Contracts for First-Class Classes

T. Stephen Strickland & Matthias Felleisen
PLT @ Northeastern University

DLS, October 18, 2010

Dynamic languages have first-class classes.

- Python
- Ruby
- Lisp
- Javascript

...

First-class classes are useful for code organization.

```
class Stack
  def initialize
    @intarr = Array.new
  end
  def push(elem)
    @intarr.push(elem)
  end
  def pop
    @intarr.pop
  end
  def isEmpty
    @intarr.length == 0
  end
end
```

```
def addFlush(cls)
  new_class = Class.new cls
  new_class.class_eval do
    def flush
      while !self.isEmpty
        self.pop
      end
    end
  end
  new_class
end

stack = addFlush(Stack).new
```

Racket[*] also has first-class classes.

```
(define stack
  (class object%
    (define intlist null)
    (define/public (empty?) (null? intlist))
    (define/public (push e)
      (set! intlist (cons e intlist)))
    (define/public (pop)
      (begin0 (first intlist)
              (set! intlist (rest intlist))))))

(define (add-flush cls)
  (class cls
    (inherit empty? pop)
    (define/public (flush)
      (unless (empty?) (begin (pop) (flush))))))

(define flush-stack (add-flush stack))
```

[*] formerly known as PLT Scheme

Dynamic languages also have contract libraries.

- Python
- Ruby
- Lisp
- Javascript

...

However, most libraries focus on Eiffel-like features.

```python
class PrimeStack:
    """A stack of prime numbers."""

    def __init__(self):
        self.data = []

    def is_empty(self):
        return (len(self.data) == 0)

    def push(self, elem):
        """Add an element to the stack.

        pre: is_prime(elem)"""
        self.data.append(elem)

    def pop(self):
        """Returns the last added element.

        pre: not self.is_empty()
        post: is_prime(__return__)"""
        return self.data.pop()
```

We add contracts that protect class and object values.

```
(define prime-stack
  (class object%
    (define intlist null)
    (define/public (empty?) (null? intlist))
    (define/public (push e)
      (set! intlist (cons e intlist)))
    (define/public (pop)
      (begin0 (first intlist)
              (set! intlist (rest intlist))))))

(provide/contract
 [prime-stack
  (class/c
   [push (-> prime? void?)]
   [pop (-> #:pre (not (send this empty?)) prime?)])])
```

Contracts in Racket

Contracts are a declarative way to specify behavior.

```
(define prime-list (list 3 5 7 13))

(define (filter-primes lst)
  (filter prime? lst))

(provide/contract
 [prime-list (listof prime?)]
 [filter-primes (-> (listof any/c) (listof prime?))])
```

Contract boundaries separate providers from users.

server | **client**

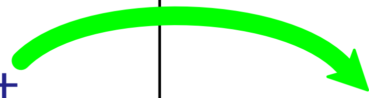
```
(define prime-list
  (list 3 5 7 13))
```

```
... prime-list ...
```

```
(listof prime?)
```

Values that flow over contract boundaries are checked.

server **client**



```
(define prime-list
  (list 3 5 7 13))
```

```
... prime-list ...
```

```
(listof prime?)
```

Values that flow over contract boundaries are checked.

server | **client**

```
(define prime-list
  (list 3 4 5 7 13))
```

```
... prime-list ...
```

```
(listof prime?)
```

server broke the contract (listof prime?) on prime-list; expected <prime?>, got 4

For values with behavior, checking must be delayed.

server

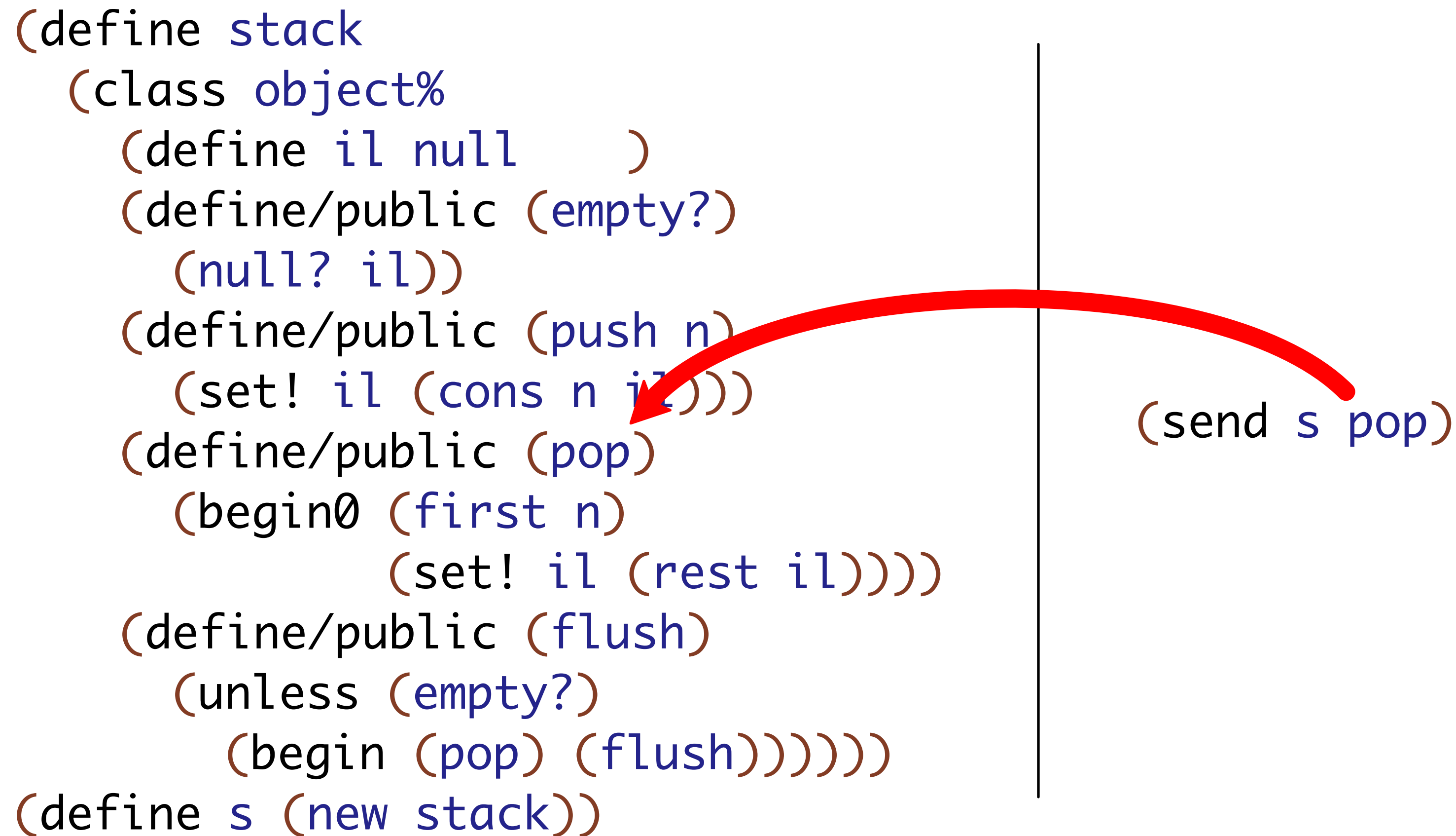
```
(define stack
  (class object%
    (define il null     )
    (define/public (empty?)
      (null? il))
    (define/public (push n)   ?
      (set! il (cons n il)))
    (define/public (pop)
      (begin0 (first n)
              (set! il (rest il))))
    (define/public (flush)
      (unless (empty?)
        (begin (pop) (flush))))))
(define s (new stack))
```

client

```
(send s pop)
```

```
(object/c [pop (-> #:pre (not (send this empty?)) prime?)])
```

For values with behavior, checking must be delayed.



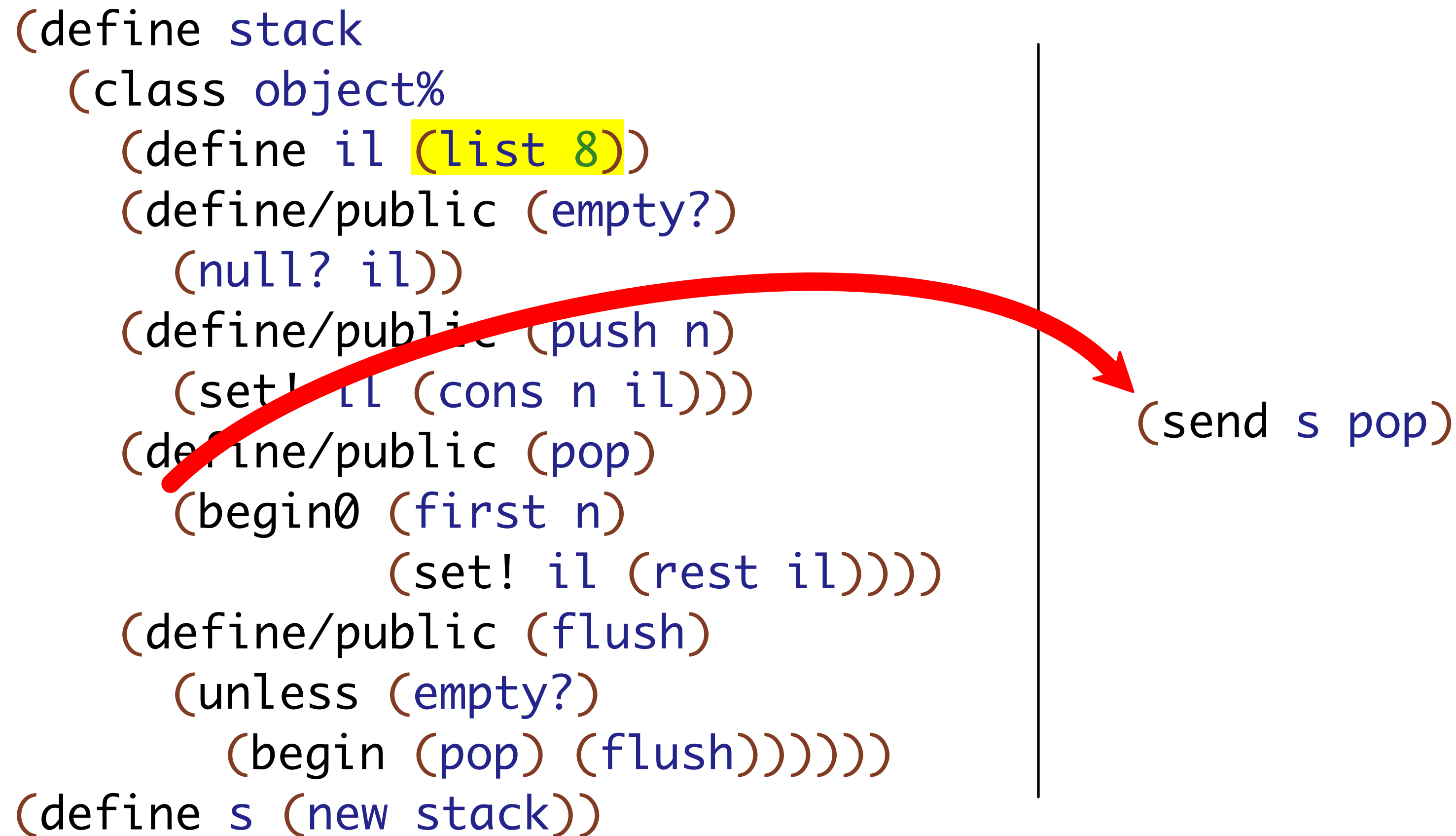

```
(object/c [pop (-> #:pre (not (send this empty?)) prime?)])
```

client broke the contract
(object/c [pop (-> #:pre (not (send this empty?)) prime?)])
on prime-list; failed pre-condition

For values with behavior, checking must be delayed.

server

client




```
(object/c [pop (-> #:pre (not (send this empty?)) prime?)])
```

server broke the contract
(object/c [pop (-> #:pre (not (send this empty?)) prime?)])
on prime-list; expected <prime?>, got 8

Internal uses are not checked.

server

```
(define stack
  (class object%
    (define il (list 8))
    (define/public (empty?)
      (null? il))
    (define/public (push n)
      (set! il (cons n il)))
    (define/public (pop)
      (begin0 (first n)
              (set! il (rest il))))
    (define/public (flush)
      (unless (empty?)
        (begin (pop) (flush))))))
(define s (new stack))
```

client

```
(send s flush)
```

```
(object/c [pop (-> #:pre (not (send this empty?)) prime?)])
```

Contracts for First-Class Classes

In traditional contract systems, classes contain contracts.

```
class PrimeStack:
    """A stack of prime numbers."""

    def __init__(self):
        self.data = []

    def is_empty(self):
        return (len(self.data) == 0)

    def push(self, elem):
        """Add an element to the stack.

        pre: is_prime(elem)"""
        self.data.append(elem)

    def pop(self):
        """Returns the last added element.

        pre: not self.is_empty()
        post: is_prime(__return__)"""
        return self.data.pop()
```

However, this limits reusability of code.

```
class PrimeStack:
    """A stack of prime numbers."""

    def __init__(self):
        self.data = []

    def is_empty(self):
        return (len(self.data) == 0)

    def push(self, elem):
        """Add an element to the stack.

                           """
        self.data.append(elem)

    def pop(self):
        """Returns the last added element.

        pre: not self.is_empty()
                                   """
        return self.data.pop()
```

These systems cannot protect existing classes.

```ruby
class Stack
  def initialize
    @intarr = Array.new
  end
  def push(elem)
    @intarr.push(elem)
  end
  def pop
    @intarr.pop
  end
  def isEmpty
    @intarr.length == 0
  end
end
```

```ruby
def addFlush(cls)
  new_class = Class.new cls
  new_class.class_eval do
    def flush
      while !self.isEmpty
        self.pop
      end
    end
  end
  new_class
end


stack = addFlush(Stack).new
```

Instead, we separate contracts from classes.

```
(define stack
  (class object%
    (define intlist null)
    (define/public (empty?) (null? intlist))
    (define/public (push e)
      (set! intlist (cons e intlist)))
    (define/public (pop)
      (begin0 (first intlist)
              (set! intlist (rest intlist))))))

(provide/contract
 (rename stack prime-stack
  (class/c
   [push (-> prime? void?)]
   [pop (-> #:pre (not (send this empty?)) prime?)])))
```

Instead, we separate contracts from classes.

```
(define (add-flush cls)
  (class cls
    (inherit empty? pop)
    (define/public (flush)
      (unless (empty?) (begin (pop) (flush))))))

(provide/contract
 [add-flush
  (-> (class/c
        [empty? (-> boolean?)]
        [pop (-> #:pre (not (send this empty?)) any/c)])
      (class/c
        [flush (-> #:post (send this empty?) void?)]))])
```

First-class contracts allow for contract abstractions.

```
(define stack
  (class object%
    (define intlist null)
    (define/public (empty?) (null? intlist))
    (define/public (push e)
      (set! intlist (cons e intlist)))
    (define/public (pop)
      (begin0 (first intlist)
              (set! intlist (rest intlist))))))

(define (stack/c e/c)
  (class/c
   [push (-> e/c void?)]
   [pop (-> #:pre (not (send this empty?)) e/c)]))

(provide/contract
 [stack (stack/c any/c)]
 (rename stack prime-stack (stack/c prime?)))
```

Our contracts distinguish different users of classes.

```
(define (add-flush cls)
  (class cls
    (inherit empty? pop)
    (define/public (flush)
      (unless (empty?) (begin (pop) (flush))))))

(provide/contract
 [add-flush
  (-> (class/c
        [empty? (-> boolean?)]
        [pop (-> #:pre (not (send this empty?)) any/c)])
      (class/c
        [flush (-> #:post (send this empty?) void?)]))])
```

Our contracts distinguish different users of classes.

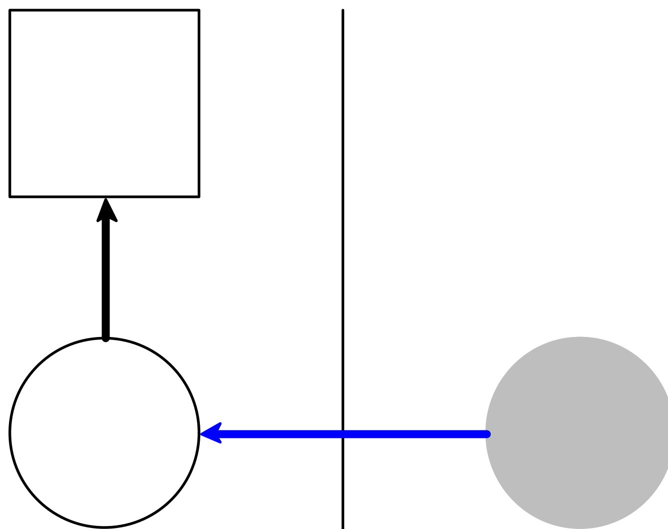
```
(define (add-flush cls)
  (class cls
    (inherit empty? pop)
    (define/public (flush)
      (unless (empty?) (begin (pop) (flush))))))

(provide/contract
 [add-flush
  (-> (class/c
        (inherit
          [empty? (-> boolean?)]
          [pop (-> #:pre (not (send this empty?)) any/c)]))
      (class/c
       [flush (-> #:post (send this empty?) void?)]))])
```

Contracts for Objects

We need object contracts as well as class contracts.

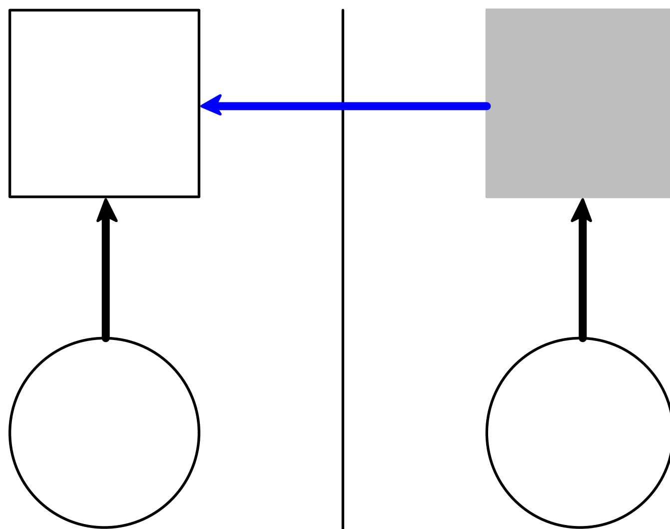
```
(require data/stack)
(define big-stack (new stack))

(define (push-list o l)
  (for ([elem l])
    (send o push elem)))

(provide/contract
 [big-stack (object/c [push (-> (>=/c 1000000) void?)]
                      [pop (-> (>=/c 1000000))])]
 [push-list (-> (object/c [push (-> any/c void?)])
                void?)])
```

We build object contracts on top of class contracts.



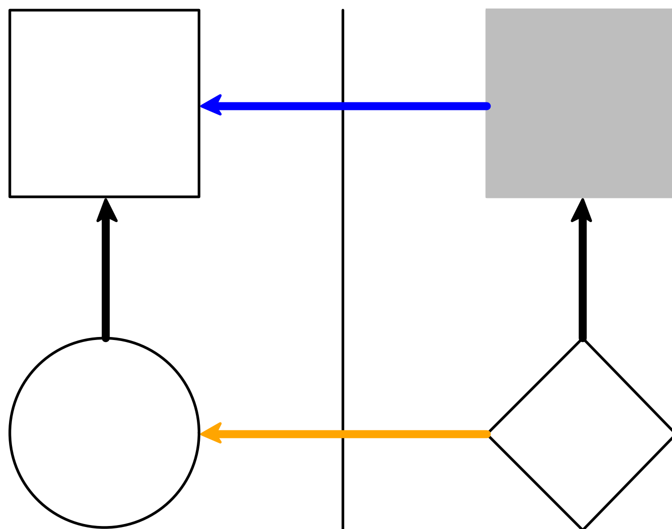
Key:
□ = class
○ = object
● = contracted object
■ = contracted class
◇ = proxy object
→ = instantiates
→ = protects
→ = proxies

We build object contracts on top of class contracts.



Key:
□ = class
○ = object
● = contracted object
■ = contracted class
◇ = proxy object
→ = instantiates
→ = protects
→ = proxies

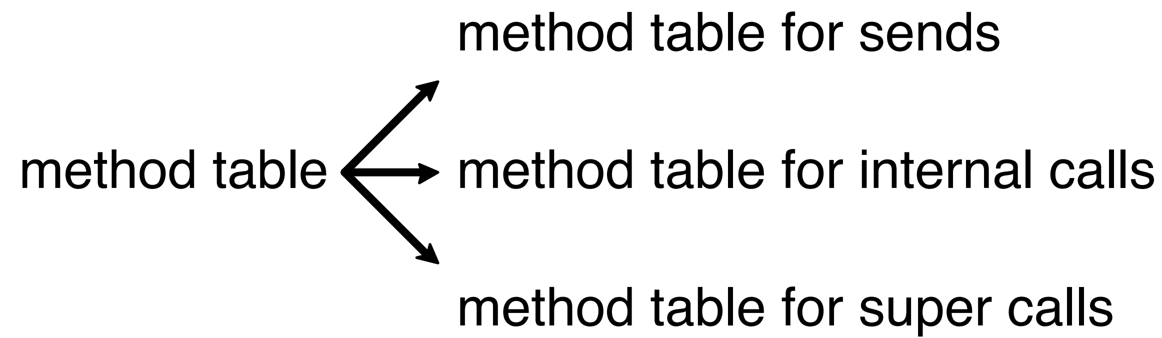
We build object contracts on top of class contracts.



Key:
= class
= object
= contracted object
= contracted class
= proxy object
= instantiates
= protects
= proxies

Implementation Issues

Distinguishing users requires changes to classes.



Internal calls that cross boundaries also require changes.

```
(define stack
  (class object%
    ...
    (define/public (push n) ...)
    (define/public (push-list ns)
      (for/list ([n ns]) (push n)))))
```

```
(define logging-stack
  (class stack
    ... (push x) ...
    (define/override (push n)
      (printf "received value ~v\n" n)
      (super push n))))
```

```
(class/c
 [push (-> number? void?)])
```

Internal calls that cross boundaries also require changes.

```
(define stack
  (class object%
    ...
    (define/public (push n) ...)
    (define/public (push-list ns)
      (for/list ([n ns]) (push n)))))
```

```
(class/c
 [push (-> number? void?)])
```

```
(define logging-stack
  (class stack
    ... (push x) ...
    (define/override (push n)
      (printf "received value ~v\n" n)
      (super push n))))
```

```
(define s (new stack))
(send s push-list (list 3 4 5 6))
```

Internal calls that cross boundaries also require changes.

```
(define stack
  (class object%
    ...
    (define/public (push n) ...)
    (define/public (push-list ns)
      (for/list ([n ns]) (push n)))))
```

```
(class/c
 [push (-> number? void?)])
```

```
(define logging-stack
  (class stack
    ... (push x) ...
    (define/override (push n)
      (printf "received value ~v\n" n)
      (super push n))))
```

```
(define s (new logging-stack))
(send s push-list (list 3 4 5 6))
```

Internal calls that cross boundaries also require changes.

```
(define stack
  (class object%
    ...
    (define/public (push n) ...)
    (define/public (push-list ns)
      (for/list ([n ns]) (push n)))))
```

```
(define logging-stack
  (class stack
    ... (push x) ...
    (define/override (push n)
      (printf "received value ~v\n" n)
      (super push n))))
```

```
(class/c
 [push (-> number? void?)])
```

```
(define s (new logging-stack))
...
```

Bad News

The implementation is not pay-as-you-go.

Up to 3x overhead on some microbenchmarks.

Good News

On macrobenchmarks and programs, no real impact.

More Evaluation

We are adding contracts to Racket's GUI library.

Conclusion

First-class classes are useful.

First-class classes with contracts are better.

Racket has them and you can have them too.

A Challenge

We haven’t found a pay-as-you-go implementation.

Can you?

Thank you

A Challenge

We haven't found a pay-as-you-go implementation.

Can you?

`racket-lang.org`