Class Design I Class Contracts

You will be expected to:

- implement a class given a contract specified by invariants, preconditions and postconditions
- write client code that adheres to the contract specified for a class using invariants, preconditions and postconditions
- describe the benefits of programming by contract for client and developer
- use assertions appropriately in code

Readings:

2nd Ed:

Section 9.5, Advanced Topic 9.2

3rd/4th Eds:

Section 8.5, Special/Advanced Topic 8.2

Some ideas come from:

- ***"Practical Object-Oriented Development with UML and Java" R. Lee, W. Tepfenhart, Prentice Hall, 2002.***
- ***"Object-Oriented Software Development Using Java", Xiaoping Jia, Addison Wesley, 2002***

Why do we need contracts?

- Write an implementation for the following methods of the `Account` class:

```
public class Account
{
    private int id;
    private static int nextAccountId = 0;
    private String name;
    private double balance;

        /**
         * Deposits an amount to the account.
         * @param amount The amount to be deposited.
         */
    public double deposit(double amount)
    {
```


Why do we need contracts?

```
   //cont'd

/**
 * Withdraws an amount from the account.
 * @param amount The amount to be withdrawn.
 */
public double withdraw(double amount)
   {
```


Questions?

- Even though the methods are nicely documented with standard Javadoc comments, the task on the previous slides probably raised some questions:
  -
  -
- That's why we need a contract.
- **A contract specifies more clearly what a method is supposed to do and not do.**

Specifying a Class Contract

- We will specify a class contract using class invariants, preconditions and postconditions.
- Each of these is a statement that we require to be true at some point in the code.
  - A **class invariant** is attached to a class. Invariants must be true from the time the call to a constructor ends until the corresponding object is destroyed.
  - A **precondition** is attached to a method. Preconditions must be true at the time that the method is called.
  - A **postcondition** is also attached to a method. Postconditions must be true at the time that the call to the method ends.
- They are all expressed in terms of the public class components

Preconditions

- Precondition: Requirement that the caller of a method must meet
- Publish preconditions so the caller won't call methods with bad parameters
- Typical use:
  - *To restrict the parameters of a method*
  - *To require that a method is only called when the object is in an appropriate state*

Preconditions (cont.)

- If precondition is violated, method is not responsible for computing the correct result. It is free to do *anything*

Preconditions

- Method may throw exception if precondition violated – more about that in a few minutes

```
if (amount < 0) throw new IllegalArgumentException();
balance = balance + amount;
```

- Method doesn't have to test for precondition. (Test may be costly)

```
// if this makes the balance negative, it's the caller's
   fault
balance = balance + amount;
```

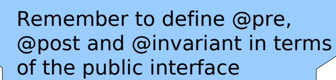
Postconditions

- Condition that is true after a method has completed
- If method call is in accordance with preconditions, it must ensure that postconditions are valid
- There are two kinds of postconditions:
  - *The return value is computed correctly*
  - *The object is in a certain state after the method call is completed*
- Don't document trivial postconditions that repeat the `@return` clause

Postconditions (cont.)

```
amount <= getBalance() // this is the way to state a
   postcondition
amount <= balance // wrong postcondition formulation
```

- Contract: If caller fulfills precondition, method must fulfill postcondition



Question

Why might you want to add a precondition to a method that you provide for other programmers?

Answer: Then you don't have to worry about checking for invalid values – it becomes the caller's responsibility.

Question

When you implement a method with a precondition and you notice that the caller did not fulfill the precondition, do you have to notify the caller?

Answer: No – you can take any action that is convenient for you.

`Account` - invariants

- An invariant is used to specify a condition that must be true about the state of an object from the time it's constructed (when the call to the constructor ends) until it is destroyed.
- What invariants can we specify on our `Account` class?

```
/**
  * @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
  * @invariant ________________________________
*/
public class Account
{
    private int id;
…
}
```


`deposit` - pre/postconditions

What preconditions and postconditions can we specify for the `deposit` method?

```
/**
    * Deposit money into the account
    * @param amount The amount to be deposited
    *
    * @pre ______________________
         *
    * @post _____________________________________________________
    * @return The current balance of the account
    */
 public double deposit(double amount)
 {
```


`withdraw` - pre/postconditions (version 1)

Can be specified as the following:

```
/**
 * Withdraw money from the account
 * @param amount The amount to be withdrawn
 * @pre
 * @pre
 *
 * @post
 *
 * @return The current balance of the account
 */
public double withdraw(double amount) {
```


`withdraw` - pre/postconditions (version 2)

Or maybe you can try to withdraw too much, but get nothing:

```
/**
 * Withdraw money from the account
 * @param amount The amount to be withdrawn
 * @pre
 * @post


 * @return The current balance of the account
 */
public double withdraw(double amount) {
```

Responsibilities and Benefits

	Method Implementer	User of the Method
Precondition	Benefit - assumes that the precondition is true - simplifies code	Obligation - must verify that the preconditions hold before a method is called
Postcondition	Obligation - ensure that if the preconditions of a method are met, its postconditions are true when the method terminates	Benefit - assumes that the postcondition is true when the method returns

Specifying Class Contracts

- We will specify invariants, preconditions and postconditions using simple English phrases coupled together by logical AND, OR, NOT, IF…THEN…ELSE as necessary.
- Contracts should be specified only in terms of the public interface of the class whenever possible so as to avoid exposing non-public elements of the class.

  So, in specifying a postcondition for the `deposit` method, for example, we use `getBalance()` (which is public) rather than `balance` (which is private).

Specifying Class Contracts (cont'd)

- Suppose we are documenting a method `m()` and that `foo()` is some other method whose value we wish to use in specifying a pre- or post-condition.

  We use `@pre.foo()` to refer to the value returned by `foo()` before `m()` is called and `foo()` to refer to its value after `m()` is called.

Formal Languages for Class Contracts

- There is a more formal way to define these conditions using Object Constraint Language (OCL). We won't use OCL in this course but if you're interested, see: http://www.omg.org/docs/formal/06-05-01.pdf
- There are also tools that will instrument Java code so that OCL constraints can be checked at runtime. See, for example, http://sourceforge.net/projects/dresden-ocl/

Java Assertions

- We can use assertions to easily verify that preconditions are true.
- Assert statements are Java constructs that assert that a given condition is true. If the condition is not true, the program is terminated with an `AssertionError`.

```
/**
 * …
 *@pre
 */
public double deposit( double amount )
{
```


Java Assertions

- By default, assert statements are not checked. To enable assertion checking, run your program with the `-enableassertions` flag:

```
java -enableassertions myJavaProgram
```

- It's good practice:
  - to use assert statements to check preconditions
  - to enable assert statements during testing and debugging
  - to disable assert statements for the final production version

Assertions

`assert condition;`

Example:

`assert amount >= 0;`

Purpose:

To assert that a condition is fulfilled. If assertion checking is enabled and the condition is false, an assertion error is thrown.

@invariant, @pre and @post

- Class contracts are the important part of the class documentation and should be included in the class javadoc
- The @invariant, @pre and @post tags are **not** standard Javadoc tags (yet).
- For instructions on how to use these tags in your programs, please see the tutorial on the course web site.
- The complete definition of the class Account (containing contracts and code) will be posted on the course web site.

More Exercises

- Review Exercises:
  - 3rd Ed, Chapter 8
    - Questions : R8.13, R8.15, R8.16, R8.17, R8.19, R8.20, R8.21, R8.22, R8.23
    - Programming Exercises: P8.1, P8.2, P8.3
  - 2nd Ed, Chapter 9
    - Questions : R9.13, R9.15, R9.16, R9.17, R9.19, R9.20, R9.21, R9.22, R9.23
    - Programming Exercises: P9.1, P9.2, P9.3

Learning Goals Review

You will be expected to:

- implement a class given a contract specified by invariants, preconditions and postconditions
- write client code that adheres to the contract specified for a class using invariants, preconditions and postconditions
- describe the benefits of programming by contract for client and developer
- use assertions appropriately in code

Tea break!

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
  - 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

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!

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:

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.

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:

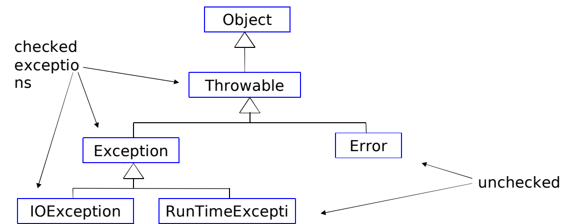
```
Throwable();
Throwable( String message );
String getMessage();
void printStackTrace();
```

- 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).

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.

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.
- Programmers can define their own exceptions as subclasses of Exception or its subclasses.

Checked and Unchecked Exceptions

•

- *Checked*
  - o The compiler checks that you don't ignore them
  - o Due to external circumstances that the programmer cannot prevent
  - o Majority occur when dealing with input and output
  - o For example, `IOException`
- *Unchecked:*
  - o Extend the class `RuntimeException` or `Error`
  - o They are the programmer's fault
  - o Examples of runtime exceptions:
    `NumberFormatException`
    `IllegalArgumentException`
    `NullPointerException`
  - o Example of error:
    `OutOfMemoryError`

Hierarchy of Exception Classes

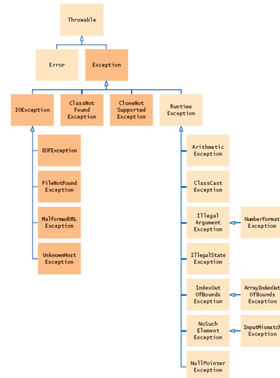


Figure 1 The Hierarchy of Exception Classes

Defining an Exception Class

- Returning to our `Account` example, suppose we decide to throw an exception when the amount is negative.
- First we must decide which exception class to use. We could use the class `Exception` in the Java library but we can capture more information by defining our own exception class.
- Let's define a class named `IllegalValueException` to represent the type of exception that will be thrown when we attempt to pass a negative amount.
- This will be a checked exception (more about this later).

Defining an exception class

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


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 IllegalValueException if amount is negative
 */
public void deposit(double amount)
                         throws IllegalValueException {
  if (amount < 0)
     throw new IllegalValueException("Error: Neg. amount");
   balance = balance + amount;
}
```


Throwing Exceptions

`throw` *`exceptionObject;`*

Example:

`throw new IllegalArgumentException();`

Purpose:

To throw an exception and transfer control to a handler for this exception type.

Handling Exceptions

- Recall that `IllegalValueException` is a checked exception. This has consequences for a client calling our `deposit` method. The client code **must** do one of the following:
  - catch the exception
  - propagate (i.e., pass on) the exception to its caller (i.e., the method that called it)

Catching Exceptions

- Statements in `try` block are executed
- If no exceptions occur, `catch` clauses are skipped
- If exception of matching type occurs, execution jumps to catch clause
- If exception of another type occurs, it is thrown until it is caught by another `try` block

Try-Catch Syntax

```
try
{
   statement
   statement
   . . .
}
catch (ExceptionClass exceptionObject)
{
   statement
   statement
   . . .
}
catch (ExceptionClass exceptionObject)
{
   statement
   statement
   . . .
}
. . .
```

Client Catching an Exception

```
public static void main( String[] args ) {

      Account instructorAccount =
        new Account ( "instructor", 100.0 );

       try {
            instructorAccount.deposit( 100 );
            System.out.println( "Balance: " +
                 instructorAccount.getBalance() );
       }
       catch( IllegalValueException e ) {
            System.out.println( e.getMessage() );
       }
}
```

- What happens when `deposit` is called?

What happens when this code executes?

```
public static void main( String[] args ) {

      Account instructorAccount =
        new Account ( “instructor”, 100.0 );

       try {
             instructorAccount.deposit( -100 );
             System.out.println( "Balance: " +
                   instructorAccount.getBalance() );
       }
       catch( IllegalValueException e ) {
             System.out.println( e.getMessage() );
       }
}
```


Client Propagating an Exception

```
public void depositToAccount( Account anAccount,
  double amount ) throws IllegalValueException
{
  anAccount.deposit( amount );
  System.out.println( "Balance: " +
  anAccount.getBalance() );
}
```

- The *method that calls* `deposit` must either:
  - catch the exception
  - propagate the exception to its caller
- If it propagates the exception then its caller must either catch or propagate the exception and so on…

Tracing an example call

- Trace the following code:

```
public static void main( String[] args ) {
 Account anAccount = new Account ( “test”, 200 );

    try {
       depositToAccount( anAccount, 100.0 );
    }
    catch( IllegalValueException e ) {
       System.out.println( e.getMessage() );
    }
}
```


Tracing an example call

- Trace the following code:

```
public static void main( String[] args ) {

   Account anAccount = new Account ( “test”, 200 );

   try {
      depositToAccount( anAccount, -100.0 );
   }
   catch( IllegalValueException e ) {
      System.out.println( e.getMessage() );
   }
}
```

Exception Propagation

- If the exception is propagated as far as `main()` and `main()` doesn't catch the exception, the program is terminated.
- The error message associated with the exception is printed on the screen along with the stack trace.
- Checked exceptions should be caught and handled somewhere in your code.
- Allowing your program to terminate when an exception is thrown is sloppy (and could lead to disaster in real code!)

Unchecked Exceptions

- If a method throws an *unchecked* exception, the rules are different:
  - it is not necessary to declare that the method throws the exception
  - there is no requirement on the calling method to handle the exception (i.e., doesn't have to catch or propagate the exception)
- If we don't handle unchecked exceptions in our code (and we usually don't), the program will terminate when an unchecked exception is thrown (e.g., `ArrayIndexOutOfBoundsException`, `NullPointerException`).

Checked or unchecked?

- When we define our own exception class, should it be checked or unchecked?
- It depends.
- In general, we make it a checked exception because we usually want to force the client to deal with it.
- The exception (excuse the pun) is when we want to respond to common problems that are the result of sloppy programming (e.g., index out of bounds exception for an array), in which case we'd probably use an unchecked exception.

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.

The `finally` Clause

```
FileReader reader = new FileReader(filename);
try
{
   Scanner in = new Scanner(reader);
   readData(in);
}
finally
{
   reader.close(); // if an exception occurs, finally
         clause is also
                    // executed before exception is passed
         to its handler
}
```


Example

```
public class ExamMarker {
   //...
   /**
    * Calculates the given mark as a percentage of max mark
    * @param mark the given mark
    * @param max the maximum mark for the exam
    * @return the mark as a percentage of max
    */
   public int percentage(double mark, double max)
          throws IllegalMarkException, IllegalMaxException {
      if ( max == 0 )
       throw new IllegalMaxException( "Max is 0" );
      if( mark < 0 || mark > max )
   throw new IllegalMarkException( "Incorrect
                                 Mark Submitted" );
      return (int)( mark / max * 100 );
   }
```



```
public static void main(String[] args)
{
   ExamMarker marker = new ExamMarker();
   Scanner input = new Scanner( System.in );
   double mark, max;
   int percent;

   System.out.println( "Enter a mark for this exam and the
                         max mark: " );

   // cont'd
```



```
   while( input.hasNext() )
   {
   mark = input.nextDouble();
   max = input.nextDouble();

       try
       {
   percent = marker.percentage( mark, max);
   System.out.println( "The exam mark is: "
                                  + percent + "%" );
   }
   catch( IllegalMaxException e )
   {
   System.out.println( "Exam Marker Error: "
                                  + e.getMessage() );
   }
   catch( IllegalMarkException e )
   {
   System.out.println( "Exam Marker Error: "
                                  + e.getMessage() );
   }
   }
}
```

```
public class ExamMarkerException extends Exception
{
   public ExamMarkerException(){ }

   public ExamMarkerException( String msg )
   {
    super(msg);
   }
}
```



```
public class IllegalMarkException extends ExamMarkerException
{
   public IllegalMarkException(){ }
   public IllegalMarkException( String msg )
   {
     super( msg );
   }
}

public class IllegalMaxException extends ExamMarkerException
{
   public IllegalMaxException(){}
   public IllegalMaxException( String msg )
   {
     super( msg );
   }
}
```


In-Class Exercise II

- What will be output if we enter the following data on the keyboard?
  20.0 50.0

  40.0 30.0

- Give an example of data that we could enter on the keyboard to cause an `IllegalMaxException` to be thrown.

Question

- What if we replace the two `catch` blocks in `main()` with the following?

```
catch( ExamMarkerException e )
{
 System.out.println( "Exam Marker Error: "
                    + e.getMessage() );
}
```

- How will the output change?

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).

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()

APPENDIX

Account Example
with
Class Contracts

Example: Class Account

```
/**
 * 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;
```


64

```java
/**
 * Initializes an account
 * @param accountName Customer name for account
 * @param initialBalance Initial balance deposited in account
 *
 * @pre true
 * @post getName() = accountName
 * @post getId() = a new number not returned by other accounts
 * @post (initialBalance>=0 AND getBalance() = initialBalance)
 *       OR  getBalance() =  0
 *
 */
public Account(String accountName, double initialBalance) {
       id = nextAccountId++;
       name = accountName;
       if (initialBalance >= 0)
         balance = initialBalance;
       else
         balance = 0;
}
```



```java
/**
 * Accessor method to return the account id
 * @pre true
 * @return the account id
 */
public int getId() {
  return id;
}

/**
 * Accessor method to return the customer name
 * @pre true
 * @return the customer name
 */
public String getName() {
  return name;
}
```



```java
/**
 * 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;
}
```



```java
/**
 * Withdraw money from the account
 * @param amount The amount to be withdrawn
 *
 * @pre amount >= 0
 * @post IF (@pre.getBalance()-amount >= 0 )
 *        THEN getBalance() = @pre.getBalance() - amount
 *        ELSE getBalance() = @pre.getBalance()
 * @return The current balance of the account
 */
public double withdraw(double amount) {

  assert  amount >= 0;


     if (balance - amount >= 0)
         balance = balance - amount;

  return balance;
}
```

```java
  /**
   * 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 + "]";
  }

}
```


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