

CSC 221: Computer Programming I

Fall 2006

Understanding class definitions

- class structure
- fields, constructors, methods
- parameters
- assignment statements
- local variables

1

Looking inside classes

recall that classes define the properties and behaviors of its objects

a 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

```
public class CLASSNAME {  
    FIELDS  
  
    CONSTRUCTOR  
  
    METHODS  
}
```

public is a visibility modifier – declaring the class to be public ensures that the user (and other classes) can use of this class

fields are optional – only needed if an object needs to maintain some state

2

Fields

fields store values for an object (a.k.a. instance variables)

- the collection of all fields for an object define its state
- when declaring a field, must specify its visibility, type, and name

```
private FIELD_TYPE FIELD_NAME;
```

for our purposes, all fields will be private (accessible to methods, but not to the user)

```
/**
 * A circle that can be manipulated and that draws itself on a canvas.
 *
 * @author Michael Kolling and David J. Barnes
 * @version 15 July 2000
 */
public class Circle {
    private int diameter;
    private int xPosition;
    private int yPosition;
    private String color;
    private boolean isVisible;

    . . .
}
```

text enclosed in `/** */` is a *comment* – visible to the user, but ignored by the compiler. Good for documenting code.

note that the fields are those values you see when you inspect an object in BlueJ

3

Constructor

a constructor is a special method that specifies how to create an object

- it has the same name as the class, public visibility (since called by the user)

```
public CLASS_NAME(OPTIONAL_PARAMETERS) {
    STATEMENTS FOR INITIALIZING OBJECT STATE
}
```

```
public class Circle {
    private int diameter;
    private int xPosition;
    private int yPosition;
    private String color;
    private boolean isVisible;

    /**
     * Create a new circle at default position with default color.
     */
    public Circle() {
        this.diameter = 30;
        this.xPosition = 20;
        this.yPosition = 60;
        this.color = "blue";
        this.isVisible = false;
    }

    . . .
}
```

within a method, can refer to fields of the object via

```
this.FIELD_NAME
```

the period denotes ownership: you are referring to a field that belongs to "this" object

note: the "this." prefix is optional, but instructive

an *assignment statement* stores a value in a field

```
this.FIELD_NAME = VALUE;
```

here, default values are assigned for a circle

4

Methods

methods implement the behavior of objects

```
public RETURN_TYPE METHOD_NAME(OPTIONAL_PARAMETERS) {  
    STATEMENTS FOR IMPLEMENTING THE DESIRED BEHAVIOR  
}
```

```
public class Circle {  
    . . .  
  
    /**  
     * Make this circle visible. If it was already visible, do nothing.  
     */  
    public void makeVisible() {  
        this.isVisible = true;  
        this.draw();  
    }  
  
    /**  
     * Make this circle invisible. If it was already invisible, do nothing.  
     */  
    public void makeInvisible() {  
        this.erase();  
        this.isVisible = false;  
    }  
  
    . . .  
}
```

void return type specifies no value is returned by the method – here, the result is shown on the Canvas

note that one method can "call" another one

this.draw() calls the draw method on this circle
this.erase() calls the erase method on this circle

5

Simpler example: Die class

```
/**  
 * This class models a simple die object, which can have any number of sides.  
 * @author Dave Reed  
 * @version 9/1/05  
 */  
  
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 Die object needs to keep track of its number of sides, number of times rolled

the default constructor (no parameters) creates a 6-sided die

can have multiple constructors (with parameters)

- a *parameter* is specified by its type and name
- a parameter represents a *temporary* value that can be used during the methods execution
- note: parameters are not prefixed with "this."

6

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 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 = this.numRolls + 1;
    return (int)(Math.random()*this.numSides + 1);
}
}
```

a *return statement* specifies the value returned by a call to the method (shows up in a box in BlueJ)

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

a method that changes the state is a *mutator* method

the `roll` method calculates a random roll (details later) and increments the number of rolls

7

PaperSheet example

```
public class PaperSheet
{
    private double thickness; // thickness in inches
    private int numFolds; // the number of folds so far

    /**
     * Constructs the PaperSheet object
     * @param initial the initial thickness (in inches) of the paper
     */
    public PaperSheet(double initial) {
        this.thickness = initial;
        this.numFolds = 0;
    }

    /**
     * Folds the sheet, doubling its thickness as a result
     */
    public void fold() {
        this.thickness = 2 * this.thickness;
        this.numFolds = this.numFolds + 1;
    }

    /**
     * Repeatedly folds the sheet until the desired thickness is reached
     * @param goalDistance the desired thickness (in inches)
     */
    public void foldUntil(double goalDistance) {
        while (this.thickness < goalDistance) {
            this.fold();
        }
    }

    /**
     * Accessor method for determining folds
     * @return the number of times the paper has been folded
     */
    public int getNumFolds() {
        return this.numFolds;
    }
}
```

note: only the black text is necessary code

(comments, which document the code, are shown here in blue)

8

Another example: Singer

```
/**
 * This class can be used to display various children's songs.
 * @author Dave Reed
 * @version 9/1/06
 */
public class Singer
{
    /**
     * Constructor for objects of class Singer
     */
    public Singer() {
    }

    /**
     * Displays a verse of "OldMacDonald Had a Farm"
     * @param animal animal name for this verse
     * @param sound sound that the animal makes
     */
    public void oldMacDonaldVerse(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 a " + animal + ", E-I-E-I-O");
        System.out.println("With a " + sound + "-" + sound + " here, and a " +
            sound + "-" + sound + " there, ");
        System.out.println(" here a " + sound + ", there a " + sound +
            ", everywhere a " + sound + "-" + sound + ".");
        System.out.println("Old MacDonald had a farm, E-I-E-I-O.");
        System.out.println();
    }

    . . .
}
```

a singer does not have any state, so no fields are needed

since no fields, constructor has nothing to initialize (should still have one, though)

System.out.println displays text in a window – can specify a String, a parameter name (in which case its value is displayed), or a combination using +

9

Another example: Singer (cont.)

```
. . .
/**
 * Displays the song "OldMacDonald Had a Farm"
 */
public void oldMacDonaldSong() {
    this.oldMacDonaldVerse("cow", "moo");
    this.oldMacDonaldVerse("duck", "quack");
    this.oldMacDonaldVerse("sheep", "baa");
    this.oldMacDonaldVerse("dog", "woof");
}

. . .
}
```

one method can call another one:

`this.METHOD_NAME(PARAMETERS)`

again, the "this." prefix is optional, but instructive (emphasizes that the method is being called on *this* object)

when calling a method, the parameter values match up with the parameter names in the method based on order

`this.oldMacDonaldVerse("cow", "moo");` → `animal = "cow", sound = "moo"`

`this.oldMacDonaldVerse("meow", "cat");` → `animal = "meow", sound = "cat"`

10

HW1: experimentation with SequenceGenerator

add a method to `SequenceGenerator` class to display 5 random sequences

```
/**
 * Displays 5 random letter sequence of the specified length
 * @param seqLength the number of letters in the random sequences
 */
public void displaySequences(int seqLength) {
    System.out.println(this.randomSequence(seqLength) + " " +
        this.randomSequence(seqLength) + " " +
        this.randomSequence(seqLength) + " " +
        this.randomSequence(seqLength) + " " +
        this.randomSequence(seqLength));
}
```

copy-and-paste 20 copies of the `System.out.println` statement so that the method displays 100 random sequences

- Java provides nicer means of doing this, but we will see them later

using this modified class, you will collect data to estimate the numbers of words with given characteristics

11

Examples from text: Bank Account & Cash Register

simple example: a bank account

- fields? account balance
- methods? construct an account (either with no money or a set amount)
deposit a set amount
withdraw a set amount

slightly more complex: a cash register

- fields? amount purchased (scanned) so far
amount paid so far
- methods? construct a cash register
purchase (scan) an item
pay a set amount
complete the purchase & get change

for now, we will assume the customer is honest

- customer will only enter positive amounts, will pay at least as much as purchase

12

CashRegister class

fields: maintain amounts
purchased and paid

constructor: initialize the
fields

methods: ???

```
/**
 * A cash register totals up sales and computes change due.
 * @author Dave Reed (based on code by Cay Horstmann)
 * @version 9/1/06
 */
public class CashRegister {
    private double purchase;
    private double payment;

    /**
     * Constructs a cash register with no money in it.
     */
    public CashRegister() {
        this.purchase = 0.0;
        this.payment = 0.0;
    }

    ...
}
```

13

CashRegister methods

recordPurchase:

- *mutator* method that returns adds to the purchase amount

enterPayment:

- *mutator* method that returns adds to the amount paid

giveChange:

- *mutator* method that returns the change owed to the customer (and resets the fields)

```
...
/**
 * Records the sale of an item.
 * @param amount the price of the item
 */
public void recordPurchase(double amount) {
    this.purchase = this.purchase + amount;
}

/**
 * Enters the payment received from the customer.
 * @param amount the amount of the payment
 */
public void enterPayment(double amount) {
    this.payment = this.payment + amount;
}

/**
 * Computes the change due and resets the machine
 * for the next customer.
 * @return the change due to the customer
 */
public double giveChange() {
    double change;
    change = this.payment - this.purchase;

    this.purchase = 0;
    this.payment = 0;

    return change;
}
}
```

14

Interface view of class

comments that use `/** ... */` are *documentation comments*

- BlueJ will automatically generate a documentation page from these comments
- can view the documentation by selecting *Interface* from the top-right menu

```
1  /**  
2   * A cash register totals up sales and computes change due.  
3   * @author Dave Reed (based on code by Cay Horstmann)  
4   * @version 9/1/06  
5   */  
6  
7  public class CashRegister {  
8      private double purchase;  
9      private double payment;  
10  
11     /**  
12      * Constructs a cash register with no money in it.  
13      */  
14     public CashRegister() {  
15         this.purchase = 0;  
16         this.payment = 0;  
17     }  
18  
19     /**  
20      * Records the sale of an item.  
21      * @param amount the price of the item  
22      */  
23     public void recordPurchase(double amount) {  
24         this.purchase = this.purchase + amount;  
25     }  
26  
27     /**  
28      * Enters the payment received from the customer.  
29      * @param amount the amount of the payment  
30      */  
31     public void enterPayment(double amount) {  
32         this.payment = this.payment + amount;  
33     }  
34  
35     /**  
36      * Computes the change due and resets the machine for the next customer.  
37     */  
38 }  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100  
101  
102  
103  
104  
105  
106  
107  
108  
109  
110  
111  
112  
113  
114  
115  
116  
117  
118  
119  
120  
121  
122  
123  
124  
125  
126  
127  
128  
129  
130  
131  
132  
133  
134  
135  
136  
137  
138  
139  
140  
141  
142  
143  
144  
145  
146  
147  
148  
149  
150  
151  
152  
153  
154  
155  
156  
157  
158  
159  
160  
161  
162  
163  
164  
165  
166  
167  
168  
169  
170  
171  
172  
173  
174  
175  
176  
177  
178  
179  
180  
181  
182  
183  
184  
185  
186  
187  
188  
189  
190  
191  
192  
193  
194  
195  
196  
197  
198  
199  
200  
201  
202  
203  
204  
205  
206  
207  
208  
209  
210  
211  
212  
213  
214  
215  
216  
217  
218  
219  
220  
221  
222  
223  
224  
225  
226  
227  
228  
229  
230  
231  
232  
233  
234  
235  
236  
237  
238  
239  
240  
241  
242  
243  
244  
245  
246  
247  
248  
249  
250  
251  
252  
253  
254  
255  
256  
257  
258  
259  
260  
261  
262  
263  
264  
265  
266  
267  
268  
269  
270  
271  
272  
273  
274  
275  
276  
277  
278  
279  
280  
281  
282  
283  
284  
285  
286  
287  
288  
289  
290  
291  
292  
293  
294  
295  
296  
297  
298  
299  
300  
301  
302  
303  
304  
305  
306  
307  
308  
309  
310  
311  
312  
313  
314  
315  
316  
317  
318  
319  
320  
321  
322  
323  
324  
325  
326  
327  
328  
329  
330  
331  
332  
333  
334  
335  
336  
337  
338  
339  
340  
341  
342  
343  
344  
345  
346  
347  
348  
349  
350  
351  
352  
353  
354  
355  
356  
357  
358  
359  
360  
361  
362  
363  
364  
365  
366  
367  
368  
369  
370  
371  
372  
373  
374  
375  
376  
377  
378  
379  
380  
381  
382  
383  
384  
385  
386  
387  
388  
389  
390  
391  
392  
393  
394  
395  
396  
397  
398  
399  
400  
401  
402  
403  
404  
405  
406  
407  
408  
409  
410  
411  
412  
413  
414  
415  
416  
417  
418  
419  
420  
421  
422  
423  
424  
425  
426  
427  
428  
429  
430  
431  
432  
433  
434  
435  
436  
437  
438  
439  
440  
441  
442  
443  
444  
445  
446  
447  
448  
449  
450  
451  
452  
453  
454  
455  
456  
457  
458  
459  
460  
461  
462  
463  
464  
465  
466  
467  
468  
469  
470  
471  
472  
473  
474  
475  
476  
477  
478  
479  
480  
481  
482  
483  
484  
485  
486  
487  
488  
489  
490  
491  
492  
493  
494  
495  
496  
497  
498  
499  
500  
501  
502  
503  
504  
505  
506  
507  
508  
509  
510  
511  
512  
513  
514  
515  
516  
517  
518  
519  
520  
521  
522  
523  
524  
525  
526  
527  
528  
529  
530  
531  
532  
533  
534  
535  
536  
537  
538  
539  
540  
541  
542  
543  
544  
545  
546  
547  
548  
549  
550  
551  
552  
553  
554  
555  
556  
557  
558  
559  
560  
561  
562  
563  
564  
565  
566  
567  
568  
569  
570  
571  
572  
573  
574  
575  
576  
577  
578  
579  
580  
581  
582  
583  
584  
585  
586  
587  
588  
589  
590  
591  
592  
593  
594  
595  
596  
597  
598  
599  
600  
601  
602  
603  
604  
605  
606  
607  
608  
609  
610  
611  
612  
613  
614  
615  
616  
617  
618  
619  
620  
621  
622  
623  
624  
625  
626  
627  
628  
629  
630  
631  
632  
633  
634  
635  
636  
637  
638  
639  
640  
641  
642  
643  
644  
645  
646  
647  
648  
649  
650  
651  
652  
653  
654  
655  
656  
657  
658  
659  
660  
661  
662  
663  
664  
665  
666  
667  
668  
669  
670  
671  
672  
673  
674  
675  
676  
677  
678  
679  
680  
681  
682  
683  
684  
685  
686  
687  
688  
689  
690  
691  
692  
693  
694  
695  
696  
697  
698  
699  
700  
701  
702  
703  
704  
705  
706  
707  
708  
709  
710  
711  
712  
713  
714  
715  
716  
717  
718  
719  
720  
721  
722  
723  
724  
725  
726  
727  
728  
729  
730  
731  
732  
733  
734  
735  
736  
737  
738  
739  
740  
741  
742  
743  
744  
745  
746  
747  
748  
749  
750  
751  
752  
753  
754  
755  
756  
757  
758  
759  
760  
761  
762  
763  
764  
765  
766  
767  
768  
769  
770  
771  
772  
773  
774  
775  
776  
777  
778  
779  
780  
781  
782  
783  
784  
785  
786  
787  
788  
789  
790  
791  
792  
793  
794  
795  
796  
797  
798  
799  
800  
801  
802  
803  
804  
805  
806  
807  
808  
809  
810  
811  
812  
813  
814  
815  
816  
817  
818  
819  
820  
821  
822  
823  
824  
825  
826  
827  
828  
829  
830  
831  
832  
833  
834  
835  
836  
837  
838  
839  
840  
841  
842  
843  
844  
845  
846  
847  
848  
849  
850  
851  
852  
853  
854  
855  
856  
857  
858  
859  
860  
861  
862  
863  
864  
865  
866  
867  
868  
869  
870  
871  
872  
873  
874  
875  
876  
877  
878  
879  
880  
881  
882  
883  
884  
885  
886  
887  
888  
889  
890  
891  
892  
893  
894  
895  
896  
897  
898  
899  
900  
901  
902  
903  
904  
905  
906  
907  
908  
909  
910  
911  
912  
913  
914  
915  
916  
917  
918  
919  
920  
921  
922  
923  
924  
925  
926  
927  
928  
929  
930  
931  
932  
933  
934  
935  
936  
937  
938  
939  
940  
941  
942  
943  
944  
945  
946  
947  
948  
949  
950  
951  
952  
953  
954  
955  
956  
957  
958  
959  
960  
961  
962  
963  
964  
965  
966  
967  
968  
969  
970  
971  
972  
973  
974  
975  
976  
977  
978  
979  
980  
981  
982  
983  
984  
985  
986  
987  
988  
989  
990  
991  
992  
993  
994  
995  
996  
997  
998  
999  
1000
```

```
Class CashRegister  
java.lang.Object  
↳ CashRegister  
  
public class CashRegister extends java.lang.Object  
A cash register totals up sales and computes change due.  
  
Version:  
9/1/06  
Author:  
Dave Reed (based on code by Cay Horstmann)  
  
Constructor Summary  
CashRegister()  
Constructs a cash register with no money in it.  
  
Method Summary  
void enterPayment(double amount)  
Enters the payment received from the customer.  
  
Loading class interface... Done.
```

15

More on assignments

recall that fields are assigned values using an *assignment statement*

```
this.FIELD_NAME = VALUE;
```

field, parameter, method, class, and object names are all *identifiers*:

- can be any sequence of letters, underscores, and digits, but must start with a letter
- e.g., `amount`, `recordPurchase`, `CashRegister`, `Circle`, `circle1`, ...

by convention: class names start with capital letters; all others start with lowercase when assigning a multiword name, capitalize inner words avoid underscores (difficult to read in text)

WARNING: capitalization matters, so `giveChange` and `givechange` are different!

each field in a class definition corresponds to a data value that must be stored for each object of that class

- when you create an object, memory is set aside to store that value
- when you perform an assignment, a value is stored in that memory location

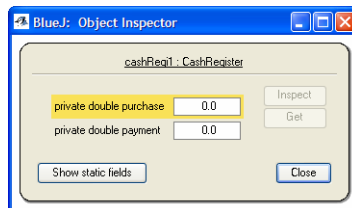
16

Variables

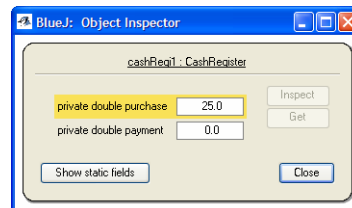
fields and parameters are examples of *variables*

- a *variable* is a name that refers to some value (which is stored in memory)
- when you assign a value to a variable, the Java interpreter finds its associated memory location and stores the value there
- if there was already a value there, it is overwritten

```
this.purchase = 0.0;
```



```
this.purchase = 25.0;
```



17

Assignments and expressions

the left-hand side of an assignment must be a variable (field or parameter);
the right-hand can be :

- a value (String, int, double, ...)

```
animal = "cow";           // the value on the right-hand side is  
this.payment = 0.0;      // assigned to and stored in the variable
```

- a variable (parameter or field name)

```
this.circleColor = color; // value represented by that variable is  
this.payment = amount;   // assigned to the field
```

- an expression using values, variables, and operators (+, -, *, /)

```
x = 2 + 3;                // can apply operators to values  
z = x - y;               // or variables  
this.num1 = this.num2 + 1; // or a combination  
  
inchesToSun = 93000000.0 * 5280 * 12; // can use more than 1 operator  
  
this.purchase = this.purchase + amount;  
// same variable can appear on both sides;  
// evaluate expression on right-hand side  
// using current value, then assign to left
```

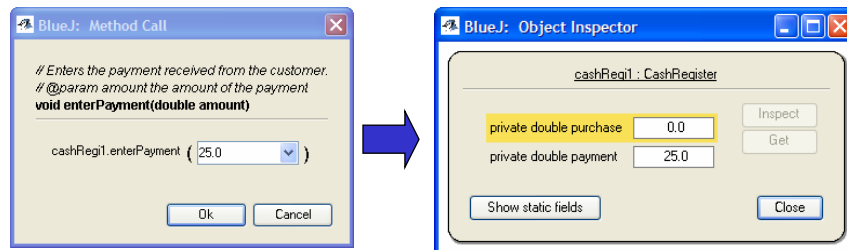
18

More on parameters

recall that a parameter represents a value that is passed in to a method

- a parameter is a variable (it refers to a piece of memory where the value is stored)
- a parameter "belongs to" its method – only exists while that method executes
- using BlueJ, a parameter value can be entered by the user – that value is assigned to the parameter variable and subsequently used within the method

```
/**
 * Enters the payment received from the customer.
 * @param amount the amount of the payment
 */
public void enterPayment(double amount) {
    this.payment = this.payment + amount;
}
```



19

Local variables

fields are one sort of variable

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

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

- they exist only as long as the method is being executed
- they are only accessible from within the method
- 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)

```
int num;
String firstName;
```

- then, can use just like any other variable (assign it a value, print its value, ...)

20

Local variable example

```
public double giveChange() {
    double change;
    change = this.payment - this.purchase;

    this.purchase = 0;
    this.payment = 0;

    return change;
}
```

you can declare and assign a local variable at the same time

- preferable since it ensures you won't forget to initialize
- the compiler will complain if you try to access an uninitialized variable

```
public double giveChange() {
    double change = this.payment - this.purchase;

    this.purchase = 0;
    this.payment = 0;

    return change;
}
```

21

When local variables?

local variables are useful when

- you need to store a temporary value (as part of a calculation or to avoid losing it)
- you are using some value over and over within a method

recall the `oldMacDonaldVerse` method from the `Singer` class

- what would have to change if we decided to spell the refrain "Eeyigh-eeyigh-oh" ?

```
public void oldMacDonaldVerse(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 a " + animal + ", E-I-E-I-O.");
    System.out.println("With a " + sound + "-" + sound + " here, and a " +
        sound + "-" + sound + " there, ");
    System.out.println(" here a " + sound + ", there a " + sound +
        ", everywhere a " + sound + "-" + sound + ".");
    System.out.println("Old MacDonald had a farm, E-I-E-I-O.");
    System.out.println();
}
```

22

A better verse...

duplication within code is dangerous

- if you ever decide to change it, you must change it everywhere!

here, we could use a local variable to store the spelling of the refrain

- `System.out.println` will use this variable as opposed to the actual text
- if we decide to change the spelling, only one change is required (in the assignment)

```
public void oldMacDonaldVerse(String animal, String sound) {
    String refrain = "E-I-E-I-O"; // change the spelling here to affect entire verse

    System.out.println("Old MacDonald had a farm, " + refrain + ".");
    System.out.println("And on that farm he had a " + animal + ", " + refrain + ".");
    System.out.println("With a " + sound + "-" + sound + " here, and a " +
        sound + "-" + sound + " there, ");
    System.out.println(" here a " + sound + ", there a " + sound +
        ", everywhere a " + sound + "-" + sound + ".");
    System.out.println("Old MacDonald had a farm, " + refrain + ".");
    System.out.println();
}
```

23

Parameters vs. local variables

parameters are similar to local variables

- they only exist when that method is executing
- they are only accessible inside that method
- they are declared by specifying type and name (no private or public modifier)
- their values can be accessed/assigned within that method

however, they differ from local variables in that

- parameter declarations appear in the header for that method
- parameters are automatically assigned values when that method is called (based on the inputs provided in the call)

parameters and local variables both differ from fields in that they belong to (and are limited to) a method as opposed to the entire object

24

Quick-and-dirty summary

a class definition consists of: fields + constructors + methods

- *fields* are the data values that define the state of an object

```
private FIELD_TYPE FIELD_NAME;
```

- *constructors* initialize the state of an object when it is created
a class can have multiple constructors with different *parameters* to initialize the state differently
in its simplest form, a constructor contains *assignments* to fields

```
this.FIELD_NAME = VALUE_TO_BE_ASSIGNED;
```

- *methods* implement the behaviors or actions for an object
when defining a method, must specify its return type (or `void` if none) and parameters (if any)
 - a *return statement* is used to return a value computed by the method

```
return VALUE_TO_BE_RETURNED;
```

- a *print or println statement* is used to display text in a console window

```
System.out.print(TEXT_MESSAGE);    System.out.println(TEXT_MESSAGE);
```

fields & parameters are examples of *variables* (names that represent values)

- *fields* are variables that belong to an object, accessible by constructors & methods
- *parameters* are variables that are passed in and exist only inside a constructor/method
- in addition, *local variables* can be defined inside constructors/methods for temporary storage

25