

Class Design

Handling Errors

You will be expected to:

- incorporate exception handling into the design of a method's contract
 - trace code that makes use of exception handling
 - write code to throw, catch or propagate an exception
 - write code that uses a finally block
 - write code to define a new exception class
 - compare and contrast checked and unchecked exceptions
- 06/24/10 • understand the consequence of using checked vs. unchecked exceptions

Reading:

2nd Ed: Chapter 15

3rd, 4th Ed: Chapter 11

Exercises

2nd Ed: P15.5, P15.6

(Hint: look at documentation for Scanner class!)

3rd Ed: P11.9, P11.11

4th Ed: P11.12, P11.14

1

Office hours posted

- Office hours / Demco Learning Centre hours posted
- See course webpage

06/24/10

2

Course Info

- Students with the following IDs please see me after class or at the break

06/24/10

3

Course Info

- Assignment 1 due Wednesday
- Midterm exam: Friday, July 9th
- Final exam: Friday, July 30th
- Both exams are at the normal time and location for the class (9:00-11:30, DMP 110)

06/24/10

4

Review

- Contracts
 - Preconditions, Postconditions, Invariants
- Assertions
- Exceptions

06/24/10

5

Contracts

- Check that your code satisfies the contracts
 - If the client has satisfied the precondition for your method, will your method satisfy the postcondition?
 - Do your methods satisfy the defined class invariants?

06/24/10

6

Assertions

- Assertions check whether some condition is true (e.g. a precondition)
- Very useful for debugging
- Can be enabled and disabled
- More powerful and flexible than the classic “print statements” form of debugging

06/24/10

7

Exceptions – Why do we need them?

- Remember the `Account` class? We added the following precondition to the `deposit` method:

```
amount >= 0
```

- What if the client fails to check the precondition? The customers won't be happy to find out that sloppy programming has resulted in losing money because of a simple mistake!

06/24/10

8

Exceptions – Why we need them?

- Rather than using a precondition, we can have the method:
 - return a special value (e.g., true/false) to indicate whether or not the operation was successful

problem:

- print an error message

problem:

- terminate the program

problem:

06/24/10

9

Exceptions – Why we need them?

- Rather than using a precondition or one of the other methods suggested on the previous slide, we can have the method **throw an exception** if the amount is negative.

Benefits:

- We can force the client to acknowledge the problem.
- We allow the client to decide how to handle the problem.

06/24/10

10

What's a Java Exception?

- An exception is an object with a specific interface, that can be thrown.
- All exception classes are subclasses of the class `Throwable` defined in the Java library.

- Here are some of the methods of this class:

```
Throwable();  
Throwable( String message );  
String getMessage();  
void printStackTrace();
```

- Exceptions encapsulate information about the kind of problem that has occurred (the message) and the sequence of method calls that led to the problem (the stack trace).

06/24/10

11

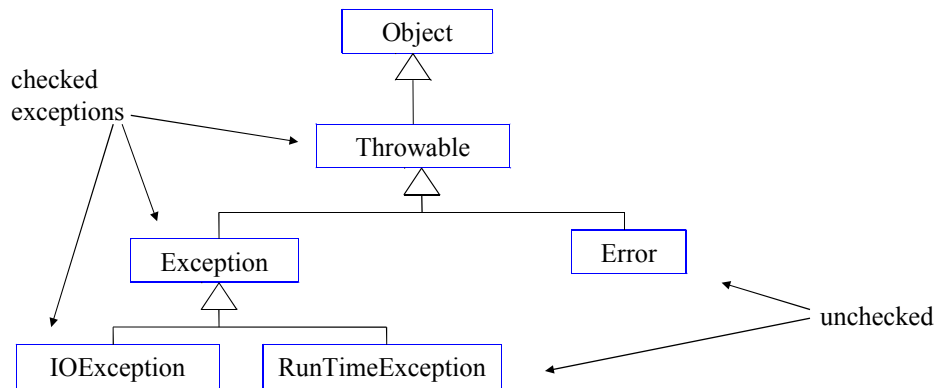
What's an exception?

- There are two types of exception: **checked** and **unchecked**.
- Unchecked exceptions are subclasses of Java's `RuntimeException` class, while all others are checked exceptions.
- There is also an `Error` class that represents abnormal conditions that a program would normally not be expected to handle. Errors are treated like unchecked exceptions.

06/24/10

12

Java Exception Hierarchy



- Numerous exceptions and errors are defined in various java packages. i.e.,
 - FileNotFoundException in java.io
 - IOException in java.io
 - NullPointerException in java.lang
 - etc.
- ^{06/24/10}Programmers can define their own exceptions as subclasses of Exception or its subclasses.¹³

Throwing an Exception

```
/**  
 * Deposit money into the account  
 * @param amount The amount to be deposited  
 *  
 * @pre true  
 * @post IF amount >= 0  
 *       THEN getBalance() = @pre.getBalance() + amount  
 *       ELSE getBalance() = @pre.getBalance()  
 * @throws IllegalArgumentException if amount is negative  
 */  
public void deposit(double amount)  
    throws IllegalArgumentException {  
    if (amount < 0)  
        throw new IllegalArgumentException("Error: Neg. amount");  
    balance = balance + amount;  
}
```

The finally clause

- A `finally` clause can follow the `catch` clauses of a `try` block (or even a `try` block with no `catch` clauses):

```
try {  
    // code that may throw checked exceptions  
}  
catch( SomeException e ) {  
    ...  
}  
finally {  
    ...  
}
```

- The `finally` clause is executed whether or not an exception is thrown, and (if thrown) whether or not it was caught.
- It is often used to ensure that resources are released.

06/24/10

15

Comments

- Note that methods can throw more than one type of exception.
- If we call a method that throws more than one type of exception we can have more than one catch block to handle each type of exception.
- Catch blocks must be ordered from the most specific type of exception (the one lowest in the inheritance hierarchy) to the least specific (the one highest in the hierarchy).

06/24/10

16

Designing Exceptions

- Need to distinguish boundary cases that can be handled by the method from exceptional cases which should throw exceptions
- Define individual exception for each type of error
 - can group them into hierarchies – allows more flexibility in handling them
- Exceptions thrown by a method are shown in the method's comment using the **@throws** tag.
- Too many exceptions may make the method difficult to use.
- Exceptions and Postconditions:
 - The postcondition should distinguish the case where an exception is thrown from the case when it is not
 - i.e., if `withdraw(amount)` throws an exception when the amount is negative, its postcondition would be:
 - IF `amount >= 0` THEN `getBalance() = @pre.getBalance() – amount`
ELSE `getBalance() = @pre.getBalance()`

06/24/10

17

Example: Class Account Re-designed

We redesign `deposit` and `withdraw` to throw exceptions in the error cases

```
/**
 * A simple bank account for which the balance can never be
 * less than zero
 *
 * @invariant getBalance() >= 0
 * @invariant getId() is unique and set when account is created
 * @invariant getName() is set when account is created
 * @invariant the values of getId() and getName() never change
 */
public class Account
{
    private int id;
    private static int nextAccountId = 0;
    private String name;
    private double balance;
    06/24/10
```

.

18

```

/**
 * Deposit money into the account
 * @param amount The amount to be deposited
 *
 * @pre amount >= 0
 * @post getBalance() = @pre.getBalance() + amount
 * @return The current balance of the account
 */
public double deposit(double amount)
{
    assert amount >= 0;

    balance = balance + amount;
    return balance
}

```

06/24/10

19

Another Design for Deposit

```

/**
 * Deposit money into the account
 * @param amount The amount to be deposited
 *
 * @pre true
 * @post IF amount >= 0
 *       THEN getBalance() = @pre.getBalance() + amount
 *       ELSE getBalance() = @pre.getBalance()
 * @return The current balance of the account
 * @throws IllegalArgumentException if amount is negative
 */
public double deposit(double amount)
    throws IllegalArgumentException {
    if (amount < 0)
        throw new IllegalArgumentException("Error: Neg. amount");
    balance = balance + amount;
    return balance
}

```

Should it have
"assert"?

YES NO

06/24/10

20

```

/**
 * Withdraw money from the account
 * @param amount The amount to be withdrawn
 * @pre true
 * @post IF (amount >= 0 AND @pre.getBalance()-amount >= 0 )
 *       THEN getBalance() = @pre.getBalance() - amount
 *       ELSE getBalance() = @pre.getBalance()
 * @return The current balance of the account
 * @throws IllegalArgumentException if amount<0
 * @throws NotEnoughMoneyException if getBalance()-amount<0
 */
public double withdraw(double amount) throws
    IllegalArgumentException, NotEnoughMoneyException {
    if (amount < 0)
        throw new IllegalArgumentException("Error: Neg. amount");
    if (balance - amount < 0)
        throw new NotEnoughMoneyException("Error: no $$$");

    balance = balance - amount;
    return balance;
}

```

21

```

/**
 * Returns the string representation of an account
 *
 * @pre true
 * @return the account represented as a string
 */
public String toString()
{
    return "[ id = " + id + ", name = " + name +
        ", balance = " + balance + " ]";
}
}

```

06/24/10

22

Account Exceptions

```
public class IllegalValueException
    extends AccountException {
    public IllegalValueException() {}
    public IllegalValueException(String msg) {
        super(msg);
    }
}

public class NotEnoughMoneyException
    extends AccountException
{
    public NotEnoughMoneyException() {}
    public NotEnoughMoneyException(String msg) {
        super(msg);
    }
}
```

- **NOTE:** We could use Java's `IllegalArgumentException` instead of defining a new exception for illegal value

06/24/10

23

Exceptions – Checked and Unchecked

- **Q:** Defining, throwing and handling checked exceptions seems like a pain. Why don't I just throw unchecked exceptions?
- **A:** This misses the point. We want to provide the client with info about what went wrong, *force them to acknowledge the problem*, and give them flexibility on how to recover

06/24/10

24

Exceptions – Checked and Unchecked

- These different exception types also have different purposes
- Checked exceptions are errors typically beyond the control of the programmer, from which the client can reasonably be expected to recover
- Unchecked exceptions are typically the result of sloppy programming, from which the client can not be expected to recover

06/24/10

25

Checked Exceptions

- **Q:** Okay, I'll use checked exceptions, but why don't I just throw general Exceptions? And why don't I replace all of my catch blocks with a single catch block for Exception?
- **A:** This still misses the point. We want define and throw specific exception types so that the client can handle each case as they see fit.

06/24/10

26

Exceptions Example

- Let's say we have a very simple method taking an integer parameter representing a month of the year, and returning the number of months until the end of the year

```
public int calcYearEnd(int month){  
    return 12-month;  
}
```

06/24/10

27

Exceptions Example

- What if month > 12?
- What if month <= 0?
- We might want to throw more than one exception and let the client respond to each as they see fit

```
public int calcYearEnd(int month){  
    return 12-month;  
}
```

06/24/10

28

Define an exception

```
public class MonthException extends
Exception{

    public MonthException(){}

    public MonthException(String msg)
    {
        super(msg);
    }
}
06/24/10
```

29

And two subclasses

```
public class HighMonthException extends MonthException{

    public HighMonthException(){}

    public HighMonthException(String msg)
    {
        super(msg);
    }
}

public class LowMonthException extends MonthException{

    public LowMonthException(){}

    public LowMonthException(String msg)
    {
        super(msg);
    }
}
06/24/10
```

30

Rewrite our method

```
public int calcYearEnd(int month) throws
HighMonthException, LowMonthException
{
    if (month > 12)
    {
        throw new HighMonthException("Error: too
high!");
    }
    if (month <= 0)
    {
        throw new LowMonthException("Error: too low!");
    }
    return 12-month;
}
```

06/24/10

31

A second method calls the first

```
public void calcRun(int month)
{
    try{
        int monthsLeft = calcYearEnd(month);
        System.out.println(monthsLeft);
    }
    catch (LowMonthException e)
    {
        System.out.println(e.getMessage());
        // the client decides to do one thing...
    }
    catch (HighMonthException e)
    {
        System.out.println(e.getMessage());
        // the client decides to do something else...
    }
}
```

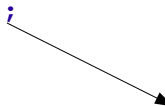
06/24/10

32

Calling the method

Assume the two methods `calcYearEnd` and `calcRun` are defined in a class `YearEnd`

```
public class YearEndTester {  
  
    public static void main(String[] args) {  
        YearEnd ye = new YearEnd();  
        ye.calcRun(14);  
  
    }  
  
}
```

 Error: too high!

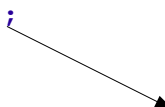
06/24/10

33

Calling the method

Assume the two methods `calcYearEnd` and `calcRun` are defined in a class `YearEnd`

```
public class YearEndTester {  
  
    public static void main(String[] args) {  
        YearEnd ye = new YearEnd();  
        ye.calcRun(-2);  
  
    }  
  
}
```

 Error: too low!

06/24/10

34

Exception methods

- So far we've seen the `getMessage()` method of an `Exception`
- The other method you are likely to find useful is `printStackTrace()`
- This prints the sequence of calls that lead to the exception being thrown

06/24/10

35

Get the stack trace

```
public void calcRun(int month)
{
    try{
        int monthsLeft = calcYearEnd(month);
        System.out.println(monthsLeft);
    }
    catch (LowMonthException e)
    {
        System.out.println(e.getMessage());
        e.printStackTrace();
        // the client decides to do one thing...
    }
    catch (HighMonthException e)
    {
        System.out.println(e.getMessage());
        e.printStackTrace();
        // the client decides to do something else...
    }
}
```

06/24/10

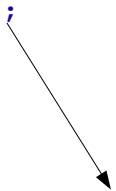
36

Calling the method

Assume the two methods `calcYearEnd` and `calcRun` are defined in a class `YearEnd`

```
public class YearEndTester {  
  
    public static void main(String[] args) {  
        YearEnd ye = new YearEnd();  
        ye.calcRun(-2);  
  
    }  
}
```

Error: too low!
LowMonthException: too low!
at YearEnd.calcYearEnd(L02Exceptions.java:13)
at YearEnd.calcRun(L02Exceptions.java:21)
at YearEndTester.main(L02Exceptions.java:5)



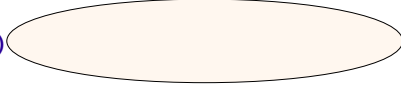
06/24/10

37

Propagation

What if get rid of the try/catch blocks and just throw an exception? What output do we get when we call the main method as before?

```
public void calcRun(int month) {  
    {  
        int monthsLeft = calcYearEnd(month);  
        System.out.println(monthsLeft);  
    }  
}
```



06/24/10

38

Propagation

We could add a try/catch block to the main method. We've got to handle the exception at some point or the program will terminate if it's propagated all the way to main and never handled

```
public class YearEndTester {  
  
    public static void main(String[] args) {  
        YearEnd ye = new YearEnd();  
        ye.calcRun(-2);  
    }  
}
```

} Exception in thread "main" java.lang.Error: Unresolved
} compilation problem:

Unhandled exception type MonthException

06/24/10

at YearEndTester.main(YearEndTester.java:5)

39

Exceptions are polymorphic

- You can DECLARE exceptions using a supertype of the exceptions you throw

```
public int calcYearEnd(int month) throws MonthException
```

- You can CATCH exceptions using a supertype of the exception thrown

```
catch(MonthException e)
```

- Just because you CAN catch everything with one big super polymorphic catch, doesn't always mean you SHOULD
- Write a different catch block for each exception you need to handle uniquely.

06/24/10

40

- - Head First Java

Order matters with catch blocks

- More specific exceptions need to be listed first
 - more specific = lower in hierarchy
- Why? What would happen if you tried to catch `Exception` first?

06/24/10

41

In-Class Exercise I

```
public class Weather
{
    String sunshine(String s) throws
        SunException, RainException
    {
        if (s != null) {
            if (s.equals("Strong sun")) {
                return "Better use sunblock!";
            }
            throw new SunException(
                "It won't last long.");
        }
        throw new RainException("No sun today.");
    }

    void fog(String x)
    {
        try {
            System.out.println(snow(x));
        }
        catch (ColdException ce) {
            System.out.println(
                "06/24/10 You should expect " + ce.getMessage());
        }
    }
}
```

```
String snow(String s) throws ColdException
{
    if (s != null && s.equals("Really cold")) {
        throw new ColdException("dry snow");
    }
    try {
        return sunshine(s);
    }
    catch (RainException re){
        return "Terrible! " + re.getMessage();
    }
    catch (SunException se) {
        return "Don't worry! " + se.getMessage();
    }
}
```

Assuming that the exceptions used here are appropriately defined, what would the following calls produce?

- `new Weather().fog("Showers");`
- `new Weather().fog("Really cold");`
- `new Weather().fog("Strong sun");`
- `new Weather().fog(null);`

42

Learning Goals Review

You will be expected to:

- incorporate exception handling into the design of a method's contract
- trace code that makes use of exception handling
- write code to throw, catch or propagate an exception
- write code that uses a finally block
- write code to define a new exception class
- compare and contrast checked and unchecked exceptions
 - understand the consequence of using checked vs. unchecked exceptions

06/24/10

43

A short video...

06/24/10

44

Tea break!

06/24/10

45

Recap + equals()

This lecture ensures you are competent with the basics of the Java you learned in 111 and 211 to this point. We'll also add in the concepts of:

- inheritance and over-riding
- equals()

If there are concepts covered today that are not clear after lecture, review of lecture materials and review of relevant parts of the book, please see an instructor to clarify the confusing points. We'll be building on all of these basics for the rest of the term.

46

46

Inheritance

- A polymorphic assignment is one of the form:

```
MyClass reference_to_object;
```

```
reference_to_object = expression;
```

where the type of `expression` must be a *subtype* of `MyClass`.

- Three types are involved here:
 - the *reference type* : the type that the reference was declared to be
 - the *expression type* : the type of the result of the expression (as can be determined at compile time)
 - the *actual type*: the type of the object that is actually returned by the expression (determined at run time)

- **The expression type must be a subclass of the reference type, otherwise this gives a compile-time error**

06/24/10

47

Inheritance

- Example: Suppose that `SavingsAccount` is a subclass of `Account`, and `SpecialSavAccount` is a subclass of `SavingsAccount` :

- ```
Account acc = new SavingsAccount();
```

reference type :  
expression type :  
actual type :

```
Account acc;
```

```
SavingsAccount sacc = new SpecialSavAccount();
```

```
acc = sacc;
```

reference type :  
expression type :  
actual type :

06/24/10

48



## Inheritance

- You can explicitly convert (cast) references between a type and its subtypes:
  - **widening**: converting a subtype to one of its super-types
    - always allowed in Java (subtype "is-a" super-type)
    - no explicit cast is needed
  - **narrowing or downcasting**: converting a supertype to one of its subtypes
    - not always possible
    - may throw `ClassCastException`
    - explicit cast is always needed

```
Account acc = new SavingsAccount();
SavingsAccount sacc;
sacc = (SavingsAccount) acc; // cast OK
```

06/24/10

49

## More Downcast Uses

- The *declared (static) type* of a reference determines which methods can be called.
- Suppose that `SavingsAccount` defines `addInterest()` and that this method is not defined in `Account`, then:

```
Account acc = new SavingsAccount();
acc.addInterest(); // Error
but
((SavingsAccount) acc).addInterest(); // ok
```

the following also works:

```
SavingsAccount sacc = (SavingsAccount) acc;
sacc.addInterest();
```

06/24/10

50

## Which method is called?

- The actual method called is determined at run time based on the actual type of the object.
- Example: Suppose that `SavingsAccount` is a subclass of `Account`, and `SpecialSavAccount` is a subclass of `SavingsAccount` :

```
Account acc;

SavingsAccount sacc = new SpecialSavAccount();

...
acc = sacc;
acc.m();
```

calls the method `m()` defined for the `SpecialSavAccount`; defined in `SpecialSavAccount` or inherited from its lowest superclass that defines `m()`.

06/24/10

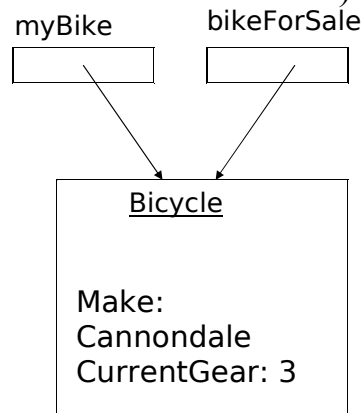
51

## Comparing References

- We can compare two object references using Java's `==` operator
  - two object references are equal if and only if they reference the same object (i.e. have the same value)

```
Bicycle myBike;
Bicycle bikeForSale;

myBike = new Bicycle();
bikeForSale = myBike;
```



- In this case  
`bikeForSale == myBike`

is true  
06/24/10

52

52

## Comparing Objects

- To compare objects we have to use Java's `.equals()` method
  - Equals is defined in `Object` to mean `==`
  - `a.equals(b)` can be overridden if `a` and `b` are the same as far as an application is concerned.

```
Bicycle myBike;
Bicycle bikeForSale;

myBike = new Bicycle();
bikeForSale=new Bicycle();
```

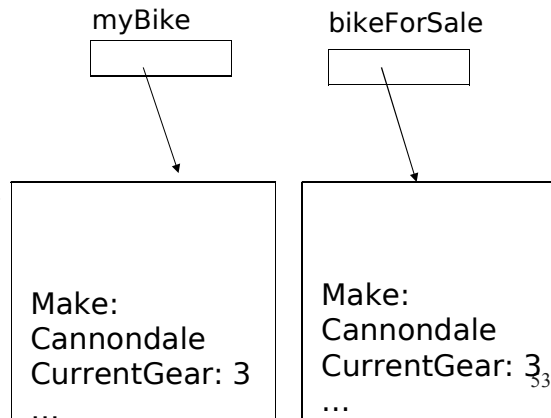
Then:

```
myBike.equals(bikeForSale)
```

May be true, but

```
myBike == bikeForSale
```

is false



53

## Overriding equals()

- Its parameter must be of type `Object` to match the method defined in the `Object` class
- The method must check if its explicit argument is null
  - must return false if it is so
- The method must check that its implicit and its explicit arguments are of the same type
  - must return false if they are of different type
- For any object references `o`, `o1`, `o2`, `o3` in the class, the following must hold
  - `o.equals(o)`
  - IF `o1.equals(o2)` THEN `o2.equals(o1)`
  - IF `o1.equals(o2)` AND `o2.equals(o3)` THEN `o1.equals(o3)`
- The method usually returns
  - true if the two objects are of the same type and their corresponding data components are equal
  - false otherwise

06/24/10

54

54

## Checking an Object's Class

- To check that two objects are of the same type, equals()
    - must use `getClass()` (defined in `Object`) to get the actual class of an object
    - `instanceof` is not specific enough; returns true if one object is a subclass of the other
  - For instance, assuming `SavAccount` extends `Account` :
    - `Account a1 = new Account();`
    - `Account a2 = new SavAccount();`
    - `SavAccount a3 = new SavAccount();`
    - `Account a4 = (Account) a3;`
- then
- `a1 instanceof Account`, `a2 instanceof Account`, `a3 instanceof Account`,  
`a4 instanceof Account`, are all true
- but
- `a1.getClass()` returns "Account"
  - `a2.getClass()` returns "SavAccount"
  - `a3.getClass()` returns "SavAccount"
  - `a4.getClass()` returns "SavAccount"

06/24/10

55

55

## Overriding equals () in Account

```
public boolean equals(Object obj) {
 if (obj == null)
 return false;
 if (getClass() != obj.getClass())
 return false;

 Account other = (Account) obj;
 return id == other.id
 && balance == other.balance
 && name.equals(other.name);
}
```

06/24/10

56

56

# APPENDIX

## Credit Card Example

06/24/10

57

```
public class SIVACard {
 private int number; // A unique number for each card
 private static int nextNumber = 1; // A real credit card # is far more
 // complicated to generate
 private CardHolder holder; // The holder of the card
 protected double rewardsBalance; // The points available to the card
 holder
 private double balance; // The balance on the card
 private Transaction[] currentTransactions; // This period's transactions
 private int currentTransactionCount;
 private Statement lastStatement; // Last statement issued
 private final static int TRANSACTION_LIMIT = 30;

 public SIVACard(CardHolder holder) {
 number = nextNumber++;
 this.holder = holder;
 rewardsBalance = 0.0;
 balance = 0.0;
 lastStatement = new Statement(this); // Blank statement
 currentTransactionCount = 0;
 currentTransactions = new Transaction[TRANSACTION_LIMIT];
 }

 public double getRewardsBalance() {
 return rewardsBalance;
 }

 public double getBalance() {
 return balance;
 }

 public Statement getLastStatement() {
 return lastStatement;
 }

 public void postPayment(double amount) {
 balance -= amount;
 }

 public void redeemRewards(double amount) {
 rewardsBalance -= amount;
 }

 public CardHolder getHolder() {
 return holder;
 }

 /**
 * Post a purchase to the card
 * @param purchase The purchase to post.
 */
 public void postPurchase(Transaction purchase) {
 currentTransactions[currentTransactionCount++] = purchase;
 computeRewards(purchase);
 }

 /**
 * Put all current transactions onto the last statement.
 */
 public void generateStatement() {
 lastStatement = new Statement(this);
 for (int i = 0; i < currentTransactionCount; i++) {
 lastStatement.addTransaction(currentTransactions[i]);
 }
 currentTransactionCount = 0;
 }

 /**
 * A basic card just gives you one reward point per dollar purchased
 * @param purchase The purchase from which to determine points
 */
 void computeRewards(Transaction purchase) {
 rewardsBalance += purchase.getValue();
 }
}
```

06/24/10

58

```

public class CardHolder {

 private String lastName;
 private String firstName;

 public CardHolder(String lastName, String firstName) {
 this.lastName = lastName;
 this.firstName = firstName;
 }

 public String toString() {
 return new String(lastName + ", " + firstName);
 }

}

```

06/24/10

59

```

public class Statement {

 private String[] lines;
 private int numberOfLines; // The lines on the statement
 private SIVACard card;

 public Statement(SIVACard card) {
 numberOfLines = 0;
 lines = new String[30];
 this.card = card;
 }

 public void addTransaction(Transaction aTransaction) {
 lines[numberOfLines++] = aTransaction.toString();
 }

 public void print(PrintStream stream) {
 stream.println("=====" + card.getHolder().toString());
 stream.println("\tTransactions:");
 for (int i = 0; i < numberOfLines; i++) {
 stream.println("\t\t" + lines[i]);
 }
 stream.println("\tRewards: " + card.getRewardsBalance());
 stream.println(":::");
 stream.println();
 }

}

```

06/24/10

60

```

public class Transaction {

 private int value;
 private String description;

 public Transaction(String description, int value) {
 this.value = value;
 this.description = description;
 }

 public int getValue() {
 return value;
 }

 public String getDescription() {
 return this.description;
 }

 public String toString() {
 return new String(description + ": " + value);
 }
}

```

06/24/10

61

```

public class Driver {

 private final static int NUMBER_OF_CARDS = 10;
 private static Object[] allCards;
 private static int allCardsIndex = 0;

 public static void main(String args[]) {

 CardHolder gail = new CardHolder("Murphy", "Gail");
 CardHolder george = new CardHolder("Tsiknis", "George");

 SIVACard gailsCard = new SIVACard(gail);
 SIVACard georgesCard = new SIVACard(george);

 gailsCard.postPurchase(new Transaction("sushi", 20));
 gailsCard.postPurchase(new Transaction("air-cdn", 750));

 georgesCard.postPurchase(new Transaction("pizza", 25));
 georgesCard.postPurchase(new Transaction("air-cdn2",
750));

 printStatement(gailsCard);
 printStatement(georgesCard);

 CardHolder gail2 = new CardHolder("Murphy", "Gail");
 if (gail.equals(gail2))
 System.out.println("The two cardholders are the
same!");
 else
 System.out.println("The two cardholders are
different");
 }
}

```

06/24/10

62

# SOLUTION

06/24/10

63

```
public class CardHolder {

 private String lastName;
 private String firstName;

 public CardHolder(String lastName, String firstName) {
 this.lastName = lastName;
 this.firstName = firstName;
 }

 public String toString() {
 return new String(lastName + ", " + firstName);
 }

 public boolean equals(Object ob){
 if (ob == null)
 return false;
 if (getClass() != ob.getClass())
 return false;
 CardHolder holder = (CardHolder)ob;
 return lastName.equals(holder.lastName) &&
 firstName.equals(holder.firstName);
 }
}
```

06/24/10

64



# Interfaces

- When we define a class that **implements** an **interface**, we are committed to providing definitions for the abstract methods listed in the interface
- The interface itself contains no method definitions, it just tells you what you need to do
- So if you need to implement an interface (e.g. for an assignment, hint hint), look at the interface definition and it will tell you which methods your class will need

06/24/16

65

## Interfaces vs. Classes

An interface type is similar to a class, but there are several important differences:

- *All methods in an interface type are abstract; they don't have an implementation*
- *All methods in an interface type are automatically public*
- *An interface type does not have instance fields*

66

## Syntax 9.1 Defining an Interface

```
public interface InterfaceName
{
 // method signatures
}
```

### Example:

```
public interface Measurable
{
 double getMeasure();
}
```

### Purpose:

To define an interface and its method signatures. The methods are automatically public.

67

## Syntax 9.2 Implementing an Interface

```
public class ClassName
implements InterfaceName, InterfaceName, ...
{
 // methods
 // instance variables
}
```

### Example:

```
public class BankAccount implements Measurable
{
 // Other BankAccount methods
 public double getMeasure()
 {
 // Method implementation
 }
}
```

68

# Advantages of Interfaces

- Polymorphism
  - Classes that implement an interface x will have objects of type x with the methods defined in x, but the method definitions will differ
- Simulating multiple inheritance
- Reducing coupling between classes

69

# A simple interface

```
public interface Moveable {

 public void moveForward();
 public void moveBackward();
}
```

70

## Implementing the interface

```
public class Car implements Moveable {
 public void moveBackward() {
 System.out.println("Going 95 in reverse");
 }

 public void moveForward() {
 System.out.println("Going 95 on the freeway");
 }
}
```

71

## Implementing the interface

```
public class Bike implements Moveable {
 public void moveBackward() {
 System.out.println("Pedaling backwards!");
 }

 public void moveForward() {
 System.out.println("Pedaling forwards!");
 }
}
```

72

# Interfaces and Polymorphism

```
public class MoveTest {
 public static void main(String[] args) {
 Moveable[] moveArr = new Moveable[2];
 moveArr[0] = new Bike();
 moveArr[1] = new Car();
 for (Moveable mover: moveArr)
 {
 mover.moveForward();
 }
 }
}
```

What gets printed?

73

# Inner Classes

- A trivial class can be defined within another class – thus “inner” class
- We will be discussing this in detail later in the term
- An inner class can use all the methods and variables of the outer class, even the private ones

74

## Inner Classes

```
class MyOuterClass {
 private int x;
 class MyInnerClass {
 void go() {
 x = 42;
 }
 }
}
```

75

## Inner Classes

```
class MyOuter {
 private int x;
 class MyInner {
 void go() {
 x = 42;
 }
 }
}
```

We can use x just as if it  
were a variable of the inner  
class

76

# Inner Classes

- An instance of the *inner* class is tied to an instance of the *outer* class

77

```
class MyOuter {
 private int x;
 MyInner inner = new MyInner();
 public void doStuff(){
 inner.go();}

 class MyInner {
 void go() {
 x = 42;}
 } // end of inner class
 } // end of outer class
```

78