UNIVERSITÄT BERN

7. Best Practice Patterns

Birds-eye view

Let your code talk — Names matter. Let the code say what it means.
Introduce a method for everything that needs to be done. Don't be afraid to delegate, even to yourself.

Roadmap

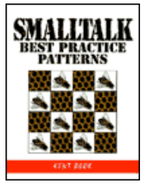
- Naming conventions
- Delegation and Double Dispatch
- Conversion and Extension
- Being Lazy
- Collections, Intervals and Streams

Selected material based on: Kent Beck, *Smalltalk Best Practice Patterns*, Prentice-Hall, 1997.

Roadmap

> **Naming conventions**
> Delegation and Double Dispatch
> Conversion and Extension
> Being Lazy
> Collections, Intervals and Streams

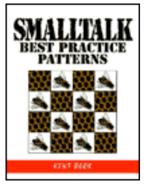
Simple Superclass Name



What should we call the root of a hierarchy?

> Use a single word that conveys its purpose in the design
  — `Number`
  — `Collection`
  — `VisualComponent`
  — `BoardSquare`

Qualified Subclass Name



What should you call a subclass that plays a role similar to its superclass?

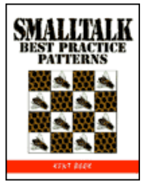
> Use names that indicate the distinct role. Otherwise prepend an adjective that communicates the relationship
  — `OrderedCollection` (vs. `Array`)
  — `UndefinedObject`
  — `FirstSquare` (vs. `Snake` and `Ladder`)

Naming methods and variables

> Choose method and variable names so that expressions can be read like (pidgin) sentences.
  - Spell out names in full
    - *Avoid abbreviations!*

```
players do: [:each | each moveTo: self firstSquare ].
```

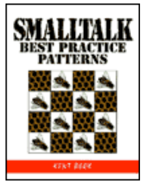
Intention Revealing Selector



What do you name a method?

> Name methods after *what* they accomplish, not how.
- — Change state of the receiver:
  - – *`translateBy:`*, *`add:`* …
- — Change state of the argument:
  - – *`displayOn:`*, *`addTo:`*, *`printOn:`*
- — Return value from receiver:
  - – *`translatedBy:`*, *`size`*, *`topLeft`*

Role Suggesting Instance Variable Name



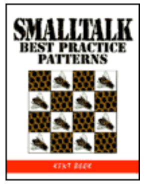
What do you name an instance variable?

> Name instance variables for the role they play in the computation.
  — Make the name plural if the variable will hold a Collection

```
Object subclass: #SnakesAndLadders
   instanceVariableNames: 'players squares turn die over'
   …
```

Type Suggesting Parameter Name

What do you call a method parameter?



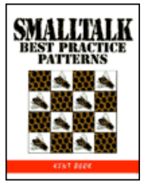
> Name parameters according to their most general expected class, preceded by "a" or "an".
  — Don't need to do this if the method name already specifies the type, or if the type is obvious.
  — If there is more than one argument with the same expected type, precede the type with its role.

```
BoardSquare>>setPosition: aNumber board: aBoard
   position := aNumber.
   board := aBoard
```

```
Collection>>reject: rejectBlock thenDo: doBlock
   "Utility method to improve readability."
   ^ (self reject: rejectBlock) do: doBlock
```

Role Suggesting Temporary Variable Name

What do you call a temporary variable?



> Name a temporary variable for the role it plays in the computation.
  - Use temporaries to:
    - *collect intermediate results*
    - *reuse the result of an expression*
    - *name the result of an expression*
  - Methods are often simpler when they don't use temporaries!

```
GamePlayer>>moveWith: aDie
    | roll destination |
    roll := aDie roll.
    destination := square forwardBy: roll.
    self moveTo: destination.
    ^ name, ' rolls ', roll asString
```

Methods from Comments

- Be suspicious of comments
  - If you feel the need to comment your code, try instead to introduce a new method
  - *“Do not comment bad code — rewrite it”*

Kernighan ’78

```
GamePlayer>>moveTo: aSquare
   square notNil ifTrue: [ square remove: self ].
      "leave the current square"
   square := aSquare landHere: self.
```

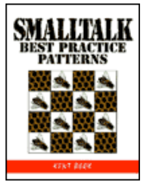
```
GamePlayer>>moveTo: aSquare
   self leaveCurrentSquare.
   square := aSquare landHere: self.

GamePlayer>>leaveCurrentSquare
   square notNil ifTrue: [ square remove: self ].
```

Exception: always write class comments!

Roadmap

> Naming conventions
> **Delegation and Double Dispatch**
> Conversion and Extension
> Being Lazy
> Collections, Intervals and Streams

Delegation

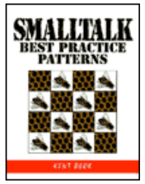


How does an object share implementation without inheritance?

> Pass part of its work on to another object
  — Many objects need to display, all objects delegate to a brush-like object (Pen in VisualSmalltalk, GraphicsContext in VisualAge and VisualWorks)
  — All the detailed code is concentrated in a single class and the rest of the system has a simplified view of the displaying.

Simple Delegation

How do you invoke a disinterested delegate?

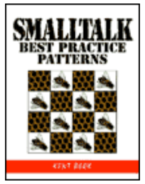


> Delegate messages unchanged
  - Is the identity of the delegating object important?
    - *No*
  - Is the state of the delegating object important?
    - *No*
  - Use simple delegation!

```
SnakesAndLadders>>at: position
   ^ squares at: position
```

Self Delegation

How do you implement delegation to an object that needs reference to the delegating object?

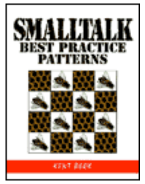


> Pass along the delegating object (i.e., `self`) in an additional parameter.
  — Commonly called “`for:`”

```
GamePlayer>>moveTo: aSquare
   self leaveCurrentSquare.
   square := aSquare landHere: self.
```

Reversing Method

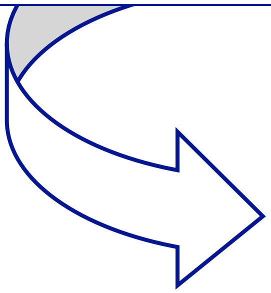
How do you code a smooth flow of messages?



> Code a method on the parameter.
  - Derive its name form the original message.
  - Take the original receiver as a parameter to the new method.
  - Implement the method by sending the original message to the original receiver.

```
Point>>printOn: aStream
   x printOn: aStream
   aStream nextPutAll: '@'.
   y printOn: aStream
```

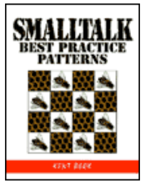
Caveat: Creating new selectors just for fun is not a good idea. Each selector must justify its existence.



```
Stream>>print: anObject
   anObject printOn: self

Point>>printOn: aStream
   aStream print: x; nextPutAll: '@'; print: y
```

Execute Around Method

How do you represent pairs of actions that have to be taken together?



> Code a method that takes a `Block` as an argument.
  — Name the method by appending “`During: aBlock`” to the name of the first method to be invoked.
  — In the body, invoke the first method, evaluate the block, then invoke the second method.

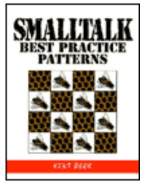
```
File>>openDuring: aBlock
   self open.
   aBlock value.
   self close
```

Or better:

```
File>>openDuring: aBlock
   self open.
   [aBlock value]
   ensure: [self close]
```

Method Object

How do you break up a method where many lines of code share many arguments and temporary variables?



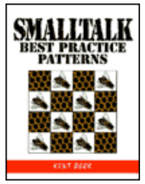
> Create a class named after the method.
  - Give it an instance variable for the receiver of the original method, each argument and each temporary.
  - Give it a Constructor Method that takes the original receiver and method arguments.
  - Give it one method, `compute`, implemented by the original method body.
  - Replace the original method with a call to an instance of the new class.
  - Refactor the `compute` method into *lots of little methods*.

Method Object

```
Obligation>>sendTask: aTask job: aJob
   | notprocessed processed copied executed |
   ... 150 lines of heavily commented code
```

```
Object subclass: #TaskSender
   instanceVariableNames: 'obligation task job
      notprocessed processed copied executed'
   ...

TaskSender class>>obligation: anObligation task: aTask job: aJob
   ^ self new
      setObligation: anObligation task: aTask job: aJob

TaskSender>>compute
   ... 150 lines of heavily commented code (to be refactored)

Obligation>>sendTask: aTask job: aJob
   (TaskSender obligation: self task: aTask job: aJob) compute
```

Choosing Object

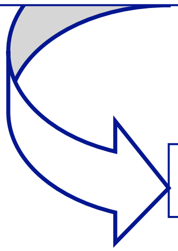


How do you execute one of several alternatives?

> Send a message to one of several different kinds of objects, each of which executes one alternative.

Choosing Object

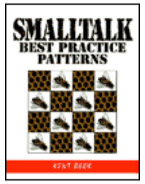
```
square isSnake
   ifTrue: [
      destination := square backwardBy: square back ]
   ifFalse: [
      square isLadder
         ifTrue: [ destination := square forwardBy: square forward ]
         ifFalse: [ destination := square ] ]
```



```
square destination
```

```
BoardSquare>>destination
   ^ self

LadderSquare>>destination
   ^ self forwardBy: forward

SnakeSquare>>destination
   ^ self backwardBy: back
```

Double Dispatch



> How can you code a computation that has many cases, the cross product of two families of classes?

> Send a message to the argument.
  - Append the class or “species” name of the receiver to the selector.
  - Pass the receiver as an argument.
  - *Caveat:* Can lead to a proliferation of messages

Maresey Doats

Mares eat oats and does eat oats,
And little lambs eat ivy,
A kid will eat ivy too,
Wouldn't you?

```
MareTest>>testEating
   self assert:
      ((mare eats: oats)
      and: [ doe eats: oats ]
      and: [ lamb eats: ivy ]
      and: [ kid eats: ivy ]
      ).
```

Bad Solutions

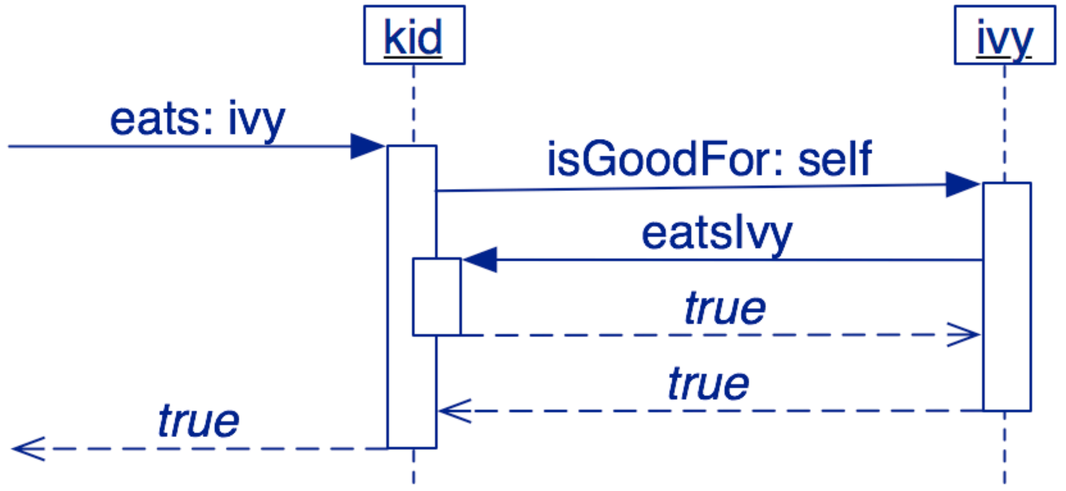
```
Mare>>eats: aFood
    ^ aFood class = Oats
```

- Breaks encapsulation
- Hard to extend
- Fragile with respect to changes

```
Mare>>eats: aFood
    ^ aFood isGoodForMares
Food>>isGoodForMares
    ^ false
Oats>>isGoodForMares
    ^ true
```

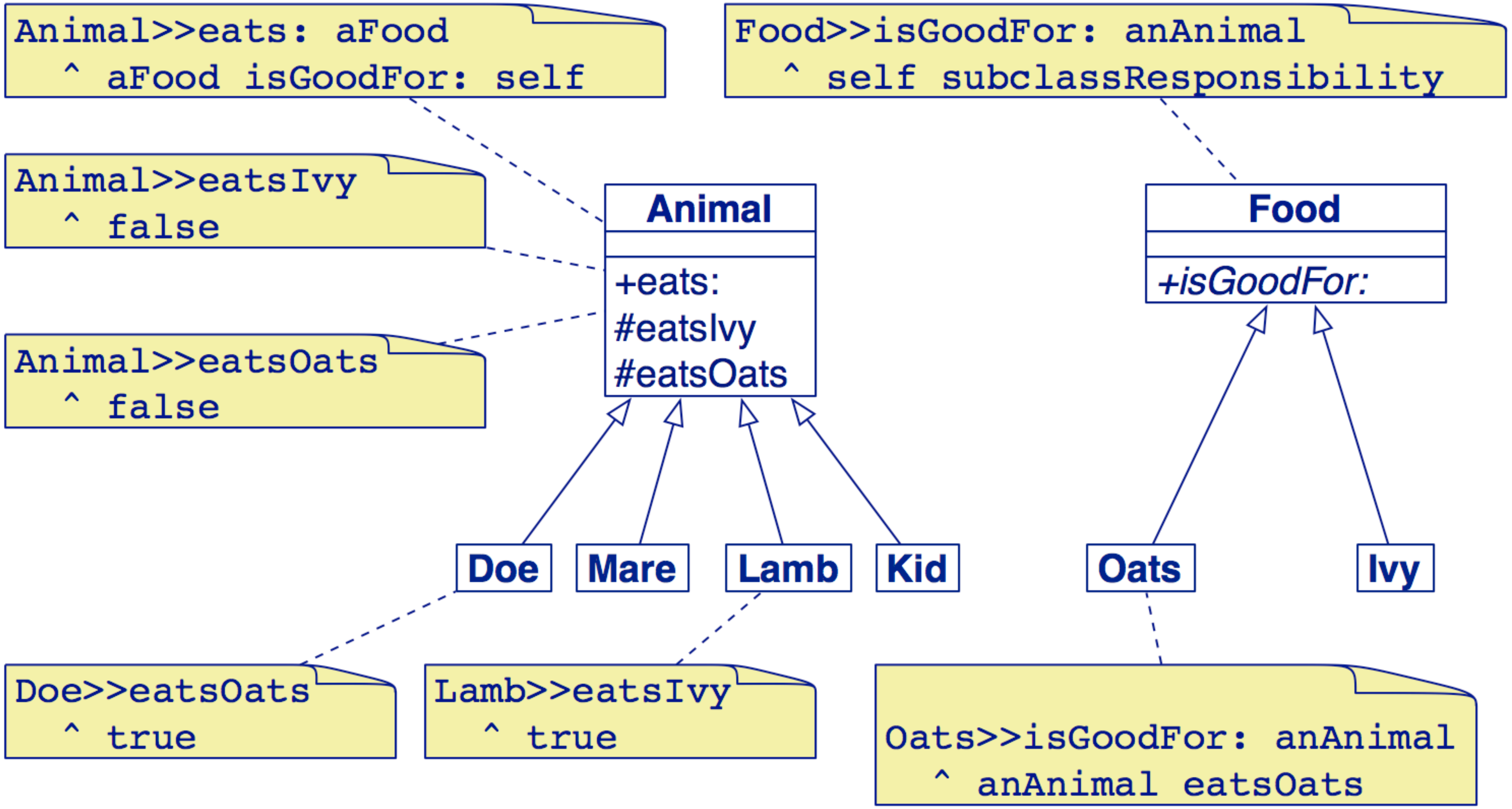
Better, but:
- Mixes responsibilities
- Still hard to extend

Double Dispatch — Interaction



- Separates responsibilities
- Easy to extend
- Handles multiple kinds of food

Double Dispatch — Hierarchy



```
Animal>>eats: aFood
  ^ aFood isGoodFor: self
```

```
Food>>isGoodFor: anAnimal
  ^ self subclassResponsibility
```

```
Animal>>eatsIvy
  ^ false
```

```
Animal>>eatsOats
  ^ false
```

```
Doe>>eatsOats
  ^ true
```

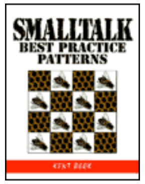
```
Lamb>>eatsIvy
  ^ true
```

```
Oats>>isGoodFor: anAnimal
  ^ anAnimal eatsOats
```

Roadmap

> Naming conventions
> Delegation and Double Dispatch
> **Conversion and Extension**
> Being Lazy
> Collections, Intervals and Streams

Converter Method

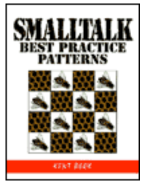


How do you convert an object of one class to that of another that supports the same protocol?

> Provide a converter method in the interface of the object to be converted.
  — Name it by prepending "as" to the class of the object returned
  — E.g., `asArray`, `asSet`, `asOrderedCollection` etc.

Converter Constructor Method

How do you convert an object of one class to that of another that supports a different protocol?



> Introduce a Constructor Method that takes the object to be converted as an argument
  — Name it by prepending “from” to the class of the object to be converted

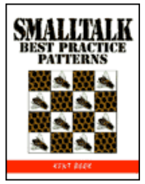
```
String>>asDate
   …
"Jan 1, 2006" asDate
```

Don’t confuse responsibilities!

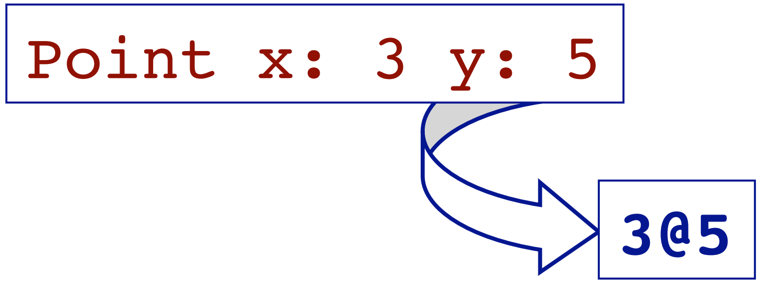
```
Date class>>fromString:
   …
Date fromString: "Jan 1, 2006"
```

Shortcut Constructor Method

What is the external interface for creating a new object when a Constructor Method is too wordy?



> Represent object creation as a message to one of the arguments of the Constructor Method.
  — Add no more than three of these methods per system you develop!

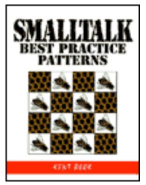


```
Point x: 3 y: 5
```

```
3@5
```

Modifying Super

> How do you change part of the behaviour of a super class method without modifying it?



> Override the method and invoke super.
  — *Then execute the code to modify the results.*

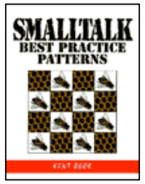
```
SnakesAndLadders>>initialize
   die := Die new.
   …

ScriptedSnakesAndLadders>>initialize
   super initialize
   die := LoadedDie new.
   …
```

Roadmap

> Naming conventions
> Delegation and Double Dispatch
> Conversion and Extension
> **Being Lazy**
> Collections, Intervals and Streams

Default Value Method

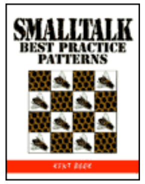
How do you represent the default value of a variable?



> Create a method that returns the value.
  - Prepend “default” to the name of the variable as the name of the method

```
DisplayScanner>>defaultFont
   ^ TextStyle defaultFont
```

Constant Method



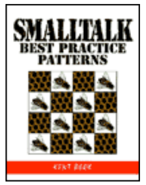
How do you code a constant?

> Create a method that returns the constant

```
Fraction>>one
    ^ self numerator: 1 denominator: 1
```

Lazy Initialization

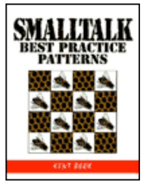
How do you initialize an instance variable to its default value?



> Write a Getting Method for the variable.
  - Initialize it if necessary with a Default Value Method
  - Useful if:
    - *The variable is not always needed*
    - *The variable consumes expensive resources (e.g., space)*
    - *Initialization is expensive.*

```
XWindows>>windowManager
   windowManager isNil ifTrue: [
      windowManager := self defaultWindowManager ].
   ^ windowManager
```

Lookup Cache



> How do you optimize repeated access to objects that are expensive to compute?

> Cache the values of the computation
  - Prepend “lookup” to the name of the expensive method
  - Add an instance variable holding a Dictionary to cache the results.
  - Make the parameters of the method be the search keys of the dictionary and the results be its values.

Slow Fibonacci

```
Fibs>>at: anIndex
   self assert: anIndex >= 1.
   anIndex = 1 ifTrue: [ ^ 1 ].
   anIndex = 2 ifTrue: [ ^ 1 ].
   ^ (self at: anIndex - 1) + (self at: anIndex - 2)
```

```
Fibs new at: 35
```

```
9227465
```

Takes 8 seconds.
Forget about larger values!

Cacheing Fibonacci

```
Object subclass: #Fibs
    instanceVariableNames: 'fibCache'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Misc'

Fibs>>initialize
    fibCache := Dictionary new

Fibs>>fibCache
    ^ fibCache
```

Introduce the cache …

Cacheing Fibonacci

Now we introduce the lookup method, and redirect all accesses to use the cache lookup

```
Fibs>>lookup: anIndex
    ^ self fibCache at: anIndex ifAbsentPut: [ self at: anIndex ]

Fibs>>at: anIndex
    self assert: anIndex >= 1.
    anIndex = 1 ifTrue: [ ^ 1 ].
    anIndex = 2 ifTrue: [ ^ 1 ].
    ^ (self lookup: anIndex - 1) + (self lookup: anIndex - 2)
```

```
Fibs new at: 100
```

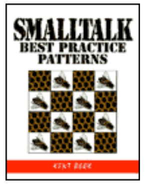
```
354224848179261915075
```

… is virtually instantaneous!

Roadmap

> Naming conventions
> Delegation and Double Dispatch
> Conversion and Extension
> Being Lazy
> **Collections, Intervals and Streams**

Comparing Method

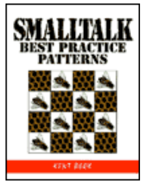


How do you order objects with respect to each other?

> Implement <= to return true if the receiver should be ordered before the argument
  — `<,<=,>,>=` are defined for `Magnitude` and its subclasses.
  — Implement <= in the “comparing” protocol

Sorted Collection

How do you sort a collection?



> Use a Sorted Collection.
  — Set its sort block if you want to sort by some other criterion than <=

```
#( 'Snakes' 'Ladders' ) asSortedCollection
```

```
a SortedCollection('Ladders' 'Snakes')
```

```
#( 'Snakes' 'Ladders' ) asSortedCollection: [:a :b | b<=a ]
```

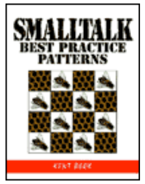
```
a SortedCollection('Snakes' 'Ladders')
```

```
#( 'Snakes' 'Ladders' ) asSortedCollection
        sortBlock: [:a :b | b<=a ]
```

```
a SortedCollection('Snakes' 'Ladders')
```

Interval

How do you code a collection of numbers in a sequence?



> Use an `Interval` with start, stop and optional step value.
  — Use the Shortcut Constructor methods *Number*>>`to:` and *Number*>>`to:by:` to build intervals

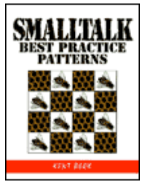
```
1 to: 5
(1 to: 5) asSet
(10 to: 100 by: 20) asOrderedCollection
```

```
(1 to: 5)
a Set(1 2 3 4 5)
an OrderedCollection(10 30 50 70 90)
```

Duplicate Removing Set

How do you remove the duplicates from a Collection?

> Send `asSet` to the collection

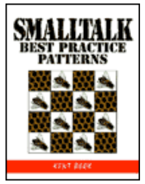


```
'hello world' asSet
```

```
a Set(Character space $r $d $e $w $h $l $o)
```

Searching Literal

How do you test if an object is equal to one of several literal values?



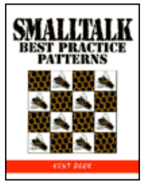
> Ask a literal Collection if it includes the element you seek

```
char = $a | char = $e | char = $i | char = $o | char = $u |
char = $A | char = $E | char = $I | char = $O | char = $U
```

```
'aeiou' includes: char asLowercase
```

Concatenation

How do you put two collections together?



> Send “,” to the first with the second as argument

```
(1 to: 3), (4 to: 6)
```

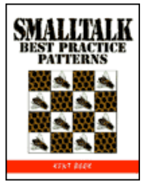
```
#(1 2 3 4 5 6)
```

```
(Dictionary newFrom: { #a -> 1}), (Dictionary newFrom: { #b -> 2})
```

```
a Dictionary(#a->1 #b->2 )
```

Concatenating Stream

How do you concatenate several Collections?



> Use a Stream on a new collection of the result type.

```
writer := WriteStream on: String new.
Smalltalk keys do: [ : each | writer nextPutAll: each, '::' ].
writer contents
```

Can be vastly more efficient than building a new collection with each concatenation.

What you should know!

- ✎ *How should you name instance variables?*
- ✎ *Why should you be suspicious of comments?*
- ✎ *How does Simple Delegation differ from Self Delegation?*
- ✎ *When would you use Double Dispatch?*
- ✎ *Why should you avoid introducing a Converter Method for an object supporting a different protocol?*
- ✎ *How do you sort a Collection?*
- ✎ *When should you use Lazy Initialization?*

Can you answer these questions?

- ✎ *Which patterns would you use to implement a transactional interface?*
- ✎ *How can Method Object help you to decompose long methods?*
- ✎ *Why is it a bad idea to query an object for its class?*
- ✎ *Why are you less likely to see Double Dispatch in a statically-typed language?*
- ✎ *How can you avoid Modifying Super?*
- ✎ *How can you avoid writing case statements?*
- ✎ *What pattern does `Object>>->` illustrate?*

License