LECTURE 7: POLYMORPHISM, PART II

CS 2110
Fall 2021

LECTURE 7: POLYMORPHISM, PART II

PART 1: SUBTYPING

CS 2110
Fall 2021

Agenda

Previously in 2110:

- Objects and classes
- Encapsulation
- Inheritance
- Polymorphism: overloading, autoboxing, generics

Today:

- Subtyping
- Static vs. dynamic types (compile-time vs. runtime)
- Dynamic dispatch
- Equality

Review: Polymorphism in Programming

- Gk. poly = many, morph = form
- Polymorphism: language treats as though same, despite differences
- General phenomenon with three occurrences in Java:
  - **Ad-hoc** polymorphism
  - **Parametric** polymorphism
  - **Subtype** polymorphism

Recall: Arrays


```
int[] a= new int[2];
a[0]= 2110;  // ok
a[1]= "2110";  // error
```

but

```
Account[] accts= new Account[2];
accts[0]= new Account("A-1”);
accts[1]= new InterestAccount("IA-1"); // ok!
for (Account a : accts) { a.printStatement(); }
```

`InterestAccount` is a subtype of `Account`...

Subtyping

Subtype rule: if S is a subtype of T, then anywhere a value of type T is required a value of type S may be used instead

We show 4 such situations

Variable assignment:

```
S s= new S();
T t= s;
```

Array assignment:

```
T[] a= new T[10];
a[0]= s;
```

Argument:

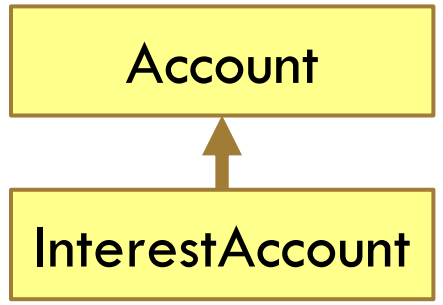
```
void foo(T t) { … }
foo(s);
```

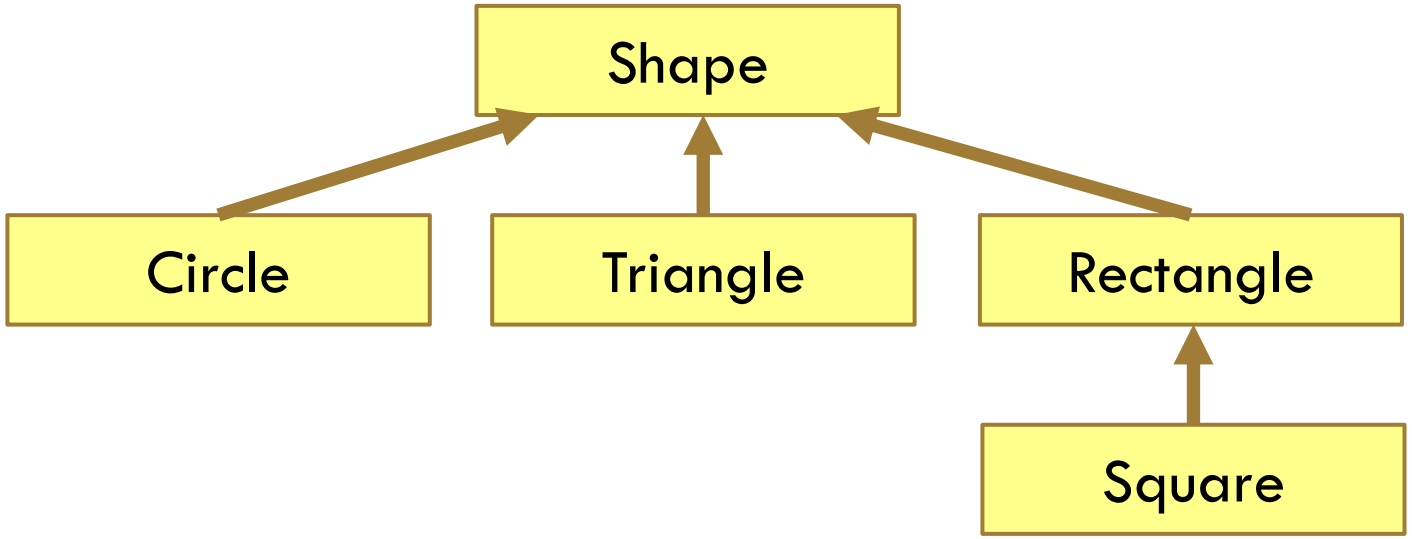
Return:

```
T bar() { return s; }
```

Subclasses are Subtypes

If S is a subclass of C,
then S is a subtype of C



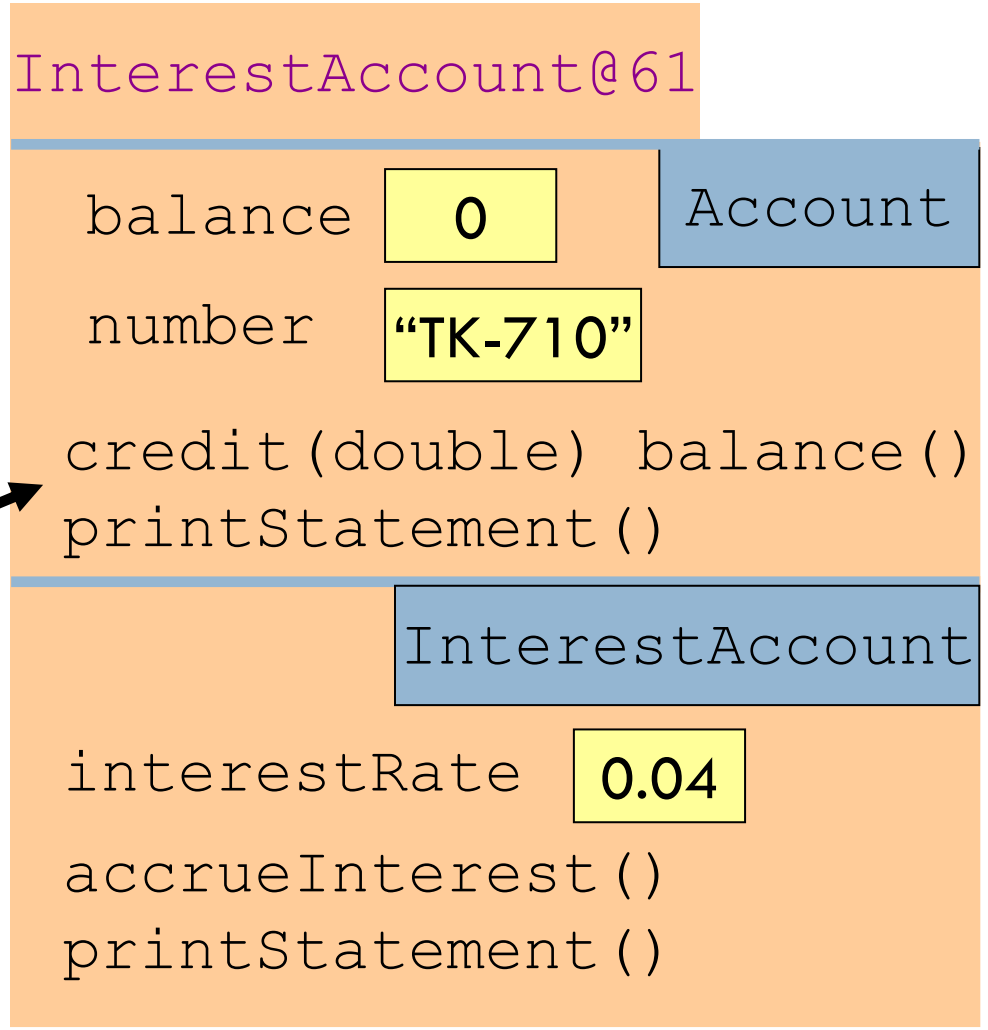


Why…?

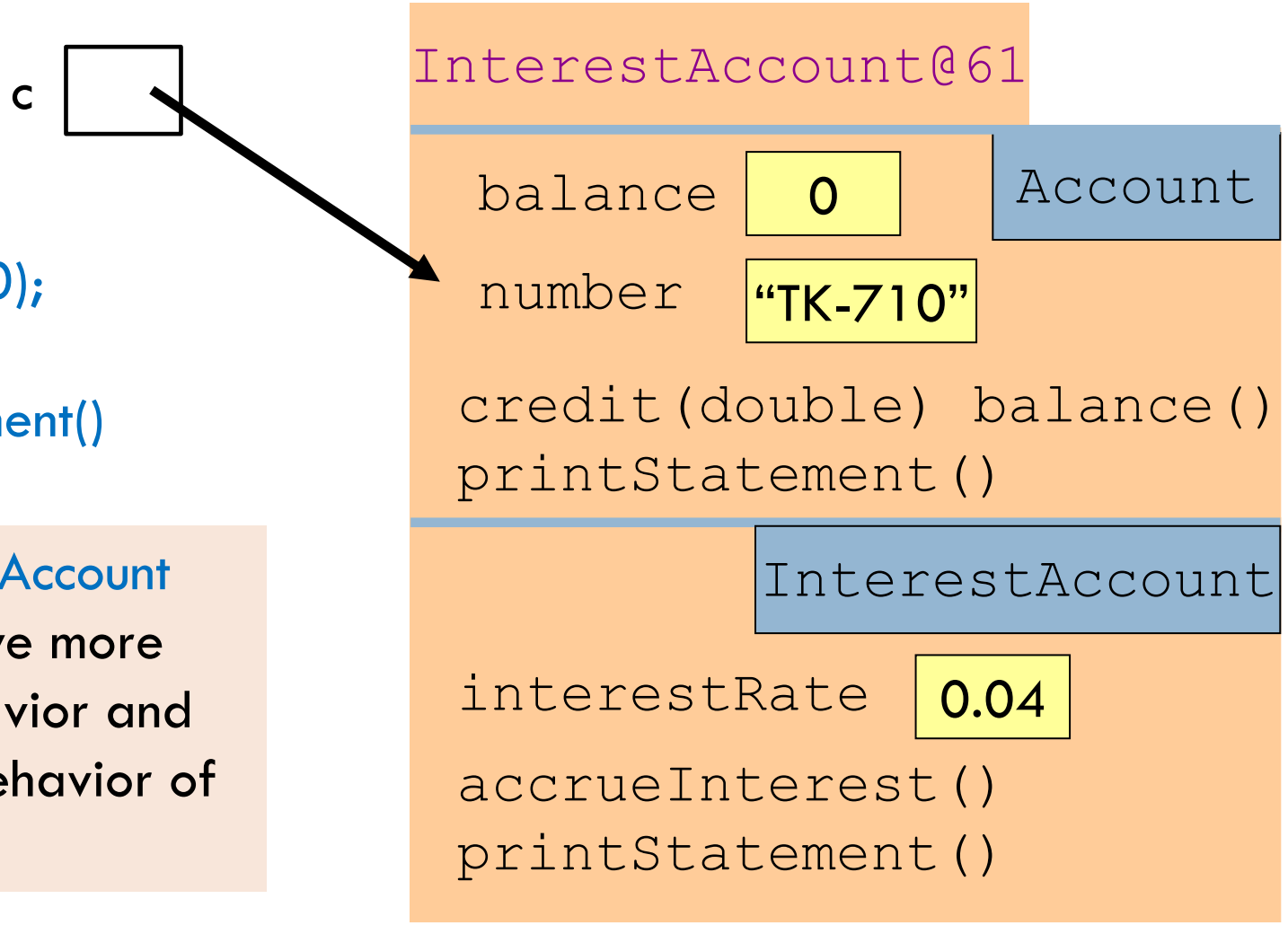
Subclasses are Subtypes

Type:
a set of values together with operations on them

An InterestAccount object can accept and respond to any Account method



Subclasses are Subtypes



c.credit(1000);
c.balance()
c.printStatement()

Though InterestAccount object may have more state and behavior and may change behavior of some methods.

Subtype Polymorphism

Why is it polymorphic?

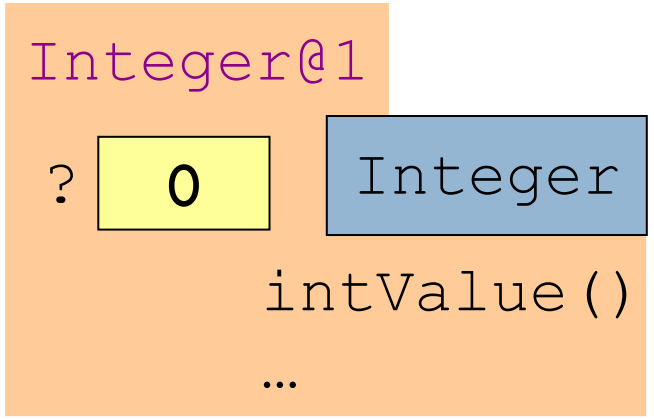
- Type can have “many forms”
- Every subtype is a different form
- But subtypes can be treated the same as supertype

Subtyping vs. Wrappers

Is `int` a subtype of `Integer`? Quiz No!!!

`int` is not a subclass of `Integer`, nor vice-versa

- You **can** use a value of one in place of the other
- But that's thanks to **ad-hoc polymorphism** (autoboxing and coercion), not to subtype polymorphism



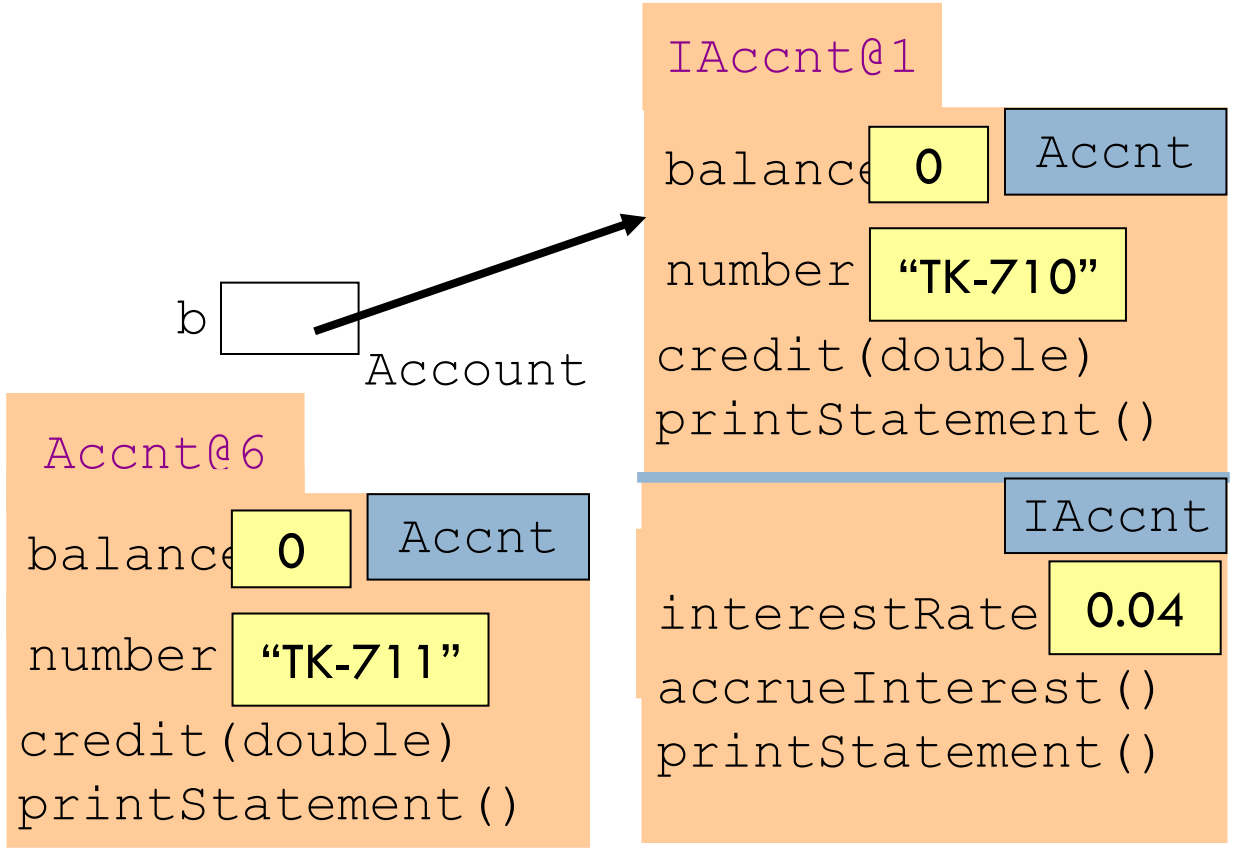
LECTURE 7: POLYMORPHISM, PART II

PART 2: COMPILE-TIME VS RUNTIME STATIC TYPES VS DYNAMIC TYPES

CS 2110
Fall 2021

(Static) Types

Every expression has a type, known at compile-time. For emphasis and later use, we call it the **static** type of the expression.

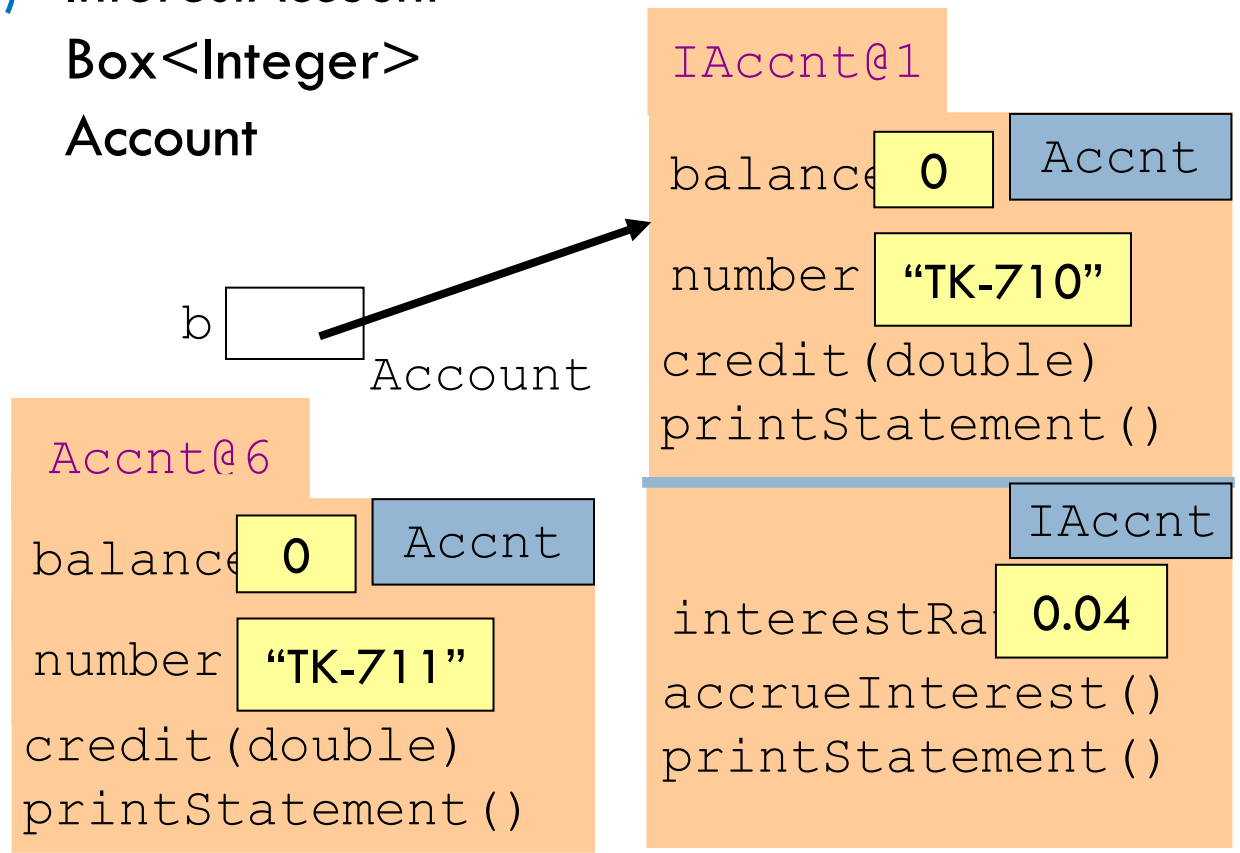


(static) Types

Expression	(static) type
new Account()	Account
new InterestAccount()	InterestAccount
new Box<Integer>()	Box<Integer>
b	Account

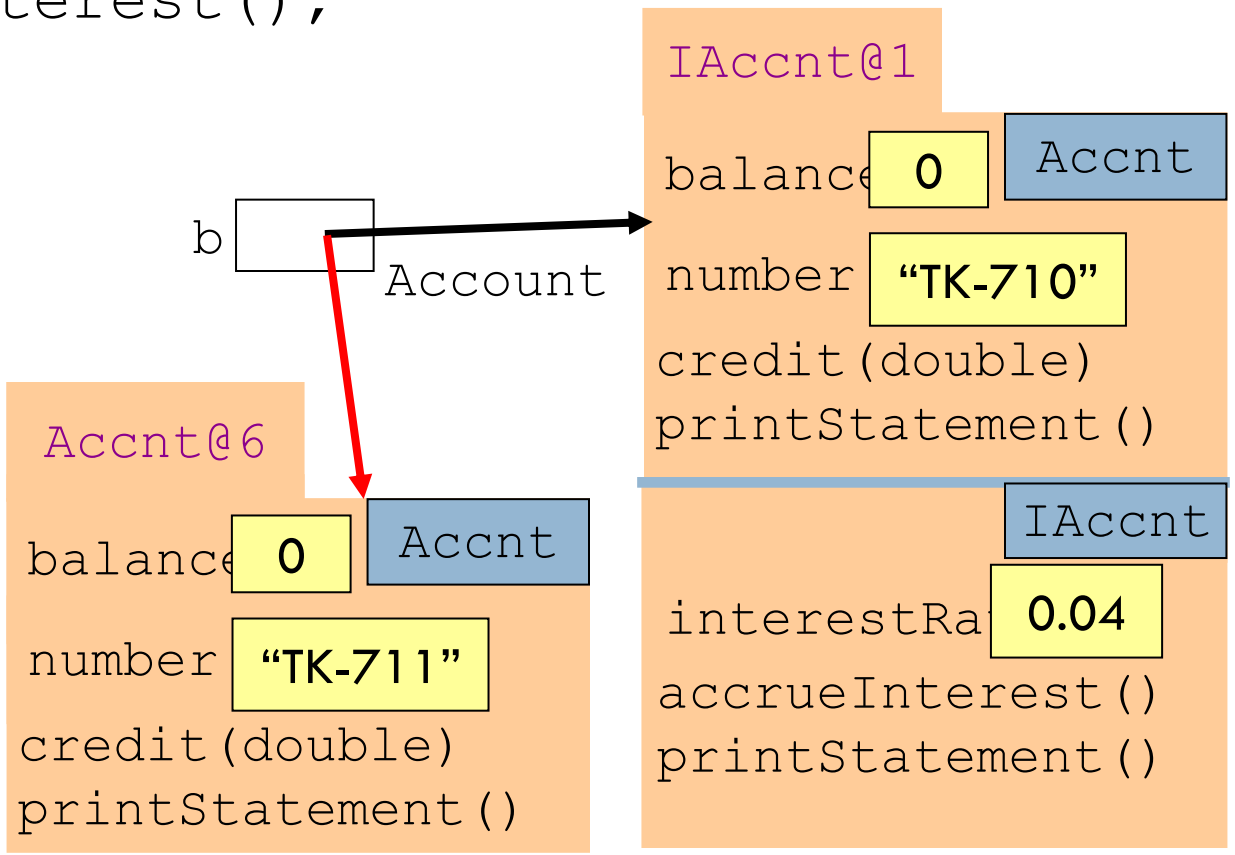
Declaration of b:

Account b;



Checking method calls

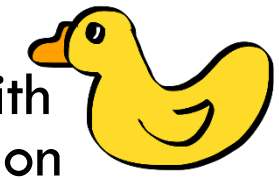

```
Account b= …;
…
b.accrueInterest();
```



Allow method call?

`obj.m(args)`

- **Possibility 1:** Always allow at compile-time
  - Check at runtime whether `obj` has a method `m` with an appropriate signature; if not, throw an exception
  - Method calls can fail at runtime
  - Python does this and calls it duck typing. It's a quack.



- **Possibility 2:** Disallow at compile-time if static type of `obj` does not have an appropriate method
  - Method calls cannot fail at runtime
  - Java does this.

Image: https://pixabay.com/illustrations/yellow-bath-duck-duck-cartoon-183052/

Compile-Time Reference Rule

Suppose the (static) type of ob is C.

Then ob.m(…) is *legal* only if:

- m(…) is declared in C or one of C's supertypes.

- Another way to think of this: compiler can only see C's partition or a partition above C's partition in the object folder that could be referenced by a variable.

Compile-Time Reference Rule

Suppose the (static) type of ob is C.

Then ob.m(…) is legal only if:

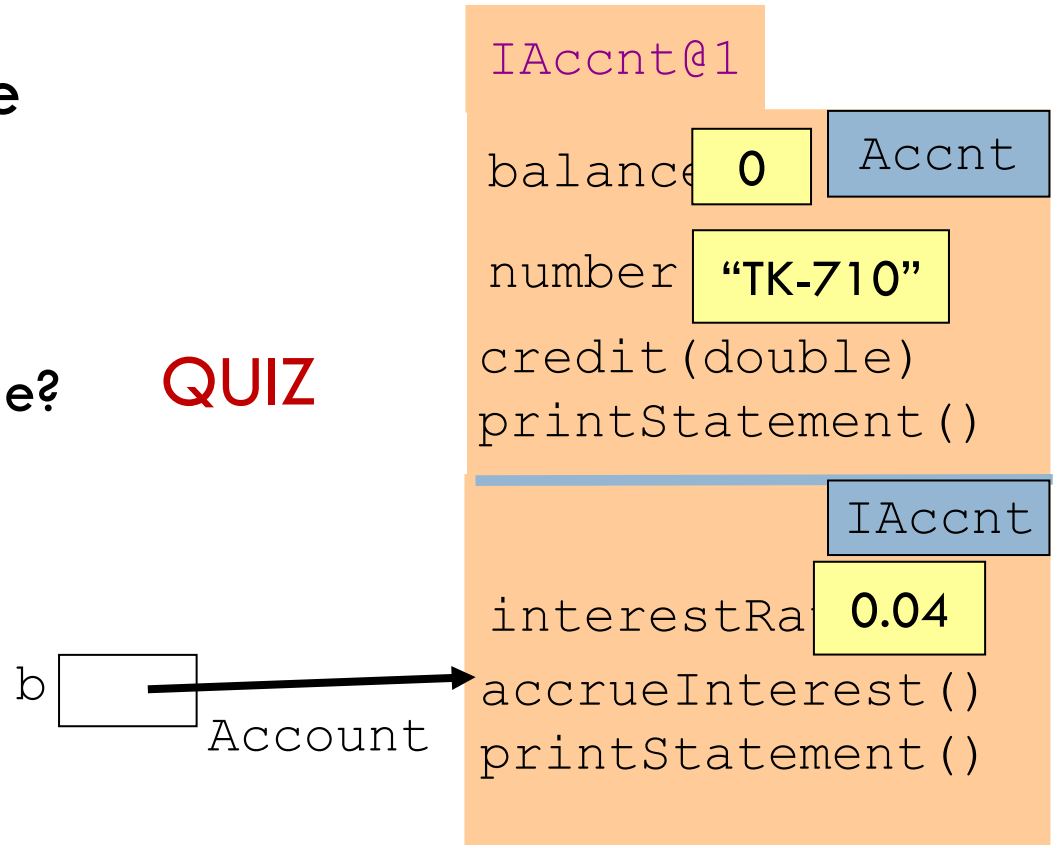
m(…) is declared in C or one of C's supertypes.

Is this legal, will it compile? QUIZ

b.accrueInterest()

Declaration of b:

Account b;



Compile-Time Reference Rule

Suppose the (static) type of ob is C.

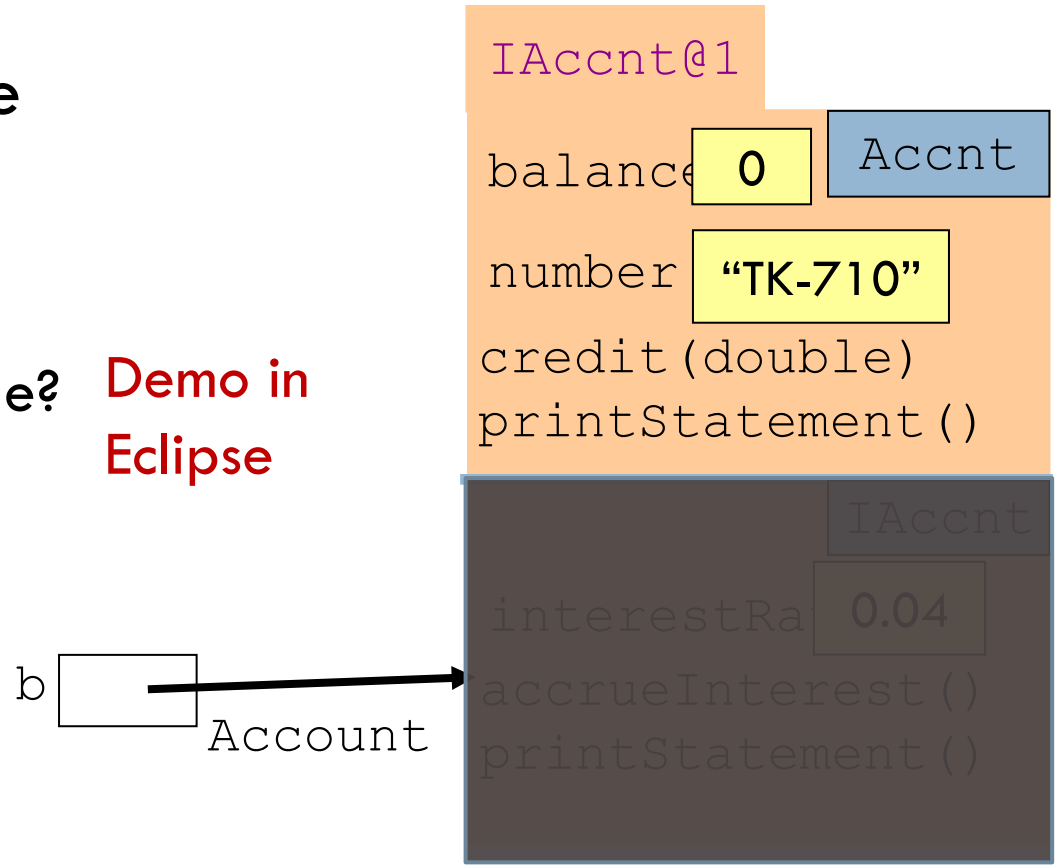
Then ob.m(…) is legal only if:

m(…) is declared in C or one of C's supertypes.

Is this legal, will it compile?
b.accrueInterest()

Demo in Eclipse

Declaration of b:
Account b;

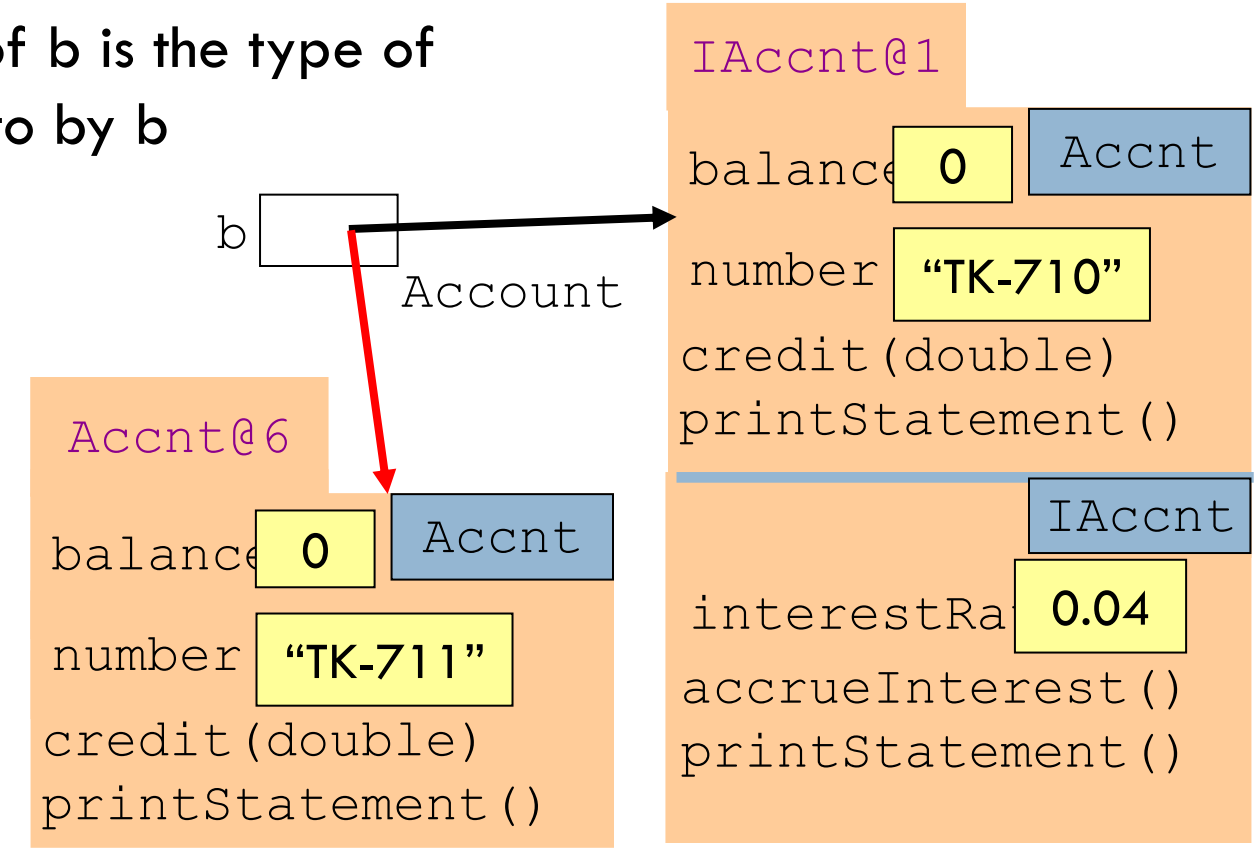


Static Types vs. Dynamic Types
Compile-time vs. Runtime

Static type: the type known at compile-time

Dynamic type: the type known at runtime

The dynamic type of b is the type of the object pointed to by b



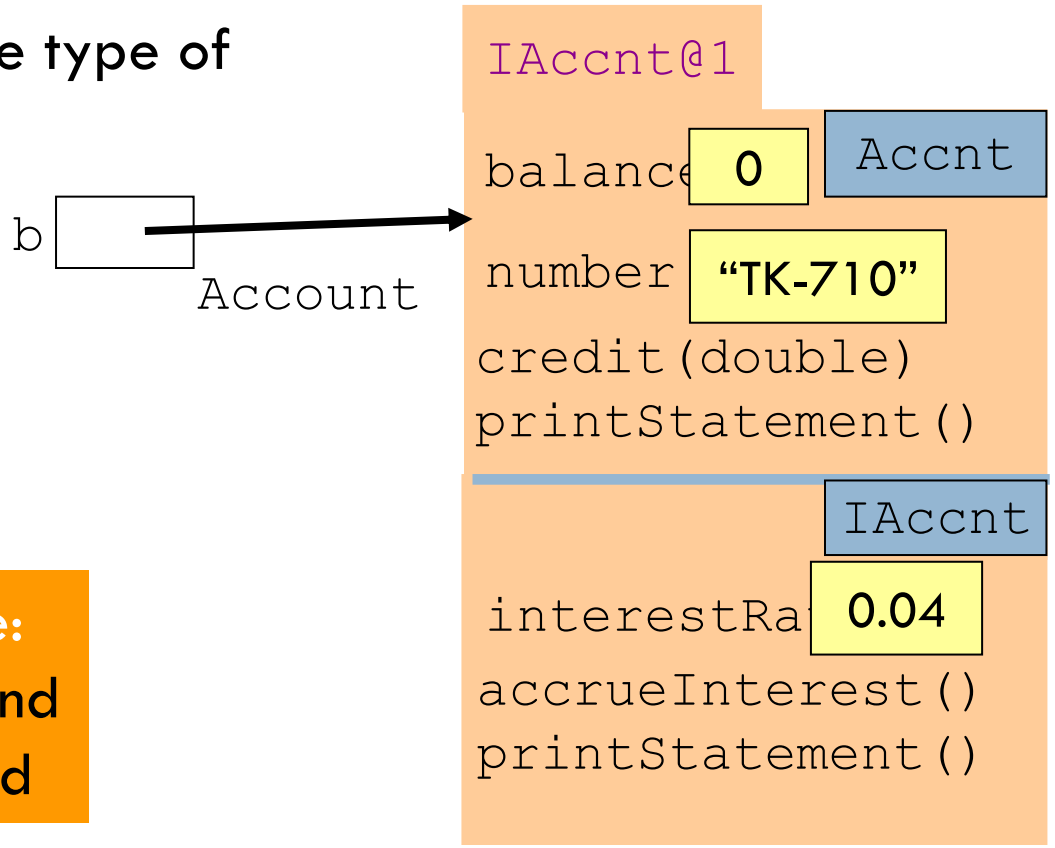
Static Types vs. Dynamic Types
Compile-time vs. Runtime

Static type: the type known at compile-time

Dynamic type: the type known at runtime

The dynamic type of b is the type of the object pointed to by b

```
b.printStatement();
```



Bottom-up/ Overriding rule:
Start at bottom of folder and work upward to find method

Bottom-Up/Overriding Rule, Revisited

- Static type is irrelevant to bottom-up/Overriding rule
- Dynamic type is what matters: what partitions the object's folder really has at runtime

- Dynamic dispatch: method to call is chosen at runtime, not compile-time
  - The object itself gets to pick the implementation
  - **A defining feature of OOP…**

Dynamic Dispatch

- "Dynamic [dispatch] in OO languages is perhaps **the most important…aspect** of these languages."
  [Milton and Schmidt, *Dynamic Dispatch in OO Languages*, 1994, p. 1]

- "Dynamic dispatch is found in all OO languages, to the point that it can be regarded as **one of their defining properties.**"
  [Abadi and Cardelli, *A Theory of Objects*, 1996, p. 18]

- "Perhaps **the most basic characteristic of the OO style** is that, when an operation is invoked on an object, the object itself determines what code gets executed…a process called dynamic dispatch."
  [Benjamin Pierce, *Types and Programming Languages*, 2002, p. 226]

Static vs. dynamic types

- Compile-time reference rule: The static type of an expression determines which method calls we are ***allowed to write***
- Dynamic dispatch (bottom-up rule): The dynamic type of an object determines which method implementation is ***executed at runtime***
  - *Some* implementation of the method is guaranteed to be executed (assuming target is not null); otherwise, we wouldn't have been allowed to write the call!

LECTURE 7: POLYMORPHISM, PART II

PART 3: MANIPULATING TYPES

CS 2110
Fall 2021

Querying Dynamic Types

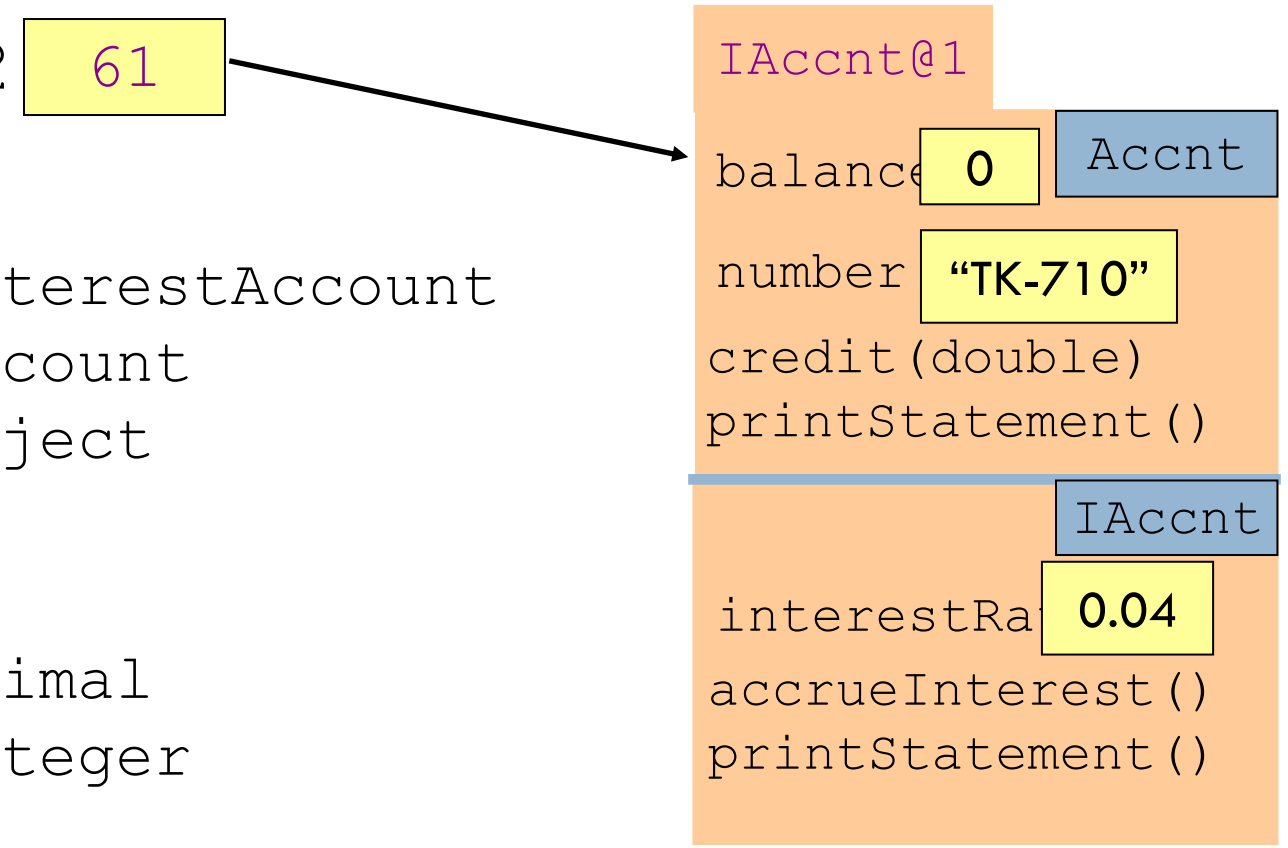
- Operator `instanceof`: `ob instanceof C`
  - Does object ob have C as a supertype or subtype?
    i.e. does ob have a partition named C?
- Method ob.`getClass()` and static field `class`:
  - Return an object that describes the dynamic type of ob
    ob= new C(…); Object ob.getClass() describes class C
    i.e. it describes the class for the lowest partition in object ob.
    Will be equal to C.class.

Operator `instanceof`

- `ob` **`instanceof`** `C` **is** `true` **iff:**
  - `ob`'s folder contains a `C` partition
  - i.e. `ob`'s dynamic type is a subtype of `C`
- `null instanceof` *<anything>* **is** `false`

Operator instanceof


```
InterestAccount a2=
        new InterestAccount("TK-710", 0.04);
```



These are true:

```
a2  instanceof  InterestAccount
a2  instanceof  Account
A2  instanceof  Object
```

These are false:

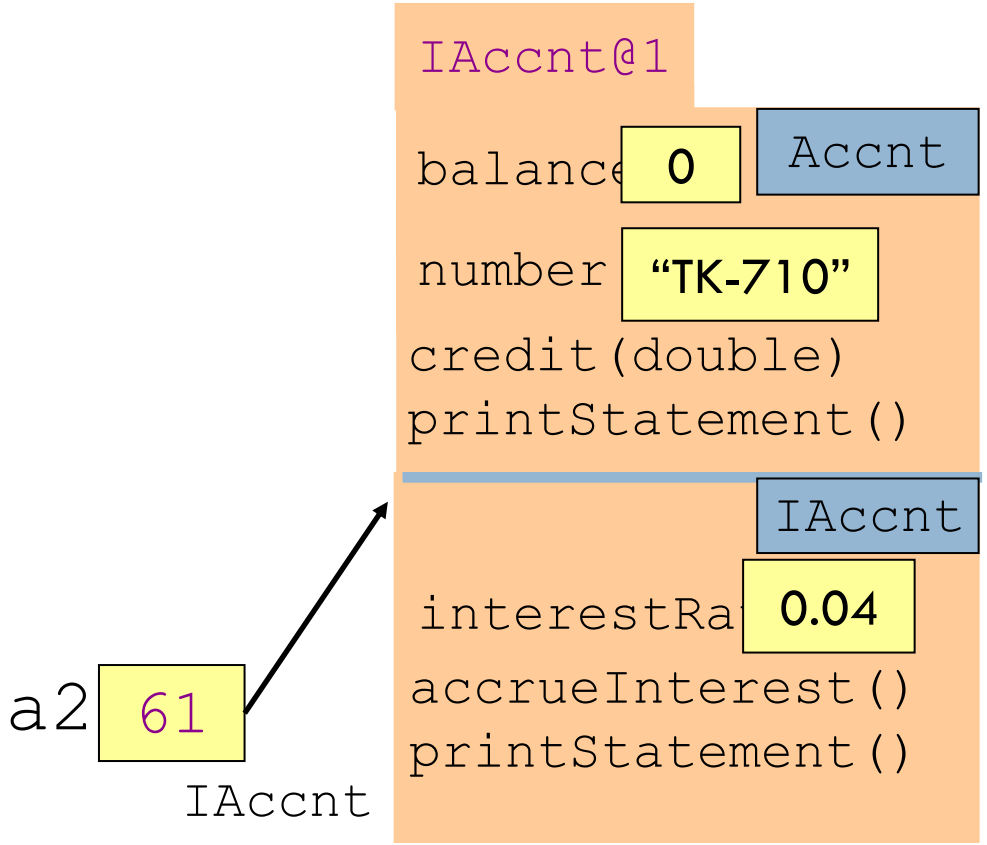
```
a2  instanceof  Animal
a2  instanceof  Integer
```

Casting among static types

Class cast expression: `(ClassName)obj`

Does not change obj, just gives a different perspective

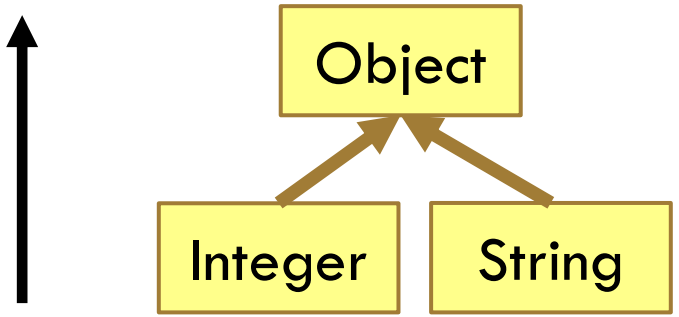
Expression	Type
`a2`	`IAccnt`
`(Accnt)a2`	`Accnt`
`(Object)a2`	`Object`
`(IAccnt)a2`	`Iaccnt`
`(Accnt)(Object)a2`	`Accnt`



Casting

Upcast: cast upward in class hierarchy

- e.g. `(Object)ob`
- No need to even write: automatically inserted as needed
- Will always succeed



- e.g. `(Integer)ob`
- Programmer must write themselves
- Might fail at run time

Downcast: cast downward in class hierarchy

Demo

JShell

Reason for downward casting

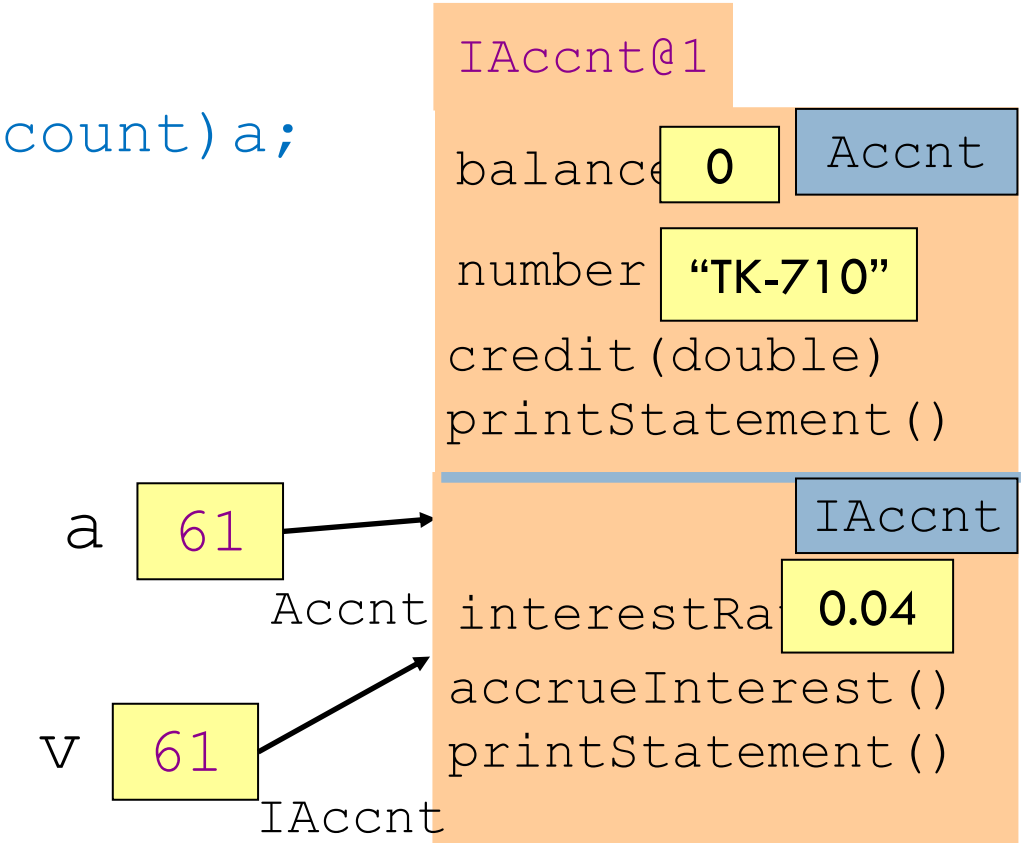
Suppose we want to call method `a.accrueInterest()`.

`a.accrueInterest()` is illegal (Compile-time Reference Rule)

```
InterestAccount v= (InterestAccount)a;
```

`v.accrueInterest()` is legal!

Downward casting to a known partition allows us to reference the fields and methods in that partition.



Checking a Downcast


```
if (ob  instanceof  C) {
     C c= (C) ob;
     // do something with c
} else {
     // do something else
}
```

Warning

Static type manipulation should be used sparingly.

It tends to be a sign of poorly-designed, non-OO code.

Prefer dynamic dispatch.

Image: public domain

Warning

Bad OOP:

```
if (ob instance of C1) {
    C1 c1=  (C1) ob;
    // do something with c1`
} else if (ob instance of C2) {
    C2 c2=  (C2) ob;
    // do something with c2
} …
```

Good OOP:

ob.do()

where do() is overridden in C1, C2…

Class Class<T>

For any class C, object C.class contains a description of C, with methods to look at C's fields, methods, and other properties.

C.class is an object of class Class!

The type of object C.class is Class<C> .

object String.class contains a description of class String.

object Account.class contains a description of class Account.

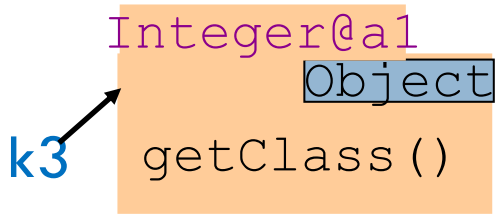
Class Class is part of Java's "reflection" mechanism. It allows a program to examine and modify its own structure at runtime.

One use of `getClass()`

We are generally interested in whether two objects were created using the same kind of new-expression:
Do the lowest partition of the objects have the same name.

k1.getClass() == k2.getClass() true

k1.getClass() == k3.getClass() false

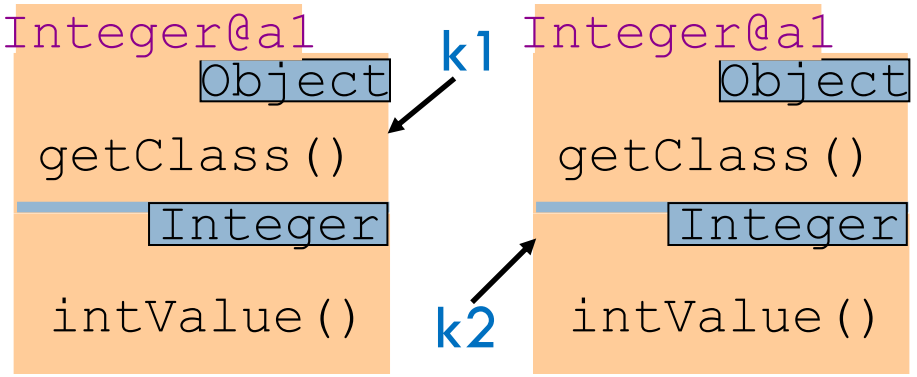


Translate

"k1 and k2 are of the same class"

into

k1.getClass() == k2.getClass()

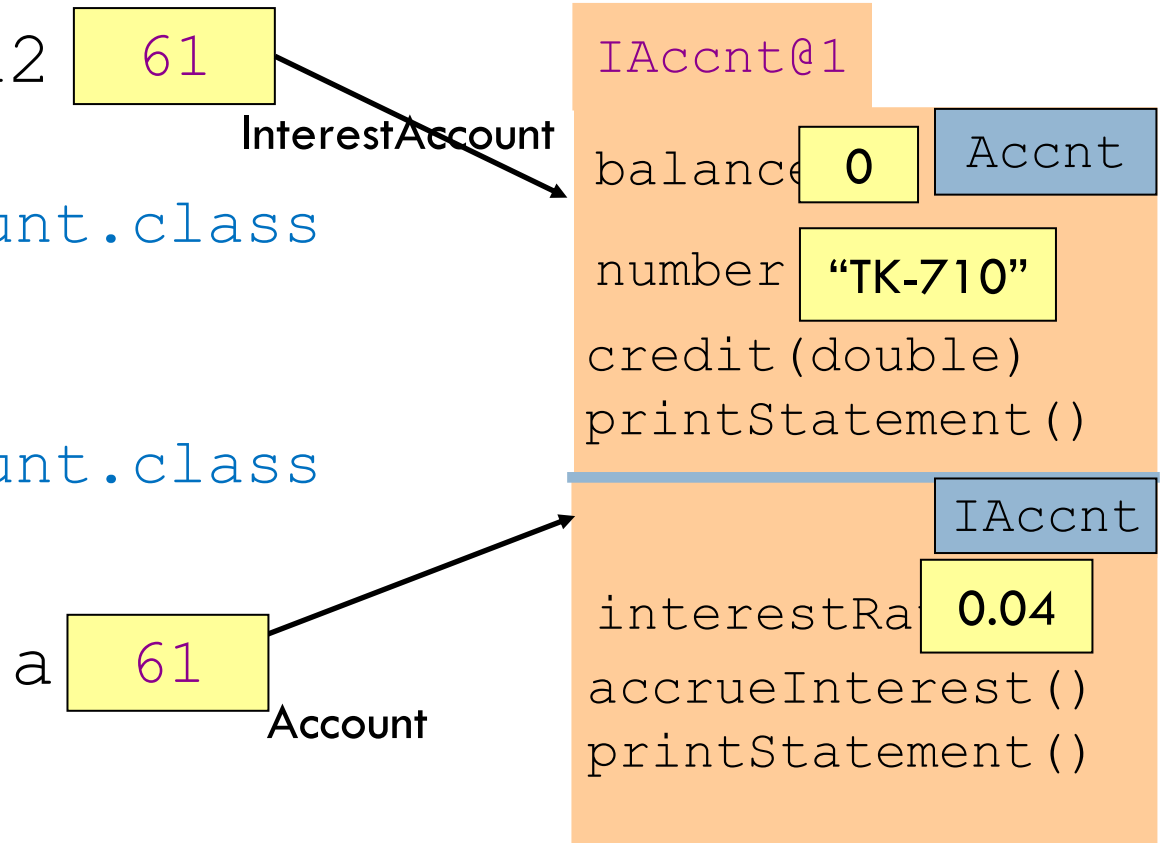


Method `getClass`


```
InterestAccount a2=
                new InterestAccount("TK-710", 0.04);
```

```
a2.getClass()  InterestAccount.class

Account a= a2;
a.getClass()   InterestAccount.class
```



LECTURE 7: POLYMORPHISM, PART II

PART 4: EQUALITY

CS 2110
Fall 2021

What Is Equality?

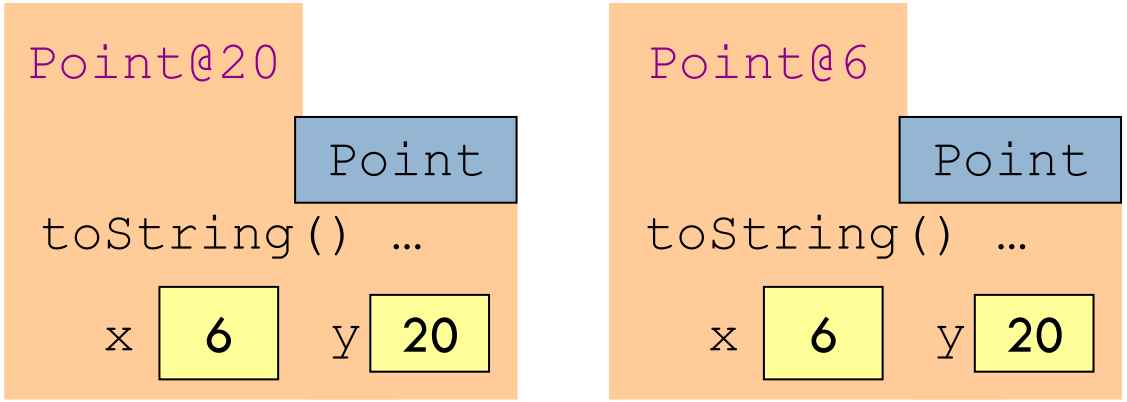
- Referential equality: two references point to the same object in memory
  - Java: `o1 == o2`

... and not this (except with primitives)

- Equivalence: two different objects are deemed "equivalent" according to the programmer
  - Java: `o1.equals(o2)`

Almost always want this!

These objects could be considered equivalent, or equal



Rules when overriding method equals()

- Not recommended if class is mutable
  - Confusing if equivalence changes over time
  - Not safe to use in Sets or as Map keys
- Must be reflexive, symmetric, and transitive
- Must return false when compared to null (may not throw NPE)
- Must also override method hashCode()

For many classes, default implementation (referential equality) is fine!

Overriding equals() is best suited for *immutable value classes*.

Equivalence

Specification: reflexive, symmetric, transitive

- Reflexive: `x.equals(x)`
- Symmetric: `x.equals(y)` iff `y.equals(x)`
- Transitive: if `x.equals(y)` and `y.equals(z)` then `x.equals(z)`

(Assuming `x` and `y` are not `null`)

Consequence: Objects of different classes should not be considered equal

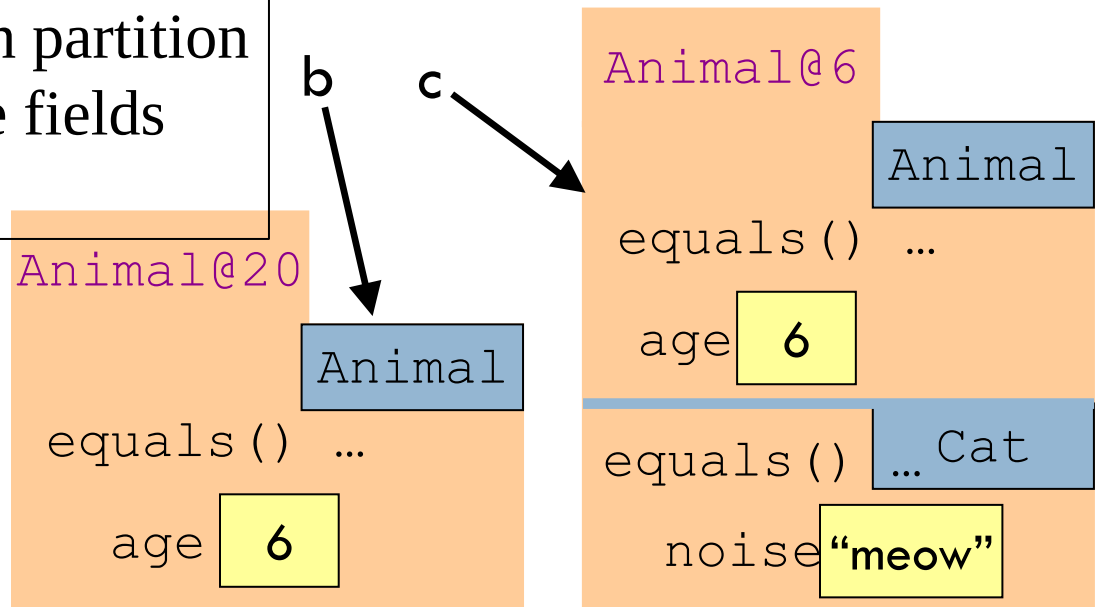
Overriding function equals

We will override function equals in classes Animal and Cat.

(Note: it wouldn't make sense to think of objects b and c as being equal).

c.equals(b)
Calls method equals in partition Cat; try to check noise fields and return false

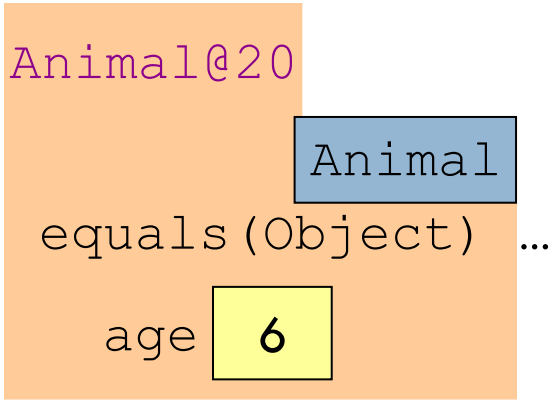
b.equals(c)
Probably check age fields and return true



Overriding equals in class Animal


```
/** = "this object and ob are of the same class"  and
          their ages are the same  */
@Override
public boolean equals(Object ob) {
        if (ob == null || getClass() != ob.getClass()) return false;
        Animal oba= (Animal)ob;
        return age == oba.age;     // return age == ((Animal)ob).age;
}
```

By compile-time reference rule,
`ob.age` is illegal



Overriding equals in class Cat

In class Cat

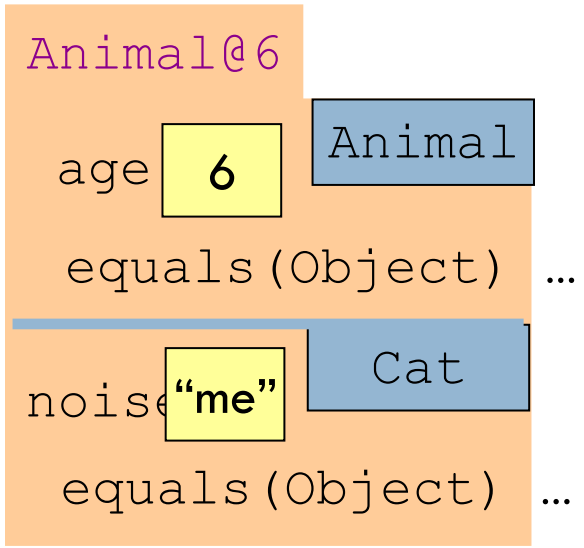
```
/** = "this object and ob are of the same class" and
            their ages and noises are the same */
public boolean equals(Object ob) {
   if (!super.equals(ob))  return false;
   Cat oba=  (Cat) ob;
   return noise.equals(oba.noise);
}
```

Quiz 1

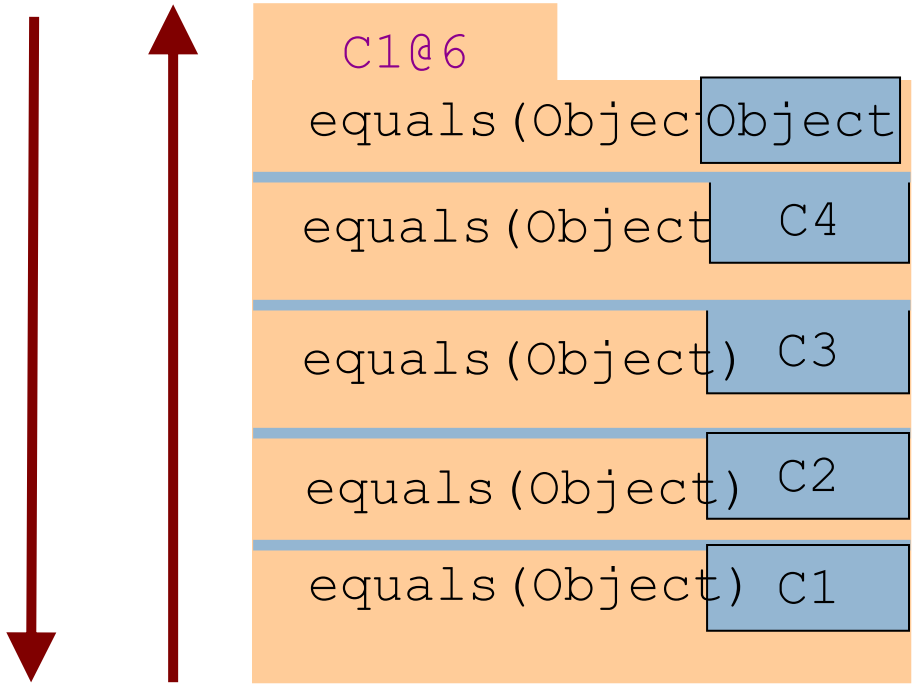
Quiz 2

In class Animal

```
/** = "this object and ob are of the same class"
          and their ages are the same */
public boolean equals(Object ob)
```



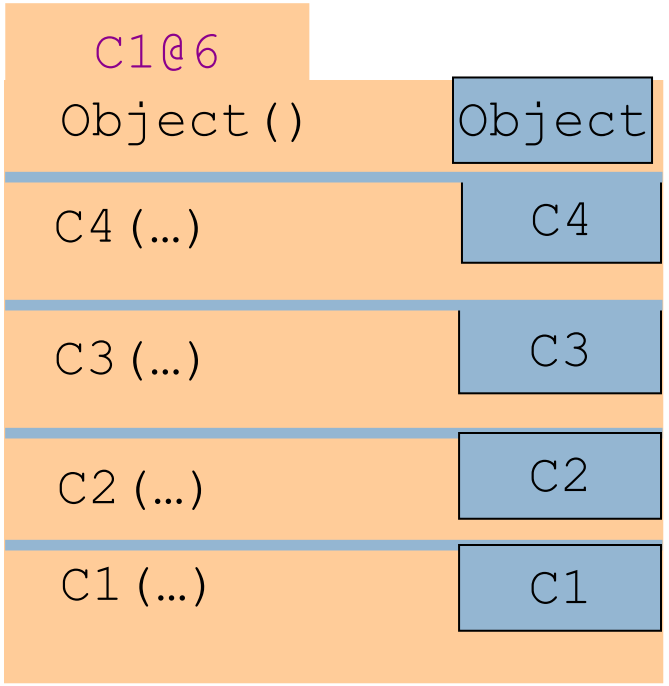
OOP: process superclass partitions first



OOP: process superclass partitions first

new C1(…)

Principle:
Fill in superclass fields first



Your Turn: Read in JavaHyperText

- Bottom-up rule / Overriding rule
- Compile-time reference rule
- `getClass`
- `instanceof`
- Cast: class cast, downcast, upcast, object casting rule
- `equals`