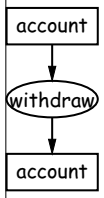
ecture #7: Object-Based Programming

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

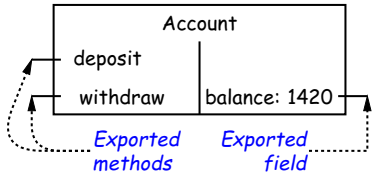
d programs are organized around the types of objects
d to represent data; methods are grouped by type of

ng-system example:

-based



Object-based



Announcements

weekly group tutoring offered by the course tutors
leased!

se on Saturday, 9/9, at 11:59PM.

ive room and time assignments on Sunday via email.

start next week and will be focused on strengthening
s.

ation pinned on Piazza.

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) {
    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);
```


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
ls (`head` and `tail`).

prefer a purely *procedural interface*, where the func-
ds) do everything—no outside access to fields.

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

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

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`.

You Also Saw It All in CS61AS

```
ccount balance0)
 (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*.

ss-wide" ≡ static

```
lass Account {

 static int funds = 0;
 int deposit(int amount) {
ace += amount; funds += amount;
rn balance;

 static int funds() {
rn funds;

/ Also change withdraw.
```

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

Calling Instance Method

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

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

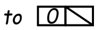
```
unt.deposit(myAccount, 100);
```

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

Constructors

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

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

tList(1,null) ⟹ { tmp = *pointer to* 
tmp.IntList(1, null);
L = tmp; }

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.

ow public access only through methods:

```
lass Account {
 int balance;

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

.balance = 1000000 is an error outside Account.

ame balance for both the field and the method. Java
h is meant by syntax: A.balance vs. A.balance(). How-
bably better to choose differing names to avoid confu-

Instance Methods

thod such as

```
sit(int amount) {
 += amount; funds += amount;
 balance;
```

of like a static method with hidden argument:

```
nt deposit(final Account this, int amount) {
alance += amount; funds += amount;
 this.balance;
```

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

Instance' and 'Static' Don't Mix

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

```
tatic int badBalance(Account A) {
 = A.balance;  // This is OK
               // (A tells us whose balance)
n balance;     // WRONG! NONSENSE!
               // (Whose balance?)
```

o balance here equivalent to this.balance,

eaningless (*whose* balance?)

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

one of each static field, so don't need to have a 'this' to
ust name the class.

nstructors and Instance Variables

iables initializations are moved inside all constructors:

```
ss Foo {
nt x = 5;
oo() {
 DoStuff();
```

$\Longleftrightarrow$

```
class Foo {
  int x;
  Foo() {
    x = 5;
    DoStuff();
  }
}
```


Constructors and Default Constructors

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

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

rloaded constructors possible, and they can use each
ugh 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)