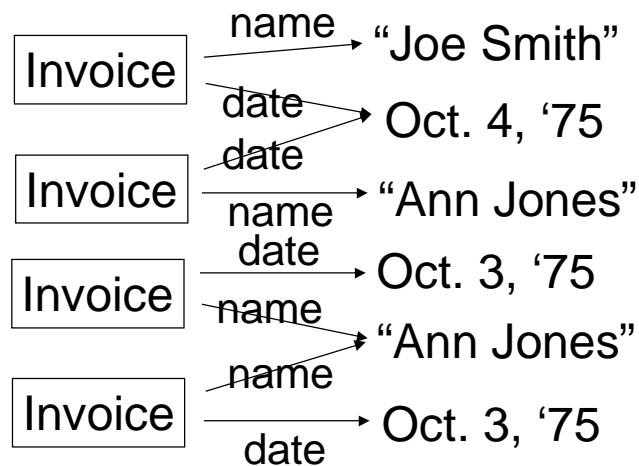
A new object is different from any existing object.

`X new == X new`  
is almost always false.

“`Rectangle new = Rectangle new`” is true.

## Sharing

---



## Sharing

---

Sharing saves space.

Sharing makes changes to one object visible to another.

Sharing is always safe when objects are immutable.

## What is the value of?

---

`(Point x: 3 y: 17) == (Point x: 3 y: 17)`

`(Point x: 3 y: 17) = (Point x: 3 y: 17)`

`'this is a string' == 'this is a string'`

`#aSymbol == #aSymbol`

## Information Hiding

---

An employees transactions are entirely hidden from clients.

To add an transaction, use  
`postTransaction:`

There is no way to access transactions outside Employee.

## Information Hiding

---

Suppose you want to iterate over transactions of an employee.

Alternative 1

- 1) Add #transactions method
- 2) anEmployee transactions do:

## Violating Information Hiding

---

An accessing method discloses information.

anEmployee transactions add:  
(Paycheck new)

anEmployee transactions remove

Very dangerous!

## Information Hiding

---

Alternative 2

**transactionsDo: aBlock**

transaction do: aBlock

Alternative 3

**transactions**

^transactions copy

## Copying

---

"shallow copy" -- copy object, but not  
contents of object

"deep copy" -- copy object and  
contents of object, recursively  
(usually VERY selectively)

Copying is usually shallow copying.

# Copying

---

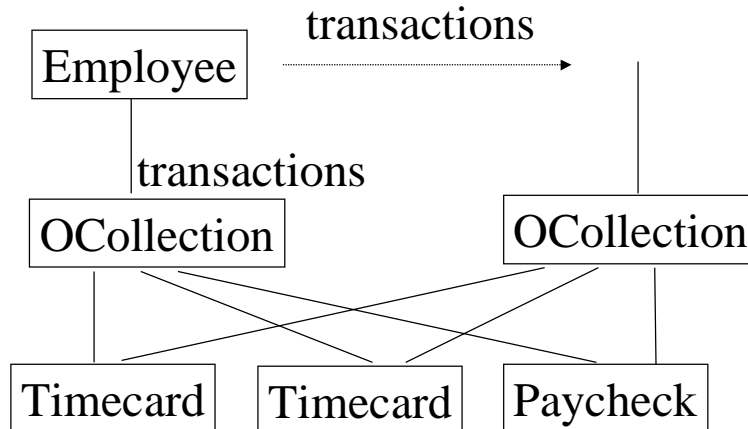
In class Object:

**copy**

^self shallowCopy postCopy

Template Method pattern!

Redefine postCopy to change the way to copy variables, not copy.



---

| t s | 1  
t := Point x: 1 y: 17.  
s := t copy.  
s x: 5.  
s = t

| t s | 2  
t := Point x: 1 y: 17.  
s := t.  
s x: 5.  
s = t

| t s | 3  
t := Point x: 1 y: 17.  
s := t copy.  
s == t

---

## Streams

Stream protocol  
Object composition

## Uses of Streams

---

External iterator  
Parsing  
Formatted output  
Dataflow computing  
File I/O

## ReadStream Protocol

---

ReadStream

- next, atEnd
- (do:, nextMatchFor:)

#next is valid if #atEnd is false.

## ReadStream

---

```
t := ReadStream on: 'this is the input'.  
[t atEnd]  
  whileFalse:  
    [t next = $p ifTrue: [^t]]]
```

## Iterator

---

Internal Iterator - do:  
External Iterator - streams

Internal iterator is easier to use, but less powerful.

External iterators needed for simultaneous iteration.

## Iterator

---

```
“return true if streams equal”  
[stream1 atEnd | stream2 atEnd]  
whileFalse:  
    [stream1 next = stream2 next  
     ifFalse: [^false]].  
^stream1 atEnd & stream2 atEnd
```

## Stream Protocol

---

```
WriteStream  
    nextPut:  
    (nextPutAll:, cr, tab, space)
```

## LineStream

---

Takes character stream and returns one input line at a time.

aStream := LineStream on: ('stuff'  
asFilename readStream).

inputLine := aStream next.

## Creating a LineStream

---

*Class method*

**on: aStream**

^self basicNew on: aStream

*Instance method*

**on: aStream**

inputStream := aStream

## **next**

```
| outputStream |
outputStream := WriteStream
                    on: (String new: 20).

inputStream
  do: [:eachChar | eachChar = CR
      ifTrue: [^outputStream contents]
      ifFalse: [outputStream
                  nextPut: eachChar]].
^outputStream contents
```

## LineStream

---

### **atEnd**

```
^inputStream atEnd
```

### *Class method*

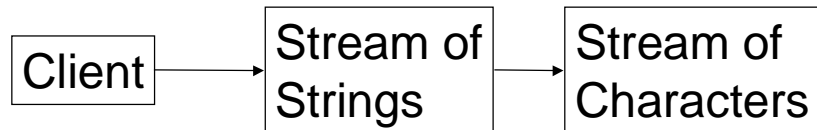
### **initialize**

```
"LineStream initialize"
```

```
CR := Character cr
```

## Adapters

---



Transcript show:  
(LineStream on:  
('stuff' asFilename readStream))  
next

*Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson*

240

## Infinite Streams

---

It is easy for streams to be infinite; they just always return false for atEnd.

IntegerStream is a class of streams that return the positive integers in order. It has one instance variable, value.

*Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson*

241

---

**atEnd**

^false

**next**

| temp |

temp := value

value := value + 1

^temp

**init**

value := 1

*Class method*

**new**

^super new init

## SelectionStreams

---

The select: message to a stream should produce a SelectionStream.

SelectionStream has four instance variables, *selectBlock*, *nextValue*, *atEnd*, and *stream*.

**atEnd**

^atEnd

### **next**

```
| temp |  
temp := nextValue.  
stream do: [:each | (selectBlock value:  
each)  
ifTrue: [nextValue := each.  
^temp]].  
atEnd := true.  
^temp
```

---

### **select: aBlock on: aStream**

```
selectBlock := aBlock.  
stream := aStream.  
atEnd := false.  
self next.
```

# Primes

---

PrimeStream has one instance variable,  
stream.

## **init**

```
stream := IntegerStream new.  
stream next "start stream with 2"
```

## **atEnd**

```
^false
```

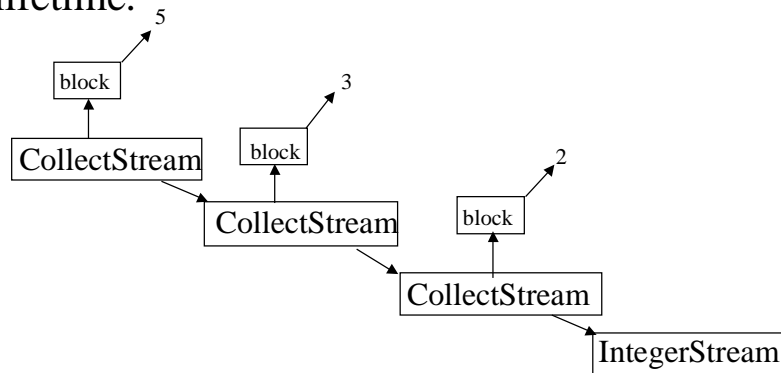
---

## **next**

```
| nextPrime |  
nextPrime := stream next.  
stream :=  
    stream  
        select: [:i |( i \\ nextPrime)~~ 0].  
^nextPrime
```

# Blocks

Blocks can cause temporaries to have a long lifetime.



# Blocks

Blocks point to method contexts, and method contexts are stored until nothing (i.e. no called contexts and no blocks) needs them.

## Reuse

---

### Inheritance vs. composition

Inheritance: Stream defines standard interface and shared code.

## Composition

---

SelectionStream can be composed with other streams to make a new kind of stream. Most of PrimeStream is reused from SelectionStream.

Composition requires new way of looking at problem.

# FileStreams

---

How to read a file

'file.out' asFilename readStream

# Other operations on files

---

'filename.st' asFilename edit.

'filename.st' asFilename fileIn.

'filename.st' asFilename delete.

Note: Filename is an adaptor  
between strings and streams.

## Inheritance as Parameterization

---

- Subclass customizes template method by implementing abstract operations.
- Any method acts as a parameter.
- Abstract class -- one that must be customized before it can be used.

## Inheritance and Polymorphism

---

- Polymorphism works best with standard interfaces.
- Inheritance creates families of classes with similar interfaces.
- Abstract class describes standard interfaces.
- Inheritance helps software reuse by making polymorphism easier.

## Specification Inheritance

---

### Reuse of specification

A program that works with Numbers will work with Fractions.

A program that works with Collections will work with Arrays.

## Inheritance for code reuse

---

Dictionary is a subclass of Set

Semaphore is a subclass of LinkedList

Subclass reuses code from superclass, but has a different specification. It cannot be used everywhere its superclass is used. Usually overrides a lot of code.

## Inheritance for code reuse

---

Inheritance for code reuse is good for:

- rapid prototyping
- getting application done quickly.

Bad for:

- easy to understand systems
- reusable software
- application with long life-time.

## How to Use Inheritance

---

- When you are in a hurry, do what seems easiest.
- Clean up later, make sure classes use “is-a” relationship, not just “is-implemented-like”.
- Is-a is a design decision, the compiler only enforces is-implemented-like!!!