CSE 143 Java

Inheritance Example

Example Domain: Bank Accounts

- We want to model different kinds of bank accounts
  - A plain bank account: standard account information (name, account #, balance)
  - a savings account: like a generic bank account, but it also earns interest when balance is above some minimum
  - a checking account: like a generic bank account, but it also is charged a fee if the balance dips below some minimum amount
- How should we program this?

Option 1: Three Separate Classes

- BankAccount class
  - The code we already saw
- SavingsAccount class
  - Copy the BankAccount code, and add a creditInterest method
- CheckingAccount class
  - Copy the BankAccount code, and add a deductFees method

- This is what we'd have to do in a non-OO language
- But is a poor solution in an OO language
  - Why?

Option 2: Introduce a Common Interface

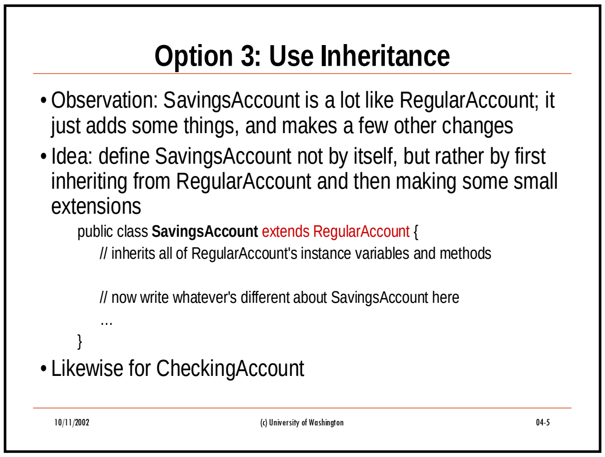
- BankAccount interface defines the common operations of all accounts

```
public interface BankAccount {
    public double getBalance( );
    public boolean deposit(double amount);
    public boolean withdraw(double amount);
}
```

- Each kind of account implements this interface

```
public class RegularAccount implements BankAccount { ... }
public class SavingsAccount implements BankAccount { ... }
public class CheckingAccount implements BankAccount { ... }
```

- What are the strengths of this approach? weaknesses?

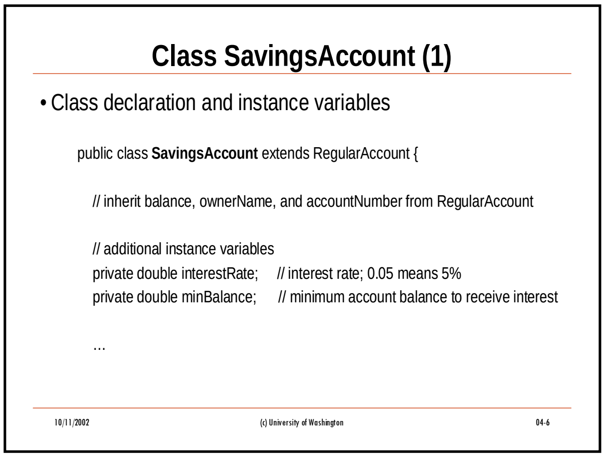


Option 3: Use Inheritance

- Observation: SavingsAccount is a lot like RegularAccount; it just adds some things, and makes a few other changes
- Idea: define SavingsAccount not by itself, but rather by first inheriting from RegularAccount and then making some small extensions

```
public class SavingsAccount extends RegularAccount {
    // inherits all of RegularAccount's instance variables and methods

    // now write whatever's different about SavingsAccount here
    ...
}
```

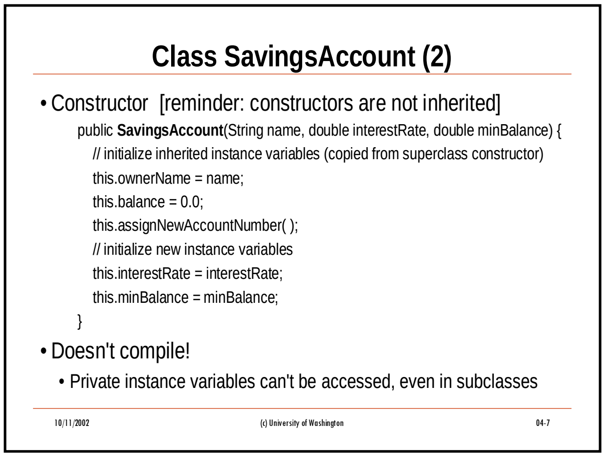
- Likewise for CheckingAccount



Class SavingsAccount (1)

- Class declaration and instance variables

```
public class SavingsAccount extends RegularAccount {

    // inherit balance, ownerName, and accountNumber from RegularAccount

    // additional instance variables
    private double interestRate;     // interest rate; 0.05 means 5%
    private double minBalance;       // minimum account balance to receive interest

    ...
```

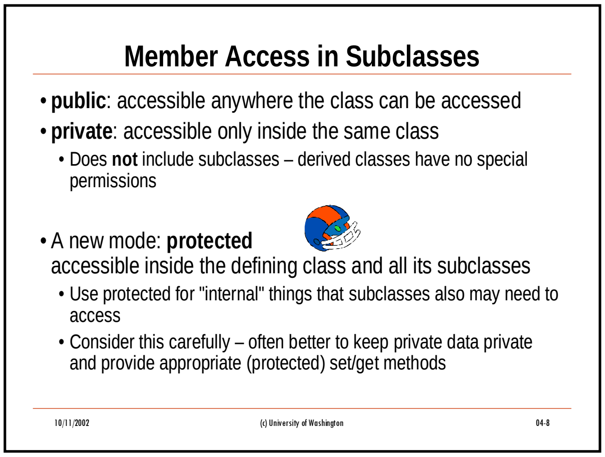


Class SavingsAccount (2)

- Constructor [reminder: constructors are not inherited]

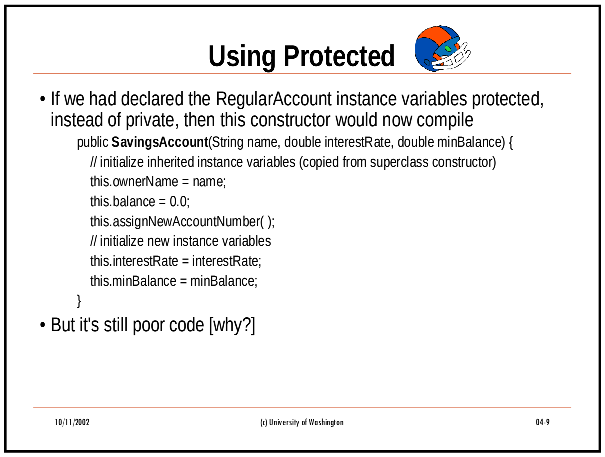
```
public SavingsAccount(String name, double interestRate, double minBalance) {
    // initialize inherited instance variables (copied from superclass constructor)
    this.ownerName = name;
    this.balance = 0.0;
    this.assignNewAccountNumber( );
    // initialize new instance variables
    this.interestRate = interestRate;
    this.minBalance = minBalance;
}
```

- Doesn't compile!
  - Private instance variables can't be accessed, even in subclasses



Member Access in Subclasses

- **public**: accessible anywhere the class can be accessed
- **private**: accessible only inside the same class
  - Does **not** include subclasses – derived classes have no special permissions
- A new mode: **protected**
  accessible inside the defining class and all its subclasses
  - Use protected for "internal" things that subclasses also may need to access
  - Consider this carefully – often better to keep private data private and provide appropriate (protected) set/get methods

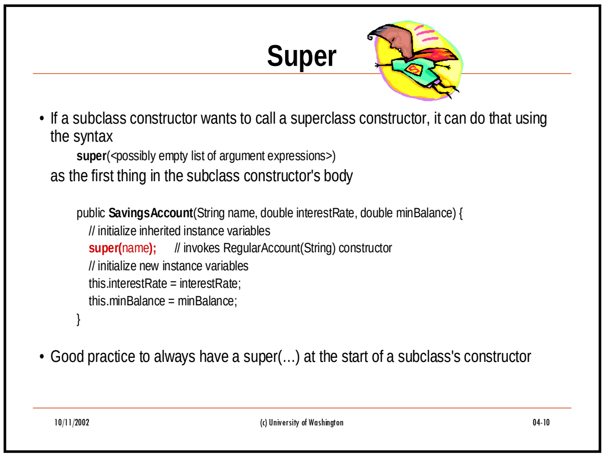


Using Protected

- If we had declared the RegularAccount instance variables protected, instead of private, then this constructor would now compile

```
public SavingsAccount(String name, double interestRate, double minBalance) {
    // initialize inherited instance variables (copied from superclass constructor)
    this.ownerName = name;
    this.balance = 0.0;
    this.assignNewAccountNumber( );
    // initialize new instance variables
    this.interestRate = interestRate;
    this.minBalance = minBalance;
}
```

- But it's still poor code [why?]



Super

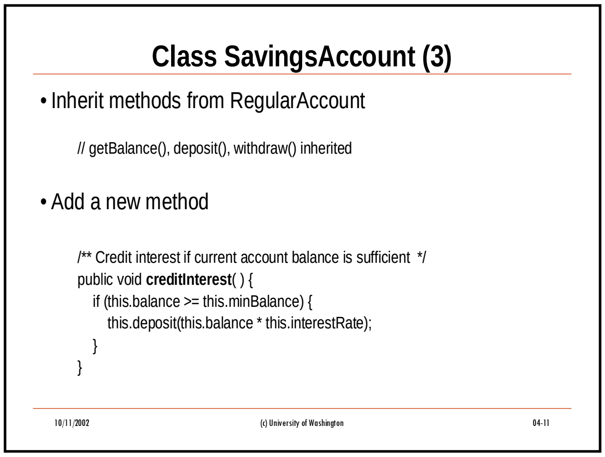
- If a subclass constructor wants to call a superclass constructor, it can do that using the syntax

  super(<possibly empty list of argument expressions>)

  as the first thing in the subclass constructor's body

```
public SavingsAccount(String name, double interestRate, double minBalance) {
    // initialize inherited instance variables
    super(name);      // invokes RegularAccount(String) constructor
    // initialize new instance variables
    this.interestRate = interestRate;
    this.minBalance = minBalance;
}
```

- Good practice to always have a super(…) at the start of a subclass's constructor



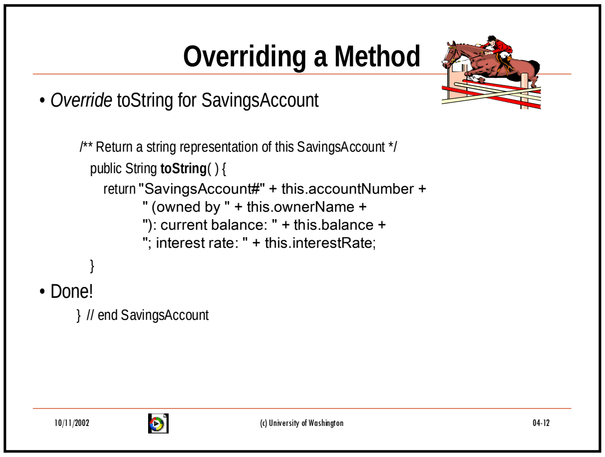
Class SavingsAccount (3)

- Inherit methods from RegularAccount

```
// getBalance(), deposit(), withdraw() inherited
```

- Add a new method

```
/** Credit interest if current account balance is sufficient  */
public void creditInterest( ) {
    if (this.balance >= this.minBalance) {
        this.deposit(this.balance * this.interestRate);
    }
}
```



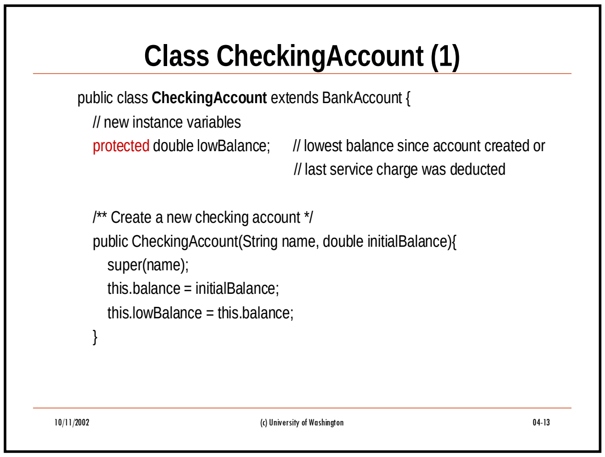
Overriding a Method

- *Override* toString for SavingsAccount

```
/** Return a string representation of this SavingsAccount */
  public String toString( ) {
     return "SavingsAccount#" + this.accountNumber +
              " (owned by " + this.ownerName +
              "): current balance: " + this.balance +
              "; interest rate: " + this.interestRate;
  }
```

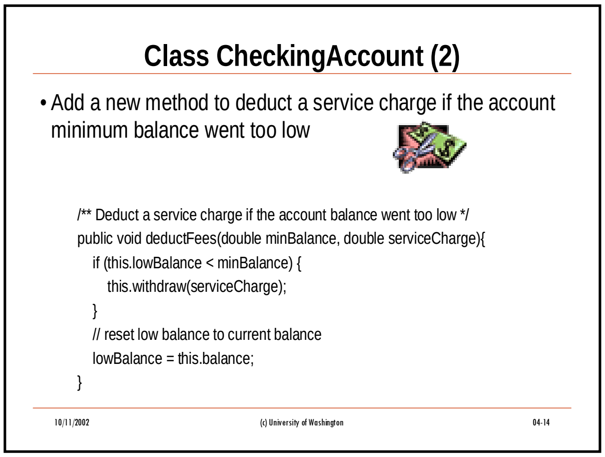
- Done!

```
} // end SavingsAccount
```



Class CheckingAccount (1)

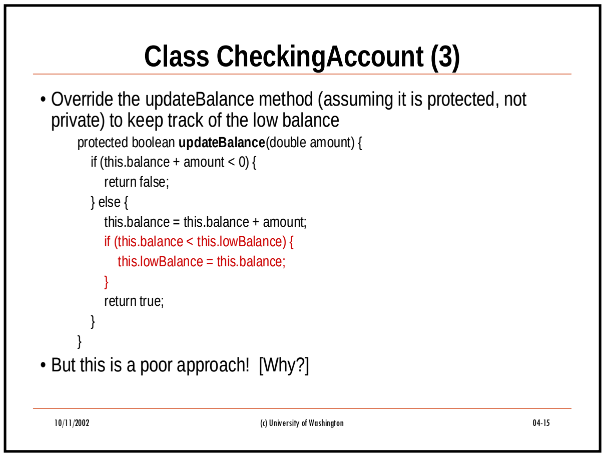
```
public class CheckingAccount extends BankAccount {
    // new instance variables
    protected double lowBalance;      // lowest balance since account created or
                                      // last service charge was deducted

    /** Create a new checking account */
    public CheckingAccount(String name, double initialBalance){
        super(name);
        this.balance = initialBalance;
        this.lowBalance = this.balance;
    }
```



Class CheckingAccount (2)

- Add a new method to deduct a service charge if the account minimum balance went too low

```
/** Deduct a service charge if the account balance went too low */
public void deductFees(double minBalance, double serviceCharge){
    if (this.lowBalance < minBalance) {
        this.withdraw(serviceCharge);
    }
    // reset low balance to current balance
    lowBalance = this.balance;
}
```

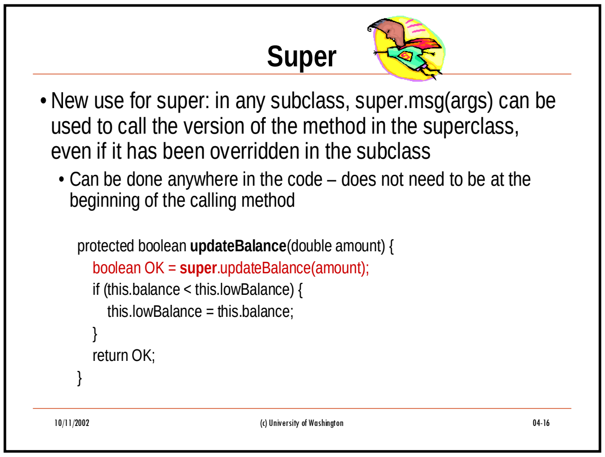


Class CheckingAccount (3)

- Override the updateBalance method (assuming it is protected, not private) to keep track of the low balance

```
protected boolean updateBalance(double amount) {
    if (this.balance + amount < 0) {
        return false;
    } else {
        this.balance = this.balance + amount;
        if (this.balance < this.lowBalance) {
            this.lowBalance = this.balance;
        }
        return true;
    }
}
```

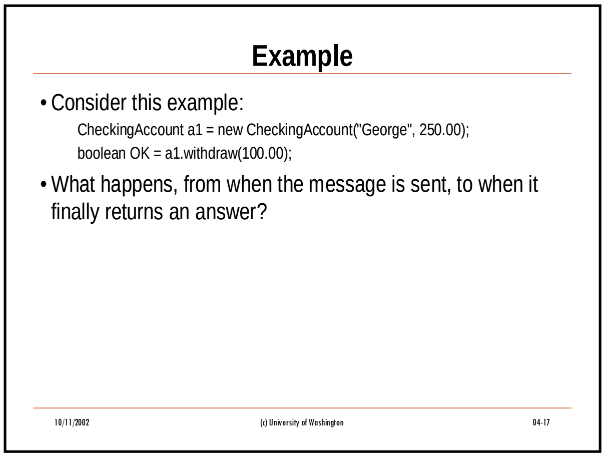
- But this is a poor approach! [Why?]



Super

- New use for super: in any subclass, super.msg(args) can be used to call the version of the method in the superclass, even if it has been overridden in the subclass
  - Can be done anywhere in the code – does not need to be at the beginning of the calling method

```
protected boolean updateBalance(double amount) {
    boolean OK = super.updateBalance(amount);
    if (this.balance < this.lowBalance) {
        this.lowBalance = this.balance;
    }
    return OK;
}
```

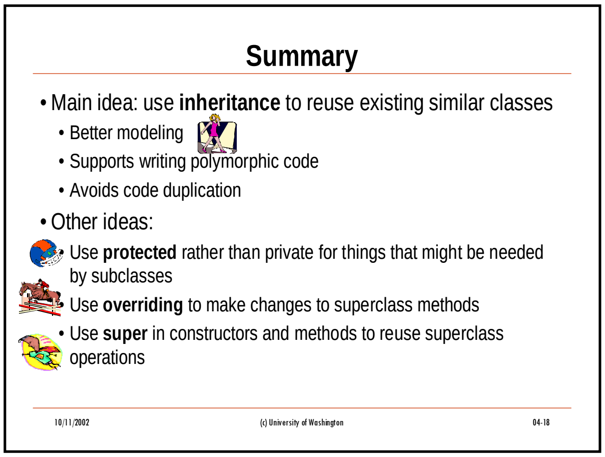


Example

- Consider this example:

```
CheckingAccount a1 = new CheckingAccount("George", 250.00);
boolean OK = a1.withdraw(100.00);
```

- What happens, from when the message is sent, to when it finally returns an answer?



Summary

- Main idea: use **inheritance** to reuse existing similar classes
  - Better modeling
  - Supports writing polymorphic code
  - Avoids code duplication
- Other ideas:
  - Use **protected** rather than private for things that might be needed by subclasses
  - Use **overriding** to make changes to superclass methods
  - Use **super** in constructors and methods to reuse superclass operations