Functional Abstraction (Chap 4)

- **Structured Programming**
  - The flow of control should be as simple as possible (Chapter 3).
  - The construction of a program should embody top-down design (Chapter 4).
- Top-down design (aka stepwise refinement) consists of repeatedly decomposing the problem into smaller problems.

```
// Message.java: Simple method use
class Message {
  public static void main(String[] args) {
    System.out.println("HELLO DEBRA!");
    printMessage(); //method call
    System.out.println("Goodbye.");
  }
  //definition of method printMessage
  static void printMessage() {
    System.out.println("A message for you:  ");
    System.out.println("Have a nice day!\n");
  }
}
```

Method call-return

The system calls main().
    main() calls System.out.println().
        "HELLO DEBRA" is printed
        System.out.println() returns.
    Main() calls printMessage().
        printMessage() calls System.out.println().
            "A message for you: " is printed.
            System.out.println() returns.
        printMessage() calls System.out.println().
            "Have a nice day!\n" is printed.
            System.out.println() returns.
        printMessage() returns.
    Main() calls System.out.println().
        "Goodbye." is printed.
        System.out.println() returns.
    Main() returns to the system.
The program ends.

Defining Simple Methods

`public static` *ReturnType Identifier* **(** *ParameterList* **) {**
  Body
}

- *ReturnType* is the type of value returned from the method/function.
- *Identifier* is the name of the method/function.
- *ParameterList* is a list of variables that will be used to pass information into the method. These are called the *formal parameters*.
- *Body* is a list of statements and declarations describing the action performed by this method.

```
// Message2.java: method parameter use
class Message2 {
  public static void main(String[] args) {
    System.out.println("HELLO DEBRA!");
    printMessage(5); //actual argument is 5
    System.out.println("Goodbye.");
  }
  static void printMessage(int howManyTimes) {
  //formal parameter  is howManyTimes
    System.out.println("A message for you:  ");
      for (int i = 0; i < howManyTimes; i++)
        System.out.println(
                       "Have a nice day!\n");
  }
}
```

Don't forget `static`

Until chapter 6, all of the methods you write should have the qualifier **`static`**. If you leave it off you will get a message like:

`Can't make static reference to method` *returnType methodName*`(…) in class` *YourClass*`.`

```
// Min2.java: return expression in a method

class Min2 {
  public static void main(String[] args) {
    int   j = 78, k = 3 * 30, m;
    System.out.println(
          "Minimum of two integers Test:");
    m = min(j, k);
    System.out.println("The minimum of : "
       + j + " , " + k + " is  " + m);
  }
  static int min(int a, int b) {
    if (a < b)
      return a;
    else
      return b;
  }
}
```

More about return

Control returns *immediately* from a method when a return is executed.

```
static int min(int a, int b) {
  if (a < b)
    return a;
  else if (b < a)
    return b;
  System.out.println("they are equal!!!");
  return a;
}
```

Watch for repeating lines.

If you find you have typed the same sequence of 3 or more lines, more than once, then you should consider creating a method to accomplish the task of those repeated lines.

Occasionally, even a one or two line method can add to program readability.

```
// Min2Bad.java - doesn't work because of scope

class Min2Bad {
  public static void main(String[] args) {
    int   j = 78, k = 3 * 30, m;
    System.out.println(
         "Minimum of two integers Test:");
    m = min();
    System.out.println("The minimum of : "
         + j + " , " + k + " is  " + m);
  }
  static int min() {
    if (j < k)
      return j;
    else
      return k;
  }
}
```

```
public class SquareRoots2 {// contains scope errors

  public static void main(String[] args) {
    int i = 99;
    double squareRoot = Math.sqrt(i);

    System.out.println("the square root of " + i +
                       " is " + squareRoot);

    for (int i = 1; i <= 10; i++) {
      double squareRoot = Math.sqrt(i);
      double square = squareRoot * squareRoot;
      System.out.println("the square root of " + i +
                       " is " + squareRoot);
      System.out.println("squaring that yields " +
                        square);
    }
    System.out.println("The final value of square"
                     + " is " + square);
  }
}
```

Top-down Design with methods

Problem: Analyze some company data represented by a series of integers. As you read each integer, you want to

- print the sum of integers read so far,
- print the count of integers read,
- print the minimum integer read up to this point,
- and print the maximum up to this point.

Also, print a nice heading for the data.

```
//RunSums.java: top level, main(), calls methods
//    to handle subproblems
import tio.*;
class RunSums {
  public static void main(String[] args) {
    printBanner();
    printHeadings();
    readAndPrintData();
  }
  // printBanner, printHeadings and
  // readAndPrintData definitions will go here

}
```

```
static void printBanner() {
   System.out.println("\n" +
     "*******************************************\n" +
     "*   RUNNING SUMS, MINIMUMS, AND MAXIMUMS   *\n" +
     "*******************************************\n");
 }

 static void printHeadings() {
 System.out.println(
            "Count\tItem\tSum\tMinimum\tMaximum");
 }
```

```
static void readAndPrintData() {
  int cnt = 0, sum = 0, item, smallest, biggest;

  item = Console.in.readInt();
  smallest = biggest = item;
  while (item != -99999) {
    cnt++;
    sum = sum + item;
    smallest = min(item, smallest);
    biggest = max(item, biggest);
    System.out.println( cnt + "\t" +  item + "\t"
          + sum + "\t" + smallest + "\t" + biggest);
    item = Console.in.readInt();
  }
}
```

```
static int min(int a, int b) {
    if (a < b)
      return a;
    else
      return b;
  }
  static int max(int a, int b) {
    if (a > b)
      return a;
    else
      return b;
  }
```

```
os-prompt>java RunSums < data

********************************************
*   RUNNING SUMS, MINIMUMS, AND MAXIMUMS   *
********************************************

Count   Item    Sum     Minimum Maximum
1       19      19      19      19
2       23      42      19      23
3       -7      35      -7      23
4       29      64      -7      29
5       -11     53      -11     29
6       17      70      -11     29
```

Simulating a Coin Toss

Problem: What is the probability that you can toss some number, n, heads in a row?

Pseudocode:

```
Input the number of heads in a row for a trial.
Input the number of trials.
Perform the specified number of trials.
Print the result.
```

Pseudocode for performing the n trials:

```
initialize the number of successes to 0
while there are more trials to run
  run one trial
  if the trial was a success
    increment the number of successes
end while loop
return the number of successful trials
```

Pseudocode for performing one trial:

```
let numTosses be the number of tosses for
                                     a successful trial
initialize the number of heads tossed to zero
while number of heads tossed is less than numTosses
  toss the coin
  if the coin comes up tails
    return failure
  increment the number of heads tossed
end while loop
return success
```

```
class CoinToss {
  public static void main(String[] args) {
    //Input the number of tosses in a row to try for.
    int numTosses = 4;      //Just use 4 for testing

    //Input the number of trials to run.
    int numTrials = 10000;  //Use 10000 for testing

    //Perform the specified number of trials
    int numSuccesses =
      performTrials(numTosses,numTrials);

    //Print the results
    double probability =
      numSuccesses / (double)numTrials;
    System.out.println("Probability found in "
           + numTrials + " is  " + probability);
  }
```

```
  // perform numTrials simulated coin tosses
  // and return the number of successes
  static int performTrials(int numTosses,
                           int numTrials) {
    System.out.println("Monte Carlo " + numTosses +
                       " in a row heads");
    int numSuccesses = 0;
    for (int trials= 0 ; trials < numTrials; trials++)
      // perform one trial
      if ( isAllHeads(numTosses))
        numSuccesses++;         // trial was a success
    return numSuccesses;
  }
```

```
  // Perform one trial.
  // return true if numTosses heads are tossed
  // before a tail
  static boolean isAllHeads(int numTosses) {
    double outcome;

    for (int numHeads = 0; numHeads < numTosses;
                           numHeads++) {
      outcome = Math.random();   // toss the coin
      if ( outcome < 0.5)
        return false;            // tossed a tail
    }
    return true;                 // tossed all heads
   }
```

```
//FailedSwap.java - Call-By-Value test
class FailedSwap {
  public static void main(String[] args) {
    int numOne = 1, numTwo = 2;

    swap(numOne, numTwo);
    System.out.println("numOne = " + numOne);
    System.out.println("numTwo = " + numTwo);
  }
```

```
  static void swap(int x, int y) {
    int temp;

    System.out.println("x = " + x);
    System.out.println("y = " + y);
    temp = x;
    x = y;
    y = temp;
    System.out.println("x = " + x);
    System.out.println("y = " + y);
  }
}
```

The Software Life Cycle

1. Requirements analysis and definition.
2. Design.
3. Implementation.
4. Testing.
5. Maintenance

A computer game: Requirements

Twenty-one pickup is a two-player game that starts with a pile of 21 stones. Each player takes turns removing 1, 2, or 3 stones from the pile. The player that removes the last stone wins.

Refining the requirements

- What is the role of the computer?
  - The computer will be one of the players. The human will always go first.
- What will be the interface between the human and the computer?
  - A simple text interface at the console.
- Does the program play a sequence of games or just one?
  - Just play a single game.

Twenty-one pickup: Design

Top-level Pseudocode:

```
print instructions
create the initial pile with 21 stones
while the game is not over
     have the user move
     if the game is not over
          have the computer move
end while loop
print the outcome
```

Pseudocode for "Have the user move":

```
get the user move from the console
remove the stones from the pile
print the user's move on the console
```

Pseudocode for "Have the computer move":

```
compute the number of stones for the computer
                                    to remove
remove the stones from the pile
print the computer's move on the console
```

Pseudocode for "Get user move from console":

```
prompt the user for the user’s next move
from the console, read the number of stones
                                   to remove
while the number read is not a legal move
     prompt the user again
     read the number of stones to remove
end while loop
return the number of stones to remove
```

```
/**
 * playerMove completes one move by the player.
 * @param numberOfStones
 *     The number of stones reamining in the pile.
 * @return
 *    The number of stones remaining after the
 *    user’s move.
 */
static int playerMove(int numberOfStones)
```

```
/**
* computerMove completes one move by the computer.
* @param numberOfStones
*    The number of stones reamining in the pile.
* @return
*    The numberOfStones remaining after the
*    computer’s move.
*/
static int computerMove(int numberOfStones)
```

```
/**
 * getUserMove reads in the user’s move, only
 *    accepting legal inputs.
 * @param numberOfStones
 *    The number of stones reamining in the pile.
 * @return
 *    The number of stones selected for removal by
 *    the user.
 */
 static int getUserMove(int numberOfStones)
```

```
// TwentyOnePickup.java - compare with pseudocode
...
  public static void main(String[] args) {
    printInstructions();
    // create the initial pile with 21 stones
    int numberOfStones = 21;
    // keep track of who moved last
    boolean playerMovedLast = false;
    while (numberOfStones > 0) {
      numberOfStones = playerMove(numberOfStones);
      playerMovedLast = true;
      if (numberOfStones > 0){
        numberOfStones = computerMove(numberOfStones);
        playerMovedLast = false;
      }
    }
    // print the outcome
    if (playerMovedLast)
      System.out.println("Congratulations, you won.");
    else
      System.out.println("Better luck next time.");
  }
```

```
/**
* printInstructions prints the initial instructions
*/
static void printInstructions() {
  System.out.println(
    "The object of this game is to remove the last"
    + " stone.\n"
    + "There are 21 stones in the pile to start"
    + " with.\n"
    + "You may remove from 1 to 3 stones on each"
    + " move.\n"
    + "Good Luck!");
}
```

```
/**
* playerMove completes one move by the player.
* @param numberOfStones
*     The number of stones reamining in the pile.
* @return
*    The number of stones remaining after the
*    user’s move.
*/
static int playerMove(int numberOfStones) {
  int move = getUserMove(numberOfStones);

  numberOfStones = numberOfStones - move;
  System.out.println("There are " + numberOfStones
                   + " stones remaining.");
  return numberOfStones;
}
//Note we cannot change numberOfStones in main directly.
```

```
/**
* computerMove completes one move by the computer.
*/
static int computerMove(int numberOfStones) {
  int move;

  if (numberOfStones <= 3) {
    move = numberOfStones; /* remove the rest */
  }
  else {
    move = numberOfStones%4;
    if (move == 0) move = 1;
  }
  numberOfStones = numberOfStones - move;
  System.out.println("The computer removes " + move
      + " stones leaving " + numberOfStones + ".");
  return numberOfStones;
}
```

```
static int getUserMove(int numberOfStones) {
  System.out.println("Your move - how many stones"
                     + " do you wish to remove?");
  int move = Console.in.readInt();

  while (move > numberOfStones || move < 1 || move > 3) {
    if (numberOfStones >= 3)
      System.out.println("Sorry," +
          " you can only remove 1 to 3 stones.");
    else
      System.out.println("Sorry, you can only "
          + "remove 1 to " + numberOfStones
          + " stones.");
    System.out.println("How many stones"
                       + " do you wish to remove?");
    move = Console.in.readInt();
  }
  return move;
}
```

Twenty-one pickup: Testing

- Statement coverage - execute every statement at least once.
- Branch coverage - make sure every possible branch has been taken at least once.

Testing example

Correct:

```
while (move > numberOfStones || move < 1 || move > 3)
```

Incorrect:

```
while (move > numberOfStones || move < 0 || move > 3)
```

Statement coverage does not guarantee that this error is detected. We must have a test case with move equal to 0.

Twenty-one pickup: Test Strategy

- main(): one case wherein the user wins and one wherein the computer wins.
- computerMove(): computer wins, computer removes 1 stone because computed move was 0, and computer removes >1.
- playerMove(): no branches.
- getUserMove(): one erroneous value when > 3 stones remain, one erroneous value when < 3 stones remain, a value that is too big, and a value that is too small, end condition when < 3 stones remain but user enters 3.

Recursion

```
// Recur.java - recursive goodbye
public class Recur {
  public static void main(String[] args) {
    sayGoodBye(5);
  }
  static void sayGoodBye(int n) {
    if (n < 1) //base case
      System.out.println("########");
    else     {
      System.out.println("Say goodbye Gracie.");
      sayGoodBye(n - 1); //recursion
    }
  }
}
```

General Form of recursion

```
If (stopping condition)      // base case
    /* do whatever at the end */;
else {
    //execute recursive step
    RecursiveMethod(arguments);
}
```

Factorial: A classic recursion

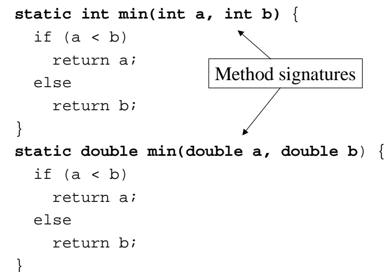
```
static long factorial(int n) {
    if (n <= 1)
        return 1;
    else
        return (n * factorial(n - 1));
}
```

Factorial:
x = factorial(4);

```
main() calls factorial(4).
  factorial(4) calls factorial(3).
    factorial(3) calls factorial(2).
      factorial(2) calls factorial(1).
        factorial(1) returns 1.
      factorial(2) returns 2.
    factorial(3) returns 6.
  factorial(4) returns 24.
  main() continues assigning 24 to x.
```

Method Overloading

```
static int min(int a, int b) {
  if (a < b)
    return a;
  else
    return b;
}
static double min(double a, double b) {
  if (a < b)
    return a;
  else
    return b;
}
```



An ambiguous overload.

```
//AmbiguousOverload.java: won't compile
class AmbiguousOverload {
  public static void main(String[] args) {
    int i = 1, j = 2;

    System.out.println(ambig(i,j));
  }
  static boolean ambig(float x, int y){
    return x < y;
  }
  static boolean ambig(int x, float y){
    return x < y;
  }
}
```

An ambiguous overload - corrected.

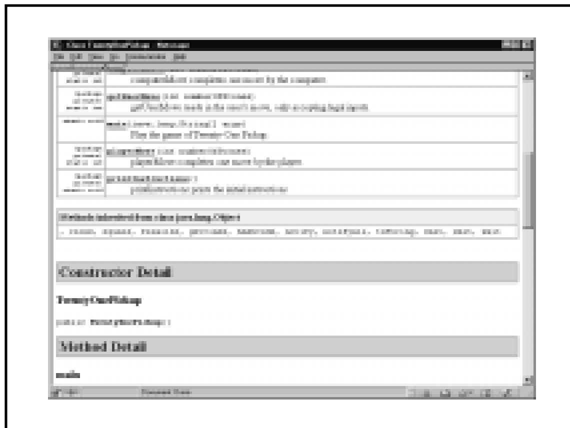
```
//The method definitions were ok, it was the call
// that was bad.
class AmbiguousOverload {
  public static void main(String[] args) {
    int i = 1, j = 2;

    System.out.println(ambig(i,(float)j));
  }
  static boolean ambig(float x, int y){
    return x < y;
  }
  static boolean ambig(int x, float y){
    return x < y;
  }
}
```

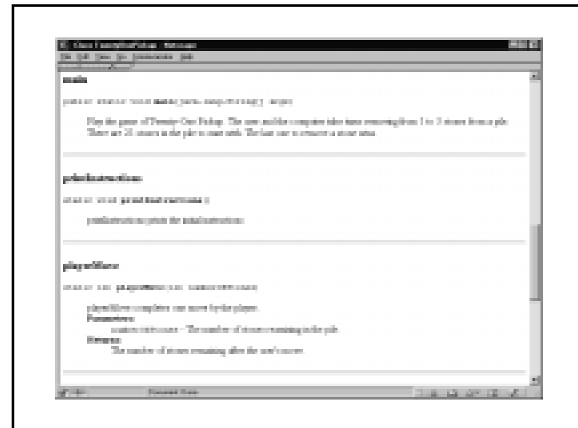
javadoc

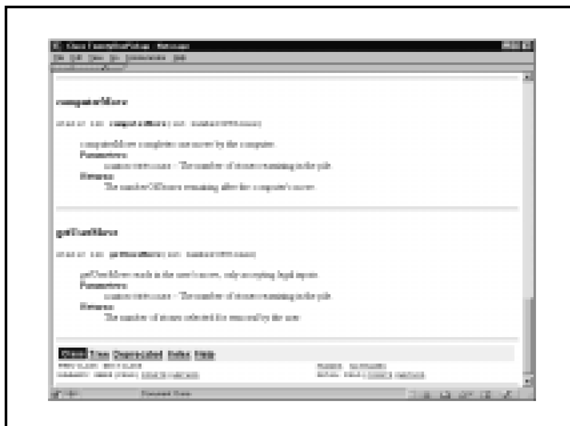
```
/**
  * playerMove completes one move by the player.
  * @param numberOfStones
  *      The number of stones reamining in the pile.
  * @return
  *     The number of stones remaining after the
  *     user's move.
  */
  static int playerMove(int numberOfStones) {
…
```

```
javadoc -package TwentyOnePickup.java
```

produces a number of html files.







A Final Example

- The following code is a solution to Chap 4 Exercise 17. I presented this example after starting the lectures on arrays and after students had submitted their solutions to the problem.

```
/*
 * Author: Charlie McDowell (charlie@cse)
 *
 * This program allows the user to play the game of
 * Craps. The rules of craps are as follows:
 * if the first roll is a 7 or a 11, you win immediately
 * if the first roll is a 2 or a 12, you lose
 * immediately otherwise, you go into "point mode" and
 * the first roll becomes the * point. Now, you keep
 * rolling the dice until you get the point again or a
 * 7. If you got the point, you win, if you get a 7,
 * you lose.
 *
 * The user bets some amount on each round of craps.
 * If she wins, she gets  additional money equal to the
 * bet. If she loses, she loses the money she bet.
 */
```

```
import tio.*;
import java.util.*;

class Craps {
  public static void main(String[] args) {
    int money = 100;

    System.out.println("Enter the seed.");
    int seed = Console.in.readInt();
    Random random = new Random(seed);

    int bet = getBet(money); // Get initial bet
```

```
    while (bet > 0 && money > 0) {
      boolean won = rollDiceUntilWinOrLose(random);
      if (won) {
        money = money + bet;
        System.out.println("You won, you now have " +
                            money);
      } else {
        money = money - bet;
        System.out.println("You lost, you now have " +
                            money);
      }
      if ( money > 0) {
        bet = getBet(money); // If user can bet, let her
      }
    }
  }
```

```
/*
 * This method plays one round of craps and tells us
 * whether the user won or lost.
 * input: random number generator (needed by rollDice)
 * output: won or lost
 */

  static boolean rollDiceUntilWinOrLose(Random random) {
    int point = rollDice(random);
    if ( point == 7 || point == 11 )
      return true;
    else if ( point == 2 || point == 12)
      return false;
    boolean winLose = rollUntilPointOr7(point, random);
    return winLose;
  }
```

```
  /*
   * This method continues the game in "point mode".
   * Here, the user already has a "point" and is looking
   * to match it by repeatedly rolling the dice. If she
   * can find the match before she rolls a 7, she wins,
   * otherwise, she loses.
   * input: point - the point we are looking for
   *         random - random number generator
   * output: won or lost in point mode
   */

  static boolean rollUntilPointOr7(int point,
                                   Random random)
```

```
  static boolean rollUntilPointOr7(int point,
                                   Random random)
  {
    int nextRoll = rollDice(random);
    while ( nextRoll != point && nextRoll != 7 ) {
      nextRoll = rollDice(random);
    }
    if ( nextRoll == 7 )
      return false;
    else
      return true;
  }
```

```
/*
 * Simulates rolling of dice by generating random
 * numbers from a random number generator.
 * input: random number generator to use.
 *         We need this because we want
 *         to use the *same* random number generator
 *         throughout and duplicate our results.
 * output: total after rolling the dice.
 */

static int rollDice(Random random) {
  int die1 = random.nextInt(6) + 1;
  int die2 = random.nextInt(6) + 1;
  System.out.println("roll is " + die1 + ", " + die2);
  return die1 + die2;
}
```

```
/* Returns the amount bet by the user. Performs input
 * checking to ensure that the user does not bet more
 * than she has and she does not bet a negative value.
 * input: amount user currently has. This is checked
 *        against the amount she wants to bet to make
 *        sure the bet is valid.
 * output: a valid bet
 */
static int getBet(int total) {
  System.out.println("Enter bet.");
  int bet = Console.in.readInt();
  while ( bet < 0 || bet > total) {
    System.out.println("Not an ok bet.");
    System.out.println("Enter bet.");
    bet = Console.in.readInt();
  }
  return bet;
}
```