CSC 222: Object-Oriented Programming

Fall 2017

Understanding class definitions

- class structure
- fields, constructors, methods
- parameters
- shorthand assignments
- local variables
- final-static fields, static methods

Looking inside classes

recall that classes define the properties and behaviors of its objects

a class definition must:

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

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 2011.07.31
 */

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

    . . .
}
```

in general, the constructor should contain an assignment for every field

- provides initial values for all of the properties

when referring to a field, the `this.` prefix is optional

- makes it clear that the variable is a field & belongs to this particular object
- I will use consistently in my code & strongly recommend you do so too

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() {
      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 drawn on the Canvas

note that one method can "call" another one

`this.draw()` calls `draw` method (on this object) to show it

`this.erase()` calls `erase` method (on this object) to hide it

5

Simpler example: Die class

note: comments in gray

```
/**
 * This class models a simple die object, which can have any number of sides.
 *   @author Dave Reed
 *   @version 1/20/2017
 */
public class Die {
  private int numSides;

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

  /**
   * 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;
  }

  /**
   * 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() {
    return (int)(Math.random()*this.numSides + 1);
  }
}
```

a `Die` object needs to keep track of its number of sides

the constructor takes the # of sides as a parameter

`getNumberOfSides` is an accessor method – enables look at private field

`roll` is a mutator method – changes the state of the object

6

Java class vs. Python class

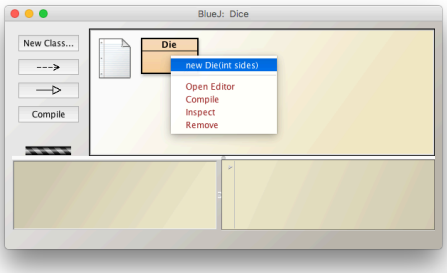
```
public class Die {
  private int numSides;

  public Die(int sides) {
    this.numSides = sides;
  }

  public int getNumberOfSides() {
    return this.numSides;
  }

  public int roll() {
    return (int)(Math.random()*this.numSides) + 1;
  }
}
```

```
class Die:
    def __init__(self, sides):
        self.numSides = sides

    def getNumberOfSides(self):
        return self.numSides

    def roll(self):
        return randint(1, self.numSides)
```

- Java provides accessibility options (all public in Python), so must specify
- Java variables (fields & parameters) are committed to one type of value, so must declare name & type
- methods that return values must specify the type
- constructor is similar to `__init__`
- `this.` in Java is similar to `self.` (but don't specify as method parameter)

7

In BlueJ

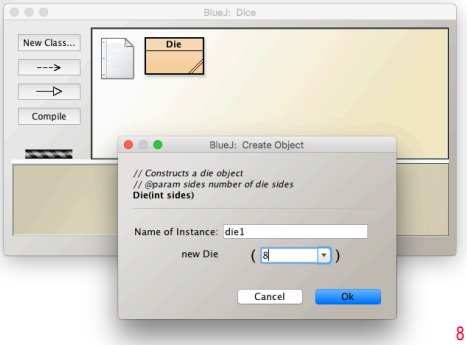


create an object from a class by:

- right-clicking on the class icon
- selecting the constructor

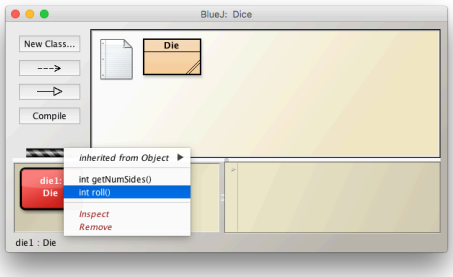
you will be prompted for:

- the name of the object
- values for any parameters



8

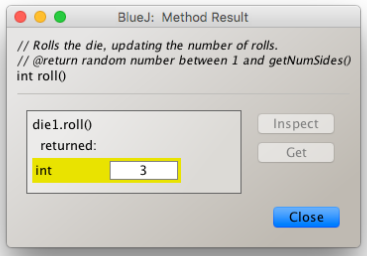
In BlueJ



the created object will appear as a red icon in the lower left

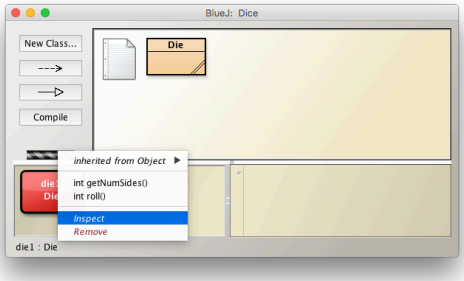
- right-clicking on the object icon to select and call methods

if the method returns a value, it will appear in a pop-up window



9

In BlueJ



BlueJ has a nice feature that allows you to see the internal state of an object

- right-click and select *Inspect*

a pop-up window will show the values of all fields

10

Variation: adding more state

```
public class Die {
  private int numSides;
  private int numRolls;

  public Die(int sides) {
    this.numSides = sides;
    this.numRolls = 0;
  }

  public int getNumberOfSides() {
    return this.numSides;
  }

  public in getNumberOfRolls() {
    return this.numRolls;
  }

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

we might want to have another field to keep track of the number of times the die has been rolled

- add a private field (`numRolls`)
- initialize it in the constructor
- provide an accessor method
- update it in `roll` method

11

Variation: adding another constructor

```
public class Die {
  private int numSides;
  private int numRolls;

 public Die() {
    this.numSides = 6;
    this.numRolls = 0;
  }

  public Die(int sides) {
    this.numSides = sides;
    this.numRolls = 0;
  }

  public int getNumberOfSides() {
    return this.numSides;
  }

  public in getNumberOfRolls() {
    return this.numRolls;
  }

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

since 6-sided dice are most common, we might want a simpler way of creating them

a class can have more than one constructor – useful if there are different ways to create/initialize an object

- add a default constructor (with no parameters) for 6-sided dice
- if a class has multiple constructors, they must differ in their parameters

12

Variation: linking constructors

```
public class Die {
  private int numSides;
  private int numRolls;

 public Die() {
    this(6);
  }

  public Die(int sides) {
    this.numSides = sides;
    this.numRolls = 0;
  }

  public int getNumberOfSides() {
    return this.numSides;
  }

  public in getNumberOfRolls() {
    return this.numRolls;
  }

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

within a class a method can call another method

- likewise, a constructor can call another constructor

here, the default constructor can just call the other constructor with 6 as parameter

13

Building complexity

```
/**
 * Class that models a fair coin.
 *   @author Dave Reed
 *   @version 8/30/17
 */
public class Coin {
    private Die d2;

    public Coin() {
       this.d2 = new Die(2);
    }

    public String flip() {
        if (this.d2.roll() == 1) {
            return "HEADS";
        }
        else {
            return "TAILS";
        }
    }

    public int getNumberOfFlips() {
        return this.d2.getNumberOfRolls();
    }
}
```

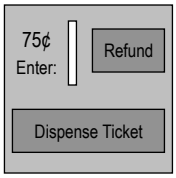
recall: can have an object as a field

here, can model a coin using a 2-sided die

- flip method converts 1 or 2 roll into HEADS or TAILS
- getNumberOfFlips method makes use of getNumberOfRolls

14

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
- once the required amount entered, customer hits a button to dispense ticket
- machine keeps track of total sales for the day

- properties/fields:

  ticket price, balance entered, total amount for day

- behaviors/methods:

  insert money, print ticket, get balance, get ticket price, get total sales

15

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

insertMoney:

- *mutator* method that adds to the customer's balance (or displays warning if not a positive amount)

```
/**
 * This class simulates a simple ticket vending machine.
 *   @author  Barnes & Kolling (modified by Dave Reed)
 *   @version 9/2/17
 */
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) {
    this.price = ticketCost;
    this.balance = 0;
    this.total = 0;
  }

 /**
   * 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)
  {
    if (amount > 0) {
        this.balance = this.balance + amount;
    }
    else {
        System.out.println("Must be a positive amount!");
    }
  }
  . . .
}
```

16

TicketMachine methods

getPrice:
- *accessor* method that returns the ticket price

getBalance:
- *accessor* method that returns the balance

getTotal:
- *accessor* method that returns the total

printTicket:
- simulates the printing of a ticket (or displays warning if not enough entered)

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

/**
 * Accessor method for the amount inserted.
 *   @return the amount inserted so far
 */
public int getBalance() {
  return this.balance;
}

/**
 * Accessor method for the total amount.
 *   @return the total amount inserted so far
 */
public int getTotal() {
  return this.total;
}

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

    this.total = this.total + this.price;
    this.balance = this.balance - this.price;
  }
  else {
    System.out.println("You must enter at least: " +
                       (this.price - this.balance) + " cents.");
  }
 }
}
```

17

Shorthand assignments

recall: the right-hand side of an assignment can be
- a value (`String`, `int`, `double`, ...) `this.balance = 0;`
- a variable (parameter or field name) `this.price = cost;`
- an expression using (+, -, *, /) `this.balance = this.balance + amount;`

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

`x += y;`	is equivalent to	`x = x + y;`			
`x -= y;`	is equivalent to	`x = x - y;`			
`x *= y;`	is equivalent to	`x = x * y;`			
`x /= y;`	is equivalent to	`x = x / y;`			
`x++;`	is equivalent to	`x = x + 1;`	is equivalent to	`x += 1;`	
`x--;`	is equivalent to	`x = x - 1;`	is equivalent to	`x -= 1;`	

+= works for Strings as well

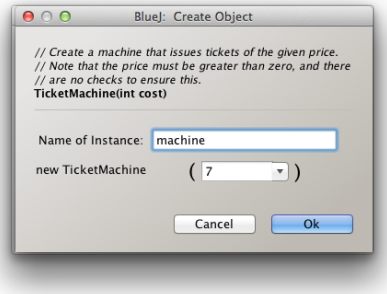
`str += "!";` is equivalent to `str = str + "!";`

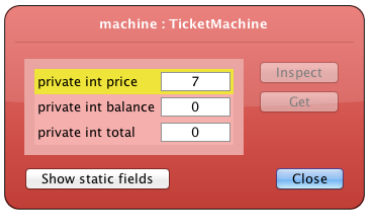
18

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





19

Local variables

fields are variables that

- are initialized by the constructor
- exist throughout the life of the object
- are accessible throughout the class

parameters are variables that

- are initialized when the method is called
- exist only while the method executes
- are accessible only within that method

in addition, methods can include short-lived *local variables*

- are declared and initialized when needed in the method
- exist only as long as the method is being executed
- are accessible only within that method
- local variables are useful whenever you need to store some temporary value (e.g., in a complex calculation)

20

New method: refundBalance

suppose we want a method to refund the ticket balance

- need to return the balance amount and reset balance to 0

```
/**
 * Refunds the customer's current balance and resets their balance to 0.
 *   @return the balance refund amount
 */
public int refundBalance() {
  return this.balance;
  this.balance = 0;
}
```

is this OK?

when a return statement is reached, the method terminates

- the Java compiler won't even allow a statement after a return!
- here, need a local variable

```
/**
 * Refunds the customer's current balance and resets their balance to 0.
 *   @return the balance refund amount
 */
public int refundBalance() {
  int amountLeft = balance;
  this.balance = 0;
  return amountLeft;
}
```

21

example: Pig

Pig is a simple, children's dice game

- 2 players take turns rolling a die
- on each turn, a player can make as many rolls as they want
  - but, if they roll 1, the turn is over and the player earns 0 points
  - if they choose to stop before rolling 1, they earn the total of rolls in that turn
- first player to 100 points wins

- e.g., possible turns

4	6	1
2	3	bust
6	1	*earns 0 points*
stop	4	
earns 12 points	5	
	1	
	bust	
	earns 0 points	

22

Pig analysis?

what is the best strategy for playing Pig?

when do you stop? when do you risk rolling again?

does it depend on the score or is it consistent?

once we have a strategy in mind, how do we test its effectiveness?

- the Law of Large numbers states that as the number of experiments approaches infinity, the results will converge on the expected outcome
- in plain English, the more experiments you conduct, the better (closer to the expected probability) your results

how many games would you have to play to be confident in your results?

computer are perfect for carrying out tedious, repetitive simulations

- we can build a model of a game & perform thousands or millions of simulations in seconds
- we can get results as accurate as we need them to be

23

Designing a PigGame class

we want to build a class that can simulate Pig games & maintain statistics

- for now, let's assume consistent (score-independent) strategies only
  e.g., stop rolling if your turn total exceeds 15
- fields/properties?
- constructor?
- methods/behaviors?

recall: end goal is to be able to determine the average number of turns it takes to reach 100, given a specific strategy

- can then compare different strategies to determine the best one

24

PigGame

many options for fields:

- `Die` for rolling
- threshold for when to stop
- points total?
- number of turns?

ideally, keep fields to a minimum

- assume each `PigGame` has its own cutoff (specified in constructor)
- don't need points or turns, methods can return these
- for testing purposes, can print each roll/round
- then comment out so not inundated with output

```
public class PigGame {
    private Die roller = new Die();
    private int threshold;

    public PigGame(int cutoff) {
        this.threshold = cutoff;
        this.roller = new Die();
    }

    public int playTurn() {
        int turnPoints = 0;

        while (turnPoints < this.threshold) {
            int roll = this.roller.roll();
            //System.out.println(roll);
            if (roll == 1) {
                return 0;
            }
            else {
                turnPoints += roll;
            }
        }
        return turnPoints;
    }

    public int playGame() {
        int totalPoints = 0;
        int turn = 0;
        while (totalPoints < 100) {
            totalPoints += this.playTurn();
            //System.out.println(totalPoints);
            turn++;
        }
        return turn;
    }
}
```

25

PigGame

it is bad practice to have "magic numbers" in code

- in 6 months, will you remember what the 100 in `playGame` represents?
- how easy would it be to change the game length?

better solution – define a *constant* at the top of the class

- a constant is a field that is declared to be "final static"

final → once assigned, it cannot be changed (so SAFE)

static → shared by all objects of the class (so EFFICIENT)

since the field belongs to the class, specify the class name to the left of the dot

```
public class PigGame {
    private final static int GOAL_POINTS = 100;

    private Die roller = new Die();
    private int threshold;

    . . .

    public int playGame() {
        int totalPoints = 0;
        int turn = 0;
        while (totalPoints < PigGame.GOAL_POINTS) {
            totalPoints += this.playTurn();
            //System.out.println(totalPoints);
            turn++;
        }
        return turn;
    }
}
```

26

PigGame (cont.)

how do we collect statistics?

- could call `playGame` repeatedly, keep track ourselves
- or, could add a method for simulating repeated games
  keeps a running total of all rounds, then calculates average per game

```
/**
 * Simulates repeated games of 1-player Pig.
 *   @param numGames the number of games to simulate (numGames > 0)
 *   @param cutoff the point total at which the player holds (cutoff > 0)
 *   @return the average number of rounds required to win
 */
public double averageLengthOfGame(int numGames) {
    int totalRounds = 0;
    for (int i = 0; i < numGames; i++) {
        totalRounds += this.playGame();
    }
    return (double)totalRounds/numGames;
}
```

27

PigStats driver class

while adding the stats generation method to `PigGame` works, it is not ideal

- generating stats about repeated games is not the job of the game itself
- better to create another class whose job is to simulate repeated games and report stats
  can use a constant to specify the number of games for each simulation

```
public class PigStats {
    private final static int NUM_GAMES = 1000000;

    public double averageLengthOfGame(int cutoff) {
        PigGame game = new PigGame(cutoff);

        int totalRounds = 0;
        for (int i = 0; i < PigStats.NUM_GAMES; i++) {
            totalRounds += game.playGame();
        }
        return (double)totalRounds/PigStats.NUM_GAMES;
    }

    public void showGameStats(int lowCutoff, int highCutoff) {
        for (int i = lowCutoff; i <= highCutoff; i++) {
            PigGame game = new PigGame(i);
            double avg = game.averageLengthOfGame(PigStats.NUM_GAMES);
            System.out.println(i + ": " + avg + " turns");
        }
    }
}
```

note: no fields & no constructor
WHY?

28

Static methods

if a class has no fields, all objects of that class would be identical

➔ that means you never need to create more than one object of that class

if that is the case, make the methods static

since static methods class belong to the class, can call directly from the con in BlueJ (saves the extra step of creating an object and then calling the method on that object)

```
public class PigStats {
    private final static int NUM_GAMES = 1000000;

    public static double averageLengthOfGame(int cutoff) {
        PigGame game = new PigGame(cutoff);

        int totalRounds = 0;
        for (int i = 0; i < PigStats.NUM_GAMES; i++) {
            totalRounds += game.playGame();
        }
        return (double)totalRounds/PigStats.NUM_GAMES;
    }

    public static void showGameStats(int lowCutoff, int highCutoff) {
        for (int i = lowCutoff; i <= highCutoff; i++) {
            PigGame game = new PigGame(i);
            double avg = game.averageLengthOfGame(PigStats.NUM_GAMES);
            System.out.println(i + ": " + avg + " turns");
        }
    }
}
```

29