# CSC 427: Data Structures and Algorithm Analysis

## Fall 2008

- Java review (or What I Expect You to Know from 221/222)
  - class, object, fields, methods, private vs. public, parameters
  - variables, primitive vs. objects, expressions, if, if-else, while, for
  - object-oriented design: cohesion, coupling
  - String, Math, arrays, ArrayList
  - interfaces, List, LinkedList, iterators
  - searching and sorting, algorithm efficiency, recursion
  - Stack class, Queue interface
  - inheritance, polymorphism



# Class structure

```
/**
 * This class models a simple die object,
 * which can have any number of sides.
 *   @author Dave Reed
 *   @version 8/25/08
 */
public class Die {

  private int numSides;
  private int numRolls;

  /**
   * Constructs a 6-sided die object
   */
  public Die() {
    this.numSides = 6;
    this.numRolls = 0;
  }

  /**
   * Constructs an arbitrary die object.
   *   @param sides the number of sides on the die
   */
  public Die(int sides) {
    this.numSides = sides;
    this.numRolls = 0;
  }

  . . .
```

a *class* defines a new type of object

- *fields* are variables that belong to the object (and maintain its state)
- typically *private,* so can only be accessed from within the class
- *note: "this." is optional, but instructive*

- *methods* define the actions that can be performed on an object
- typically *public*, so can be called by client code
- *note: "this." is optional, but instructive*

- a *constructor* is a special method that automatically initializes the object when it is created
- can have more than one constructor

## Class structure (cont.)

```
  . . .

  /**
   * Gets the number of sides on the die object.
   *   @return  the number of sides (an N-sided die can roll 1 through N)
   */
  public int getNumberOfSides() {
    return this.numSides;
  }

  /**
   * Gets the number of rolls by on the die object.
   *   @return  the number of times roll has been called
   */
  public int getNumberOfRolls() {
    return this.numRolls;
  }

  /**
   * Simulates a random roll of the die.
   *   @return  the value of the roll (for an N-sided die,
   *            the roll is between 1 and N)
   */
  public int roll() {
    this.numRolls++;
    return (int)(Math.random()*this.getNumberOfSides() + 1);
  }
}
```

a *return statement* specifies the value returned by a call to the method

*accessor method:* provides access to a private field

*mutator method:* changes one or more fields

3

## public static void main

### using the BlueJ IDE, we could

- create objects by right-clicking on the class icon
- call methods on an object by right-clicking on the object icon

### the more general approach is to have a separate "driver" class

- if a class has a "public static void main" method, it will automatically be executed
- a `static` method belongs to the entire class, you don't have to create an object in order to call the method

```
public class DiceRoller {
   public static void main(String[] args) {
      Die d6 = new Die();

      int roll = d6.roll() + d6.roll();

      System.out.println("You rolled a " + roll);
   }
}
```

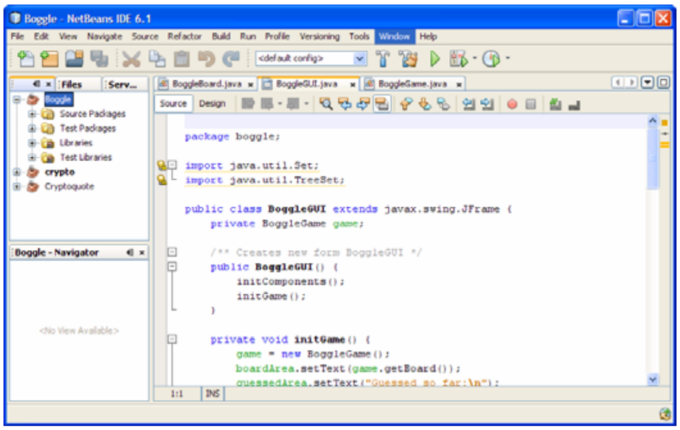
- you can have a `public static void main` method in any class – makes it executable (good for testing purposes)

4

## NetBeans

### in class, we will be using the NetBeans IDE

- see [http://cs.creighton.edu/info/software.html](http://cs.creighton.edu/info/software.html) for dowload & installation instructions




### nice features:

- can easily set preferences & defaults
- can automatically generate javadocs
- code completion
- code refactoring

5

## Java variables

### variable names consist of letters, digits, and underscores

- must start with a letter (can use underscore, but don't)

### naming conventions:

- class names start with a capital letter
  - e.g., `Die`, `String`
- methods, fields, parameters, and local variables start with a lowercase letter
  - e.g., `roll`, `getNumberOfRolls`, `numRolls`, `numSides`, `sides`, `i`
- constants (i.e., `final` values) are in all capitals
  - e.g., `MAX_SCORE`, `DEFAULT_SIZE`

### primitive types are predefined in Java, e.g., `int, double, boolean, char`

```
int num;            double x = 5.8;
num = 0;
```

### object types are those defined by classes, e.g., `Die, String`

```
Die d8 = new Die(8);
int result = d8.roll();
```

6

## Class composition

fields of a class can be instances of other classes

- consider a Dot class, to be used in dot race simulations

predefined mathematical ops:

+, -, *, /, %, ++, --

arithmetic assignments:

+=, -=, *=, /=, %=

when applied to Strings, '+' concatenates

```
public class Dot {

    private Die die;
    private String dotColor;
    private int dotPosition;

    public Dot(String color, int maxStep) {
        this.die = new Die(maxStep);
        this.dotColor = color;
        this.dotPosition= 0;
    }

    public int getPosition() {
        return this.dotPosition;
    }

    public void step() {
        this.dotPosition += this.die.roll();
    }

    public void reset() {
        this.dotPosition = 0;
    }

    public void showPosition() {
        System.out.println(this.dotColor + ": " +
                           this.dotPosition);
    }
}
```

7

## DotRace class

`final` denotes a *constant* variable

- once assigned a value, it cannot be changed

`static` denotes a class variable

- belongs to the class, is shared by all instances

```
public class DotRace {

   public final static int GOAL = 20;
   public final static int MAX_STEP = 3;

   public static void main(String[] args) {
      Dot redDot = new Dot("RED", MAX_STEP);
      Dot blueDot = new Dot("BLUE", MAX_STEP);

      while (redDot.getPosition() < GOAL && blueDot.getPosition() < GOAL) {
         redDot.step();
         blueDot.step();

         redDot.showPosition();
         blueDot.showPosition();
      }

      if (redDot.getPosition() >= GOAL && blueDot.getPosition() >= GOAL) {
         System.out.println("It is a tie");
      }
      else if (redDot.getPosition() >= GOAL) {
         System.out.println("RED wins!");
      }
      else {
         System.out.println("BLUE wins!");
      }
   }
}
```

*if statement* (w/ optional else) defines conditional execution

*while loop* defines conditional repetition

- both driven by a boolean test
- can use relational ops: `>  >=  <  <=  ==  !=`
- can use logical connectives: `&&` (and), `||` (or), `!` (not)

8

## Example: VolleyballSimulator

consider a volleyball simulation in which each team's power ranking determines their likelihood of winning a point

```
public class VolleyballSimulator {

    private Die roller;       // Die for simulating points
    private int ranking1;     // power ranking of team 1
    private int ranking2;     // power ranking of team 2

    /**
     * Constructs a volleyball game simulator.
     *   @param team1Ranking the power ranking (0-100) of team 1, the team that serves first
     *   @param team2Ranking the power ranking (0-100) of team 2, the receiving team
     */
    public VolleyBallSimulator(int team1Ranking, int team2Ranking) {
        this.roller = new Die(team1Ranking+team2Ranking);
        this.ranking1 = team1Ranking;
        this.ranking2 = team2Ranking;
    }

    /**
     * Simulates a single rally between the two teams.
     *   @return the winner of the rally (either "team 1" or "team 2")
     */
    public String playRally() {
        if (this.roller.roll() <= this.ranking1) {
            return "team 1";
        }
        else {
            return "team 2";
        }
    }

    . . .
```

9

## Example: VolleyballSimulator (cont.)

`java.lang.Math` contains many useful static fields & methods

- Math.PI, Math.E
- Math.abs, Math.sqrt, Math.random

```
    . . .

    /**
     * Simulates an entire game using the rally scoring system.
     *   @param winningPoints the number of points needed to win the game (winningPoints > 0)
     *   @return the winner of the game (either "team 1" or "team 2")
     */
    public String playGame(int winningPoints) {
        int score1 = 0;
        int score2 = 0;
        String winner = "";

        while ((score1 < winningPoints && score2 < winningPoints)
                 || (Math.abs(score1 - score2) <= 1)) {
            winner = this.playRally();
            if (winner.equals("team 1")) {
                score1++;
            }
            else {
                score2++;
            }

            System.out.println(winner + " wins the point (" + score1 + "-" + score2 + ")");
        }
        return winner;
    }
}
```

note: always use `equals` to compare objects, not ==

10

## Example: Interactive VolleyballStats

```
import java.util.Scanner;

/**
 * Performs a large number of volleyball game simulations and displays statistics.
 *   @author Dave Reed
 *   @version 8/25/08
 */
public class VolleyballStats {

    public final static int WINNING_POINTS = 15;
    public final static int NUM_GAMES = 10000;

    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

        System.out.print("What is the ranking for team 1? ");
        int ranking1 = input.nextInt();
        System.out.print("What is the ranking for team 2? ");
        int ranking2 = input.nextInt();

        VolleyballSimulator sim = new VolleyballSimulator(ranking1, ranking2);
        int team1Wins = 0;
        for (int game = 0; game < NUM_GAMES; game++) {
            if (sim.playGame(WINNING_POINTS).equals("team 1")) {
                team1Wins++;
            }
        }

        System.out.println("Out of " + NUM_GAMES + " games to " + WINNING_POINTS +
                           ", team 1 (" + ranking1 + "-" + ranking2 + ") won: " +
                           100.0*team1Wins/NUM_GAMES + "%");
    }
}
```

Java 5.0 introduced the `Scanner` class
- simple console or file input

*for loop:* neater version of while, for deterministic loops

11

## Design issues

*cohesion* describes how well a unit of code maps to an entity or behavior

in a highly cohesive system:

- each class maps to a single, well-defined entity – encapsulating all of its internal state and external behaviors
- each method of the class maps to a single, well-defined behavior
- leads to code that is easier to read and reuse

*coupling* describes the interconnectedness of classes

in a loosely coupled system:

- each class is largely independent and communicates with other classes vi a small, well-defined interface
- leads to code that is easier to develop and modify

12

# Java Strings

the `String` class includes many useful methods (in addition to '+')

| | |
|---|---|
| `int length()` | returns number of chars in String |
| `char charAt(int index)` | returns the character at the specified index |
| `int indexOf(char ch)`<br>`int indexOf(String str)` | returns index where the specified char/substring first occurs in the String (-1 if not found) |
| `String substring(int start, int end)` | returns the substring from indices start to (end-1) |
| `String toUpperCase()`<br>`String toLowerCase()` | returns copy of String with all letters uppercase<br>returns copy of String with all letters lowercase |
| `bool equals(String other)`<br>`int compareTo(String other)` | returns true if other String has same value<br>returns *neg* if < other; 0 if = other; *pos* if > other |

ALSO, from the **`Character`** class:

| | |
|---|---|
| `char Character.toLowerCase(char ch)` | returns lowercase copy of ch |
| `char Character.toUpperCase(char ch)` | returns uppercase copy of ch |
| `boolean Character.isLetter(char ch)` | returns true if ch is a letter |
| `boolean Character.isLowerCase(char ch)` | returns true if lowercase letter |
| `boolean Character.isUpperCase(char ch)` | returns true if uppercase letter |

13

# Example: Pig Latin

```
. . .

public String pigLatin(String str) {
    int firstVowel = this.findVowel(str);

    if (firstVowel <= 0) {
        return str + "way";
    }
    else {
        return str.substring(firstVowel, str.length()) +
               str.substring(0, firstVowel) + "ay";
    }
}

private boolean isVowel(char ch) {
    String VOWELS = "aeiouAEIOU";
    return (VOWELS.indexOf(ch) != -1);
}

private int findVowel(String str) {
    for (int i = 0; i < str.length(); i++) {
        if (this.isVowel(str.charAt(i))) {
            return i;
        }
    }
    return -1;
}
```

14

# Java arrays

## arrays are simple lists

- stored contiguously, with each item accessible via an index
- must specify content type when declare, size when create
- once created, the size cannot be changed (without copying entire contents)

```
public class DiceStats {
   public final static int DIE_SIDES = 6;
   public final static int NUM_ROLLS = 10000;

   public static void main(String[] args) {
      int[] counts = new int[2*DIE_SIDES+1];

      Die die = new Die(DIE_SIDES);
      for (int i = 0; i < NUM_ROLLS; i++) {
         counts[die.roll() + die.roll()]++;
      }

      for (int i = 2; i < counts.length; i++) {
         System.out.println(i + ": " + counts[i] + "  ("
                              + (100.0*counts[i]/NUM_ROLLS) + "%)");
      }
   }
}
```

15

# Java ArrayLists

## an `ArrayList` is a more robust, general purpose list of objects

- must specify content type when declare, size is optional (default is 0)
- can be dynamically expanded/reduced; can easily add/remove from middle

## common methods:

| | |
|---|---|
| `T get(int index)` | returns object at specified index |
| `T add(Object obj)` | adds obj to the end of the list |
| `T add(int index, T obj)` | adds obj at index (shifts to right) |
| `T remove(int index)` | removes object at index (shifts to left) |
| `int size()` | removes number of entries in list |
| `boolean contains(T obj)` | returns true if obj is in the list |

## other useful methods:

| | |
|---|---|
| `T set(int index, T obj)` | sets entry at index to be obj |
| `int indexOf(T obj)` | returns index of obj in the list (assumes obj has an `equals` method) |
| `String toString()` | returns a String representation of the list e.g., "[foo, bar, biz, baz]" |

16

# Example: Dictionary

one constructor can call another via `this()`

a `Scanner` object can be used to read from a file

- must create a `File` object
- in case the file isn't there, the code is required to catch `FileNotFoundException`

```
try {
  // CODE TO TRY
}
catch (ExceptionType e) {
  // CODE IN CASE IT OCCURS
}
```

```
import java.util.ArrayList;
import java.util.Scanner;
import java.io.File;

public class Dictionary {
    private ArrayList<String> words;

    public Dictionary() {
        this.words = new ArrayList<String>();
    }

    public Dictionary(String filename) {
        this();

        try {
            Scanner infile = new Scanner(new File(filename));
            while (infile.hasNext()) {
                String nextWord = infile.next();
                this.words.add(nextWord.toLowerCase());
            }
        }
        catch (java.io.FileNotFoundException e) {
            System.out.println("FILE NOT FOUND");
        }
    }

    public void add(String newWord) {
        this.words.add(newWord.toLowerCase());
    }

    public void remove(String oldWord) {
        this.words.remove(oldWord.toLowerCase());
    }

    public boolean contains(String testWord) {
        return this.words.contains(testWord.toLowerCase());
    }
}
```

17

# Arraylists & primitives

ArrayLists can only store objects, but Java 5.0 (and above) will automatically box and unbox primitives into *wrapper classes* (Integer, Double, Character, …)

```
import java.util.ArrayList;

public class DiceStats {
   public final static int DIE_SIDES = 6;
   public final static int NUM_ROLLS = 10000;

   public static void main(String[] args) {
      ArrayList<Integer> counts = new ArrayList<Integer>();
      for (int i = 0; i <= 2*DIE_SIDES; i++) {
         counts.add(0);
      }

      Die die = new Die(DIE_SIDES);
      for (int i = 0; i < NUM_ROLLS; i++) {
         int roll = die.roll() + die.roll();
         counts.set(roll, counts.get(roll)+1);
      }

      for (int i = 2; i < counts.length; i++) {
         System.out.println(i + ": " + counts.get(i) + "  ("
                              + (100.0*counts.get(i)/NUM_ROLLS) + "%)");
      }
   }
}
```

18

## List interface

### an *interface* defines a generic template for a class

- specifies the methods that the class must implement
- but, does not specify fields nor method implementations

```
public interface List<T> {
    boolean add(T obj);
    boolean add(int index, T obj);
    void clear();
    boolean contains(Object obj);
    T get(int index);
    T remove(int index);
    boolean remove(T obj)
    T set(int index, T obj);
    int size();
    . . .
}
```

### advantage: can define different implementations with different tradeoffs

```
public class ArrayList<T> implements List<T> { … }     // uses array, so direct access
                                                       // but must shift when add/remove

public class LinkedList<T> implements List<T> { … }    // uses doubly-linked list, so
                                                       // sequential access but easy
                                                       // add/remove
```

- so, can write generic code that works on a `List` → either implementation will work

19

## Example: Dictionary

*polymorphism:* the capability of objects to react differently to the same method call

here, can declare the field to be of type `List` (the more generic interface)

- if choose to instantiate with an `ArrayList`, it's methods will be called
- if choose to instantiate with a `LinkedList`, it's methods will be called

this style leads to more general-purpose code

```
import java.util.List;
import java.util.ArrayList;
import java.util.Scanner;
import java.io.File;

public class Dictionary {
    private List<String> words;

    public Dictionary() {
        this.words = new ArrayList<String>();
    }

    public Dictionary(String filename) {
        this();

        try {
            Scanner infile = new Scanner(new File(filename));
            while (infile.hasNext()) {
                String nextWord = infile.next();
                this.words.add(nextWord.toLowerCase());
            }
        }
        catch (java.io.FileNotFoundException e) {
            System.out.println("FILE NOT FOUND");
        }
    }

    public void add(String newWord) {
        this.words.add(newWord.toLowerCase());
    }

    public void remove(String oldWord) {
        this.words.remove(oldWord.toLowerCase());
    }

    public boolean contains(String testWord) {
        return this.words.contains(testWord.toLowerCase());
    }
}
```

20

## Collections class

`java.util.Collections` provides a variety of static methods on Lists

```
static int binarySearch(List<T> list, T key);   // where T is Comparable
static T max(List<T> list);                     // where T is Comparable
static T min(List<T> list);                     // where T is Comparable
static void reverse(List<T> list);
static void shuffle(List<T> list);
static void sort(List<T> list);                 // where T is Comparable
```

since the `List` interface is specified, can make use of polymorphism

- these methods can be called on both `ArrayLists` and `LinkedLists`

```
ArrayList<String> words = new ArrayList<String>();
…
sort(words);

LinkedList<Integer> nums = new LinkedList<Integer>();
…
sort(nums);
```

21

## Searching an ArrayList

sequential search traverses the list from beginning to end

- check each entry in the list
- if matches the desired entry, then FOUND (return its index)
- if traverse entire list and no match, then NOT FOUND (return -1)

recall: the `ArrayList` class has `indexOf`, `contains` methods

```
public int indexOf(T desired) {
    for(int k=0; k < this.size(); k++) {
        if (desired.equals(this.get(k))) {
            return k;
        }
    }
    return -1;
}

public boolean contains(T desired) {
    return this.indexOf(desired) != -1;
}
```

22

## Sequential search: Big-Oh analysis

to represent an algorithm's performance in relation to the size of the problem, computer scientists use *Big-Oh* notation

an algorithm is O(N) if the number of operations required to solve a problem is proportional to the size of the problem

sequential search on a list of N items requires *roughly* N checks (+ other constants)
➔ O(N)

*<we will revisit the technical definition of Big-Oh later in the course>*

for an O(N) algorithm, doubling the size of the problem requires double the amount of work (in the worst case)

- if it takes 1 second to search a list of 1,000 items, then
  it takes 2 seconds to search a list of 2,000 items
  it takes 4 seconds to search a list of 4,000 items
  it takes 8 seconds to search a list of 8,000 items
  . . .



## Quick quiz:

what is the Big-Oh complexity of the following method?

```
public int sumOfNums(List<Integer> numbers) {
    int sum = 0;
    for (int i = 0; i < numbers.size(); i++) {
        sum += numbers.get(i);
    }
    return sum;
}
```

does it matter if the method is passed an `ArrayList` or `LinkedList`?

### alternative: iterator

```
Iterator<Integer> iter = numbers.iterator();
while (iter.hasNext()) {
    sum += iter.next();
}
```

the iterator executes an efficient traversal, regardless of the List type → O(N)

### alternative: enhanced for loop

```
for (Integer n : numbers) {
    sum += n;
}
```

hides the underlying iterator → O(N)
*note: can't be used if altering the list while traversing*

## Binary search

the `Collections` utility class contains a `binarySearch` method

- `T extends Comparable<? super T>` is UGLY notation

  refers to the fact that the class must implement the `Comparable` interface, or if a derived class then one of its parents must

```
public static <T extends Comparable<? super T>> int binarySearch(List<T> items, T desired) {
    int left = 0;                    // initialize range where desired could be
    int right = items.length-1;

    while (left <= right) {
        int mid = (left+right)/2;       // get midpoint value and compare
        int comparison = desired.compareTo(items.get(mid));

        if (comparison == 0) {          // if desired at midpoint, then DONE
            return mid;
        }
        else if (comparison < 0) {      // if less than midpoint, focus on left half
            right = mid-1;
        }
        else {                          // otherwise, focus on right half
            left = mid + 1;
        }
    }
    return -1;                          // if reduced to empty range, NOT FOUND
}
```

25

## Binary search: Big-Oh analysis

an algorithm is O(log N) if the number of operations required to solve a problem is proportional to the logarithm of the size of the problem

binary search on a list of N items requires *roughly* $\log_2 N$ checks (+ other constants)

➔ O(log N)

for an O(log N) algorithm, doubling the size of the problem adds only a constant amount of work

- if it takes 1 second to search a list of 1,000 items, then

  searching a list of 2,000 items will take time to check midpoint + 1 second

  searching a list of 4,000 items will take time for 2 checks + 1 second

  searching a list of 8,000 items will take time for 3 checks + 1 second

  . . .

26

## Comparison: searching a phone book

| Number of entries in phone book | Number of checks performed by sequential search | Number of checks performed by binary search |
|---|---|---|
| 100 | 100 | 7 |
| 200 | 200 | 8 |
| 400 | 400 | 9 |
| 800 | 800 | 10 |
| 1,600 | 1,600 | 11 |
| … | … | … |
| 10,000 | 10,000 | 14 |
| 20,000 | 20,000 | 15 |
| 40,000 | 40,000 | 16 |
| … | … | … |
| 1,000,000 | 1,000,000 | 20 |

to search a phone book of the United States (~300 million) using binary search?

to search a phone book of the world (6.7 billion) using binary search?

27

## $O(N^2)$ sorts

### a variety of algorithms exist for sorting a list

- *insertion sort* takes one item at a time and inserts it into an auxiliary sorted list
- *selection sort* traverses to find the next smallest, then swaps it into place
- both are $O(N^2)$, so doubling the list size quadruples the amount of work

```
public void selectionSort(ArrayList<String> items)
{
  for (int i = 0; i < items.size()-1; i++) {        // traverse the list to
    int indexOfMin = i;                             // find the index of the
    for (int j = i+1; j < items.size(); j++) {      // next smallest item
      if (items.get(j).compareTo(items.get(indexOfMin)) < 0) {
        indexOfMin = j;
      }
    }

    String temp = items.get(i);                     // swap the next smallest
    items.set(i, items.get(indexOfMin));            // item into its correct
    items.set(indexOfMin, temp);                    // position
  }
}
```

28

# O(N log N) sorts

## faster, but more complex sorts exist

- *quick sort* partitions the list around a pivot, then recursively sorts each partition
- *merge sort* recursively sorts each half of the list, then merges the sorted sublists
- both are O(N log N), so doubling the list size increases the work by a little more than double

```
public void mergeSort(ArrayList<String> items)
{
    mergeSort(items, 0, items.size()-1);
}

private void mergeSort(ArrayList<String> items, int low, int high)
{
    if (low < high) {
        int middle = (low + high)/2;
        mergeSort(items, low, middle);
        mergeSort(items, middle+1, high);
        merge(items, low, high);
    }
}

. . .
```

note: merging two lists of size N can be done in O(N) steps

29

# Recursion

## recursion is useful when a task can be broken down into smaller, similar tasks

- method: directly or indirectly calls itself
- class: contains a smaller version of itself as a field

## EXAMPLE: recursive permutation generator:

- for each letter in the word
  1. remove that letter
  2. get the next permutation of the remaining letters (using recursive field)
  3. add the letter back at the front

key to understanding recursion: don't think too hard – only 1 level deep!

```
public class PermutationGenerator
{
  private String word;
  private int current;
  private PermutationGenerator tailGenerator;

  public PermutationGenerator(String word) {
    this.word = word;
    this.current = 0;
    if (this.word.length() > 1) {
      this.tailGenerator =
          new PermutationGenerator(this.word.substring(1));
    }
  }

  public String nextPermutation() {
    if (this.word.length() == 1) {
      this.current++;
      return this.word;
    }

    String r = this.word.charAt(this.current) +
               this.tailGenerator.nextPermutation();

    if (!this.tailGenerator.hasMorePermutations()) {
      this.current++;
      if (this.current < this.word.length()) {
        String tailString = this.word.substring(0, this.current)
                          + this.word.substring(this.current+1);
        this.tailGenerator = new PermutationGenerator(tailString);
      }
    }
    return r;
  }

  public boolean hasMorePermutations()
  {
    return this.current < this.word.length();
  }
}
```

30

## Stacks & Queues

the `java.util.Stack` class defines the basic operations of a stack

```
public class Stack<T> {
    public Stack<T>() { … }
    T push(T obj) { … }
    T pop() { … }
    T peek() { … }
    boolean isEmpty() { … }
    …
}
```

the `java.util.Queue` interface defines the basic operations of a queue

- `LinkedList` implements the `Queue` interface

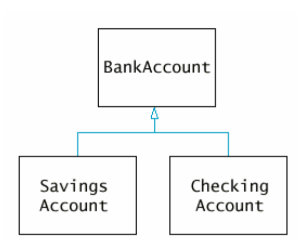
```
public interface Queue<T> {
    boolean add(T obj);
    T remove();
    T peek();
    boolean isEmpty();
    …
}

Queue<Integer> numQ = new LinkedList<Integer>();
```

31

## Inheritance

inheritance is a mechanism for enhancing existing classes

- one of the most powerful techniques of object-oriented programming
- allows for large-scale code reuse




- here, a static field is used so that each account has a unique number

```
public class BankAccount {

  private double balance;
  private int accountNumber;
  private static int nextNumber = 1;

  public BankAccount() {
      this.balance = 0;
      this.accountNumber = this.nextNumber;
      this.nextNumber++;
  }

  public int getAccountNumber() {
      return this.accountNumber;
  }

  public double getBalance() {
      return this.balance;
  }

  public void deposit(double amount) {
      this.balance += amount;
  }

  public void withdraw(double amount) {
      if (amount >= this.balance) {
          this.balance -= amount;
      }
  }
}
```

32

## Derived classes

```
public class SavingsAccount extends BankAccount
{
  private double interestRate;

  public SavingsAccount(double rate)
  {
    this.interestRate = rate;
  }

  public void addInterest()
  {
    double interest =
      this.getBalance()*this.interestRate/100;
    this.deposit(this.interest);
  }
}
```

a derived class automatically inherits all fields and methods (but private fields are inaccessible)

- can override existing methods or add new fields/methods as needed

```
public class CheckingAccount extends BankAccount
{
  private int transCount;
  private static final int NUM_FREE = 3;
  private static final double TRANS_FEE = 2.0;

  public CheckingAccount()
  {
    this.transCount = 0;
  }

  public void deposit(double amount)
  {
    super.deposit(amount);
    this.transCount++;
  }

  public void withdraw(double amount)
  {
    super.withdraw(amount);
    this.transCount++;
  }

  public void deductFees()
  {
    if (this.transactionCount > NUM_FREE) {
      double fees =
        TRANS_FEE*(this.transCount – NUM_FREE);
      super.withdraw(fees);
    }
    this.transCount = 0;
  }
}
```

33

## Inheritance & polymorphism

polymorphism applies to classes in an inheritance hierarchy

- can pass a derived class wherever the parent class is expected
- the appropriate method for the class is called

```
public void showAccount(BankAccount acct) {
    System.out.println("Account " + acct.getAccountNumber() + ": $" +
                       acct.getBalance());
}


BankAccount acct1 = new BankAccount();
…
showAccount(acct1);

SavingsAccount acct2 = new SavingsAccount();
…
showAccount(acct2);

CheckingAccount acct3 = new CheckingAccount();
…
showAccount(acct3);
```

34