Big Idea™: Interface vs Implementation

Interface

What something does

Example: a stack
- **create** an empty stack
- **push** an item to top
- **pop** an item from top
- get stack **length** (size)

```python
s = makeStack()
push(s, 42)
print(len(s))  # 1
value = pop(s)
print(value)  # 42
print(len(s))  # 0
```
## Big Idea™: Interface vs Implementation

<table>
<thead>
<tr>
<th>Interface</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>What</strong> something does</td>
<td><strong>How</strong> it’s done</td>
</tr>
<tr>
<td>Example: a stack</td>
<td>Example: a stack</td>
</tr>
<tr>
<td>• create an empty stack</td>
<td>• How to store elements?</td>
</tr>
<tr>
<td>• push an item to top</td>
<td>• What is an empty stack?</td>
</tr>
<tr>
<td>• pop an item from top</td>
<td>• Where is top of stack?</td>
</tr>
<tr>
<td>• get stack length (size)</td>
<td></td>
</tr>
</tbody>
</table>

```
s = makeStack()
push(s, 42)
print(len(s))  # 1
value = pop(s)
print(value)  # 42
print(len(s))  # 0
```
**Big Idea™: Interface vs Implementation**

<table>
<thead>
<tr>
<th>Interface</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>What</strong> something does</td>
<td><strong>How</strong> it’s done</td>
</tr>
<tr>
<td>Example: a stack</td>
<td>Example: a stack</td>
</tr>
<tr>
<td>• create an empty stack</td>
<td>• store items in a list</td>
</tr>
<tr>
<td>• push an item to top</td>
<td>• top is end of list</td>
</tr>
<tr>
<td>• pop an item from top</td>
<td></td>
</tr>
<tr>
<td>• get stack length (size)</td>
<td></td>
</tr>
</tbody>
</table>

---

`s = makeStack()`  
`push(s, 42)`  
`print(len(s))`  # 1  
`value = pop(s)`  
`print(value)`  # 42  
`print(len(s))`  # 0

**Encoding**

how to use preexisting data structures to implement new data structure
**Big Idea™: Interface vs Implementation**

### Interface

**What something does**

Example: a stack
- **create** an empty stack
- **push** an item to top
- **pop** an item from top
- get stack **length** (size)

```python
s = makeStack()
push(s, 42)
print(len(s))  # 1
value = pop(s)
print(value)   # 42
print(len(s))  # 0
```

### Implementation

**How it’s done**

Example: a stack
- store items in a **list**
- top is **end** of list

```python
def makeStack():
    return []
def push(stack, item):
    return stack.append(item)
def pop(stack):
    return stack.pop()
```
## Big Idea™: Interface vs Implementation

<table>
<thead>
<tr>
<th>Interface</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>What</strong> something does</td>
<td><strong>How</strong> it’s done</td>
</tr>
<tr>
<td>Example: a stack</td>
<td>Example: a stack</td>
</tr>
<tr>
<td>• <strong>create</strong> an empty stack</td>
<td>• store items in a <strong>list</strong></td>
</tr>
<tr>
<td>• <strong>push</strong> an item to top</td>
<td>• top is <strong>beginning</strong> of list</td>
</tr>
<tr>
<td>• <strong>pop</strong> an item from top</td>
<td>We can change implementation without changing interface!</td>
</tr>
<tr>
<td>• get stack <strong>length</strong> (size)</td>
<td></td>
</tr>
</tbody>
</table>

```python
def makeStack():
    return []

def push(stack, item):
    return stack.insert(0, item)

def pop(stack):
    return stack.pop(0)
```

```python
s = makeStack()
push(s, 42)
print(len(s))  # 1
value = pop(s)
print(value)   # 42
print(len(s))  # 0
```
Drawbacks of procedural implementation?

**Client code**

```python
import stack1
import stack2

s1 = stack1.makeStack()
stack1.push(s1, 1)
stack1.push(s1, 2)

s2 = stack2.makeStack()
stack2.push(s2, 3)
stack2.push(s2, 4)

value = stack2.pop(s1)
print(value)
```

**Implementations**

```python
def makeStack():
    return []

def push(stack, item):
    return stack.append(item)

def pop(stack):
    return stack.pop()
```

```python
def makeStack():
    return []

def push(stack, item):
    return stack.insert(0, item)

def pop(stack):
    return stack.pop(0)
```

```
T. 11 / 13
```
Drawbacks of procedural implementation?

**Client code**

```python
import stack1
import stack2

s1 = stack1.makeStack()
stack1.push(s1, 1)
stack1.push(s1, 2)

s2 = stack2.makeStack()
stack2.push(s2, 3)
stack2.push(s2, 4)

value = stack2.pop(s1)
print(value)

implementation details can "leak", causing data and behavior to become separated
```

**Implementations**

```python
# stack1.py

def makeStack():
    return []

def push(stack, item):
    return stack.append(item)

def pop(stack):
    return stack.pop()

# stack2.py

def makeStack():
    return []

def push(stack, item):
    return stack.insert(0, item)

def pop(stack):
    return stack.pop(0)
```
programs = data + behavior

encapsulation (separate interface from implementation)

object

- combines data & behavior
- access only through interface
- knows about itself (can access its own data and behavior)

an object is sort-of like a little state machine!
Object-oriented programming languages differ in:

- how the programmer specifies an object's **interface**
- how the programmer specifies an object's **implementation**
- how objects are **created**, **initialized**, **queried**, and **updated**
- **encapsulation** mechanism
  - how strictly the language *enforces* the separation between interface & implementation
Procedural vs object-oriented clients in Python

**Procedural**

```python
import stack1
import stack2

s1 = stack1.makeStack()
stack1.push(s1, 1)
stack1.push(s1, 2)

s2 = stack2.makeStack()
stack2.push(s2, 3)
stack2.push(s2, 4)

value = stack2.pop(s1)
print(value)
```

**Object-oriented**

```python
import stack1
import stack2

s1 = stack1.Stack()
s1.push(1)
s1.push(2)

s2 = stack2.Stack()
s2.push(3)
s2.push(4)

value = s1.pop()
print(value)
```

Implementation details can "leak", causing data and behavior to become separated.

Object stores its own data, method operates on the object's data.
Classes are patterns for objects

A class is like

a cookie cutter

Objects are like

cookies
A procedural stack implementation

Encoded as a list, where the top of the stack is at the end of the list.

```python
def makeStack():
    '''Creates a new stack'''
    return []

def push(stack, item):
    '''Push item onto top of stack'''
    stack.append(item)

def pop(stack):
    '''Removes and returns the item on top of stack'''
    return stack.pop()

The next several slides transform the procedural implementation to an OO implementation.
```
An object-oriented stack implementation

Encoded as a list, where the top of the stack is at the front of the list.

class Stack:
    '''A LIFO data structure'''
    def makeStack():
        '''Creates a new stack'''
        return []
    def push(stack, item):
        '''Push item onto top of stack'''
        stack.append(item)
    def pop(stack):
        '''Removes and returns the item on top of stack'''
        return stack.pop()
An object-oriented stack implementation

Encoded as a list, where the top of the stack is at the front of the list.

class Stack:
    '''A LIFO data structure'''

    def makeStack():
        '''Initializes a new stack'''
        values = []

    def push(item):
        '''Push item onto top of stack'''
        values.append(item)

    def pop():
        '''Removes and returns the item on top of stack'''
        return values.pop()
An object-oriented stack implementation

Encoded as a list, where the top of the stack is at the front of the list.

class Stack:
    '''A LIFO data structure'''

    def makeStack(self):
        '''Initializes a new stack'''
        self.values = []

    def push(self, item):
        '''Push item onto top of stack'''
        self.values.append(item)

    def pop(self):
        '''Removes and returns the item on top of stack'''
        return self.values.pop()
An object-oriented stack implementation

Encoded as a list, where the top of the stack is at the front of the list.

```python
class Stack:
    '''A LIFO data structure'''
    def __init__(self):
        '''Initializes a new stack'''
        self.values = []
    def push(self, item):
        '''Push item onto top of stack'''
        self.values.append(item)
    def pop(self):
        '''Removes and returns the item on top of stack'''
        return self.values.pop()
```

Correct, but not idiomatic

constructor

__init__ is a "special method" that Python calls automatically, to initialize an instance.
An object-oriented stack implementation

Encoded as a list, where the top of the stack is at the front of the list.

```python
class Stack:
    '''A LIFO data structure'''
    def __init__(self):
        '''Initializes a new stack'''
        self._values = []
    def push(self, item):
        '''Push item onto top of stack'''
        self._values.append(item)
    def pop(self):
        '''Removes and returns the item on top of stack'''
        return self._values.pop()
```

"private" field

By convention, if a member begins with _, it's not part of the interface. Clients shouldn't access it directly.
An object-oriented stack implementation

Encoded as a list, where the top of the stack is at the front of the list.

```python
class Stack:
    '''A LIFO data structure'''
    def __init__(self):
        '''Initializes a new stack'''
        self._values = []
    def push(self, item):
        '''Push item onto top of stack'''
        self._values.append(item)
    def pop(self):
        '''Removes and returns the item on top of stack'''
        return self._values.pop()
```

Client code

```python
s = Stack()
s.push(1)
print(s.pop())
```
An object-oriented stack implementation
Encoded as a list, where the top of the stack is at the front of the list.

```python
class Stack:
    '''A LIFO data structure'''
    def __init__(self):
        '''Initializes a new stack'''
        self._values = []
    def push(self, item):
        '''Push item onto top of stack'''
        self._values.append(item)
    def pop(self):
        '''Removes and returns the item on top of stack'''
        return self._values.pop()
    def len(self):
        '''Returns the number of elements in the stack'''
        return len(self._values)
```

Correct, but not idiomatic

Client code
```python
s = Stack()
s.push(1)
print(s.len())
```
An object-oriented stack implementation

Encoded as a list, where the top of the stack is at the front of the list.

class Stack:
    '''A LIFO data structure'''
    def __init__(self):
        '''Initializes a new stack'''
        self._values = []
    def push(self, item):
        '''Push item onto top of stack'''
        self._values.append(item)
    def pop(self):
        '''Removes and returns the item on top of stack'''
        return self._values.pop()
    def __len__(self):
        '''Returns the number of elements in the stack'''
        return len(self._values)

Client code

s = Stack()
s.push(1)
print(len(s))
```python
class Stack:
    ...

s = Stack()
```

Scopes (determined by program code)

Namespaces (a snapshot of program execution)

Built-in (and others)
```python
class Stack:
    ...

s = Stack()
s.x = 42
print(s.x)
print(s.y)
```

**Scopes** (determined by program code)

**Namespaces** (a snapshot of program execution)
```python
class Stack:
    def makeStack():
        values = []
    def push(item):
        values.append(item)
    def pop():
        return values.pop()

s = Stack()
s.makeStack()
s.push(42)
```

**Scopes** (determined by program code)

**Namespaces** (a snapshot of program execution)
```python
class Stack:
    def __init__(self):
        self._values = []
    def push(self, item):
        self._values.append(item)
    def pop(self):
        return self._values.pop()
s = Stack()
s.push(42)
```

**Scopes**
(determined by program code)

**Namespaces**
(a snapshot of program execution)
```python
class Stack:
    def __init__(self):
        self._values = []
    def push(self, item):
        self._values.append(item)
    def pop(self):
        return self._values.pop()
s = Stack()
s.push(42)
```

**Scopes**
(determined by program code)

**Namespaces**
(a snapshot of program execution)

**Built-in**
(print and others)

**Local**
s.__init__, called @ line 12
```python
class Stack:
    def __init__(self):
        self._values = []
    def push(self, item):
        self._values.append(item)
    def pop(self):
        return self._values.pop()

s = Stack()
s.push(42)
```

(scopes (determined by program code))

(built-in)

(namespaces (a snapshot of program execution))

(s)

(instance of Stack)

(global)

(print and others)
```python
class Stack:
    def __init__(self):
        self._values = []
    def push(self, item):
        self._values.append(item)
    def pop(self):
        return self._values.pop()

s = Stack()
s.push(42)
```

**Scopes**
(determined by program code)

**Namespaces**
(a snapshot of program execution)
```python
class Stack:
    def __init__(self):
        self._values = []
    def push(self, item):
        self._values.append(item)
    def pop(self):
        return self._values.pop()
s = Stack()
s.push(42)
```

**scopes**
(determined by program code)

**namespaces**
(a snapshot of program execution)
Python 3.6

```python
1 class Stack:
2     def __init__(self):
3         self._values = []
4     def push(self, item):
5         self._values.append(item)
6     def pop(self):
7         return self._values.pop()
8 s = Stack()
9 s.push(42)
```

Frames

- Global frame
- Stack

Objects

- Stack class
- __init__
- __init__(self)
- pop
- pop(self)
- push
- push(self, item)

Values

- list
  - 42

Try it out! For help, click the exclamation mark in the left margin.