CSSE 220

Inheritance

Import *Inheritance* from the repo

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

Q1

Look at Code

- BankAccount
- SavingsAccount
- Create Driver.java to illustrate:
  - how objects declared from SavingsAccount inherit from BankAccount

Examples

- `class SavingsAccount extends BankAccount`
  - adds interest earning, keeps other traits
- `class Employee extends Person`
  - adds pay information and methods, keeps other traits
- `class Manager extends Employee`
  - adds information about employees managed, changes the pay mechanism, keeps other traits

Notation and Terminology

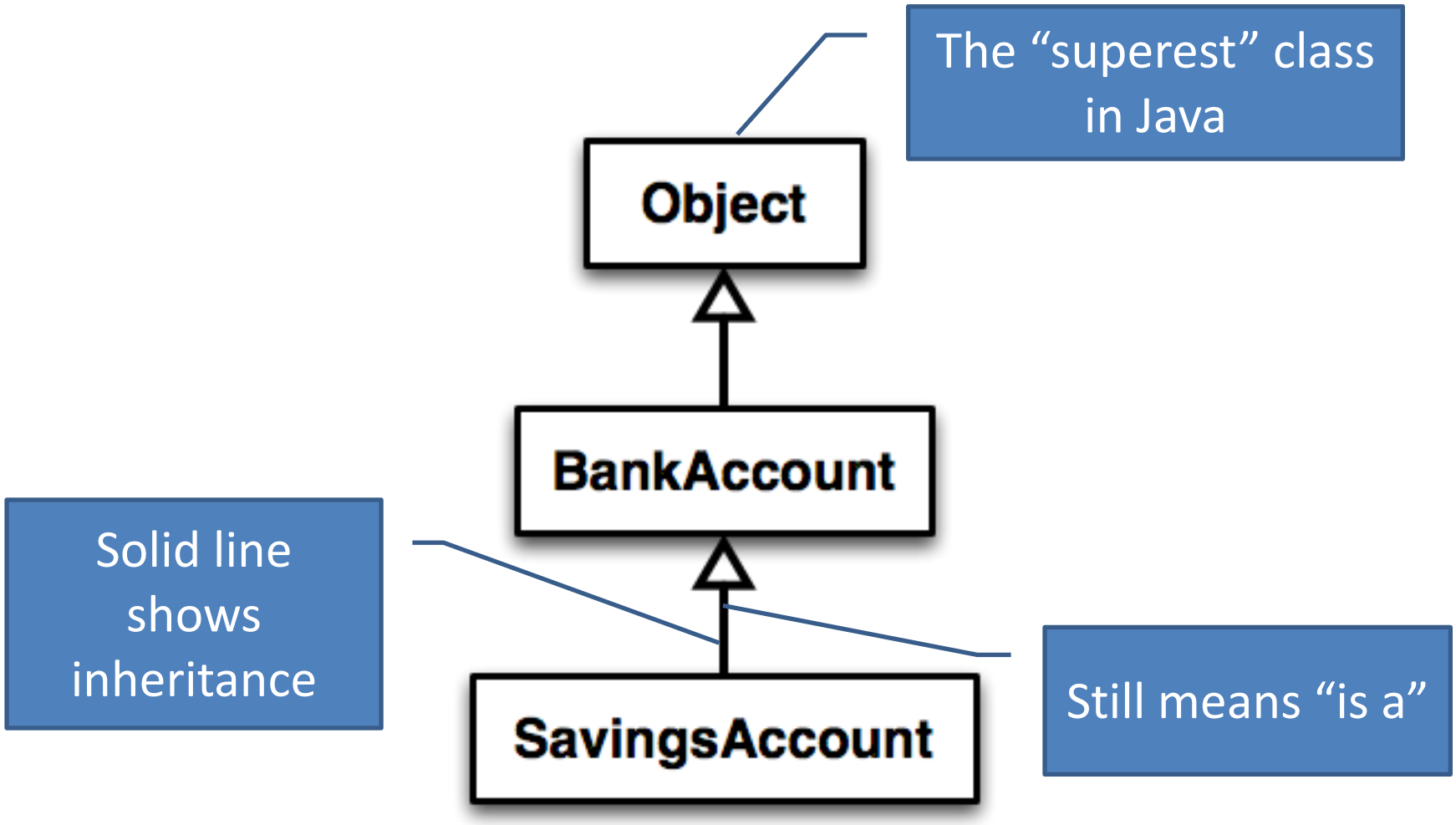
- ```
  class SavingsAccount extends BankAccount {
      // added fields
      // added methods
  }
  ```

Java keyword

- Say "SavingsAccount **is a** BankAccount"
- **Superclass**: BankAccount
- **Subclass**: SavingsAccount

English we use to talk about inheritance
  ```

Inheritance in UML



Q2

Interfaces vs. Inheritance

- **`class ClickHandler implements MouseListener`**
  - ClickHandler **promises** to implement all the methods of MouseListener

For **client** code reuse

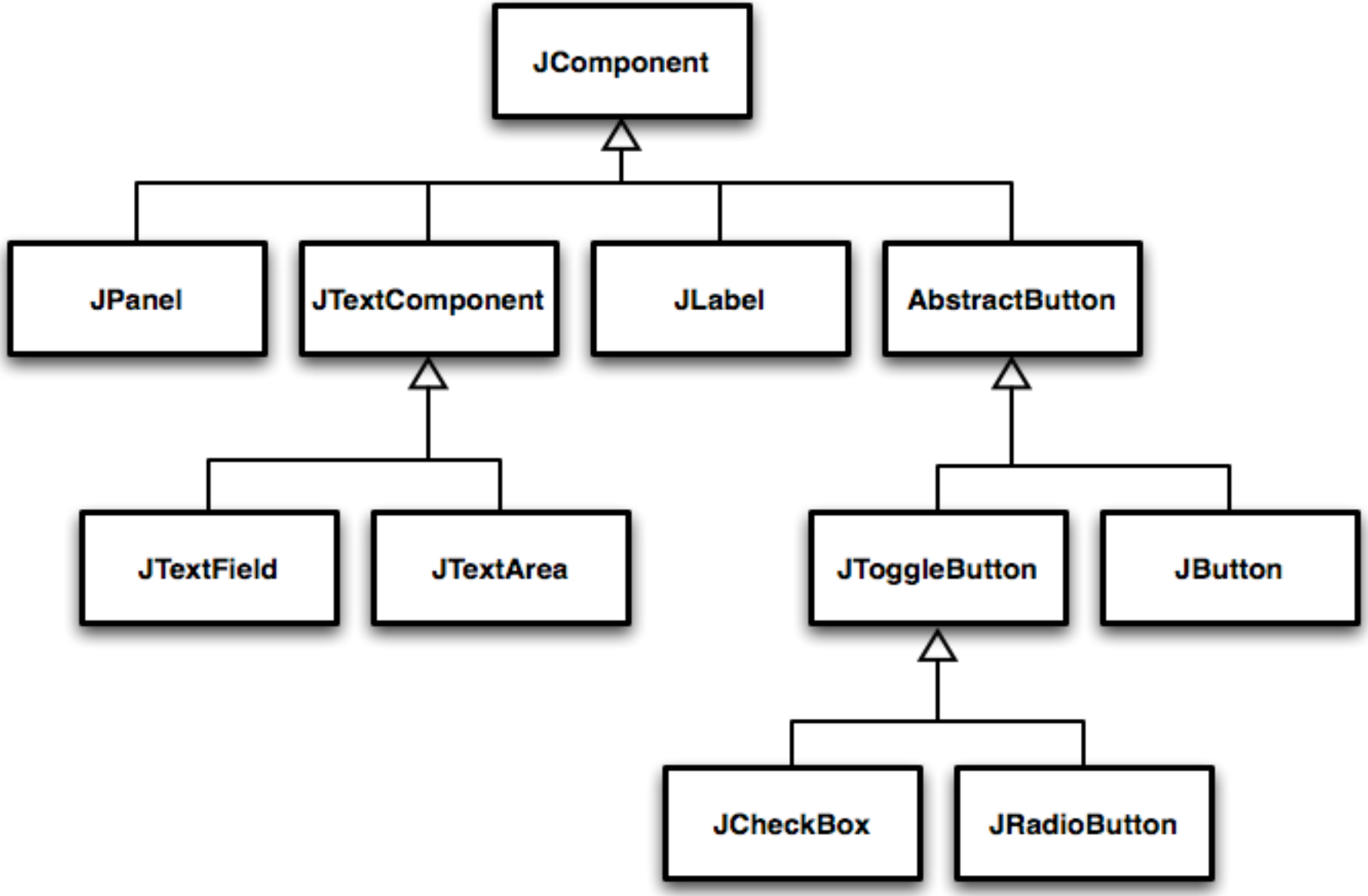
- **`class CheckingAccount extends BankAccount`**
  - CheckingAccount **inherits** (or overrides) all the methods of BankAccount

For **implementation** code reuse

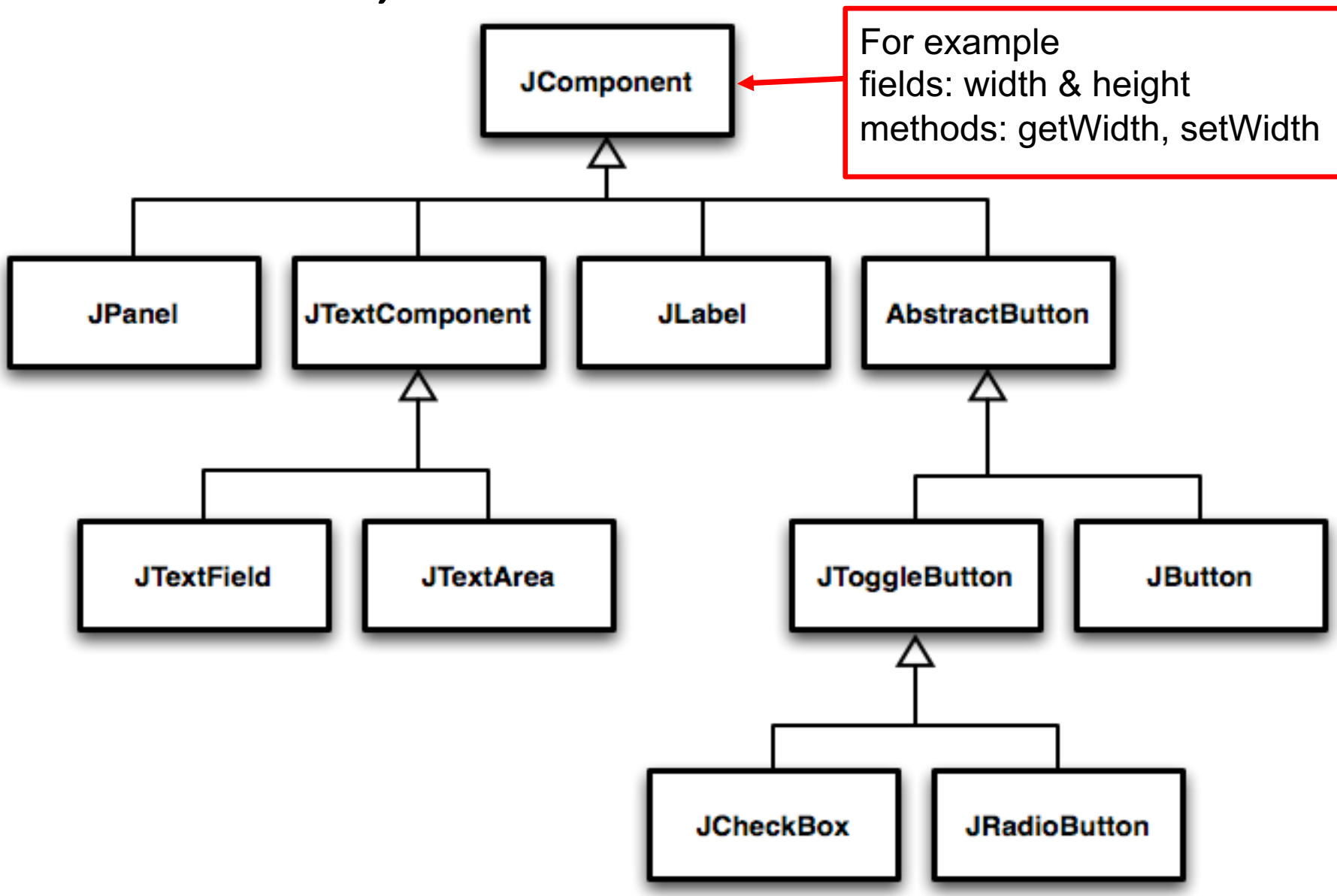
Interface in Bet Example

- By creating and using an Interface in the betting example, we removed duplication in the *handleRoll* method of the client
- We were able to make one ArrayList<Bet> in the client, and then have *handleRoll* walk that ArrayList object and process each of the different bets
- The methods of the 3 different bet classes were still independent and there might have been a way to reduce duplication across those 3 classes by using inheritance

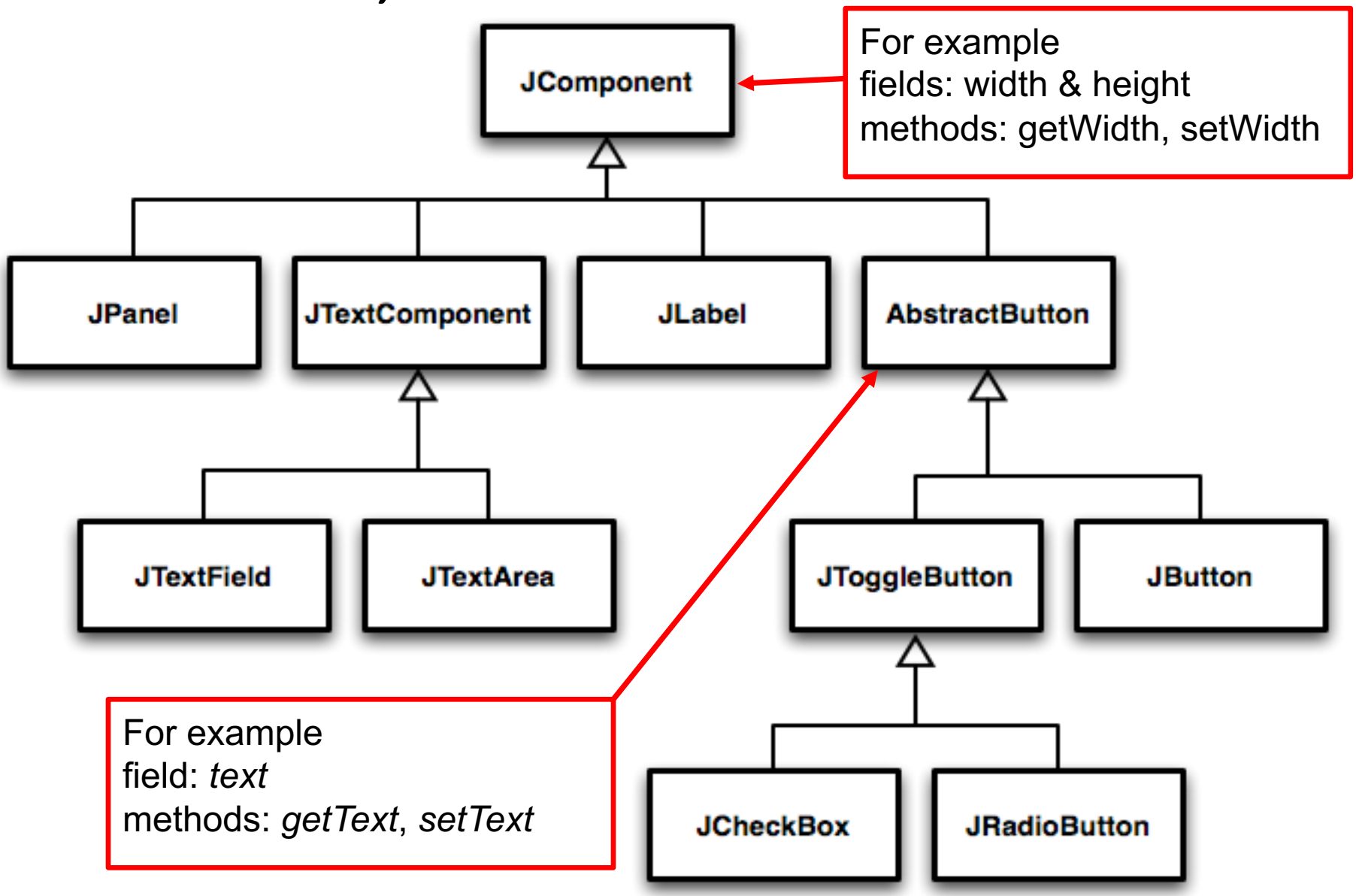
In Java, Inheritance Run Amok?



In Java, Inheritance Run Amok?



In Java, Inheritance Run Amok?



With Methods, Subclasses can:

1. **Inherit** methods unchanged
   - Simply by using 'extends' w/o doing anything else
   - Client of subclass gets to call all methods inherited from superclass

2. **Override** methods
   - Declare a new method with same signature to use **instead of** superclass method
   - Client of subclass only gets to call overridden method

3. **Add** entirely new methods not already in superclass
   - Client can call these new subclass methods

Q3

With Fields, Subclasses:

- **ALWAYS inherit** all fields unchanged
  - But only have access to: protected, public, and package level fields
  - No access to superclass's private fields

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

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

Q4

Super Calls

- Calling superclass **method**:
  - `super.methodName(args);`
  - **Joe: Demonstrate in Eclipse w/ BankAccount**

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

Must be the first line of the subclass constructor

Q5

Let’s Code CheckingAccount

- A special type of BankAccount
- Has 3 free transactions each month
  - Withdraw
  - Deposit
- Every additional transaction (beyond) costs $1.50
  - 4 cost $1.50
  - 5 cost $3.00
- At end of each month fees are deducted (all together)
  - Transaction count is reset at this time

Joe Todo

- Switch to Eclipse
- Create subclass CheckingAccount from making BankAccount a superclass
- Add fields to CheckingAccounts that keep track of # of transactions
- Create constructor to init: initial balance and # of transactions
- Override *deposit* method (from superclass)
  - Discuss @Override
  - Predefined Annotation Types in Java
- Override *withdraw* method
- Implement *deductFees* method

Polymorphism and Subclasses

- A subclass instance **is a** superclass instance
  - Polymorphism still works!
  - `// Original declaration:`
  - `BankAccount ba = new BankAccount();`

  - `// Change declaration, use CheckingAccount`
  - `BankAccount ba = new CheckingAccount();`
    `ba.deposit(100);`
  - Client used BankAccount in original *new* (above), client still works after change because CheckingAccout has all of BankAccount methods, e.g., *deposit*

Q6

Polymorphism and Subclasses

- A subclass instance **is a** superclass instance
  - Polymorphism still works!
  - ```
    BankAccount ba = new CheckingAccount();
    ba.deposit(100);
    ```

- But not the other way around!
  - ```
    CheckingAccount ca = new BankAccount();
    ca.deductFees();
    ```

- Why not?

BOOM!

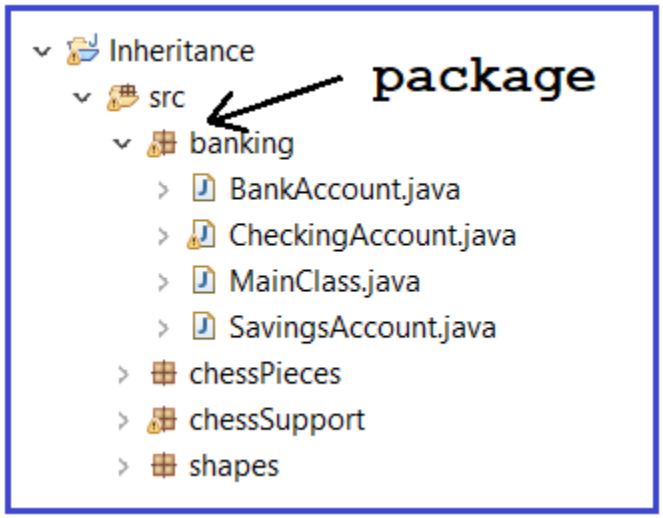
Q6

Another Example

- Can use:
  - ```
    public void transfer(double amount, BankAccount o){
        this.withdraw(amount);
        o.deposit(amount);
    }
    ```
    in BankAccount
- To transfer between different accounts:
  - `SavingsAccount sa = …;`
  - `CheckingAccount ca = …;`
  - `sa.transfer(100, ca);`
    ```

Access Modifiers

- **`public`**— any client can access it
- **`protected`** — package and subclasses can access it, but clients that *new* objects cannot access
- **`private`** — only the class itself can see it (no client can access it, nothing else in the package can access it)
- **`default`**—anything in the package can access it



Access Modifiers

- `public` — any client can access it
- `protected` — package and subclasses can access it, but clients that *new* objects cannot access
- `private` — only the class itself can see it (no client can access it, nothing else in the package can access it)
- `default` — anything in the package can access it

- Notes:
  - **default** (i.e., no modifier)—only code in the same `package` can see it
    - good choice for classes
  - `protected`—like default, but subclasses also have access
    - sometimes useful for helper methods

Bad for fields!

Q7

Live coding

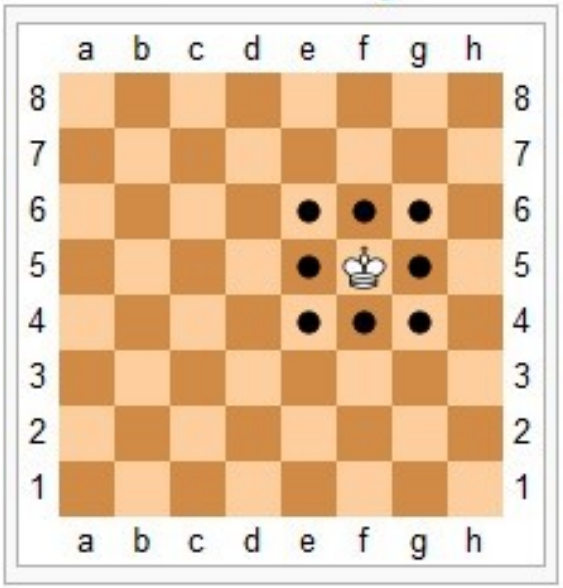
- Joe: examine at chessPieces/chessSupport
  - Let's Look at King and ChessPiece
  - StandardBoardProvider (uncomment King lines)

- In-class work:
  - Create Rook class, start by duplicating King class and renaming and modifying copied methods

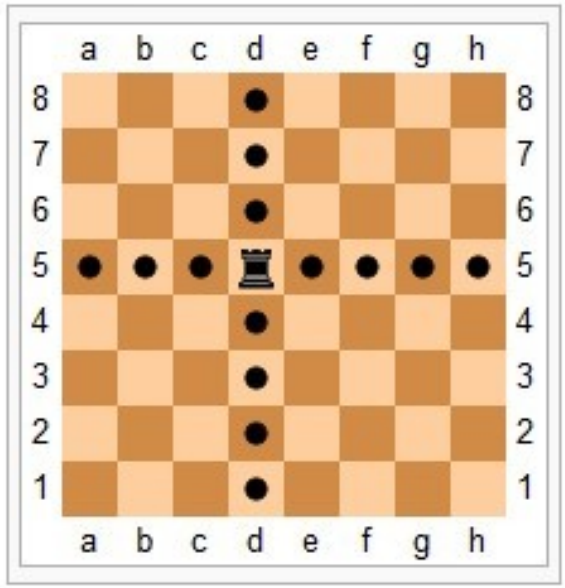
Live coding

- In-class work, continued:
  - Change King and Rook to use super class ChessPiece
  - Implement Bishop and Queen next, and then any piece you want after that

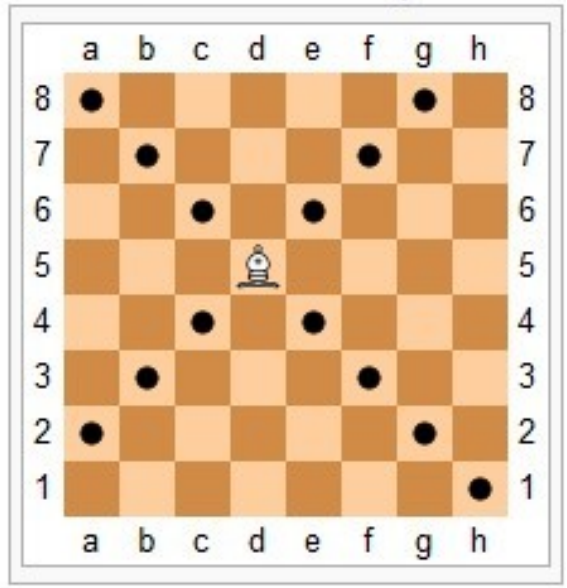
Moves of a king



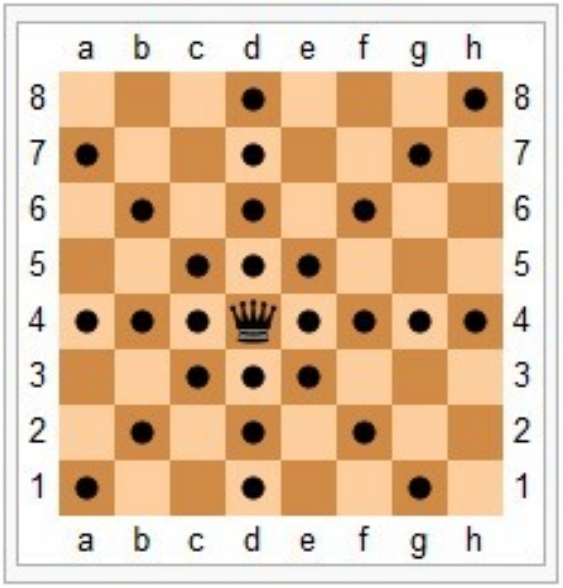
Moves of a rook



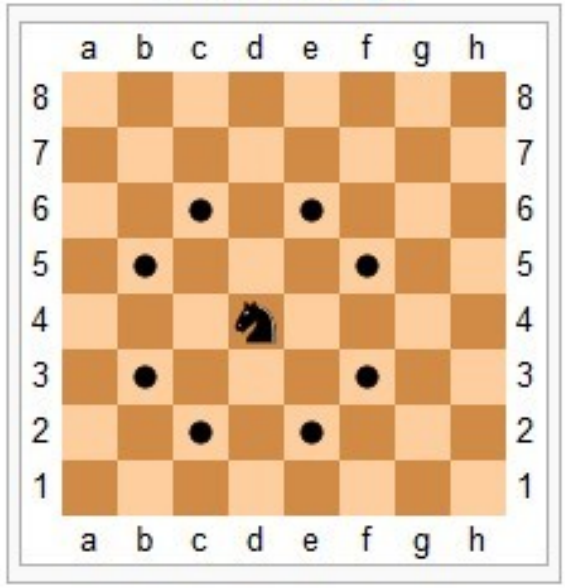
Moves of a bishop



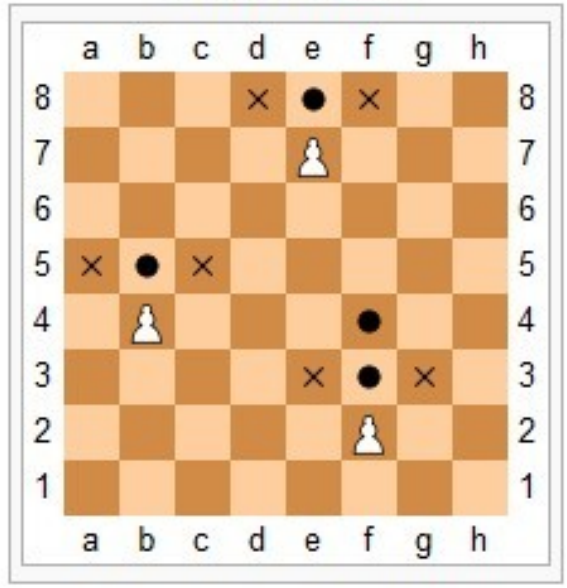
Moves of a queen



Moves of a knight



Moves of a pawn



Abstract Classes

Also look at the code in the shapes package, especially ShapesDemo (during or after class)

- Hybrid of superclasses and interfaces
  - Like regular superclasses:
    - Provide implementation of some methods
  - Like interfaces
    - Just provide signatures and docs of other methods
    - Cannot be instantiated
- Example:
  - ```
    public abstract class BankAccount {
        /** documentation here */
        public abstract void deductFees();
        …
    }
    ```

Elided methods as before
    ```

Chess

Ball World

It's a solo project, but feel free to talk with others as you do it.

And to ask instructor/assistants for help

WORK TIME

Q8-Q9