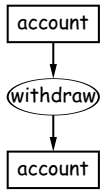
ecture #7: Object-Based Programming

ed programs are organized primarily around the func-ds, etc.) that do things. Data structures (objects) are eparate.

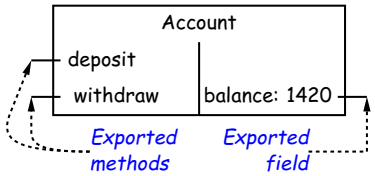
d programs are organized around the *types of objects* d to represent data; methods are grouped by type of

ng-system example:

n-based



Object-based



All (Maybe) in CS61A: The Account Class

```
(self, balance0):
ance = balance0

(self, amount):
ance += amount
elf.balance

(self, amount):
balance < amount:
e ValueError \
("Insufficient funds")

.balance -= amount
elf.balance
```

```
unt(1000)
balance)
t(100)
aw(500)
```

```
public class Account {
  public int balance;
  public Account(int balance0) {
    this.balance = balance0;
  }
  public int deposit(int amount) {
    balance += amount; return balance;
  }
  public int withdraw(int amount) {
    if (balance < amount)
      throw new IllegalStateException
        ("Insufficient funds");
    else balance -= amount;
    return balance;
  }
}
```

```
Account myAccount = new Account(1000);
print(myAccount.balance)
myAccount.deposit(100);
myAccount.withdraw(500);
```


The Pieces

ation defines a *new type of object,* i.e., new type of ontainer.

iables such as `balance` are the simple containers within s (*fields* or *components*).

thods, such as `deposit` and `withdraw` are like ordinary ods that take an invisible extra parameter (called **this**).

rator creates (*instantiates*) new objects, and initializes onstructors.

such as the method-like declaration of `Account` are ods that are used only to initialize new instances. They guments from the **new** expression.

ction picks methods to call. For example,

```
myAccount.deposit(100)
```

all the method named `deposit` that is defined for the ed to by `myAccount`.

Recreation

$$\log(1+x) = x - \frac{1}{2}x^2 + \frac{1}{3}x^3 - \ldots$$

case that

$$+ 1/3 - 1/4 + 1/5 - 1/6 + 1/7 - 1/8 + 1/9 - \ldots$$
$$3 + 1/5 + 1/7 + 1/9 + \ldots) - (1/2 + 1/4 + 1/6 + 1/8 + \ldots)$$
$$3 + 1/5 + 1/7 + 1/9 + \ldots) + (1/2 + 1/4 + 1/6 + 1/8 + \ldots)$$
$$+ 1/4 + 1/6 + 1/8 + \ldots)$$
$$2 + 1/3 + 1/4 + \ldots) - (1 + 1/2 + 1/3 + 1/4 + \ldots)$$

Philosophy

970s and before): An *abstract data type* is

ossible values (a *domain*), plus

perations on those values (or their containers).

for example, the domain was a *set of pairs:* `(head,tail)`, s an `int` and `tail` is a pointer to an `IntList`.

operations consisted only of assigning to and accessing ds (`head` and `tail`).

e prefer a purely *procedural interface,* where the func-ds) do everything—no outside access to the internal on (i.e., instance variables).

plementor of a class and its methods has complete con-avior of instances.

preferred way to write the "operations of a type" is as *thods.*

You Also Saw It All in CS61AS

```
ccount balance0)
s (balance 0))

e balance0))

it amount)
e (+ balance amount))

raw amount)
ce amount)
sufficient funds")

lance (- balance amount))
)) )
```

```
nt
ccount 1000))
'balance)
'deposit 100)
'withdraw 500)
```

```
public class Account {
  public int balance;
  public Account(int balance0) {
    balance = balance0;
  }
  public int deposit(int amount) {
    balance += amount; return balance;
  }
  public int withdraw(int amount) {
    if (balance < amount)
      throw new IllegalStateException
        ("Insufficient funds");
    else balance -= amount;
    return balance;
  }
}
```

```
Account myAccount = new Account(1000);
myAccount.balance
myAccount.deposit(100);
myAccount.withdraw(500);
```

Class Variables and Methods

want to keep track of the bank's total funds.

is not associated with any particular Account, but is ll—it is *class-wide*. In Java, "class-wide" ≡ `static`.

```
lass Account {

 static int _funds = 0;
 int deposit(int amount) {
nce += amount;
ls += amount;       // or this._funds or Account._funds
n _balance;

 static int funds() {
n _funds;           // or Account._funds

 Also change withdraw.
```

, can refer to either `Account.funds()` or to `unds()` (same thing).

Getter Methods

em with Java version of `Account`: anyone can assign to field

the control that the implementor of Account has over es of the balance.

w public access only through methods:

```
lass Account {
 int _balance;

 int balance() { return _balance; }
```

`._balance = 1000000` is an error outside `Account`.

nvention of putting '_' at the start of private instance distinguish them from local variables and non-private uld actually use `balance` for both the method and the please don't.)

Calling Instance Method

```
nal) equivalent of deposit instance method. */
deposit(final Account this, int amount) {
nce += amount;
 amount;
is._balance;
```

instance-method call `myAccount.deposit(100)` is like fictional static method:

```
nt.deposit(myAccount, 100);
```

instance method, as a convenient abbreviation, one can e leading '`this.`' on field access or method call if not Unlike Python)

Instance Methods

thod such as

```
it(int amount) {
e += amount;
+= amount;
balance;
```

of like a static method with hidden argument:

```
t deposit(final Account this, int amount) {
balance += amount;
+= amount;
this._balance;
```

explanatory: Not real Java (not allowed to declare l *is* real Java; means "can't change once initialized.")

'Instance' and 'Static' Don't Mix

atic methods don't have the invisible `this` parameter, se to refer directly to instance variables in them:

```
atic int badBalance(Account A) {
= A._balance;  // This is OK
               // (A tells us whose balance)

 _balance;     // WRONG! NONSENSE!
```

_balance here equivalent to `this._balance`,

eaningless (*whose* balance?)

makes perfect sense to access a static (class-wide) field an instance method or constructor, as happened with deposit method.

one of each static field, so don't need to have a 'this' just name the class (or use no qualification inside the e been doing).

Constructors

y control objects of some class, you must be able to set contents.

r is a kind of special instance method that is called by ator right after it creates a new object, as if

tList(1,null) ⟹
```
tmp = pointer to [0|\]
tmp.IntList(1, null);
L = tmp;
```

nstructors and Instance Variables

iables initializations are moved inside constructors that
ith this(...).

```
{                  class Foo {
 5;                   int x;

 y) {                 Foo(int y) {
uff(y);                  x = 5;
                         DoStuff(y);
          ⟺           }

                      Foo() {
(42);                    this(42); // Assigns to x
                      }
                   }
```


Constructors and Default Constructors

ave constructors. In the absence of any explicit con-
the **default constructor**, as if you had written:

```
ass Foo {
c Foo() {   }
```

rloaded constructors are possible, and they can use
lthough the syntax is odd):

```
ass IntList {
c IntList(int head, IntList tail) {
his.head = head; this.tail = tail;


c IntList(int head) {
his(head, null);    // Calls first constructor.
```


Summary: Java vs. Python

Java	Python

...;
)
. }
..)
}
int y = 21;
void g(...)
}
	class Foo: ...
	x = ...
	def __init__(self, ...):
	...
	def f(self, ...):
	...
	y = 21 # Referred to as Foo.y
	@staticmethod
	def g(...):
	...

)	aFoo.f(...) aFoo.x Foo(...) self # (typically)