CSSE 220 Day 17

Inheritance recap
`Object`: the superest class of all
Inheritance and text in GUIs

Check out *MoreGUIness* from SVN

Questions?

Inheritance Review

- A quick recap of last session

Inheritance

- Sometimes a new class is **a special case** of the concept represented by another
- Can "borrow" from an existing class, changing just what we need
- The new class **inherits** from the existing one:
  - all methods
  - all instance fields

Notation and Terminology

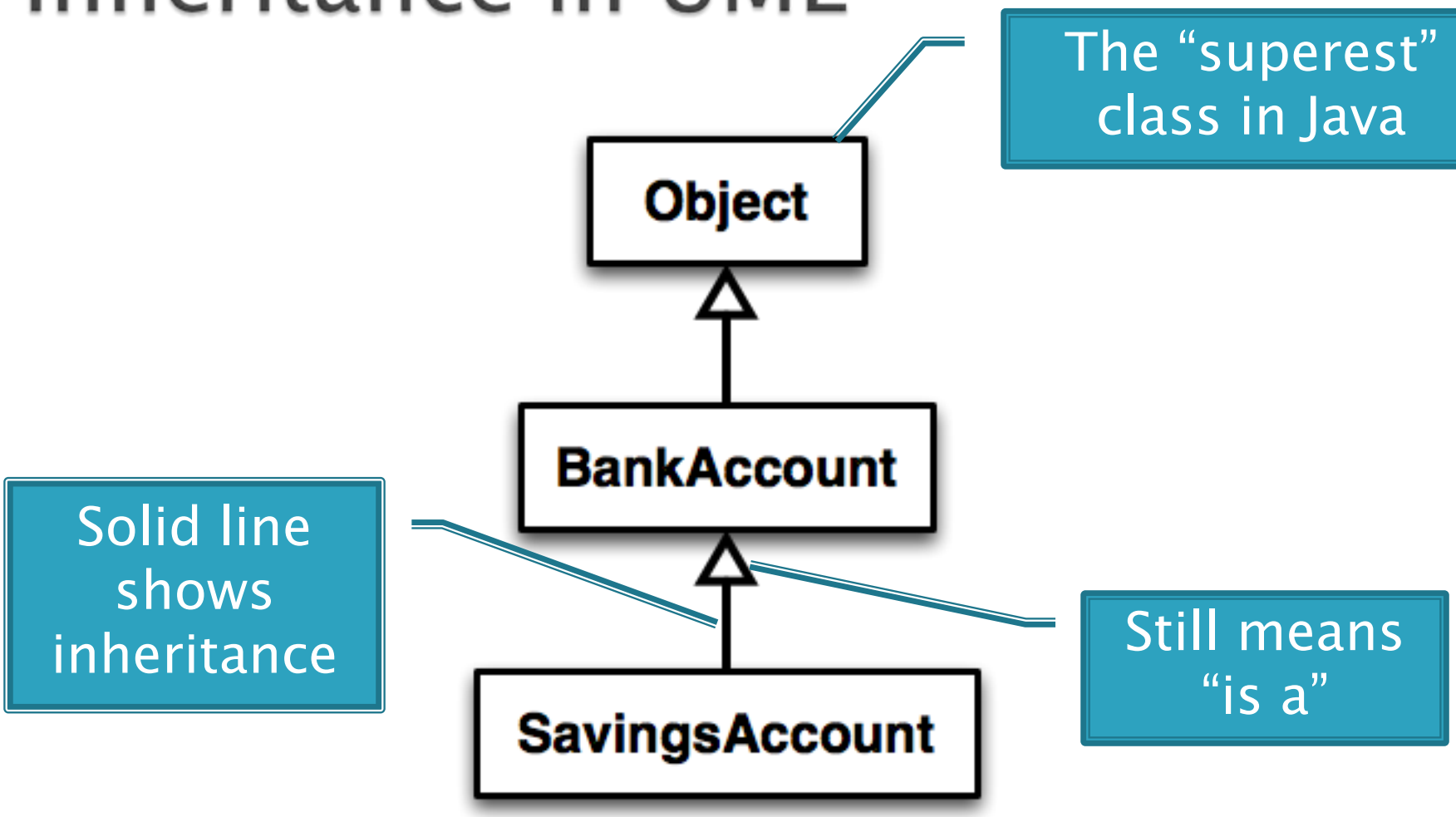
- ```
  class SavingsAccount extends BankAccount {
      // added fields
      // added methods
  }
  ```

- Say "SavingsAccount **is a** BankAccount"

- **Superclass**: BankAccount

- **Subclass**: SavingsAccount
  ```

Inheritance in UML



With Methods, Subclasses can:

- **Inherit** methods unchanged
- **Override** methods
  - Declare a new method with same signature to use instead of superclass method
- **Add** entirely new methods not in superclass

With Fields, Subclasses:

- **ALWAYS inherit** all fields unchanged

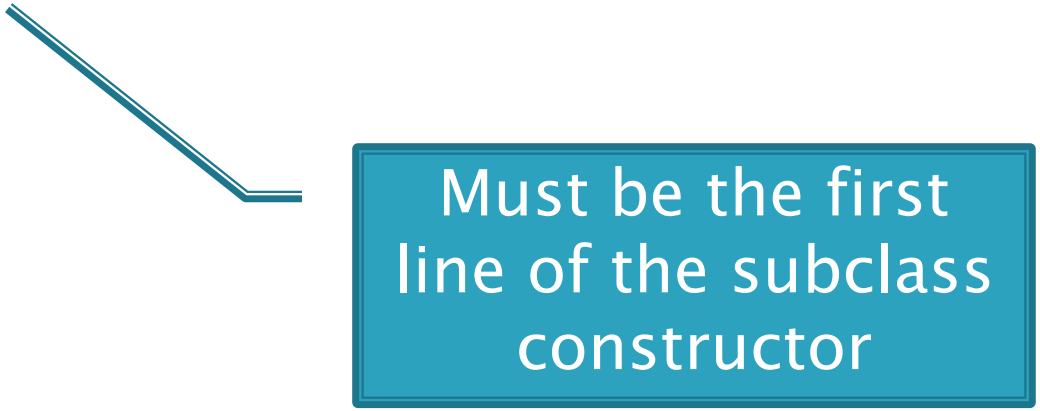
- **Can add** entirely new fields not in superclass

DANGER! Don't use the same name as a superclass field!

Super Calls

- Calling superclass **method**:
  - `super.methodName(args);`

- Calling superclass **constructor**:
  - `super(args);`



Access Modifiers

- `public`—any code can see it
- `private`—only the class itself can see it
- **default** (i.e., no modifier)—only code in the same `package` can see it
- `protected`—like default, but subclasses also have access

I, Object

- The superest class in Java

Object

- **Every** class in Java inherits from **`Object`**
  - Directly and **explicitly**:
    - `public class String extends Object {…}`
  - Directly and **implicitly**:
    - `class BankAccount {…}`
  - **Indirectly**:
    - `class SavingsAccount extends BankAccount {…}`

Object Provides Several Methods

- `String toString()`
- `boolean equals(Object otherObject)`
- `Class getClass()`
- `Object clone()`
- …

Often overridden

Often useful

Often dangerous!

Overriding `toString()`

- Return a concise, human-readable summary of the object state

- Very useful because it's called automatically:
  - During string concatenation
  - For printing
  - In the debugger

- `getClass().getName()` comes in handy here…

Overriding `equals(Object o)`

- Should return true when comparing two objects of same type with same “meaning”
- How?
  - Must check types—use `instanceof`
  - Must compare state—use **cast**
- Example…

The Reason for clone()

- Avoiding representation exposure:
  - returning an object that lets other code muck with our object's state

```
public class Customer {
    private String name;
    private BankAccount acct;
    …
    public String getName() {
        return this.name;  // ← OK!
    }

    public BankAccount getAccount() {
        return this.acct; // ← Rep. exposure!
    }
```

Book says (controversiallly) to use
return (BankAccount) this.acct.clone();"

Q3,4

The Trouble with `clone()`

- **`clone()`** is supposed to make a *deep copy*
  1. Copy the object
  2. Copy any mutable objects it points to
- **`Object`**'s **`clone()`** handles 1 **but not 2**
- *Effective Java* includes a seven page description on overriding **`clone()`**:
  - "[You] are probably better off providing some alternative means of object copying or simply not providing the capability."

Effective Java, by Joshua Block

Alternatives to clone()

- Copy constructor in Customer:
  - `public Customer(Customer toBeCopied) {…}`
- Copy factory in BankAccount:
  - `public abstract BankAccount getCopy();`
- Fixed Example:
  - ```
    public BankAccount getAccount() {
        return this.acct.getCopy();
    }
    ```
    ```

Better Frames Through Inheritance

- main() got complicated in LinearLightsOut, better to create a subclass…