Lecture 2
JAVA (46-935)
Somesh Jha
package interestRate;

/**
 * The interface definition used by Node.
 * @author Somesh Jha
 */

public interface KeyInterface {

/**
 * Checks whether two instances of KeyInterfaces are equal.
 */
public boolean isEqual(KeyInterface key);

/**
 * Prints a Key interface.
 */
public void print();

};
Interfaces (Contd)

- **KeyInterface** is an interface not a class (cannot instantiate).

- Think about interfaces as templates.

- Any class that *implements* or follows the template KeyInterface has to have the two methods: `isEqual` and `print` (with exactly the same type).

- More on this later.
Package?

- What is that funny package interestRate statement?

- JAVA allows one to organize class files into packages.

- You can group classes that are inter-related into one package.

- More on this later.
package interestRate;

/**
 * A node of a linked list.
 * @author Somesh Jha
 */

public class Node {

    // Next and previous links
    public Node prev, next;
    public KeyInterface key;

    // No of successors of this node
    // Initially this is only 2 because
    // of the binomial model
    static final int NO_OF_SUCC = 2;

    Node succ[] = new Node[NO_OF_SUCC];

    /**
     * @param k Key of that node.
     */
    public Node(KeyInterface k) {
        key = k;
        prev=next=null;
    }// end of Node

    /**
     * Search a node that is a successor of this node
     * and has a specified key.
     */
@param k Specified Key
*/

public Node Search(KeyInterface k) {
    Node x = this;
    while (x != null) {
        if (x.key.isEqual(k)) return(x);
        else x = x.next;
    }
    //didn’t find the key
    return(x);
} //end of Search

/**
 * Delete this node
 */

public void Delete() {
    Node prev1 = this.prev;
    Node next1 = this.next;
    System.out.print("Node:Delete: prev and next ");
    prev1.key.print();
    next1.key.print();

    if (prev1 != null) prev1.next = next1;
    if (next1 != null) next1.prev = prev1;
} //end of Delete

/**
 * Insert a node after a specified node
 */

public void InsertAfter(Node x) {
    Node temp = this.next;
    this.next = x;
    x.prev = this;
    x.next = temp;
    if (temp != null) temp.prev = x;
}

/**
 * Insert a node before a specified node.
 */
/*
* Insert element in the link list
starting from the node if not already there
*
public Node InsertElement(KeyInterface k) {
    Node x = Search(k);
    //element not there
    if (x == null) {
        x = new Node(k);
        InsertAfter(x);
    }
    return(x);
} //end of InsertElement

/**
print the linklist starting at the node
*/
public void print() {
    Node x=this;
    while (x != null) {
        x.key.print();
        x = x.next;
    }
} //end of print

}//end of class Node
Node (Contd)

• **Node** is a subclass of **Object**.

• Every class of non-primitive type is a subclass of **Object**.

• **Primitive types** are **int**, **double**, **String**, etc.

• Anything declared **public** is visible by everybody.
Static

- Constants are always declared in the following manner:

  \texttt{static final int NO\_OF\_SUCC}

- \texttt{static} says it is a constant, i.e., different object instances of class \texttt{Node} share that constant.

- \texttt{final} says that you can’t override it.
Constructors

- Constructor is declared as
  
  \[
  \text{public Node(KeyInterface } k) \]

- Constructors are called when we allocate an object of the particular class.

- You can have several constructors.
Destructors

- There are none.

- JAVA is garbage collected (remember!).

- You can simulate destruction, but we will get to that later.
package interestRate;

/**
   * @author Somesh Jha
   * If one tries to delete or print an empty
   * LinkList, this exception occurs.
   */
public class LinkListException extends Exception {

    public LinkListException() { super(); }
    public LinkListException(String s) { super(s); }

}//end of LinkListException
Exceptions (Contd)

- LinkListException is a subclass of Exception.

- Exception is a JAVA defined class.

- super in the constructor calls the constructor for the super class (in this case Exception).

- Exceptions are raised or thrown when something weird happens.
package interestRate;

/**
 * Implements a doubly linked list.
 * @author Somesh Jha
 * /
 public class LinkList {

 //Head of the list
 Node head;

 static final boolean DEBUG=false;

 //constructor
 public LinkList() {
   head = null;
 }//end of LinkList

/**
 * Insert a new node if doesn’t
 * exist with key k
 * */
 public Node Insert(KeyInterface k) {
   if (DEBUG) {
     System.out.print("Inserting node with key: ");
     k.print();
   }
   Node x;
   if (head == null) {
     x = new Node(k);
     head=x;
   }
   else {
     // Insertion code...
   }
}
x = head.InsertElement(k);
}
return(x);
}//end of Insert

/**
 * Delete a node with a certain key if it exist.
 * @exception LinkListException Thrown when the linked-list is empty.
 */
public void Delete(KeyInterface k) throws LinkListException {
  if (DEBUG) {
    System.out.print("Deleting node with key: ");
    k.print();
  }

  if (head == null) {
    throw new LinkListException("Deleting from an empty list");
  } else {
    //Find the element
    Node x = head.Search(k);
    if (x != null) {
      x.print();

      if (head == x) {
        head = head.next;
      }
      x.Delete();
    }
  }
}//end of Delete

/**
 * print the LinkList
 * @exception LinkListException Thrown when the linked-list is empty.
 */
public void print() throws LinkListException {
  if (head == null)
    throw new LinkListException("Printing an empty list");
  else {

System.out.println("--------------BEGIN----------------");
head.print();
System.out.println("--------------END----------------");
}

}//end of LinkList
Throwing an Exception

- Notice that the print () method throws and exception if the linked-list is empty.

- The statement that throws the exception is:

```java
if (head == null)
    throw new LinkListException("Printing an empty list");
```

- How to catch or handle an exception? Will get to that later!
Abstraction

- Notice that the actual structure of the *Key* was never mentioned in the classes *Node* and *LinkList*.

- We always worked with *KeyInterface*.

- As long as an actual key implements the interface the classes *LinkList* and *Node* will work.

- Abstracting away inessential details is very important in OO programming.
Key class

package interestRate;

import java.io.*;

/**
   Implements a Key used in the interest rate lattice.
*/
public class Key implements KeyInterface {

    //time
    public int t;

    //Number of up-ticks from the root to that node
    public int up_ticks;

    //short rate at that node
    public double short_rate;

    //option value
    public double option_value;

    //constructor
    public Key(int t, int u) {
        this.t = t;
        up_ticks = u;
    }//end of constructor

    public void print() {
        System.out.print("time: "); System.out.print(t);
    }
}
System.out.print(" up_ticks: "); System.out.print(up_ticks);
System.out.print(" short_rate: "); System.out.print(short_rate);
System.out.print(" option_value: "); System.out.print(option_value);
System.out.println();
});//end of print

public boolean isEqual(KeyInterface key) {
    Key k = (Key)key;
    if (k.t == t && k.up_ticks == up_ticks) return(true);

    return(false);
}//end of isEqual

}//end of class Key
Key (Contd)

• **Key** will hold the data for our interest rate model.

• Notice that **Key** implements the interface **KeyInterface**.

• Notice that **Key** provide the methods **isEqual** and **print**.
Testing Linked-list

```java
package testPrograms;

import interestRate.*;

public class testLinkList {
    static public void main(String argv[]) throws LinkListException {
        LinkList list = new LinkList();
        list.print();

        Key key_0 = new Key(0, 0);
        list.insert(key_0);

        Key key_u = new Key(1, 1);
        list.insert(key_u);

        Key key_d = new Key(1, -1);
        list.insert(key_d);

        Key key_uu = new Key(2, 2);
        Key key_ud = new Key(2, 0);
        Key key_du = new Key(2, 0);
        Key key_dd = new Key(2, -2);

        list.delete(key_du);
        list.insert(key_uu);
        list.insert(key_ud);
        list.insert(key_du);
    }
```
list.Insert(key_dd);
list.print();
list.Delete(key_du);
list.print();
}
// end of main method
}
Testing Linked-List

- JAVA program always call the main method first.

- Notice we are trying to print an empty list.

- Here is what we get:

  interestRate.LinkListException: Printing an empty list
  at java/lang/Throwable.<init>(line unknown, pc 59c58)
  at java/lang/Exception.<init>(line unknown, pc 52dac)
  at interestRate/LinkListException.<init>(11)
  at interestRate/LinkList.print(72)
  at interestRate/testLinkList.main(9)

- What happened?
Exception Handling

• print method of LinkedList object threw an LinkedListException.

• main never did anything.

• It got up to the JVM.

• JVM printed it.
Exception Handling

• Exceptions keep going up the call-stack until somebody catches it.

• If nobody catches it, JVM prints the exception on the screen and terminates the program.

• How to catch an exception?
package testPrograms;

import interestRate.*;

public class testLinkList1 {
    static public void main(String argv[]){
        try {
            LinkedList list = new LinkedList();
            list.print();

            Key key_0 = new Key(0,0);
            list.Insert(key_0);

            Key key_u = new Key(1,1);
            list.Insert(key_u);

            Key key_d = new Key(1,-1);
            list.Insert(key_d);

            Key key_uu = new Key(2,2);
            Key key_ud = new Key(2,0);
            Key key_du = new Key(2,0);
            Key key_dd = new Key(2,-2);

            list.Delete(key_du);
            list.Insert(key_uu);
            list.Insert(key_ud);
        }
    }
}
list.Insert(key_du);
list.Insert(key_dd);

list.print();
list.Delete(key_du);
list.print();

}  // end of try
catch (LinkListException e) {
System.out.println("Caught LinkListException: "+e.getMessage());
}   
finally {
System.out.println("Executing Finally section ");
}

}  //end of main

}  //end of class testLinkList-1
Catching Exceptions

- **try** encloses the actual block of code.

- **catch** gets executed if that exception is raised while executing the block of code.

- **finally** gets executed at the end (even if an exception is caught).
Catching Exceptions (Contd)

- What happens when we run the program.

Caught LinkedListException: Printing an empty list
Executing Finally section
Newton Raphson Method

- Assume we have $n$ functions $f_1, \cdots, f_n$ of $n$ variables $x_1, \cdots, x_n$.

- Objective is to find a vector $\vec{x}$ such that $f_i(\vec{x}) = 0$ for all $i$.

- Jacobian $J(x_1, \cdots, x_n)$ is a $n \times n$ matrix given by the following equation:

$$
\begin{pmatrix}
\frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\
\vdots & \ddots & \vdots \\
\frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_n}
\end{pmatrix}
$$
Newton Raphson (Contd)

• Start with initial vector $\vec{x}_0$.

• Update the vectors using the following equation:

$$\vec{x} = \vec{x} - J^{-1}(\vec{x}) \cdot f(\vec{x})$$

• Keep iterating until desired accuracy is achieved.
Abstract Function Object

package mathUtil;

/**
 * Abstract Function class used by Newton Raphson
 */
public abstract class AbstractFunctionObject {

    // no of variables for each function
    int n;

    private static final boolean DEBUG = false;

    /**
     * Constructor takes number of functions.
     */
    public AbstractFunctionObject(int n) {
        this.n = n;
    } // end of constructor

    /**
     * evaluate the i-th function
     */
    public abstract double evaluate(int i, double val[]);

    /**
     * evaluate the j-th derivative for the i-th function
     */
    public double derivative(int i, int j, double val[], double delta) {
        double new_val[] = new double[val.length];
        System.arraycopy(val, 0, new_val, 0, val.length);

        if (DEBUG) {
            System.out.println("i and j ");
        }
    }
}
System.out.print(i);
System.out.print(" ");
System.out.println(j);
}

// perturb the j-th entry by delta
new_val[j]=new_val[j]+delta;

if (DEBUG) {
    System.out.print("New and old values ");
    for(int k=0; k < n; k++) {
        System.out.print("|");
        System.out.print(val[k]);
        System.out.print("");
        System.out.println(new_val[k]);
    }
}

double eval1 = evaluate(i,new_val);
double eval2 = evaluate(i,val);

if (DEBUG) {
    System.out.print("Eval1 Eval2 ");
    System.out.print(eval1);
    System.out.print(" ");
    System.out.println(eval2);
}

double diff = eval1-eval2;

if (DEBUG) {
    System.out.print("Diff/Delta/return value ");
    System.out.print(diff);
    System.out.print(" ");
    System.out.print(delta);
    System.out.print(" ");
}
System.out.println(diff/delta);
}

return(diff/delta);
}//end of derivative

}//end of AbstractFunctionObject
Abstract Class

• An abstract class cannot be instantiated.

• Notice evaluate method is abstract.

• This means the class extending the AbstractFunctionObject has to provide an implementation for this method.

• Class extending the class inherits the derivative method.
package mathUtil;

/** Implements a Newton Raphson Solver
 * @author Somesh Jha
 */
public class NewtonRaphson {
    int noOfVars;
    AbstractFunctionObject funcObj;
    private static boolean DEBUG=false;
    private static double DELTA = 0.00001;
    private static double EPSILON=0.00001;
    private static double MAX=10;

    /** Solver takes argument as a function object. */
    public NewtonRaphson(AbstractFunctionObject funcObj) {
        noOfVars = funcObj.n;
        this.funcObj = funcObj;
    }

    private double[] evaluate(double values[]) {
        double functionValues[] = new double[noOfVars];
        for(int i=0; i < noOfVars; i++)
            functionValues[i] = funcObj.evaluate(i,values);
        return(functionValues);
    }

    private double norm(double functionVals[]) {
        // Implement the norm function here
    }
}
double returnVal = 0.0;
for(int i=0; i < noOfVars; i++)
    returnVal += functionVals[i]*functionVals[i];

return(returnVal);
} //end of norm

private Matrix getJacobian(double values[]) {
    double input[][] = new double[noOfVars][noOfVars];

    for(int i=0; i < noOfVars; i++)
        for(int j=0; j < noOfVars; j++)
            input[i][j] = funcObj.derivative(i,j,values,DELTA);

    return(new Matrix(noOfVars,input));
} //end of getJacobian

/*** Provided the initial seed solve the
 system of equations. */
public double[] solve(double initialVal[]) {
    double val[]=initialVal;

    double functionVal[] = evaluate(val);
    if (DEBUG) {
        System.out.print("NewtonRaphson:solve ");
        for(int i=0; i < noOfVars; i++)
            System.out.println(functionVal[i]);
    }

    int counter=1;
    while (norm(functionVal) > EPSILON) {
        Matrix jacobian = getJacobian(val);
        if (DEBUG) jacobian.print();

        Matrix inverseJacobian = jacobian.invertMatrix();
        if (DEBUG) inverseJacobian.print();

        if (DEBUG) {

Matrix tempMatrix = jacobian.multiplyLeft(inverseJacobian);
    tempMatrix.print();
}

double newVal[] = inverseJacobian.multiplyVector(functionVal);
    for(int i=0; i < noOfVars; i++)
    val[i] = val[i] - newVal[i];
    functionVal = evaluate(val);

    if (DEBUG) {
        System.out.print("NewtonRaphson:solve ");
        for(int i=0; i < noOfVars; i++)
            System.out.println(functionVal[i]);
    }

    counter++;
    if (counter > MAX) break;
}

return(val);
}//end of solve

}//end of class NewtonRaphson
NewtonRaphson class

• Notice that this class works with AbstractFunctionObject so can work for arbitrary system of equations.

• Abstraction again.
package testPrograms;

import mathUtil.AbstractFunctionObject;

public class SimpleFunction extends AbstractFunctionObject {

    public SimpleFunction() {
        //Only have two functions
        super(2);
    }

    public double evaluate(int i, double val[]) {
        switch (i) {
            case 0:
                return (val[0]*val[0]-4);
            case 1:
                return (val[0]*val[1] - 6);
            default:
                System.err.println("SimpleFunction: evaluate <Bad argument>");
        } //end of switch

        return(-1);

    } //end of evaluate

} //end of SimpleFunction
Actual Function

- Notice that SimpleFunction extends AbstractFunctionObject.

- Provides implementation of evaluate.

- SimpleFunction is in package testPrograms.

- AbstractFunctionObject is in package mathUtil.

- How do they find each other?
Packages revisited

- One needs to tell JVM where to find missing classes.

- Consider the statement given below:

  ```java
  import mathUtil.AbstractFunctionObject;
  ```

- The statement given above says that `import` the class `AbstractFunctionObject` from the package `mathUtil`.

- How do we actually find it?
package testPrograms;

import mathUtil.*;

public class testNewtonRaphson {

    static public void main(String argv[]) {

        AbstractFunctionObject funObj = new SimpleFunction();

        NewtonRaphson solver = new NewtonRaphson(funObj);

        double initialVal[] = new double[2];

        initialVal[0] = 1.0;
        initialVal[1] = 1.5;

        double result[] = solver.solve(initialVal);

        System.out.print("Value of x: ");
        System.out.println(result[0]);
        System.out.print("Value of y: ");
        System.out.println(result[1]);

    } // end of main

}// end of testNewtonRaphson
Testing the Newton Raphson

- We pass the SimpleFunction object to instantiate the solver.

- `solver.solve` actually solves it.

- Output.

  Value of $x$: 2
  Value of $y$: 3