

# CSC 221: Computer Programming I

Fall 2004

## Understanding class definitions

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

1

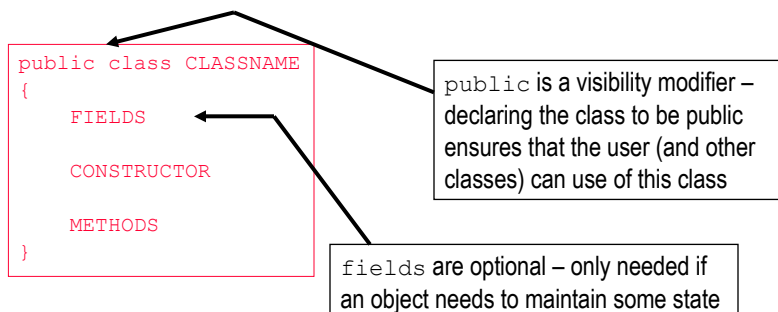
## Looking inside classes

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



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 1.0 (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()
{
    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()
    {
        diameter = 30;
        xPosition = 20;
        yPosition = 60;
        color = "blue";
        isVisible = false;
    }

    . . .
}
```

an *assignment statement* stores a value in a field

```
FIELD_NAME = VALUE;
```

here, default values are assigned for a circle

4

# Methods

methods implement the behavior of objects

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

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

draw() calls the draw method to show the circle

erase() calls the erase method to hide the circle

5

# Simpler example: Die class

```
/**  
 * This class models a simple die object, which can have any number of sides.  
 * @author Dave Reed  
 * @version 9/1/04  
 */  
  
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

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
- here, the user's input is stored in the sides parameter (of type int)
- that value is assigned to the numSides field

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 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);
}
}
```

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*

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

7

## Another example: Singer

```
/**
 * This class can be used to display various children's songs.
 * @author Dave Reed
 * @version 9/3/04
 */
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 +

8

## Another example: Singer (cont.)

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

one method can call another one

a method call consists of the method name followed by parentheses containing any parameter values

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

```
oldMacDonaldVerse("cow", "moo");    → animal = "cow", sound = "moo"
oldMacDonaldVerse("meow", "cat");    → animal = "meow", sound = "cat"
```

9

## 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(randomSequence(seqLength) + " " + randomSequence(seqLength) + " " +
        randomSequence(seqLength) + " " + randomSequence(seqLength) + " " +
        randomSequence(seqLength));
}
```

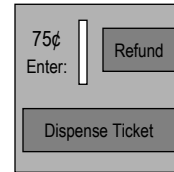
copy-and-paste 20 copies of the `System.out.println` statement so that the method displays 100 random sequences

- 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

10

## Example from text: Ticket Machine



### consider a simple ticket machine

- machine supplies tickets at a fixed price
- customer enters cash, possibly more than one coin in succession
- when correct amount entered, customer hits a button to dispense ticket
  - if not enough entered, no ticket + instructions to add more money
- customer can receive a refund if overpaid by hitting another button
- machine keeps track of total sales for the day

### for now, we will assume the customer is honest

- customer will only enter positive amounts
- customer only hits the button when the exact amount has been inserted
- so, don't need to worry about checking the amount or giving change (for now)

11

## TicketMachine class

### 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)

12

## 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;
}
}
```

13

## More on assignments

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

```
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`, `balance`, `getPrice`, `TicketMachine`, `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 `getPrice` and `getprice` are different names!

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

14

## 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

```
balance = 0;
```



```
balance = 25;
```



whenever a variable is used in code, its value is substituted

```
price = ticketCost; // value in ticketCost is assigned  
  
System.out.println("# " + price + " cents."); // value in price is displayed
```

15

## Assignments and expressions

the right-hand side of an assignment can be

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

```
animal = "cow"; // the value on the right-hand side  
balance = 0; // is assigned to and stored in the field
```

- a variable (parameter or field name)

```
circleColor = color; // value represented by that variable is  
price = ticketCost; // assigned to the field
```

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

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

16

## Assignments and expressions (cont.)

updating an existing value is a fairly common task, so *arithmetic assignments* exist as shorthand notations:

<code>x += y;</code>	is equivalent to	<code>x = x + y;</code>	
<code>x -= y;</code>	is equivalent to	<code>x = x - y;</code>	
<code>x *= y;</code>	is equivalent to	<code>x = x * y;</code>	
<code>x /= y;</code>	is equivalent to	<code>x = x / y;</code>	
<code>x++;</code>	is equivalent to	<code>x = x + 1;</code>	is equivalent to
<code>x--;</code>	is equivalent to	<code>x = x - 1;</code>	is equivalent to

e.g., `total = total + balance;` ↔ `total += balance;`

when `+` is applied to Strings, they are concatenated end-to-end

```
System.out.println("Hi " + "there."); // outputs "Hi there"
```

- when `+` is applied to a String and a number, the number is converted to a string (enclosed in `" "` then concatenated)

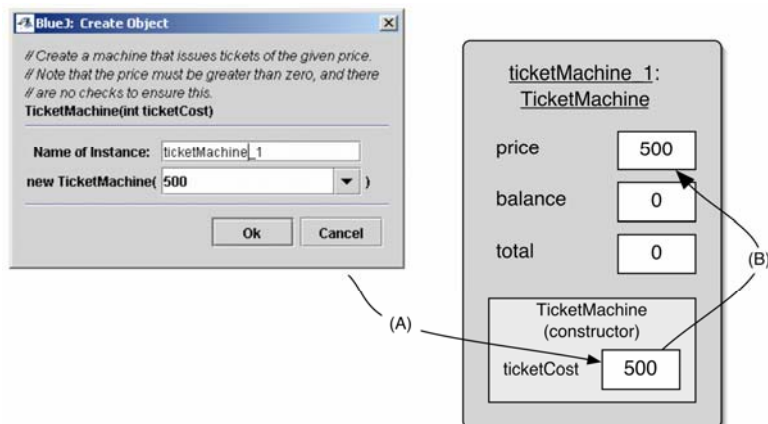
```
age = 19;  
System.out.println("You are " + age); // outputs "You are 19"
```

17

## More on parameters

recall that a parameter is 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



18

## Extending the Ticket Machine

now let us consider a fully-functional ticket machine

- will need to make sure that the user can't insert a negative amount
- will need to make sure that the amount entered  $\geq$  ticket price before dispensing
- will need to give change if customer entered too much

all of these involve *conditional* actions

- add amount to balance *only if it is positive*
- dispense ticket *only if amount entered  $\geq$  ticket price*
- give change *only if amount entered  $>$  ticket price*

in Java, an *if statement* allows for conditional execution

- i.e., can choose between 2 alternatives to execute

```
if (perform some test) {  
    Do the statements here if the test gave a true result  
}  
else {  
    Do the statements here if the test gave a false result  
}
```

19

## Conditional execution via if statements

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

if the test evaluates to true ( $\text{amount} > 0$ ), then this statement is executed

otherwise ( $\text{amount} \leq 0$ ), then this statement is executed to alert the user

you are not required to have an else case to an if statement

- if no else case exists and the test evaluates to false, nothing is done
- e.g., could have just done the following

```
public void insertMoney(int amount)  
{  
    if(amount > 0) {  
        balance += amount;  
    }  
}
```

but then no warning to user if a negative amount were entered (not as nice)

20

## Relational operators

standard relational operators are provided for the if test

<	less than	>	greater than
<=	less than or equal to	>=	greater than or equal to
==	equal to	!=	not equal to

a comparison using a relational operator is known as a *Boolean expression*, since it evaluates to a *Boolean* (true or false) value

```
public void printTicket()
{
    if (balance >= price) {
        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 += price;
        balance -= price;
    }
    else {
        System.out.println("You must enter at least: " + (price - balance) + " cents.");
    }
}
```

21

## Multiway conditionals

a simple if statement is a 1-way conditional (*execute this or not*)

```
if (number < 0) {                // if number is negative
    number = -1 * number;        // will make it positive
}                                 // otherwise, do nothing and move on
```

an if statement with else case is a 2-way conditional (*execute this or that*)

```
if (number < 0) {                // if number is negative
    System.out.println("negative"); // will display "negative"
}                                 //
else {                            // otherwise,
    System.out.println("non-negative"); // will display "non-negative"
}
```

if more than 2 possibilities, can "cascade" if statements

```
if (number < 0) {
    System.out.println("negative");
}
else {
    if (number > 0) {
        System.out.println("positive");
    }
    else {
        System.out.println("zero");
    }
}
```

can omit some braces and line up neatly:

```
if (number < 0) {
    System.out.println("negative");
}
else if (number > 0) {
    System.out.println("positive");
}
else {
    System.out.println("zero");
}
```

22

## 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, ...)

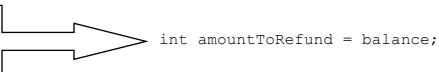
23

## New method: refundBalance

```
/**  
 * Refunds the customer's current balance and resets their balance to 0.  
 */  
public int refundBalance()  
{  
    int amountToRefund;           // declares local variable  
  
    amountToRefund = balance;     // stores balance in local variable so  
    balance = 0;                 // not lost when balance is reset  
    return amountToRefund;       // returns original balance  
}
```

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

```
int amountToRefund;  
amountToRefund = balance;
```



24

## 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-eeigh-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();
}
```

25

## 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();
}
```

26

## 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

27

## HW2: ESPTester

for HW2, you will complete a class that allows the user to test for ESP

you are given:

- `public ESPTester(int max)` constructor that creates a tester where the numbers to be guessed are in the range 1..max
- `public String guessNumber(int guess)` method that takes the user's guess, picks a random int in the range 1..max, and tells the user whether they guessed correctly

you will add:

- accessor method so that user can determine the maximum number
- fields to keep track of the # of guesses, # of correct guesses
- accessor methods for these fields
- methods to compute the percentage of correct guesses, compare with expected

28