**Computer Programming II (CS-141)**

**Note: This document is designed for the purpose of quick revision of the course. Don’t consider this as the final exam study material. This is the abstract for the core concepts of the course.**

**====== Chapter 9 ======**

**Interfaces vs. Classes**

An interface defines a *contract by specifying a set of method prototypes, but no implementation. The methods in an interface are all* implicitly abstract and public by virtue of their definition. A *method prototype has the same syntax as an abstract method*. However, only the modifiers abstract and public are allowed, but these are invariably omitted.

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

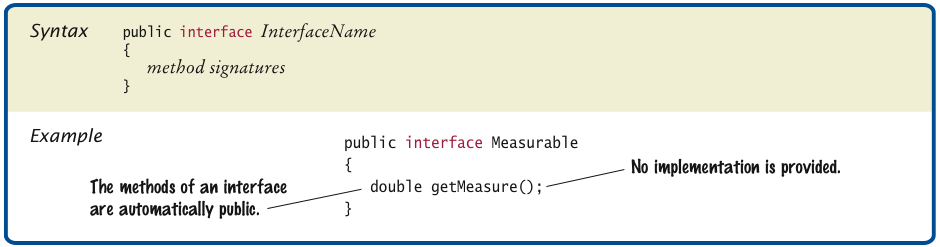
* + *All methods in an interface type are* ***abstract****; they don’t have an implementation*
  + *All methods in an interface type are automatically public*
  + *An interface type does not have instance fields*

**Declaring an Interface**

An interface type is similar to a class, but an interface does not provide any implementation of its methods and is, therefore, abstract by definition. This means that it cannot be instantiated, but classes can implement it by providing implementations for its method prototypes. Declaring an interface abstract is superfluous and seldom done.

The interface body can contain *member declarations which comprise*

* + *constant declarations*
  + *method prototype declarations*
  + *nested class and interface declarations*



Using **interface, you can specify a set of methods** which can be implemented by one or more classes.

An interface variable holds a reference to object of any class that implements the interface.

Interfaces can reduce the coupling between classes

UML notation:

* *Interfaces are tagged with a “stereotype” indicator «interface»*
* *A dotted arrow with a triangular tip denotes the “is-a” relationship between a class and an interface*
* *A dotted line with an open v-shaped arrow tip denotes the “uses” relationship or dependency*

You can convert from a class type to an interface type, provided the class implements the interface.

**Callback** is a mechanism for specifying code that is executed at a later time.

**Inner Classes**

Classes can be declared inside method or inside other classes. If inner class is declared inside an enclosing class, but outside its methods, it is available to all methods of enclosing class. Compiler turns an inner class into a regular class file.

An *inner class is a class that is defined inside another class. Why would you want to do that?*

* There are four reasons:
  + An object of an inner class can access the implementation of the object that created
  + it—including data that would otherwise be private.
  + Inner classes can be hidden from other classes in the same package.
  + *Anonymous inner classes are handy when you want to define callbacks on the fly.*
  + Inner classes are very convenient when you are writing event-driven programs.
* **Event listener:** 
  + *Notified when event happens*
  + *Belongs to a class that is provided by the application programmer*
  + *Its methods describe the actions to be taken when an event occurs*
  + *A program indicates which events it needs to receive by installing event listener objects*
* **Event source:** 
  + *User interface component that generates a particular event*
  + *Add an event listener object to the appropriate event source*
  + *When an event occurs, the event source notifies all event listeners*

A **mock object** provides the same services as another object, but in a simplified manner.

**====== Chapter 10 ======**

**Inheritance** (OOP) is when an object or class is based on another object (prototypal inheritance) or class (class-based inheritance), using the same implementation (inheriting from an object or class) specifying implementation to maintain the same behavior (realizing an interface; inheriting behavior).

Superclass: more general class

Subclass: more specialized class that inherits from the superclass.

A subclass has no access to private instance variables of its superclass

Inheritance is a mechanism for extending existing classes by adding instance variables and methods.

**Method overriding** allows a subclass or child class to provide a specific implementation ofa **method** that is already provided by one of its superclasses or parent classes.

**Method Overloading** allows a class to have two or more methods having same name, if their argument lists are different.

Use the super reserved word to call a method of the superclass.

To call the superclass constructor, use the super reserved word in the first statement of the subclass constructor.

When subclass constructor doesn't call superclass constructor, the superclass must have a constructor with no parameters. If, however, all constructors of the superclass require parameters, then the compiler reports an error.

It is OK to convert subclass reference to superclass reference.

Instanceof operator tests whether an object belongs to a particular type.

Dynamic method lookup: When the virtual machine calls an instance method, it locates the method of the implicit parameter's class.

Polymorphism: Ability to treat objects with differences in behavior in a uniform way.

Protected features can be accessed by all subclasses and by all classes in the same package. Protected data can be accessed by all methods of classes in the same package. It is best to leave all data private and provide accessor methods for the data.

All classes defined without an explicit extends clause automatically extend *Object*. Most useful methods:

String toString()

boolean equals(Object otherObject)

Object clone()

Good idea to override these methods in your classes.

**====== Chapter 12 ======**

**The Software Life Cycle**: Encompasses all activities from initial analysis until obsolescence.

Formal process for software development describes phases of the development process and gives guidelines for how to carry out the phases.

Development process consists of following five phases.

1. Analysis
2. Design
3. Implementation
4. Testing
5. Deployment

Analysis

* Decide what the project is supposed to do
* Do not think about how the program will accomplish tasks
* Output: Requirements document
  + *Describes what program will do once completed*
  + *User manual: Tells how user will operate program*
  + *Performance criteria*

Design

* Plan how to implement the system
* Discover structures that underlie problem to be solved
* Decide what classes and methods you need
* Output:
  + *Description of classes and methods*
  + *Diagrams showing the relationships among the classes*

Implementation

* Write and compile the code
* Code implements classes and methods discovered in the design phase
* Program Run: Completed program

Testing

* Run tests to verify the program works correctly
* Program Run: A report of the tests and their results

Deployment

* Users install program
* Users use program for its intended purpose

**The waterfall model:** This is the classic SDLC model, with a linear and sequential method that has goals for each development phase. The waterfall model simplifies task scheduling, because there are no iterative or overlapping steps. One drawback of the waterfall is that it does not allow for much revision. The best-known and oldest process is the waterfall model, where developers follow these steps in order. They state requirements, analyze them, design a solution approach, architect a software framework for that solution, develop code, test, deploy, and maintain. After each step is finished, the process proceeds to the next step.

**The spiral model:** This model of development combines the features of the prototyping model and the waterfall model. The spiral model is favored for large, expensive, and complicated projects.

**Extreme Programming**, XP, is the best-known agile process. In XP, the phases are carried out in extremely small (or "continuous") steps compared to the older, "batch" processes. The (intentionally incomplete) first pass through the steps might take a day or a week, rather than the months or years of each complete step in the Waterfall model. First, one writes automated tests, to provide concrete goals for development. Next is coding (by a pair of programmers), which is complete when all the tests pass, and the programmers can't think of any more tests that are needed. Design and architecture emerge out of refactoring, and come after coding. Design is done by the same people who do the coding. The incomplete but functional system is deployed or demonstrated for the users (at least one of which is on the development team). At this point, the practitioners start again on writing tests for the next most important part of the system.

**Discovering Classes**

A class represents some useful concept

Concrete entities: Bank accounts, ellipses, and products

Abstract concepts: Streams and windows

Find classes by looking for nouns in the task description

Define the behavior for each class

Find methods by looking for verbs in the task description

**CRC Card:** Describes a class, its responsibilities, and its collaborators

Use an index card for each class

Pick the class that should be responsible for each method (verb)

Write the responsibility onto the class card

**Relationships Between Classes**

Inheritance

* *Is-a* relationship
* Relationship between a more general class (superclass) and a more specialized class (subclass)
* Every savings account is a bank account
* Every circle is an ellipse (with equal width and height)
* It is sometimes abused
  + *Should the class Tire be a subclass of a class Circle?* 
    - *The has-a relationship would be more appropriate*

Aggregation

* *Has-a* relationship
* Objects of one class contain references to objects of another class
* Use an instance variable
  + *A tire has a circle as its boundary:*

class Tire   
{   
 ...   
 private String rating;   
 private Circle boundary;   
}

* Every car has a tire (in fact, it has four)

Dependency

* *Uses* relationship
* Example: Many of our applications depend on the Scanner class to read input
* Aggregation is a stronger form of dependency
* Use aggregation to remember another object between method calls

Association

A class is associated with another if you can navigate from objects of one class to objects of the other

* Association: More general relationship between classes
* Use early in the design phase

**====== Chapter 13 ======**

**RECURSION**

* A recursive computation solves a problem by using the solution of the same problem with simpler values
* For recursion to terminate, there must be special cases for the simplest inputs
* To complete our Triangle example, we must handle width <= 0:

if (width <= 0)  return 0;

* Two key requirements for recursion success:
  + *Every recursive call must simplify the computation in some way*
  + *There must be special cases to handle the simplest computations directly*

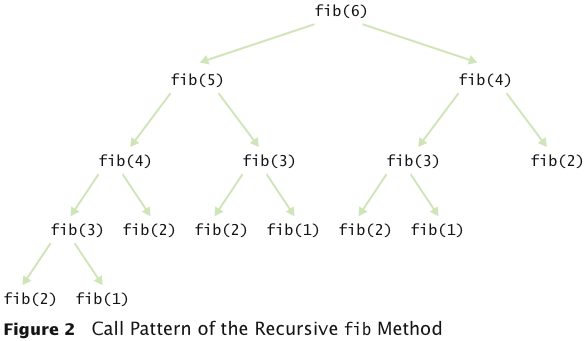
**FIBONACCI SERIES**

* Fibonacci sequence is a sequence of numbers defined by

*f*1 = 1  
*f*2 = 1  
*fn* = *fn*-1 + *fn*-2

* First ten terms:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55



* Occasionally, a recursive solution runs much slower than its iterative counterpart
* In most cases, the recursive solution is only slightly slower
* The iterative isPalindrome performs only slightly better than recursive solution
  + *Each recursive method call takes a certain amount of processor time*
* Smart compilers can avoid recursive method calls if they follow simple patterns
* Most compilers don’t do that
* In many cases, a recursive solution is easier to understand and implement correctly than an iterative solution
* “To iterate is human, to recurse divine.” L. Peter Deutsch

**PERMUTATIONS**

* Design a class that will list all permutations of a string
* A permutation is a rearrangement of the letters
* The string "eat" has six permutations:

"eat"   
"eta"

"ate"   
"aet"   
"tea"   
"tae"

* Generate all permutations that start with 'e', then 'a', then 't'
* To generate permutations starting with 'e', we need to find all permutations of "at"
* This is the same problem with simpler inputs
* Use recursion
* getPermutations: Loop through all positions in the word to be permuted
* For each position, compute the shorter word obtained by removing *i*th letter:

String shorterWord = word.substring(0, i) + word.substring(i + 1);

* Construct a permutation generator to get permutations of the shorter word:

PermutationGenerator shorterPermutationGenerator   
 = new PermutationGenerator(shorterWord); ArrayList<String> shorterWordPermutations = shorterPermutationGenerator.getPermutations();

* Finally, add the removed letter to front of all permutations of the shorter word:

for (String s : shorterWordPermutations)   
{   
 result.add(word.charAt(i) + s);   
}

* Special case: Simplest possible string is the empty string; single permutation, itself

Question: What are all permutations of the four-letter word beat?

**Answer:** They are b followed by the six permutations of eat, e followed by the six permutations of bat, a followed by the six permutations of bet, and t followed by the six permutations of bea.

**MUTUAL RECURSION**

To see the mutual recursion clearly, trace through the expression (3+4)\*5:

* + getExpressionValue calls getTermValue
    - getTermValue calls getFactorValue
      * getFactorValue consumes the ( input
      * getFactorValue calls getExpressionValue
        + getExpressionValue returns eventually with the value of 7, having consumed 3 + 4. This is the recursive call.
      * getFactorValue consumes the )input
      * getFactorValue returns 7
    - getTermValue consumes the inputs \* and 5 and returns 35
  + getExpressionValue returns 35

====== CHAPTER 14 ======

**SELECTION SORT**

* Sorts an array by repeatedly finding the smallest element of   
  the unsorted tail region and moving it to the front
* Slow when run on large data sets
* Example: sorting an array of integers
* Doubling the size of the array more than doubles the time needed to sort it

**PERFORMANCE OF SELECTION SORT**

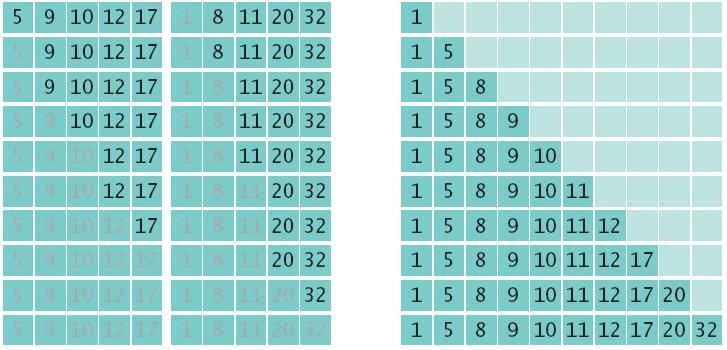
* The number of visits is of the order *n2*
* Using big-Oh notation: The number of visits is O(*n2*)
* Multiplying the number of elements in an array by **2** multiplies the processing time by **4**
* Big-Oh notation “*f*(*n*) = *O*(*g*(*n*))”   
  expresses that *f* grows no faster than *g*
* To convert to big-Oh notation: Locate fastest-growing term, and ignore constant coefficient

**QUESTION:** If you increase the size of a data set tenfold, how much longer does it take to sort it with the selection sort algorithm?

**Answer:** It takes about 100 times longer.

**MERGE SORT**

* Sorts an array by
  + *Cutting the array in half*
  + *Recursively sorting each half*
  + *Merging the sorted halves*
* Dramatically faster than the selection sort
* Merge the two sorted arrays into a single sorted array



**Question:** Manually run the merge sort algorithm on the array 8 7 6 5 4 3 2 1.

**Answer:**

First sort 8 7 6 5.

Recursively, first sort 8 7.

Recursively, first sort 8. It’s sorted.

Sort 7. It’s sorted.

Merge them: 7 8.

Do the same with 6 5 to get 5 6.

Merge them to 5 6 7 8.

Do the same with 4 3 2 1: Sort 4 3 by sorting 4 and 3 and merging them to 3 4.

Sort 2 1 by sorting 2 and 1 and merging them to 1 2.

Merge 3 4 and 1 2 to 1 2 3 4.

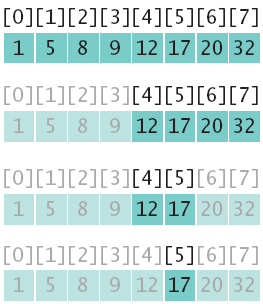
Finally, merge 5 6 7 8 and 1 2 3 4 to 1 2 3 4 5 6 7 8.

**SEARCHING**

* **Linear search: also called sequential search**
* **Examines all values in an array until it finds a match or reaches the end**
* **Number of visits for a linear search of an array of *n* elements:** 
  + ***The average search visits n/2 elements***
  + ***The maximum visits is n***
* **A linear search locates a value in an array in O(*n*) steps**

**BINARY SEARCH**

* **Locates a value in a sorted array by** 
  + ***Determining whether the value occurs in the first or second half***
  + ***Then repeating the search in one of the halves***

****

**Question: Suppose you need to look through a sorted array with 1,000,000 elements to find a value. Using the binary search algorithm, how many records do you expect to search before finding the value?**

**Answer: You would search about 20. (The binary log of 1,024 is 10.)**

**Question: What is Recursion?**

**Answer:**     **Recursion:** A problem solving / programming technique in which a method (function) can call itself in order to solve the problem. In the process, the problem is solved by reducing it to smaller versions of itself.

General format for recursive functions:

if(some easily-solved problem) // base case   
    solution statement   
else                        // general case   
    recursive function call

**Question:** How many types of recursion?

**Answer:** There are two types of recursion:

1. Direct recursion

2. Indirect recursion.

**Direct recursion:** When in the body of a method there is a call to the same method, we say that the method is **directly recursive**.

**Indirect recursion**: If method A calls method B, method B calls method C, and method C calls method A we call the methods A, B and C **indirectly recursive** or **mutually recursive**.

**Question: What are the criteria to select recursion or iterative method?**

**Answer:** There are no absolute rules when choosing between a recursive or an iterative solution. The most powerful benefit of recursive methods is the fact that they are concise, which makes them easier to maintain and read. On the other hand, recursive methods consume time and computer storage, which means that they may not be very efficient. These are some guidelines when considering the alternatives:

1.    Design a recursive method if the problem is stated recursively and the recursive algorithm is less complex. Keep in mind that in many cases, recursion is a technique that reduces the complexity of the algorithms you want to implement.

2.   Design an iterative method if similar complexities for the recursive and the iterative algorithms (the iterative solution is likely to be more efficient)

**Question:** Why is not it easy to develop an iterative solution for the permutation generator?

**Answer:** An iterative solution would have a loop whose body computes the next permutation from the previous ones. But there is no obvious mechanism for getting the next permutation. For example, if you already found permutations eat, eta, and aet, it is not clear how you use that information to get the next permutation. Actually, there is an ingenious mechanism for doing just that, but it is far from obvious.

**Question:** What are all permutations of the four letter word ‘beat’?

**Answer:** They are b followed by the six permutations of eat, e followed by the six permutations of bat, a followed by the six permutations of bet and t followed by the six permutations of bea.

**Question:** Given the following method declaration, what will redo(82, 3) return?

public int redo(int i, int j)

{

if (i==0)

return 0;

else

return redo(i/j, j)+1;

}

**Answer :** output is 5

**Question:** Consider the recursive method myPrint. What is printed for the call myPrint(821)?

public void myPrint(int n)

{

if (n < 10)

System.out.print(n);

else

{

int m = n % 10;

System.out.print(m);

myPrint(n/10);

}

}

**Answer:** output=128

**Question:** Write a program in Java to find xy recursively and conditions are as follows:

         {1                  if y=0

power(x, y) =   {x             if y=1

         {x + power(x, y-1) if y>1

                        1 if y<0,

power(x, y) =     \_\_\_\_\_\_\_\_\_\_\_\_\_

                    power(x, -y)

**Answer:** code:

   public class Power

{

    public static double power(double base, double basePow)

    {         if(basePow==0)

         {  return 1;

         }

 else if(basePow==1)

         {

           return base;

        }

         else if(basePow>1)

         {

            return base\*power(base,basePow-1);

         }

        else

         {

              return 1/power(base, -1 \* basePow);

        }

 }

 }

//main class

import java.util.Scanner;

public class Main {

    public static void main(String[] args)

    {

      Scanner input = new Scanner(System.in);

       System.out.print("Enter the base number: ");

      int base = input.nextInt();

       System.out.print("Enter the base power: ");

      int basePow = input.nextInt();

      Power access = new Power();

       System.out.print(base + " to the power of " + basePow + " is: " + access.power(base, basePow));

    }

 }

**Question:** Write a recursive method to find n factorial (n! = n \* (n-1) \* (n-2)... 3 \* 2 \* 1).

 The method call factorial(5) should have value 120,

because that is 5 \* 4 \* 3 \* 2 \* 1 .

**Answer:** Code

   public class CalFactorial

{

public static int factorial(int n){   
    if (n == 1)  // base case   
       return  1;   
    else     // general case   
       return n \* factorial(n - 1);    
}

}

 import java.util.Scanner;

public class CalFactorialTest {

    public static void main(String[] args)

    {

CalFactorial f=new CalFactorial();

Scanner input = new Scanner(System.in);

      System.out.print("Enter the number: ");

      int num = input.nextInt();

      int fact= f.factorial(num);

      System.out.println(“Factorial value of the”+num+”=”+fact);

     }

}

**Question: If you increase the size of a data set tenfold, how much longer does it take to sort it with the selection sort algorithm?**

Answer: It takes about 100 times longer. O (n2)

**Question: Manually run the merge sort algorithm on the array 8 7 6 5 4 3 2 1.**

Answer:

First sort 8 7 6 5.

Recursively, first sort 8 7.

Recursively, first sort 8. It’s sorted.

Sort 7. It’s sorted.

Merge them: 7 8.

Do the same with 6 5 to get 5 6.

Merge them to 5 6 7 8.

Do the same with 4 3 2 1: Sort 4 3 by sorting 4 and 3 and merging them to 3 4.

Sort 2 1 by sorting 2 and 1 and merging them to 1 2.

Merge 3 4 and 1 2 to 1 2 3 4.

Finally, merge 5 6 7 8 and 1 2 3 4 to 1 2 3 4 5 6 7 8.

**Question: Given the timing data for the merge sort algorithm in the table, how long would it take to sort an array of 100,000 values?**

|  |  |
| --- | --- |
| ***n*** | **Merge Sort (milliseconds)** |
| 10,000 | 40 |
| 20,000 | 73 |
| 30,000 | 134 |
| 40,000 | 170 |
| 50,000 | 192 |
| 60,000 | 205 |

**Answer:** Approximately 100,000 × log(100,000) / 50,000 × log(50,000) = 2 × 5 / 4.7 = 2.13 times the time required for 50,000 values. That’s 2.13 × 97 milliseconds or approximately 207 milliseconds.

**Question: Suppose you need to look through a sorted array with 1,000,000 elements to find a value. Using the binary search algorithm, how many records do you expect to search before finding the value?**

Answer: You would search about 20. (The binary log of 1,024 is 10.)

**Question: Why can’t you use a “for each” loop for (int element : a) in the search method?**

Answer: The search method returns the index at which the match occurs, not the data stored at that location.

**Question:** Find the error in the following code:

A)

int[] a = {1,2,3,4,6,7,7,3};

public void int method1(int v)

{

foreach (int i == 0, i < a.length; i++)

{

if (a[i] = v)

return i;

}

return -1;

}

**B)**

public **C**lass Demo{

public static void main(**s**tring[10] args){

Scanner in = new Scanner(**s**ystem.in);

boolean done == false;

while (!done) {

System.out.print("Enter -1 to quit: ");

int n = in.next**i**nt();

if (n == -1)

done = true;

}

}

**Question: Write Java method for binary search algorithm.**

public int search(int v){

int low = 0;

int high = a.length - 1;

while (low <= high)

{

int mid = (low + high) / 2;

int diff = a[mid] - v;

if (diff == 0) // a[mid] == v

return mid;

else if (diff < 0) // a[mid] < v

low = mid + 1;

else

high = mid - 1;

}

return -1;

}

**Question: Write java method to sort an array using insertion sort**

/\*\*

Sorts the array managed by this insertion sorter

\*/

public void sort()

{

for (int i = 1; i < a.length; i++)

{

int next = a[i];

// Move all larger elements up

int j = i;

while (j > 0 && a[j - 1] > next)

{

a[j] = a[j - 1];

j--;

}

// Insert the element

a[j] = next;

}

}

Chapter 15 and 16 are very important, therefore you have to study these chapters thoroughly. Following are some sample questions from these chapters.

====== Chapter 15 ======

What are the advantages of using linked lists?

What is the initial position of a list iterator?

After inserting a new element, this element will be before or after the list iterator?

Write the code to visit all elements of a linked list in two manners

Compare the complexity of adding, remove, and research elements in an array and in linked list

Write the code to add and remove the first element of the linked list

Write the code to add and remove an element in the list iterator position

What is the difference between a stack and a queue?

====== Chapter 16 ======

What are the characteristics of a Set?

When we prefer using TreeSet instead of HashSet?

What is the advantage of Sets comparing with Arrays and LinkedLists?

Compare between Sets and Maps

Give examples of hash functions

What is a bucket in a hash table?

How to find an element in a Hash Table?

What is the complexity of adding, locating, and removing elements in a hash table?

How to choose the hash table size?

Write to code add and remove an element from a HashSet

What does method hasNext() of HashSetIterator return?

How to calculate the hash code of integer and strings?

What is the characteristics of a binary Search Tree?

How to create a binary search tree?

How to delete an element from a binary search tree?

Write the code to add a node in a binary search tree

What is the complexity to add a new element in a binary search tree?

Compare between balanced and unbalanced tree

What is the difference between the different tree traversal schemes?

When using the postorder traversal?

What is the data structure using to implement the priority queue?

What is the characteristics of a Heap?

How to add and remove elements from a Heap?

What is the Heap efficiency?

Compare between the heapsort algorithm and the mergesort algorithm