CSC110 Lecture 28: Defining a Shared Public Interface with Inheritance

David Liu and Tom Fairgrieve, Department of Computer Science

*Navigation tip for web slides: press **?** to see keyboard navigation controls.*

Announcements & Today's Plan

Announcements

- Term Test 3 info has been posted
  - And the Reference Sheets
  - And the cover page
- **No tutorial this Friday** (to give you more time to prepare for the term test)
- **No more preps**

Story so far

This week, we learned about three new abstract data types: **Stack**, **Queue**, and **Priority Queue**.

For each one, we:

1. Learned the (abstract) definition.
2. Wrote that definition as the public interface of a Python class.
3. Wrote two different implementations of that public interface.
4. Compared the running times of those two implementations.

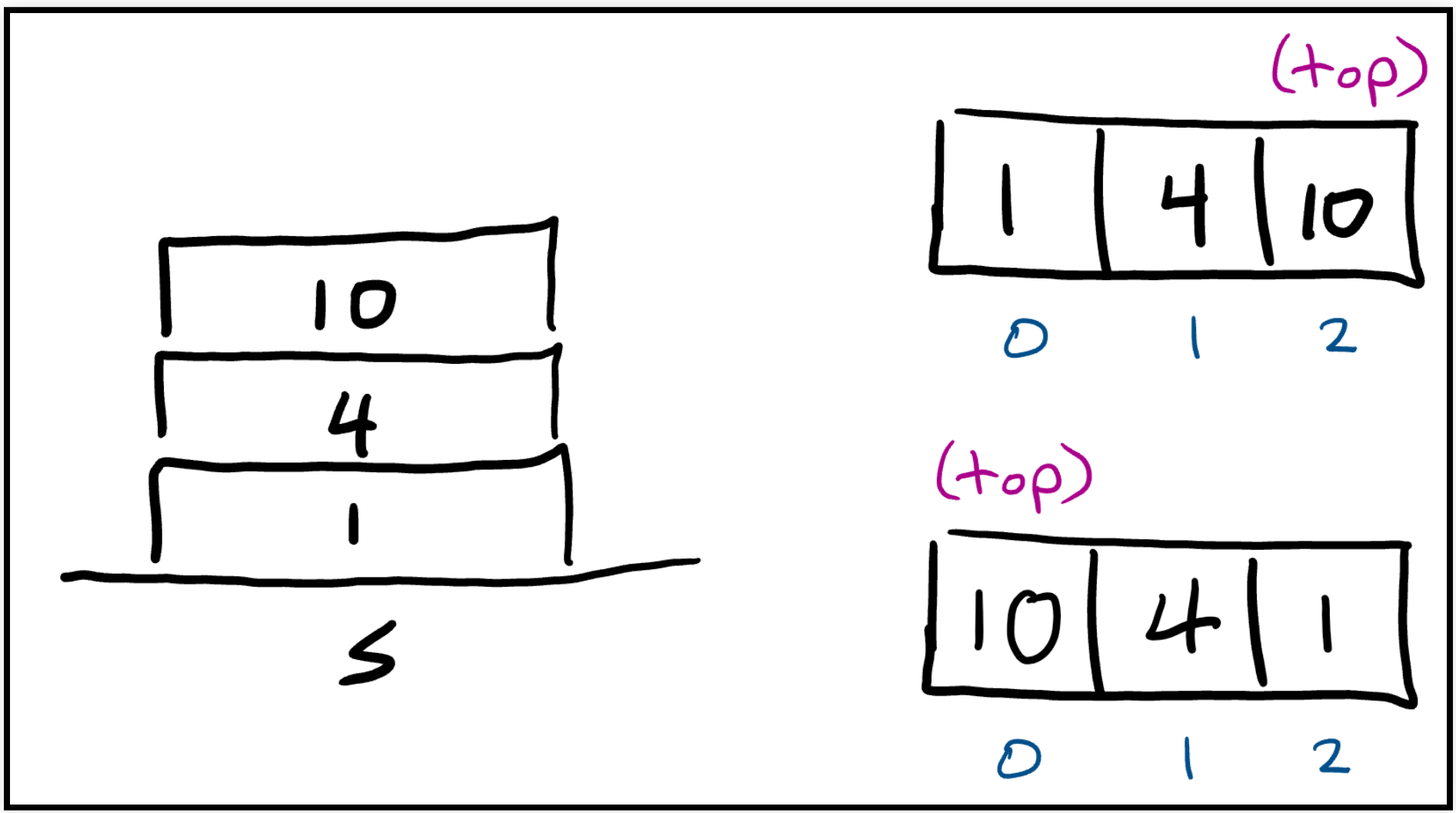
Today, we'll learn about how to represent (in Python) the relationship between multiple, different, implementations of the **same** public interface.

Today you'll learn to...

1. Define the term inheritance and explain its use when designing interfaces.
2. Define the term polymorphism and identify polymorphic functions.
3. Explain the meaning of the `object` superclass in Python.
4. Predict the behaviour of code involving inheritance.

Defining a shared public interface

Recall: we've seen two different stack implementations, `Stack1` and `Stack2`.



```
def push_and_pop(stack: ..., item: Any) -> None:
    stack.push(item)
    stack.pop()
```

What should the type annotation of `stack` be?

Too specific

```
def push_and_pop(stack: Stack1, item: Any) -> None:
    stack.push(item)
    stack.pop()
```

```
def push_and_pop(stack: Stack2, item: Any) -> None:
    stack.push(item)
    stack.pop()
```

Too general

```
def push_and_pop(stack: Any, item: Any) -> None:
    stack.push(item)
    stack.pop()
```

Idea: define a Python data type that represents "an arbitrary stack class".

Stack ADT

- Data: A collection of items
- Operations:
  - determine whether the stack is empty
  - add an item (`push`)
  - remove the most recently-added item (`pop`)

The **public interface** of the Stack ADT, written as Python code:

```
class Stack:
    def is_empty(self) -> bool:
        """Return whether this stack contains no items."""
        raise NotImplementedError

    def push(self, item: Any) -> None:
        """Add a new element to the top of this stack."""
        raise NotImplementedError

    def pop(self) -> Any:
        """Remove and return the element at the top of this stack.

        Precondition: not self.is_empty()
        """
        raise NotImplementedError
```

What does `raise NotImplementedError` mean?

An **abstract method** is a method whose body is `raise NotImplementedError`.

An **abstract class** is a class with at least one abstract method.
(Opposite: **concrete class**)

Abstract classes aren't meant to be instantiated directly…

But abstract classes can be used as type annotations!

```
def push_and_pop(stack: Stack, item: Any) -> None:
    stack.push(item)
    stack.pop()
```

What about `Stack1` and `Stack2`?

Inheritance and Polymorphism

In Python, we specify that one class implements an abstract interface through **inheritance**.

```
class Stack1(Stack):
    ...


class Stack2(Stack):
    ...
```

Terminology:

- `Stack` is the superclass or parent class
- `Stack1`, `Stack2` are subclasses or child classes
- `Stack1` and `Stack2` inherit from `Stack`
- `Stack` defines a shared public interface that is implemented by both `Stack1` and `Stack2`

Inheritance as a contract

Given an abstract class `A` and concrete class `B` that inherits from `A`:

- The implementor of the subclass `B` is required to implement all abstract methods from the abstract superclass `A`.
- Any user of the subclass `B` may assume that they can call the superclass methods on instances of the subclass.

Calling `push_and_pop` (1)

```
def push_and_pop(stack: Stack, item: Any) -> None:
    stack.push(item)
    stack.pop()


>>> s1 = Stack1()
>>> push_and_pop(s1, 10)
```

When `push_and_pop(s1, 10)` gets called, `stack` and `s1` are aliases, so...

- `s1.push(10)` gets called (`Stack1.push` method)
- `s1.pop()` gets called (`Stack1.pop` method)

Calling `push_and_pop` (2)

```
def push_and_pop(stack: Stack, item: Any) -> None:
    stack.push(item)
    stack.pop()


>>> s2 = Stack2()
>>> push_and_pop(s2, 10)
```

When `push_and_pop(s2, 10)` gets called, `stack` and `s2` are aliases, so...

- `s2.push(10)` gets called (`Stack2.push` method)
- `s2.pop()` gets called (`Stack2.pop` method)

Polymorphism

```
def push_and_pop(stack: Stack, item: Any) -> None:
    stack.push(item)
    stack.pop()


>>> s1 = Stack1()
>>> push_and_pop(s1, 10)

>>> s2 = Stack2()
>>> push_and_pop(s2, 10)
```

We say function `push_and_pop` is **polymorphic**, because it can be given argument values that have different concrete data types.

(polymorphic means many forms)

Polymorphism

In general, a function is **polymorphic** if it can take in input values of different concrete data types.

Many built-in functions (`len`, `sum`, `abs`) are polymorphic!

But this is the first time we've written polymorphic code on data types we've defined ourselves.

Exercises 1 (Inheritance) and 2 (Polymorphism)

Two technical Python notes

object dot notation vs. class dot notation

This version is correct:

```
def push_and_pop(stack: Stack, item: Any) -> None:
    stack.push(item)
    stack.pop(stack)
```

This version is **incorrect**:

```
def push_and_pop(stack: Stack, item: Any) -> None:
    Stack.push(stack, item)
    Stack.pop(stack)
```

`type` vs. `isinstance`

How can we check the type of an object?

```
>>> my_stack = Stack1()

>>> type(my_stack) is Stack1
True
>>> isinstance(my_stack, Stack1)
True
```

But:

```
>>> type(my_stack) is Stack
False
>>> isinstance(my_stack, Stack)
True
```

`type` vs. `isinstance`

```
>>> type(my_stack) is Stack
False
>>> isinstance(my_stack, Stack)
True
```

`type(x) is t` returns whether `x` is an object of type `t`.

`isinstance(x, t)` returns whether `x` is an object of type `t` **or any subclass of `t`.**

The `object` superclass and method inheritance

`object` is a special built-in Python class that is the default superclass for all classes (even abstract ones).

```
class MyClass:
    ...

# The above definition contains an implicit "(object)":
class MyClass(object):
    ...
```

`object` special methods

`object` defines a few special methods, including:

- `__init__`
- `__str__`
- `__eq__`

These methods aren't abstract, they have an implementation that acts as a default for all classes.

Demo!

But how does this "default" implementation actually work?

Method inheritance

Let `A` and `B` be Python classes, and assume `B` is a subclass of `A`.

If `A` has a method `m` and `B` does not implement the same method, then `B` inherits the method `m` from `A`.

All instances of `B` can call `A.m`!

```
class A:
    def m(self) -> int:
        return 1


class B(A):
    # no method m defined
```

```
>>> my_b = B()
>>> my_b.m()
1
```

Method overriding

Let `A` and `B` be Python classes, and assume `B` is a subclass of `A`.

If `A` has a method `m` and `B` implements the same method, then `B` overrides the method `m` from `A`.

All instances of `B` call the `B.m` method.

```
class A:
    def m(self) -> int:
        return 1


class B(A):
    def m(self) -> int:
        return 100
```

```
>>> my_b = B()
>>> my_b.m()
100
```

Parting thoughts: abstraction and interfaces

As computer scientists and software developers, identifying and communicating public interfaces and separating interfaces from implementation are critical skills.

An **Application Programming Interface (API)** is a public interface that an application provides to allow other programs to interact with it.

For example, the Instagram Basic Display API "allows users of your app to get basic profile information, photos, and videos in their Instagram accounts."

Abstract data types are a common language of interfaces that transcends any one particular programming language.

> *"I'm going to use a Stack to store this data"*

vs.

> *"I'm going to use a List to store this data"*

Separating interface from implementation let implementers modify and improve implementations without disrupting their users.

Operation	`PriorityQueueUnsorted` runtime	`PriorityQueueSorted` runtime	H r
`enqueue`	$\Theta(1)$	$\Theta(n \log n)$	Θ
`dequeue`	$\Theta(n)$	$\Theta(1)$	Θ

Inheritance in Python allows us to explicitly specify public interfaces separate from their implementations (and allow multiple implementations of the same interface).

```
def push_and_pop(stack: Stack, item: Any) -> None:
    stack.push(item)
    stack.pop()
```

Summary

Today you learned to...

1. Define the term inheritance and explain its use when designing interfaces.
2. Define the term polymorphism and identify polymorphic functions.
3. Explain the meaning of the `object` superclass in Python.
4. Predict the behaviour of code involving inheritance.

Homework

- Today's readings: 10.9, 10.10
- Next week: Chapter 11 (last chapter!)
- Study for Term Test 3 (on **Monday**)
- No more preps!
- No tutorial tomorrow; last tutorial on Friday December 2