**Administrivia**

- Lecture slides (day by day) are on the web:
- Reminder: Assignment #1 is on the web site
  - Due Thursday, January 28th, 10:00pm

**Where are we?**

**We’re learning about exceptions**

- An alternative way for a method to complete
- Checked vs. unchecked
- Throwing and catching
Deposit

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

Deposit to account

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

• Trace the following code:

```java
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() );
    }
}
```

Balance: 300.00

Tracing an example call

• Trace the following code:

```java
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() );
    }
}
```

Error: Neg amount
Exception Propagation

• If an 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):

  ```java
  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.
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 );
    }
}

**Question**

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

  **The exam mark is 40%**

  40.0 30.0

  **Exam Mark Error: Incorrect**

  **Illegal mark submitted**

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

  -10.0 0.0 -400 -200
Question

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

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

• How will the output change?

   Not at all

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, each one handling a different 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:
    o IF amount>=0 THEN getBalance() = @pre.getBalance() – amount
    ELSE getBalance() = @pre.getBalance()

Example: Class Account Re-designed

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

```java
/**
 * 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;
  ...
```

```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;
}
```

Another Design for Deposit

```java
/**
 * 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 IllegalValueException if amount is negative
 */
public double deposit(double amount)

    throws IllegalValueException {
        if (amount < 0)
            throw new IllegalValueException("Error: Neg. amount");
        balance = balance + amount;
        return balance;
    }
```
/**
 * 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 IllegalValueException if amount<0
 * @throws NotEnoughMoneyException if getBalance()-amount<0
 */

public double withdraw(double amount) throws IllegalValueException, NotEnoughMoneyException {
    if (amount < 0)
        throw new IllegalValueException("Error: Neg. amount");
    if (balance - amount < 0)
        throw new NotEnoughMoneyException("Error: no $$$");
    balance = balance - amount;
    return balance;
}

/**
 * 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 + "]";
}
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

Exercise

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

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

    String snow(String s) throws ColdException {
        if (s != null && s.equals("Really cold")) {
            throw new ColdException("dry snow");
        } else {
            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);