

# CSC 222: Computer Programming II

Spring 2005

## Review of Java basics

- > class structure: data fields, methods, constructors
- > `public static void main, JCreator LE`
- > parameters vs. data fields vs. local variables
- > control: if, if-else, while, for
- > `Scanner` class, `user input`
- > predefined classes: `String`, `array`, `ArrayList`

1

## Object-oriented programming

### the *object-oriented* approach to programming:

- solve problems by modeling real-world objects  
e.g., if designing a banking system, model clients, accounts, deposits, ...
- a program is a collection of interacting objects
- in software, objects are created from classes  
*the class describes the kind of object (its properties and behaviors)*  
*the objects represent individual instantiations of the class*

### REAL WORLD CLASS: `Die`

### REAL WORLD OBJECTS: new 6-sided die, worn-out 8-sided die, ...

- the class encompasses all dice  
they all have common properties: number of sides, number of times rolled, ...  
they all have common behaviors: can roll them, count sides, check number of rolls, ...
- each car object has its own specific characteristics and ways of producing behaviors  
rolling a 6-sided die produces a different range of possible values than an 8-sided die

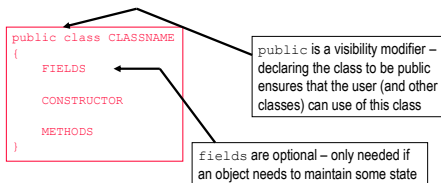
2

## Java classes

### a Java class definition must:

- specify those properties and their types
- define how to create an object of the class
- define the behaviors of objects

FIELDS  
CONSTRUCTOR  
METHODS



3

## Simple example: Die class

```
/**
 * This class models a simple die object, which can have any number of sides.
 * @author Dave Reed
 * @version 9/1/04
 */
public class Die
{
    private int numSides;
    private int numRolls;

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

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

a `Die` object needs to keep track of its number of sides, number of times rolled  
fields are declared by specifying protection (private), type, and name

the default constructor (no parameters) creates a 6-sided die  
fields are assigned values using '='

can have multiple constructors (with parameters)  
• a parameter is specified by its type and name  
• here, the user's input is stored in the `sides` parameter (of type `int`)  
• that value is assigned to the `numSides` field

4

## Simpler example: Die class (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 numSides;
}

/**
 * Gets the number of rolls by on the die object.
 * @return the number of times roll has been called
 */
public int getNumberOfRolls()
{
    return 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()
{
    numRolls = numRolls + 1;
    return (int)(Math.random()*getNumberOfSides() + 1);
}
}
```

note: methods and constructors are *public*, so can be called from other classes/programs

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

a method that simply provides access to a private field is known as an *accessor method*

the *roll* method calculates a random rolls and increments the number of rolls

5

## 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
- recall: 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);
    }
}
```

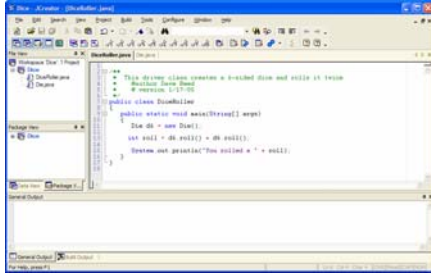
when declaring an object, specify *new* followed by constructor call  
to call a method, specify object name, a period, followed by method call  
output to screen via `System.out.print` and `System.out.println`

6

## JCreator LE

in class, we will be using the JCreator LE development environment

- free, can be downloaded from [www.jcreator.com](http://www.jcreator.com)
- a professional version with advanced features can be purchased as well



7

## Creating/executing a project in JCreator LE

- double-click on the JCreator LE icon to start up the environment
- to create a new project (collection of related classes),
  1. select New Project under the File menu
  2. click on the Empty Project icon, then Next
  3. enter the name of the project and its location (e.g., Desktop, flash drive)
  4. click Finish
- to create a class file within the project
  1. select New Class under the Project menu
  2. enter the name of the file (must be the same name as the class)
  3. click Finish

a document will appear in a new window, enter code here
- to execute the project
  1. select Compile Project under the Build menu
  2. assuming no errors, select Execute Project under the Build menu

8

## fields vs. parameters vs. local variables

fields are one sort of variable

- they store values through the life of an object; are accessible throughout the class

parameters are variables that belong to a method

- parameter value is passed in when method is called
- variable only exists and is accessible while method executes

methods can also include shorter-lived variables (called *local variables*)

- similar to parameter, only exists and is accessible while method executes
- local variables are useful whenever you need to store some temporary value (e.g., in a complex calculation)

before you can use a local variable, you must *declare it*

- specify the variable type and name (similar to fields, but no private modifier)

```
double temperature;
String firstName;

int roll = d6.roll() + d6.roll(); // can initialize at same time
```

9

## primitive types vs. object types

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

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

when you declare a variable of primitive type, memory is allocated for it

- to store a value, simply assign that value to the variable
- can combine the variable declaration and initialization (good practice when possible)

```
int x;                                double height = 72.5;
x = 0;
```

when you declare a variable of object type, it is NOT automatically created

- to initialize, must call the object's constructor: `OBJECT = CLASS (PARAMETERS)` ;
- to call a method: `OBJECT.METHOD (PARAMETERS)`

```
Circle circle1;                       Die d8 = new Die(8)
circle1 = new Circle();
circle1.changeColor("red");           System.out.println( d8.roll() );
```

10

## TicketMachine class

consider a class that models a simple ticket machine

fields: will need to store

- price of the ticket
- amount entered so far by the customer
- total intake for the day

constructor: will take the fixed price of a ticket as parameter

- must initialize the fields

methods: ???

```
/**
 * This class simulates a simple ticket vending machine.
 * @author Dave Reed
 * @version 9/13/04
 */

public class TicketMachine
{
    private int price; // price of a ticket
    private int balance; // amount entered by user
    private int total; // total intake for the day

    /**
     * Constructs a ticket machine.
     * @param ticketCost the fixed price of a ticket
     */
    public TicketMachine(int ticketCost)
    {
        price = ticketCost;
        balance = 0;
        total = 0;
    }
    . . .
}
```

Note: `//` can be used for inline comments (everything following `//` on a line is ignored by the compiler)

11

## TicketMachine methods

`getPrice`:

- accessor method that returns the ticket price

`insertMoney`:

- mutator method that adds to the customer's balance

`printTicket`:

- simulates the printing of a ticket (assuming correct amount has been entered)

```
/**
 * Accessor method for the ticket price.
 * @return the fixed price (in cents) for a ticket
 */
public int getPrice()
{
    return price;
}

/**
 * Mutator method for inserting money to the machine.
 * @param amount the amount of cash (in cents)
 * inserted by the customer
 */
public void insertMoney(int amount)
{
    balance = balance + amount;
}

/**
 * Simulates the printing of a ticket.
 * Note: this naive method assumes that the user
 * has entered the correct amount.
 */
public void printTicket()
{
    System.out.println("*****");
    System.out.println("The BlueJ Line");
    System.out.println("Ticket");
    System.out.println(" " + price + " cents.");
    System.out.println("*****");
    System.out.println();

    // Update the total collected & clear the balance.
    total = total + balance;
    balance = 0;
}
}
```

12

## Better: conditional execution

```
public void insertMoney(int amount)
{
    if (amount > 0) {
        balance += amount;
    }
    else {
        System.out.println("Use a positive amount: " + amount);
    }
}
```

if statements selects code to execute based on Boolean condition

relational operators:  
< > <= >= == !=

```
public void printTicket()
{
    if (balance >= price) {
        System.out.println("#####");
        System.out.println("# The Blue Line#");
        System.out.println("# ticket#");
        System.out.println("# " + price + " cents.");
        System.out.println("#####");
        System.out.println();
        // Update the total collected & clear the balance.
        total += price;
        balance -= price;
    }
    else {
        System.out.println("You must enter at least: " + (price - balance) + " cents.");
    }
}
```

13

## Singer class

when the method has parameters, the values specified in the method call are matched up with the parameter names by order

- the parameter variables are assigned the corresponding values
- these variables exist and can be referenced within the method
- they disappear when the method finishes executing

```
public class OldMacdonald
{
    private static void Verse(String animal, String sound)
    {
        System.out.println("Old MacDonald had a farm, E-I-E-I-O.");
        System.out.println("And on that farm he had " + animal + ", E-I-E-I-O.");
        System.out.println("With a " + sound + " sound = " + sound + " here, and a " +
            sound + " sound = " + sound + " there, ");
        System.out.println(" here = " + sound + ", there a " + sound +
            " , everywhere a " + sound + " = " + sound + " .");
        System.out.println("Old MacDonald had a farm, E-I-E-I-O.");
        System.out.println();
    }

    public static void main(String[] args)
    {
        oldMacdonaldVerse("cow", "moo");
        oldMacdonaldVerse("duck", "quack");
        oldMacdonaldVerse("sheep", "baa");
        oldMacdonaldVerse("dog", "woof");
    }
}
```

the values in the method call are sometimes referred to as *input values* or *actual parameters*

the parameters that appear in the method header are sometimes referred to as *formal parameters*

14

## Dot races

consider the task of simulating a dot race (as on stadium scoreboards)

- different colored dots race to a finish line
- at every step, each dot moves a random distance
- the dot that reaches the finish line first wins!

behaviors

- create a race (dots start at the beginning)
- step each dot forward a random amount
- access the positions of each dot
- display the status of the race
- reset the race

could try to capture the race behavior in a single, self-contained class

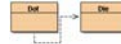
- better to modularize, build up abstractions
- e.g., DotRace class manipulates Dot objects, which manipulate Die objects

15

## Dot class

more naturally:

- fields store a Die (for generating random steps), color & position



- constructor creates the Die object and initializes the color and position fields
- methods access and update these fields to maintain the dot's state

```
public class Dot
{
    private Die die;
    private String dotColor;
    private int dotPosition;

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

    public int getPosition()
    {
        return dotPosition;
    }

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

    public void reset()
    {
        dotPosition = 0;
    }

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

16

## DotRace class

the keyword `final` denotes a *constant* variable

- once assigned a value, it cannot be changed
- use to avoid "magic numbers" in the code (easier to read and update)

```
public class DotRace
{
    private final static int GOAL = 20;
    private final static int MAX_STEP = 3;

    public static void main(String[] args)
    {
        redDot = new Dot("red", MAX_STEP);
        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!");
        }
    }
}
```

a while loop defines conditional repetition

- similar to if statement, driven by Boolean test
- as long as test succeeds, code repeatedly executes

logical connectives: && (and), || (or), ! (not)

17

## Class/object summary

a class defines the content and behavior of a new type

- fields:** variables that maintain the state of an object of that class fields persist as long as the object exists, accessible to all methods if want a single field to be shared by all objects in a class, declare it to be *static* to store a primitive value: declare a variable and assign it a value to store an object: declare a variable, call a constructor and assign to the variable
- methods:** collections of statements that implement behaviors methods will usually access and/or update the fields to produce behaviors statement types so far: assignment, println, return, if, if-else, method call (internal & external) *parameters* are variables that store values passed to a method (allow for generality)
  - parameters persist only while the method executes, accessible only to the method
  - local variables are variables that store temporary values within a method
    - local variables exist from point they are declared to the end of method execution
- constructors:** methods (same name as class) that create and initialize an object a constructor assigns initial values to the fields of the object (can have more than one)

18

## Setting up Java and JCreator on your machine

first, download and install the latest version of Java

- go to [java.sun.com](http://java.sun.com)
- click on J2SE 5.0 from the Popular Downloads table on the right
- click on Download JDK and follow directions to download the Installer
- once saved on your computer, double-click on the Installer (by default, will install the Java Development Kit in C:\Program Files\Java\jdk1.5.0\_01)

next download Java documentation

- from the Java download page, click on the Download button next to J2SE 5.0 Documentation and follow directions
- once saved on your computer, right-click on the ZIP file, select Extract All, and specify C:\Program Files\Java\jdk1.5.0\_01 as the destination directory

finally, download and install the JCreator LE IDE (or Pro version for \$\$\$)

- go to [www.jcreator.com](http://www.jcreator.com)
- click on the Download tab to the left, select JCreator LE, and follow directions
- once saved on your computer, double-click on the Installer (by default, will install the JCreator in C:\Program Files\Java\Xinox Software\JCreator LE

19

## Counters and sums

two common uses of variables:

- to count the number of occurrences of some event
  - ✓ initialize once to 0
  - ✓ increment the counter for each occurrence
- to accumulate the sum of some values
  - ✓ initialize once to 0
  - ✓ add each new value to the sum

```
public class GradeCalculator
{
    private int numGrades; // number of grades entered
    private double gradeSum; // sum of all grades entered

    /**
     * Constructs a grade calculator with no grades so far
     */
    public GradeCalculator()
    {
        numGrades = 0;
        gradeSum = 0;
    }

    /**
     * Adds a new grade to the running totals.
     * @param newGrade the new grade to be entered
     */
    public void addGrade(double newGrade)
    {
        gradeSum += newGrade;
        numGrades++;
    }

    /**
     * Calculates the average of all grades entered so far.
     * @return the average of all grades entered
     */
    public double averageGrade()
    {
        if (numGrades == 0)
            return 0.0;
        else
            return gradeSum/numGrades;
    }
}
```

20

## int vs. real division

if gradeSum is an int variable, then int division occurs

$$(90+91)/2 \rightarrow 181/2 \rightarrow 90$$

since the return type for averageGrade is double, 90 is converted to 90.0 before returning

can force real division by casting an int value to a double before dividing

```
public class GradeCalculator
{
    private int numGrades;
    private int gradeSum;
    . . .
    public double averageGrade()
    {
        if (numGrades == 0) {
            return 0.0;
        }
        else {
            return gradeSum/numGrades;
        }
    }
}

public class GradeCalculator
{
    private int numGrades;
    private int gradeSum;
    . . .
    public double averageGrade()
    {
        if (numGrades == 0) {
            return 0.0;
        }
        else {
            return (double)gradeSum/numGrades;
        }
    }
}
```

21

## FoldPuzzle class

```
public class FoldPuzzle
{
    private double thickness;

    /**
     * Constructs the FoldPuzzle object
     * @param initial the initial thickness (in inches) of the paper
     */
    public FoldPuzzle(double initial)
    {
        thickness = initial;
    }

    /**
     * Computes how many folds it would take for paper thickness to reach the goal distance
     * @param goalDistance the distance (in inches) the paper thickness must reach
     * @return number of folds required
     */
    public int FoldUntil(double goalDistance)
    {
        double current = thickness;
        int numFolds = 0;

        while (current < goalDistance) {
            current *= 2;
            numFolds++;
            System.out.println(numFolds + ": " + current);
        }
        return numFolds;
    }
}
```

22

## For loops

since counter-controlled loops are fairly common, Java provides a special notation for representing them

- a for loop combines all of the loop control elements in the head of the loop

```
int rep = 0;
while (rep < NUM_REPS) {
    STATEMENTS_TO_EXECUTE
    rep++;
}

for (int rep = 0; rep < NUM_REPS; rep++) {
    STATEMENTS_TO_EXECUTE
}
```

execution proceeds exactly as the corresponding while loop

- the advantage of for loops is that the control is separated from the statements to be repeatedly executed
- also, since all control info is listed in the head, much less likely to forget something

```
int sevens = 0;
for (int i = 0; i < 1000; i++) {
    if (d.roll() + d.roll() == 7) {
        sevens++;
    }
}
```

23

## Volleyball simulations

conducting repeated games under different scoring systems may not be feasible

- may be difficult to play enough games to be statistically valid
- may be difficult to control factors (e.g., team strengths)
- might want to try lots of different scenarios

simulations allow for repetition under a variety of controlled conditions

VolleyBallSimulator class:

- must specify the relative strengths of the two teams, e.g., power rankings (0-100) if team1 = 80 and team2 = 40, then team1 is twice as likely to win any given point
- given the power ranking for the two teams, can simulate a point using a Die must make sure that winner is probabilistically correct
- can simulate a game by repeatedly simulating points and keeping score

24

## VolleyBallSimulator class

```
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)
    {
        roller = new Die(team1Ranking+team2Ranking);
        ranking1 = team1Ranking;
        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 (roller.roll() <= ranking1) {
            return "team 1";
        }
        else {
            return "team 2";
        }
    }
}
```

25

## VolleyBallSimulator class (cont.)

```
...
/**
 * 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 = playRally();
        if (winner.equals("team 1")) {
            score1++;
        }
        else {
            score2++;
        }
        System.out.println(winner + " wins the point (" + score1 + "-" + score2 + ")");
    }
    return winner;
}
```

Math.abs function for absolute value

use equals to compare objects, not ==

26

## VolleyBallStats driver class

```
/**
 * Performs a large number of volleyball game simulations and displays statistics.
 * @author Dave Reed
 * @version 1/18/05
 */
public class VolleyBallStats
{
    private final static int WINNING_POINTS = 15;
    private final static int NUM_GAMES = 10000;
    public static void main(String[] args)
    {
        int teamWins = 0;
        for (int game = 0; game < NUM_GAMES; game++) {
            if (playGame(WINNING_POINTS).equals("team 1")) {
                teamWins++;
            }
        }
        System.out.println("Out of " + NUM_GAMES + " games to " + WINNING_POINTS +
            ", team 1 (" + ranking1 + "-" + ranking2 + ") won: " +
            100.0*teamWins/NUM_GAMES + "%");
    }
}
```

27

## User input via the Scanner class

Java 1.5 added the Scanner class for handling simple user input

```
Scanner input = new Scanner(System.in); // creates Scanner object for
// reading from keyboard
```

```
Scanner infile = new Scanner("data.txt"); // creates Scanner object for
// reading from file
```

methods exist for reading values and determining if at end of input

```
input.next() // reads and returns String (separated by whitespace)
input.nextInt() // reads and returns int value
input.nextDouble() // reads and returns double value
infile.hasNext() // returns true if more input values remain to be read
```

28

## Interactive VolleyBallStats

```
import java.util.Scanner;
/**
 * Performs a large number of volleyball game simulations and displays statistics.
 * @author Dave Reed
 * @version 1/18/05
 */
public class VolleyBallStats
{
    public static void main(String[] args)
    {
        Scanner input = new Scanner(System.in);
        System.out.print("How many games do you want to simulate? ");
        int numGames = input.nextInt();
        System.out.print("How many points are required for a win? ");
        int winningPoints = input.nextInt();
        int teamWins = 0;
        for (int game = 0; game < numGames; game++) {
            if (playGame(winningPoints).equals("team 1")) {
                teamWins++;
            }
        }
        System.out.println("Out of " + numGames + " games to " + winningPoints +
            ", team 1 (" + ranking1 + "-" + ranking2 + ") won: " +
            100.0*teamWins/numGames + "%");
    }
}
```

29

## In-class exercise

write a simple class with a main method that

- prompts the user and reads in two integers, the low and high values in a range
- calculates the sum of all integers in that range (inclusive)
- displays the sum

what is the sum from 1 to 100?

what is the sum from 1066 to 2005?

what is the sum from 10,000,000 to 20,000,000?

30

## Java strings

recall: String is a Java class that is automatically defined for you

- a String object encapsulates a sequence of characters
  - you can declare a String variable and assign it a value just like any other type
- ```
String firstName = "Dave";
```
- you can display Strings using `System.out.print` and `System.out.println`
- ```
System.out.println(firstName);
```
- the '+' operator concatenates two strings (or string and number) together

```
String str = "foo" + "lish";  
str = str + "ly";  
  
int age = 19;  
System.out.println("Next year, you will be " + (age+1));
```

31

## String methods

```
int length() returns number of chars in String  
char charAt(int index) returns the character at the specified index  
(indices range from 0 to str.length()-1)  
  
int indexOf(char ch) returns index where the specified char/substring  
int indexOf(String str) 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() returns copy of String with all letters uppercase  
String toLowerCase() returns copy of String with all letters lowercase  
  
boolean equals(String other) returns true if other String has same value  
int compareTo(String other) returns -1 if less than other String,  
0 if equal to other String,  
1 if greater than other String
```

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
```

32

## Pig Latin

```
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 (isVowel(str.charAt(i))) {  
            return i;  
        }  
    }  
    return -1;  
}  
  
public String pigLatin(String str)  
{  
    int firstVowel = findVowel(str);  
    if (firstVowel <= 0) {  
        return str + "way";  
    }  
    else {  
        return str.substring(firstVowel, str.length()) +  
            str.substring(0, firstVowel) + "ay";  
    }  
}
```

33

## Composite data types

String is a composite data type

- each String object represents a collection of characters in sequence
- can access the individual components & also act upon the collection as a whole

many applications require a more general composite data type, e.g.,

- ✓ a to-do list will keep track of a sequence/collection of notes
- ✓ a dictionary will keep track of a sequence/collection of words
- ✓ a payroll system will keep track of a sequence/collection of employee records

Java provides several library classes for storing/accessing collections of arbitrary items

34

## Arrays

arrays are simple lists

- stored contiguously, with each item accessible via an index
- to declare an array, designate the type of value stored followed by []

```
String[] words;           int[] counters;
```

- to create an array, must use `new` (an array is an object)
- specify the type and size inside brackets following `new`

```
words = new String[100];   counters = new int[26];
```

- to access an item in an array, use brackets containing the desired index

```
String str = word[0];      // note: index starts at 0  
  
for (int i = 0; i < 26, i++) {  
    counters[i] = 0;  
}
```

35

## Array example

arrays are useful if the size is unchanging, no need to shift entries

- e.g., to keep track of dice stats, can have an array of counters

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

36

## ArrayList class

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

- create an `ArrayList` by calling the `ArrayList` constructor (no inputs)
- starting with Java 1.5, `ArrayLists` use generics to specify type of object stored

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

- add items to the end of the `ArrayList` using `add`

```
words.add("Billy"); // adds "Billy" to end of list
words.add("Bluejay"); // adds "Bluejay" to end of list
```

- can access items in the `ArrayList` using `get` (indices start at 0)

```
String first = words.get(0); // assigns "Billy"
String second = words.get(1); // assigns "Bluejay"
```

- can determine the number of items in the `ArrayList` using `size`

```
int count = words.size(); // assigns 2
```

37

## ArrayList methods

common methods:

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

other useful methods:

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

38

## Notebook class

consider designing a class to model a notebook (i.e., a to-do list)

- will store notes as `Strings` in an `ArrayList`
- will provide methods for adding notes, viewing the list, and removing notes

```
import java.util.ArrayList;
public class Notebook
{
    private ArrayList<String> notes;
    public Notebook() { ... }
    public void storeNote(String newNote) { ... }
    public void storeNote(int priority, String newNote) { ... }
    public int numberOfNotes() { ... }
    public void listNotes() { ... }
    public void removeNote(int noteNumber) { ... }
    public void removeNote(String note) { ... }
}
```

any class that uses an `ArrayList` must load the library file that defines it

39

```
...
/**
 * Constructs an empty notebook.
 */
public Notebook()
{
    notes = new ArrayList<String>();
}
...
/**
 * Store a new note into the notebook.
 * @param newNote note to be added to the notebook list
 */
public void storeNote(String newNote)
{
    notes.add(newNote);
}
...
/**
 * Store a new note into the notebook with the specified priority.
 * @param priority index where note is to be added
 * @param newNote note to be added to the notebook list
 */
public void storeNote(int priority, String newNote)
{
    notes.add(priority, newNote);
}
...
/**
 * @return the number of notes currently in the notebook
 */
public int numberOfNotes()
{
    return notes.size();
}
...
}
```

constructor creates the (empty) `ArrayList`

one version of `storeNote` adds a new note at the end

another version adds the note at a specified index

`numberOfNotes` calls the `size` method

40

## Notebook class (cont.)

```
...
/**
 * Show a note.
 * @param noteNumber the number of the note to be shown (first note is # 0)
 */
public void showNote(int noteNumber)
{
    if (noteNumber < 0 || noteNumber >= numberOfNotes()) {
        System.out.println("There is no note with that index.");
    }
    else {
        System.out.println(notes.get(noteNumber));
    }
}
...
/**
 * list all notes in the notebook.
 */
public void listNotes()
{
    System.out.println("NOTEBOOK CONTENTS");
    System.out.println("-----");
    for (int i = 0; i < notes.size(); i++) {
        System.out.print(i + ": ");
        showNote(i);
    }
}
...
}
```

`showNote` checks to make sure the note number is valid, then calls the `get` method to access the entry

`listNotes` traverses the `ArrayList` and shows each note (along with its #)

41

## Notebook class (cont.)

```
...
/**
 * Removes a note.
 * @param noteNumber the number of the note to be removed (first note is # 0)
 */
public void removeNote(int noteNumber)
{
    if (noteNumber < 0 || noteNumber >= numberOfNotes()) {
        System.out.println("There is no note with that index.");
    }
    else {
        notes.remove(noteNumber);
    }
}
...
/**
 * Removes a note.
 * @param note the note to be removed
 */
public void removeNote(String note)
{
    boolean found = false;
    for (int i = 0; i < notes.size(); i++) {
        if (note.equals(notes.get(i))) {
            notes.remove(i);
            found = true;
        }
    }
    if (!found) {
        System.out.println("There is no such note.");
    }
}
...
}
```

one version of `removeNote` takes a note #, calls the `remove` method to remove the note with that number

another version takes the text of the note and traverses the `ArrayList` – when a match is found, it is removed

uses `boolean` variable to flag whether found or not

42

## ArrayLists & primitives

ArrayLists can only store objects, but Java 1.5 will automatically box and unbox primitive types 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 <= 2*DIE_SIDES; i++) {
            System.out.println(i + ": " + counts.get(i) + " (" +
                (100.0*counts.get(i)/NUM_ROLLS) + "%");
        }
    }
}
```

43

## 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 via a small, well-defined interface
- leads to code that is easier to develop and modify

44