

CSC 221: Computer Programming I

Fall 2009

Class design & strings

- design principles
- cohesion & coupling
- example: graphical DotRace
- objects vs. primitives
- String methods: length, charAt, substring, indexOf, ...
- StringUtils class/library
- example: Pig Latin
- example: Palindromes

1

Object-oriented design principles so far:

- a **class** should model some entity, encapsulating all of its state and behaviors
e.g., Circle, Die, Dot, DotRace, ...
- a **field** should store a value that is part of the state of an object (and which must persist between method calls)
e.g., xPosition, yPosition, color, diameter, isVisible, ...
 - can be primitive (e.g., int, double) or object (e.g., String, Die) type
 - should be declared private to avoid outside tampering with the fields – provide public accessor methods if needed
 - static fields should be used if the data can be shared among all objects
 - final-static fields should be used to define constants with meaningful names
- a **constructor** should initialize the fields when creating an object
 - can have more than one constructor with different parameters to initialize differently
- a **method** should implement one behavior of an object
e.g., moveLeft, moveRight, draw, erase, changeColor, ...
 - should be declared public to make accessible – helper methods can be private
 - local variables should be used to store temporary values that are needed
 - if statements for conditional execution; while loops for conditional repetition; for loops for counter-driven repetition

2

Cohesion

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

advantages of cohesion:

- highly cohesive code is easier to read
 - don't have to keep track of all the things a method does
 - if the method name is descriptive, it makes it easy to follow code
- highly cohesive code is easier to reuse
 - if the class cleanly models an entity, can reuse it in any application that needs it
 - if a method cleanly implements a behavior, it can be called by other methods and even reused in other classes

3

Coupling

coupling describes the interconnectedness of classes

in a loosely coupled system:

- each class is largely independent and communicates with other classes via small, well-defined interfaces

advantages of loose coupling:

- loosely coupled classes make changes simpler
 - can modify the implementation of one class without affecting other classes
 - only changes to the interface (e.g., adding/removing methods, changing the parameters) affect other classes
- loosely coupled classes make development easier
 - you don't have to know how a class works in order to use it
 - since fields/local variables are encapsulated within a class/method, their names cannot conflict with the development of other classes.methods

4

Recall our Dot class

in the final version:

- a constant represented the maximum step size
- the die field was static to avoid unnecessary duplication

```
public class Dot {
    private static final int MAX_STEP = 5;
    private static Die die = new Die(Dot.MAX_STEP);

    private String dotColor;
    private int dotPosition;

    public Dot(String color) {
        this.dotColor = color;
        this.dotPosition = 0;
    }

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

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

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

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

5

```
public class FinalDotRace {
    private Dot redDot;
    private Dot blueDot;

    public FinalDotRace() {
        this.redDot = new Dot("red");
        this.blueDot = new Dot("blue");
    }

    public void step() {
        this.redDot.step();
        this.blueDot.step();
    }

    public void showStatus() {
        this.redDot.showPosition();
        this.blueDot.showPosition();
    }

    public void reset() {
        this.redDot.reset();
        this.blueDot.reset();
    }

    public void runRace(int goalDistance) {
        this.reset();
        this.showStatus();
        while (this.redDot.getPosition() < goalDistance &&
            this.blueDot.getPosition() < goalDistance) {
            this.step();
            this.showStatus();
        }
        if (this.redDot.getPosition() >= goalDistance &&
            this.blueDot.getPosition() >= goalDistance) {
            System.out.println("IT IS A TIE");
        }
        else if (this.redDot.getPosition() >= goalDistance) {
            System.out.println("RED WINS");
        }
        else {
            System.out.println("BLUE WINS");
        }
    }
}
```

Recall our DotRace class

added a runRace method for running an entire race

- distance to finish line passed in as a parameter
- updates & displays dot positions until one or both cross the finish line
- displays which dot won (or a tie)

6

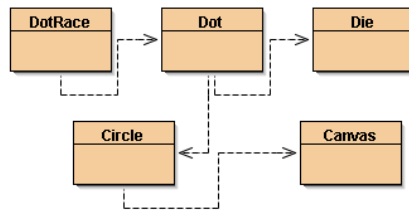
Adding graphics

what if we wanted to display the dot race visually?

- we could utilize the `Circle` class to draw the dots
- recall: `Circle` has methods for relative movements
`moveLeft()`, `moveRight()`, `moveUp()`, `moveDown()`
`moveHorizontal(dist)`, `slowMoveHorizontal(dist)`
`moveVertical(dist)`, `slowMoveVertical(dist)`
- in principle, each step of size `N` can be drawn by calling `slowMoveHorizontal(N)`

due to our cohesive design, changing the display is easy

- each `Dot` object will maintain and display its own `Circle` image

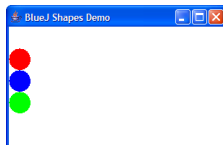


7

Adding graphics

`showPosition` now
moves the `Circle` image

- instead of displaying the position as text



note: limited methods for `Circle` class make this code tedious & complex

- have to keep track of # for each dot so that they can be aligned vertically
- have to keep track of distance traveled since the position was last shown

```
public class Dot {
    private static final int SIZE = 50;
    private static final int MAX_STEP = 5;
    private static Die die = new Die(GraphicalDot.MAX_STEP);
    private static int nextDotNumber = 0;
    private String dotColor;
    private int dotPosition;
    private Circle dotImage;
    private int distanceToDraw;

    public Dot(String color) {
        this.dotColor = color;
        this.dotPosition = 0;
        this.distanceToDraw = 0;
        this.dotImage = new Circle();
        this.dotImage.setColor(color);
        this.dotImage.setSize(Dot.SIZE);
        this.dotImage.moveVertical(Dot.SIZE * (Dot.nextDotNumber));
        this.dotImage.setVisible();
        Dot.nextDotNumber++;
    }

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

    public String getColor() { return this.dotColor; }

    public void step() {
        int distance = GraphicalDot.die.roll();
        this.dotPosition += distance;
        this.distanceToDraw += distance;
    }

    public void reset() {
        this.dotImage.moveHorizontal(-this.dotPosition);
        this.dotPosition = 0;
        this.distanceToDraw = 0;
    }

    public void showPosition() {
        this.dotImage.slowMoveHorizontal(this.distanceToDraw);
        this.distanceToDraw = 0;
    }
}
```

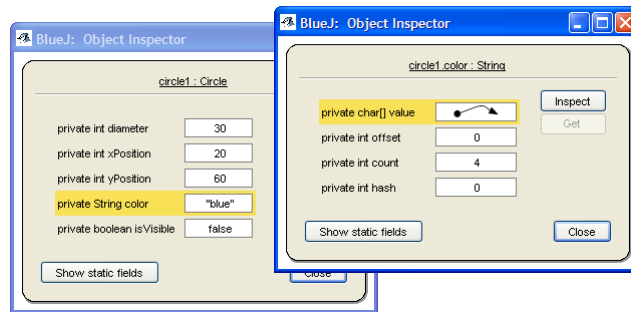
recall: a static field is shared among the class – initialized by the first object created

8

Strings vs. primitives

although they behave similarly to primitive types (int, double, char, boolean),
Strings are different in nature

- String is a class that is defined in a separate library: `java.lang.String`
→ a String value is really an object
- you can call methods on a String
- also, you can *inspect* the String fields of an object



9

Comparing strings

comparison operators (< <= > >=) are defined for primitives but not objects

```
String str1 = "foo"; // EQUIVALENT TO String str1 = new String("foo");
String str2 = "bar"; // EQUIVALENT TO String str2 = new String("bar");
if (str1 < str2) ... // ILLEGAL
```

`==` and `!=` are defined for objects, but don't do what you think

```
if (str1 == str2) ... // TESTS WHETHER THEY ARE THE
// SAME OBJECT, NOT WHETHER THEY
// HAVE THE SAME VALUE!
```

Strings are comparable using the `equals` and `compareTo` methods

```
if (str1.equals(str2)) ... // true IF THEY REPRESENT THE
// SAME STRING VALUE
```

```
if (str1.compareTo(str2) < 0) ... // RETURNS -1 if str1 < str2
// RETURNS 0 if str1 == str2
// RETURNS 1 if str1 > str2
```

10

In HW5...

note: the `repeatedGames` method in `RouletteTester` uses `equals` to compare the `betType` with the two possibilities ("color" and "number")

```
if (betType.equals("color")) {
    game.makeBet(nextBet, RouletteTester.COLOR_DEFAULT);
}
else if (betType.equals("number")) {
    game.makeBet(nextBet, RouletteTester.NUMBER_DEFAULT);
}
```

similarly, you should use `equals` when comparing the spin color with the user's pick in `makeBet`

```
if (spinColor.equals(color)) {
    ...
}
else {
    ...
}
```

never use `==` for comparing Strings

- sometimes it works, but not always

11

String methods

in addition, there are many useful methods defined for Strings

`int length()`
returns the length of the String `str`

```
String str = "foobar";
System.out.println( str.length() );
System.out.println( str.charAt(0) );
System.out.println( str.charAt(1) );
System.out.println( str.charAt(str.length()-1) );
```

`char charAt(int index)`
returns the character at specified index

- first index is 0
- last index is `str.length()-1`

```
String str = "foobar";
for (int i = 0; i < str.length(); i++) {
    System.out.print(str.charAt(i));
}
```

if `index < 0` or `index >= str.length()`, an error occurs

```
String str = "foobar";
for (int i = str.length()-1; i >= 0; i--) {
    System.out.print(str.charAt(i));
}
```

12

Traversing & constructing Strings

since the length of a String can be determined using the `length` method, a for loop can be used to traverse the String

- as you access individual characters, can test and act upon values
- can even construct a new string out of individual characters

```
String str = "zaboomofoo";

int count = 0;
for (int i = 0; i < str.length(); i++) {
    if (str.charAt(i) == 'o') {
        count++;
    }
}
```

```
String copy = "";
for (int i = 0; i < str.length(); i++) {
    copy = copy + str.charAt(i);
}
```

```
String copy = "";
for (int i = 0; i < str.length(); i++) {
    copy = str.charAt(i) + copy;
}
```

13

String utilities

we can define and encapsulate additional string operations in a class

- `StringUtils` will not have any fields, it simply encapsulates methods
- can define methods to be *static* – static methods can be called directly on the class

```
public class StringUtils
{
    /**
     * Reverses a string.
     * @param str the string to be reversed
     * @return a copy of str with the order of the characters reversed
     */
    public static String reverse(String str) {
        String copy = "";
        for (int i = 0; i < str.length(); i++) {
            copy = str.charAt(i) + copy;
        }
        return copy;
    }
    .
    .
    .
}
```

14

Stripping a string

consider the task of removing spaces from a string

- need to traverse the string and check each char to see if it is a space
- if it is not, add that char to the copy string
- if it is a space?

```
/**
 * Strips all spaces out of a string.
 * @param str the string to be stripped
 * @return a copy of str with each space removed
 */
public static String stripSpaces(String str) {
    String copy = "";
    for (int i = 0; i < str.length(); i++) {
        if (str.charAt(i) != ' ') {
            copy += str.charAt(i);
        }
    }
    return copy;
}
```

15

Censoring a string

consider the task of censoring a word, i.e., replacing each vowel with '*'

- need to traverse the string and check each char to see if it is a vowel
- if it is a vowel, add '*' to the copy string
- if it is not, add the char to the copy string

```
/**
 * Censors a string by replacing all vowels with asterisks.
 * @param str the string to be censored
 * @return a copy of str with each vowel replaced by an asterisk
 */
public static String censor(String str) {
    String copy = "";
    for (int i = 0; i < str.length(); i++) {
        if (StringUtils.isVowel(str.charAt(i))) {
            copy += '*';
        } else {
            copy += str.charAt(i);
        }
    }
    return copy;
}

/**
 * Determines if a character is a vowel (either upper or lower case).
 * @param ch the character to be tested
 * @return true if ch is a vowel, else false
 */
public static boolean isVowel(char ch) {
    ?????????
}
```

16

Testing for a vowel

a brute force approach would be to test every vowel separately

TEDIOUS!

```
public static boolean isVowel(char ch) {
    if (ch == 'a') {
        return true;
    }
    else if (ch == 'A') {
        return true;
    }
    else if (ch == 'e') {
        return true;
    }
    else if (ch == 'E') {
        return true;
    }
    .
    .
    .
    else if (ch == 'u') {
        return true;
    }
    else if (ch == 'U') {
        return true;
    }
    else {
        return false;
    }
}
```

17

Testing for a vowel (cont.)

we could simplify the code using the `Character.toLowerCase` method

- `toLowerCase` is a static method of the `Character` class
- it takes a character as input, and returns the lower case equivalent
- if the input is not a letter, then it simply returns it unchanged

```
public static boolean isVowel(char ch) {
    ch = Character.toLowerCase(ch);

    if (ch == 'a') {
        return true;
    }
    else if (ch == 'e') {
        return true;
    }
    else if (ch == 'i') {
        return true;
    }
    else if (ch == 'o') {
        return true;
    }
    else if (ch == 'u') {
        return true;
    }
    else {
        return false;
    }
}
```

18

Testing for a vowel (cont.)

could
simplify
using ||

```
public static boolean isVowel(char ch) {
    ch = Character.toLowerCase(ch);

    if (ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u') {
        return true;
    }
    else {
        return false;
    }
}
```

since the
code returns
the same
value as the
test, can
avoid the if
altogether

```
public static boolean isVowel(char ch) {
    ch = Character.toLowerCase(ch);

    return (ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u');
}
```

boolean expressions involving || or && are evaluated intelligently via *short-circuit evaluation*

- expressions are evaluated left to right
- can stop evaluating || expression as soon as a part evaluates to true
→ entire expression is true
- can stop evaluating && expression as soon as a part evaluates to false
→ entire expression is false

19

Testing for a vowel (cont.)

best solution involves the String method:

```
int indexOf(char ch)
int indexOf(String str)
```

returns the index where ch/str first appears in the string (-1 if not found)

for `isVowel`:

- create a String that contains all the vowels
- to test a character, call `indexOf` to find where it appears in the vowel String
- if return value `!= -1`, then it is a vowel

```
public static boolean isVowel(char ch) {
    String VOWELS = "aeiouAEIOU";

    return (VOWELS.indexOf(ch) != -1);
}
```

20

Substring

the last String method we will consider is substring:

```
String substring(int start, int end)
```

returns the substring starting at index `start` and ending at index `end-1`

e.g., `String str = "foobar";`
`str.substring(0,3)` → "foo"
`str.substring(3, str.length())` → "bar"

```
/**  
 * Capitalizes the first letter in the string.  
 * @param str the string to be capitalized  
 * @return a copy of str with the first letter capitalized  
 */  
public static String capitalize(String str) {  
    return Character.toUpperCase(str.charAt(0)) + str.substring(1, str.length());  
}
```

21

Pig Latin

suppose we want to translate a word into Pig Latin

- simplest version

`nix` → `ixnay`

`latin` → `atinlay`

`pig` → `igpay`

`banana` → `ananabay`

- to translate a word, move the last letter to the end and add "ay"

```
/**  
 * Translates a string into Pig Latin  
 * @param str the string to be converted  
 * @return a copy of str translated into Pig Latin  
 */  
public static String pigLatin(String str) {  
    return str.substring(1, str.length()) + str.charAt(0) + "ay";  
}
```

22

opsoay?

using our method,

oops → opsoay apple → ppleaay

for "real" Pig Latin, you must consider the first letter of the word

- if a consonant, then translate as before (move first letter to end then add "ay")
- if a vowel, simply add "way" to the end

oops → oopsway apple → appleway

```
public static String pigLatin(String str) {  
    if (StringUtils.isVowel(str.charAt(0))) {  
        return str + "way";  
    }  
    else {  
        return str.substring(1, str.length()) + str.charAt(0) + "ay";  
    }  
}
```

23

reightoncay?

using our method,

creighton → reightoncay thrill → hrilltay

for "real" Pig Latin, if the word starts with a sequence of consonants,
must move the entire sequence to the end then add "ay"

creighton → eightoncay thrill → illthray

so, we need to be able to find the first occurrence of a vowel

HOW?

24

Handling multiple consonants

```
/**
 * Finds the first occurrence of a vowel in a string.
 * @param str the string to be searched
 * @return the index where the first vowel in str occurs (-1 if no vowel)
 */
private static int findVowel(String str) {
    for (int i = 0; i < str.length(); i++) {
        if (StringUtils.isVowel(str.charAt(i))) {
            return i;
        }
    }
    return -1;
}

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

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

25

In-class exercise

modify `pigLatin` so that it preserves capitalization

computer → omputercay

science → iencesay

Creighton → Eightoncray

Nebraska → Ebraskanay

Omaha → Omahaway

26

Testing code

when you design and write code, how do you know if it works?

- run it a few times and assume it's OK?

to be convinced that code runs correctly in all cases, you must analyze the code and identify special cases that are handled

- then, define a test data set (inputs & corresponding outputs) that covers those cases
- e.g., for Pig Latin,
 - words that start with single consonant: "foo"→"oofay" "banana"→"ananabay"
 - words that start with multiple consonants: "thrill"→"illthray" "cheese"→"eesechay"
 - words that start with vowel: "apple"→"appleway" "oops"→"oopsway"
 - words with no vowels: "nth"→"nthway"
 - words that are capitalized: "Creighton"→"Eightoncray" "Omaha"→"Omahaway"

27

StringUtil class/library

Method Summary	
static java.lang.String	capitalize (java.lang.String str) Capitalizes the first letter in the string.
static java.lang.String	censor (java.lang.String str) Censors a string by replacing all vowels with asterisks.
static int	findOneOf (java.lang.String seq, java.lang.String str) Finds the first occurrence of a character in a string
static int	findVowel (java.lang.String str) Finds the first occurrence of a vowel in a string.
static boolean	isOneOf (char ch, java.lang.String str) Determines whether a character is contained in a string.
static boolean	isVowel (char ch) Determines if a character is a vowel (either upper or lower case).
static java.lang.String	pigLatin (java.lang.String str) Translates a string into Pig Latin
static java.lang.String	reverse (java.lang.String str) Reverses a string.
static java.lang.String	stripAllOf (java.lang.String seq, java.lang.String str) Removes all occurrences of select letters from a string.
static java.lang.String	stripNonLetters (java.lang.String str) Removes all non-letters from a string.
static java.lang.String	stripSpaces (java.lang.String str) Strips all spaces out of a string.

28

Palindrome

suppose we want to define a method to test whether a word is a palindrome (i.e., reads the same forwards and backwards)

```
isPalindrome("bob") → true           isPalindrome("madam") → true
isPalindrome("blob") → false        isPalindrome("madame") → false
```

download `StringUtils.java`

define `isPalindrome` method returns true if the word is a palindrome

extend your `isPalindrome` method to handle phrases

- need to ignore spaces, punctuation & capitalization

```
isPalindrome("Madam, I'm Adam.") → true
isPalindrome("Able was I ere I saw Elba.") → true
isPalindrome("A man, a plan, a canal: Panama.") → true
```

29

String method summary

<code>int length()</code>	returns number of chars in String
<code>char charAt(int index)</code>	returns the character at the specified index (indices range from 0 to <code>str.length()-1</code>)
<code>int indexOf(char ch)</code> <code>int indexOf(String str)</code>	returns index where the specified char/substring first occurs in the String (-1 if not found)
<code>String substring(int start, int end)</code>	returns the substring from indices start to (end-1)
<code>String toUpperCase()</code> <code>String toLowerCase()</code>	returns copy of String with all letters uppercase returns copy of String with all letters lowercase
<code>boolean equals(String other)</code> <code>int compareTo(String other)</code>	returns true if other String has same value returns -1 if less than other String, 0 if equal to other String, 1 if greater than other String

ALSO, from the Character class:

<code>char Character.toLowerCase(char ch)</code>	returns lowercase copy of ch
<code>char Character.toUpperCase(char ch)</code>	returns uppercase copy of ch
<code>boolean Character.isLetter(char ch)</code>	returns true if ch is a letter
<code>boolean Character.isLowerCase(char ch)</code>	returns true if lowercase letter
<code>boolean Character.isUpperCase(char ch)</code>	returns true if uppercase letter ³⁰