Software Development (CS2500)

Lecture 19: Class Design (Continued)

M. R. C. van Dongen

November 4, 2013

Relation with Owner

- `Java` has *class* and *instance* methods.
- It also has *class* and *instance* variables.
- *Class methods & class variables* are owned by the class.
  - There is one method/variable per class.
- *Instance methods & instance variables* are owned by instances.
  - There is one method/variable per instance of the class.

Encapsulation

- Consider a properly encapsulated (`private`) attribute, `attr`.
  - The attribute is only visible inside the class.
- Consider an instance, `instance`, of the defining class.
- You can only access `instance.attr` if:
  - You're inside an instance method that was called using `instance`.
  - You have access to the reference `instance`.

Using the Instance Method

Java

```
public class Example {
    private int attribute;

    public Example( int initialValue ) {
        attribute = initialValue;
    }

    public static void main( ) {
        final Example example = new Example( 42 );
        example.method( );
    }

    private void method( ) {
        System.out.println( attribute );
    }
}
```

Using the Instance Method

Java

```
public class Example {
    private int attribute;

    public Example( int initialValue ) {
        attribute = initialValue;
    }

    public static void main( ) {
        final Example example = new Example( 42 );
        example.method( );
    }

    private void method( ) {
        System.out.println( this.attribute );
    }
}
```

Using the Object Reference

Java

```
public class Example {
    private int attribute;

    public Example( int initialValue ) {
        attribute = initialValue;
    }

    public static void main( ) {
        final Example good = new Example( 42 );
        final Example bad = new Example( 666 );

        bad.method( good );
        method( good, bad );
    }

    private void method( final Example object ) {
        object.attribute = 666;
    }

    private static void method( final Example first, final Example second ) {
        System.out.println( (first.attribute + second.attribute) );
    }
```

Notation for Class

- The notation for class methods depends on where “you” are.
- You may always write ‘⟨`class`⟩`.`⟨`method`⟩`(` ⟨`arguments`⟩ `)`.’
- In the defining class you may write ‘⟨`method`⟩`(` ⟨`arguments`⟩ `)`.’
- Same for variables: you may always write ‘⟨`class`⟩`.`⟨`variable`⟩.’
- Inside the defining class you may also write ‘⟨`variable`⟩.’

Example

Java

```
public class Inside {
    public static int attribute;

    public static void method( ) {
        int var1 = attribute;
        int var2 = Inside.attribute;
        System.out.println( var1 + " = " + var2 );
    }
}
```

Java

```
public class Outside {
    public static void method( ) {
        // System.out.println( attribute ); // Not allowed.
        System.out.println( Inside.attribute );
    }
}
```

Notation for Instances

- The notation for instance variables and methods is similar.
- You may always use ‘⟨`reference`⟩`.`⟨`method`⟩`(` ⟨`arguments`⟩ `)`.’
- You may use ‘⟨`reference`⟩`(` ⟨`arguments`⟩ `)`’ in defining class.
- For attributes you may write ‘⟨`reference`⟩`.`⟨`variable`⟩.’
- But inside the defining class you may also write ‘⟨`attribute`⟩.’

Notation for Instances: Contined

- The dotless notation is only allowed inside instance methods.
- Inside instance methods you use '`this`' for the "current" object.
- Using '⟨`instance variable`⟩' without dot-notation means
  - '`this.`⟨`instance variable`⟩`.`'
- For instance methods this is the same.
- So '⟨`instance method`⟩`(` ⟨arguments⟩ `)`' means
  - '`this.`⟨`instance method`⟩`(` ⟨arguments⟩ `).`'

Example

Java

```
public class Inside {
    private int attribute;

    private static void classMethod( int var ) {
        System.out.println( var );
    }

    public void instanceMethod1( ) {
        classMethod( attribute );
    }

    public void instanceMethod2( ) {
        classMethod( this.attribute );
    }
}
```

Java

```
public class Outside {
    public static void main( String args[] ) {
        Inside inside = new Inside( );
        inside.instanceMethod1( );
        inside.instanceMethod2( );
    }
}
```

Simulating Instance Methods

Java

```
public class Simulation {
    private int attribute;

    public static void classMethod( Simulation current ) {
        System.out.println( current.attribute );
    }

    public void instanceMethod( ) {
        classMethod( this );
    }
}
```

Java

```
public class Main {
    public static void main( String args[] ) {
        Simulation simulation = new Simulation( );
        // The following calls are effectively identical.
        simulation.instanceMethod( );
        Simulation.classMethod( simulation );
    }
}
```

Common Patterns

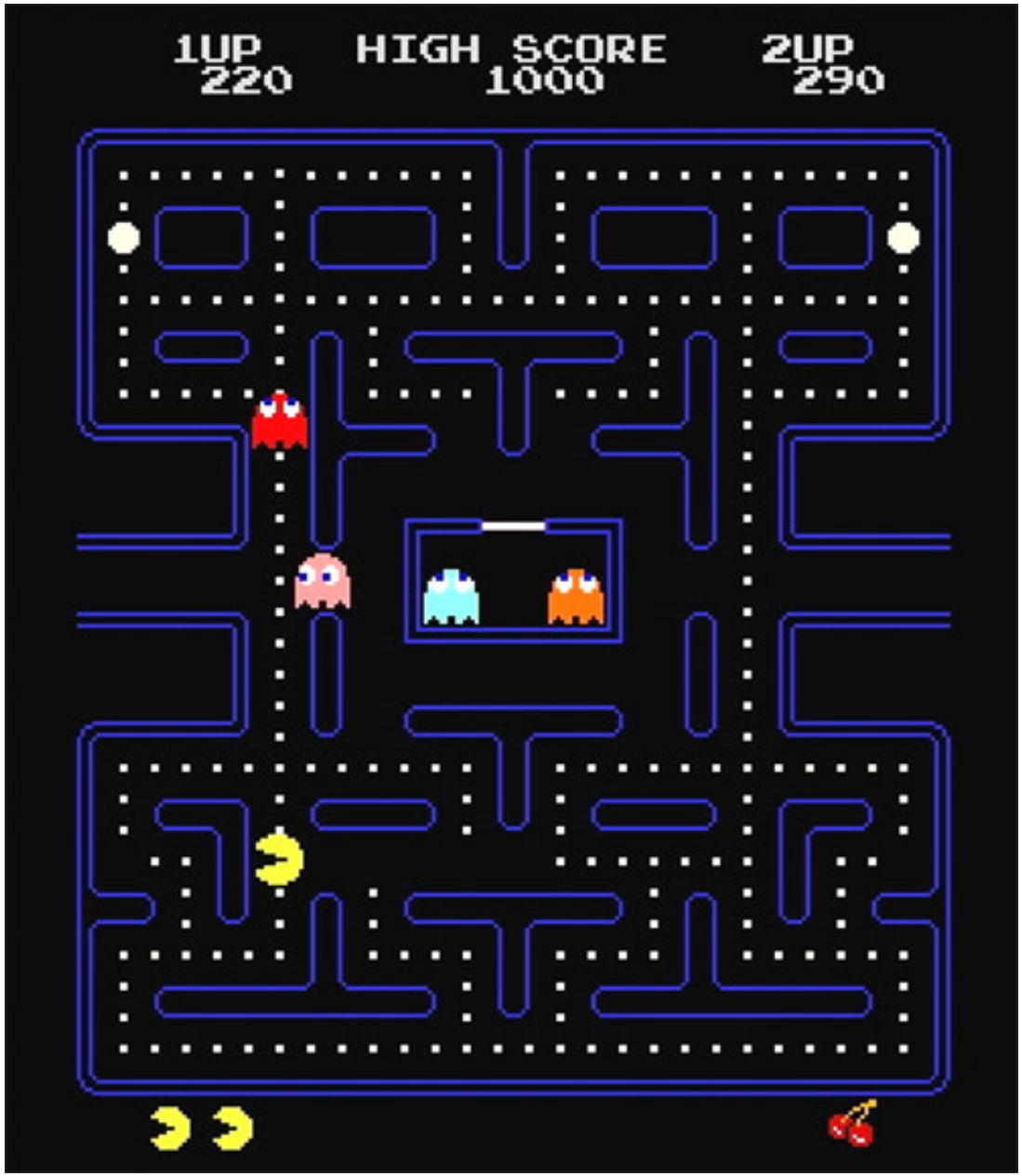
- Keeping a total;
- Counting object events;
- Collecting Values;
- Managing object properties;
- Modelling object state;
- Navigating 2-d space.

Keeping a Total

- Bank account's balance:
  - Increases when moneys are deposited;
  - Decreases when moneys are withdrawn.
- Cash register's total:
  - Increases when goods are sold;
  - Cleared at the end of the day.
- Student's total CA mark:
  - Increases when a new assignment has been processed.

Solution Template

- Introduce *instance* attribute for the total;
- Define getter and setter methods to adjust the total.



Implementation

Java

```
public class PacMan {
    private static int NORMAL_SCORE_INCREMENT = 1;
    private static int BONUS_SCORE_INCREMENT = 10;
    private int score;

    public PacMan( ) {
        score = 0;
    }

    private void setScore( final int value ) {
        score = value;
    }

    private void getScore( ) {
        return score;
    }

    private void addToScore( final int increment ) {
        setScore( score + increment );
    }

    // omited

}
```

Implementation

Java

```
public class PacMan {
    private static int NORMAL_SCORE_INCREMENT = 1;
    private static int BONUS_SCORE_INCREMENT = 10;
    private int score;

    // omited

    public void eatPacDot( ) {
        addToScore( NORMAL_SCORE_INCREMENT );
    }

    public void setBonus( ) {
        addToScore( BONUS_SCORE_INCREMENT );
    }

    public void die( ) {
        setScore( 0 );
    }

}
```

Counting Object Events

- Count number of bank transactions;
- Keep track of the number of seconds on the clock;
- ...

Solution Template

- Introduce *instance* attribute for the total;
- Define getter, setter, and increment methods to adjust the total.

Implementation

Java

```
public class PacMan {
    private static int NORMAL_SCORE_INCREMENT = 1;
    private static int BONUS_SCORE_INCREMENT = 10;
    private int score;
    private int levels;

    public PacMan( ) {
        score = INITIAL_SCORE;
        levels = 0;
    }

    public int getLevels( ) {
        return levels;
    }

    public void incrementLevels( ) {
        levels++;
    }

    // omited
}
```

Collecting Values

- Collect choices for multiple choice questions;
- Collect strategies for characters in a game;
- Collect best solutions to a problem;
- ...

Solution Template

- Introduce *instance* attribute to store the options:
  - Array or `ArrayList`.
- Store the options.

Implementation

Java

```
pubnlic class Question {
    private final ArrayList<Choice> choices;

    public Question( ) {
        choices = new ArrayList<Choice>( );
    }

    public void add( final Choice choice ) {
        choices.add( choice );
    }

    // omitted
}
```

Managing Object Properties

- Name and ID of a student;
- Licence number and owner of a car;
- Species and price of a pet in a pet shop;
- ...

Solution Template

- Introduce an *instance* attribute for each property;
- Define getter and setter methods for the attribute.

Implementation

Java

```
public class Student {
    private final String name;
    private final String id;

    public Student( final String name ) {
        this.name = name;
        id = nextId( );
    }

    private static int nextId( ) {
        return ...;
    }

    // omitted
}
```

Modelling Object State

- Implement behaviour that depends on the current state.
- Use the behaviour and current state to set the future state.
- Example:
  - A shark is in a *very hungry* state;
  - Because it's very hungry, it tries hard to find a prey;
  - When the shark catches a prey, it eats it;
  - This size of the prey determines the next state;
  - If it is a big prey, the state will change to *satisfied;*
    - With this state there's no need to look for prey;
  - If it is a small prey, the state will change to *somewhat hungry;*
    - With this state the shark will keep looking for prey.

Solution Template

- Introduce a *class* constant for each state;
- Each constant should have a different value;
  - For the moment we use `ints` for the constants;
  - Later we study a *much* better way: enumerated values.
- Introduce an *instance* attribute to represent the current state;
- Determine how the current state affects the current behaviour;
- Determine how the behaviour afects the next state.

Implementation

For the Moment Use `int`s for States

Java

```
public class Shark {
    private static final int SATISFIED = 0;
    private static final int SOMEWHAT_HUNGRY = 1;
    private static final int VERY_HUNGRY = 2;

    private int hungerLevel;

    public void eat( final Fish fish ) {
        if (fish.isBig( )) {
            hungerLevel = SATISFIED;
        } else if (hungerLevel == SOMEWHAT_HUNGRY) {
            hungerLevel = SATISFIED;
        } else {
            hungerLevel = SOMEWHAT_HUNGRY;
        }
    }

    // omitted

}
```

Implementation

For the Moment Use ints for States

Java

```
public class Shark {
    private static final int SATISFIED = 0;
    private static final int SOMEWHAT_HUNGRY = 1;
    private static final int VERY_HUNGRY = 2;

    private int hungerLevel;

    // omitted

    public void move( ) {
        if (hungerLevel == VERY_HUNGRY) {
            // very actively look for prey
        } else if (hungerLevel == SOMEWHAT_HUNGRY) {
            // look for prey but not too hard
        } else {
            // relax
        }
    }

}
```

Representing Location

- Keep track of position of ambulance;
- Keep track of position of plane;
- Determine shortest path from A to B;
- ...

Solution Template

- Introduce proper class to represent the position;
  - May depend on application;
  - E.g. 2-d versus 3-d.
- Introduce *instance* attribute to represent object's position;
- Define getter and setter methods to adjust the position.

Implementation

Java

```
public class TwoDCoordinate {
    private double x;
    private double y;

    public TwoDCoordinate( final double x, final double y ) {
        this.x = x;
        this.y = y;
    }

    public void moveInDirection( final TwoDCoordinate direction ) {
        this.x += direction.x;
        this.y += direction.y;
    }

    // omitted
}
```

Implementation

Java

```
public class Ambulance {
    private final TwoDCoordinate location;
    private final TwoDCoordinate baseLocation;

    public Ambulance( final TwoDCoordinate base ) {
        location = base;
        baseLocation = base;
    }

    public void callOut( final TwoDCoordinate target ) {
        moveTo( target );
    }

    public void returnToBase( ) {
        moveTo( baseLocation );
    }

    public void moveTo( final TwoDCoordinate target ) {
        while (!location.equals( target )) {
            location.moveInDirection( nextDirection( ) );
        }
    }

    // omitted
}
```

Two Kinds of Methods

`void`: Compute but don't return value.

- Drivers of other computations.

Non-`void`: Compute and return value.

- Including object reference values.

Side Effects

- Class methods may change class variables.

Java

```
public class MySystem {
    private static int printlnCalls;

    public static void println( String str ) {
        printlnCalls ++;
        System.out.println( str );
    }

    public static int getPrintlnCalls( ) {
        return printlnCalls;
    }
}
```

- Instance methods may change class and instance variables.
- Side effects may be caused implicitly by submethod calls.

Managing Computations

- A method should organise and manage its computation.
- Statements should be in natural order: first this, then that, ...
- Each sub-computation should be well-defined.
- This includes the input, output, and purpose of the computation.
- Computations prepare input of subsequent computation.
- Improve the clarity and flow of your computations:
  - Avoid `return` statements in `void` methods.
  - Avoid more than one `return` statement in non-`void` methods.

Example #1

Don't Try This at Home

```
public static void print( boolean condition, String str ) {
    if (!condition) {
        return;
    }
    System.out.println( str );
}
```

Example #1

Don't Try This at Home

```
public static void print( boolean condition, String str ) {
    if (!condition) {
        return;
    }
    System.out.println( str );
}
```

Java

```
public static void print( boolean condition, String str ) {
    if (condition) {
        System.out.println( str );
    }
}
```

Example #2

Java

```
public static int fib( int n ) {


    if (n <= 1) {
        return 1;
    } else {
        return fib( n - 1 ) + fib( n - 2 );
    }


}
```

Example #2

Java

```
public static int fib( int n ) {
    final int result;

    if (n <= 1) {
        result = 1;
    } else {
        result = fib( n - 1 ) + fib( n - 2 );
    }

    return result;
}
```

A Computation should have one Entry/Exit Point

Don't Try This at Home

```
while (condition1) {
    if (condition2) {
        break;
    } else {
        ⟨stuff⟩
    }
}
```

A Computation should have one Entry/Exit Point

Don't Try This at Home

```
while (condition1) {
    if (condition2) {
        break;
    } else {
        ⟨stuff⟩
    }
}
```

Java

```
while ((condition1) && (!condition2)) {
    ⟨stuff⟩
}
```

Strive for Simplicity

- Aim for Simplicity.
- By adopting this rule, the quality of your methods will improve.
- It makes your methods easier to write and maintain.

Writing Method Definitions

- Method definitions should be short and concise.
- Write sub-computations using short, simple statements.
- No method should be longer than approximately 40 lines.
- If a method is too long, introduce submethod calls.

Developing an Algorithm

- Your methods should be well-defined: input, output, and task.
- Developing a complex algorithm is an art, not a science.
- Stay in control by developing it in a *top-down* fashion.
  - Start with a coarse version of the algorithm.
  - You implement the algorithm as a method.
  - You implement the basic steps in pseudo-code.
  - When the design looks right: use submethods calls & basic `Java`.
  - You *refine* your methods until you end up with basic `Java`.

Developing the Method

- If the task requires a few simple statements:
  - Write it down using simple statements.
- If the task requires many statements or is not easy to formulate:
  - Think of sub-computations that drive the overall computation.
  - Initially you state sub-computations using *pseudo-code*.
  - Each pseudo-code statement should be well defined.
  - This includes the main task, input, and output.
  - Easy pseudo-computations can be implemented "directly:"
    - Either you use existing methods/classes/libraries.
    - Or you write simple statements without method calls.
  - Implement complex sub-computations as submethod calls.
  - Continue the design in a "recursive" fashion.

Requirements

- Two contestants are playing a game.
- The names of the contestants are John and Paul.
- The contestants compete until there's a winner.
- The program announces the winner and the final score.
- The match is won by the first player who wins 3 sets.
- The rules for playing a set are as follows:
  - Each player starts with 0 games won.
  - The players play a sequence of games.
  - The first player who wins 6 games wins the set.
- The rules for playing a game are as follows:
  - Each game has its own referee.
  - The game consists of a sequence of rounds.
  - The game is won by the first player that wins a round without ties.
  - Each round the referee and each player choose a random `boolean`.
  - A tie occurs if both players guess the same `boolean`.
  - Otherwise, the winner is the player that guesses the same `boolean` as the referee.

PseudoCode

```
Contestant john = new Contestant( "John" );
Contestant paul = new Contestant( "Paul" );

// Determine winner and loser of match.

// Announce winner and score.
```

PseudoCode

```
Contestant john = new Contestant( "John" );
Contestant paul = new Contestant( "Paul" );

// Determine winner and loser of match.

// Announce winner and score.
```

- It seems natural to introduce a method `playMatch( )` that plays a match and returns the winner.
- Implementing "Announce winner and score" seems trivial.
  - Depends on implementation of `Contestant` and `playMatch( )`.

playMatch( )

Two Options: Instance Method or Class Method

- `Contestant playMatch( Contestant player1, Contestant player2 ).`
- `Contestant playMatch( Contestant that ).`

playMatch( )

Two Options: Instance Method or Class Method

- `Contestant playMatch( Contestant player1, Contestant player2 ).`
- `Contestant playMatch( Contestant that ).`

playMatch( )

Two Options: Instance Method or Class Method

- `Contestant playMatch( Contestant player1, Contestant player2 ).`
- `Contestant playMatch( Contestant that ).`

Implementing the Second Option

PseudoCode

```
/**
 * Play match against opponent and return winner.
 *
 * @param that the opponent.
 * @return the winner of the match.
 */
public Contestant playMatch( Contestant that ) {
    boolean matchOver = false;
    // Initialise numbers of sets won.
    while (!matchOver) {
        // Determine setWinner: the winner of the next set.
        // Increase the number of sets won of setWinner.
        // Set matchOver to true if setWinner has won the match.
    }
    return (this.winsMatch( ) ? this : that);
}
```

New Attributes and Methods

- Initialise numbers of sets won.
- Determine `setWinner`: the winner of the next set.
- Increase the number of sets won of `setWinner`.
- Set `matchOver` to true if `setWinner` has won the match.

New Attributes and Methods

- Initialise numbers of sets won.
- Determine `setWinner`: the winner of the next set.
- Increase the number of sets won of `setWinner`.
- Set `matchOver` to true if `setWinner` has won the match.

Let's introduce:

- Instance attribute: `int setsWon`.

  Counts number of sets won by `this Contestant`.
- Instance method: `boolean winsMatch( )`.

  Returns `true` iff `this Contestant` has won the match.

Initial Class Design

Java

```
public class Contestant {
    /**
     * Number of sets a {@code Contestant} needs to win
     * in order to win match.
     */
    private static int SETS_REQUIRED_TO_WIN_MATCH = 3;
    /**
     * Number of times {@code Contestant} has won a set.
     */
    private int setsWon;

    /**
     * Determine if {@code Contestant} has won the match.
     *
     * @return {@code true} iff {@code this Contestant} has won the match.
     */
    public boolean winsMatch( ) {
        return setsWon == SETS_REQUIRED_TO_WIN_MATCH;
    }
}
```

The Constructor

Java

```
/**
 * Name of this {@code Contestant}.
 */
private final String name;

/**
 * Main constructor.
 *
 * @param name The name of the {@code Contestant}.
 */
public Contestant( String name ) {
    this.name = name;
}

@Override
public String toString( ) {
    return name;
}
```

How far have we Got?

- Initialise numbers of sets won.
- Determine `setWinner`: the winner of the next set.
- Increase the number of sets won of `setWinner`.
- Set `matchOver` to true if `setWinner` has won the match.

How far have we Got?

- Initialise numbers of sets won.
  - `this.setSetsWon( 0 );`

    `that.setSetsWon( 0 );`
- Determine `setWinner`: the winner of the next set.
- Increase the number of sets won of `setWinner`.
- Set `matchOver` to true if `setWinner` has won the match.

How far have we Got?

- Initialise numbers of sets won.
  - `this.setSetsWon( 0 );`

    `that.setSetsWon( 0 );`
- Determine `setWinner`: the winner of the next set.
  - `Contestant setWinner = playSet( that );`
- Increase the number of sets won of `setWinner`.

- Set `matchOver` to true if `setWinner` has won the match.

How far have we Got?

- Initialise numbers of sets won.
  - `this.setSetsWon( 0 );`

    `that.setSetsWon( 0 );`
- Determine `setWinner`: the winner of the next set.
  - `Contestant setWinner = playSet( that );`
- Increase the number of sets won of `setWinner`.
  - `setWinner.setSetsWon( setWinner.getSetsWon( ) + 1 );`
- Set `matchOver` to true if `setWinner` has won the match.

How far have we Got?

- Initialise numbers of sets won.
  - `this.setSetsWon( 0 );`

    `that.setSetsWon( 0 );`
- Determine `setWinner`: the winner of the next set.
  - `Contestant setWinner = playSet( that );`
- Increase the number of sets won of `setWinner`.
  - `setWinner.setSetsWon( setWinner.getSetsWon( ) + 1 );`
- Set `matchOver` to true if `setWinner` has won the match.
  - `matchOver = setWinner.winsMatch( );`

How far have we Got?

Java

```
public Contestant playMatch( Contestant that ) {
    boolean matchOver = false;
    this.setSetsWon( 0 );
    that.setSetsWon( 0 );

    while (!matchOver) {
        Contestant setWinner = playSet( that );
        setWinner.setSetsWon( setWinner.getSetsWon( ) + 1 );
        System.out.println( "Set won by " + setWinner );
        matchOver = setWinner.winsMatch( );
    }

    return (this.winsMatch( ) ? this : that);
}
```

For Wednesday

- Study [Horstmann 2013, 7.3–7.4].
- Using the specifications of the case study, implement the game from scratch.
- Packages are not examinable; do *not* use them for assignments.

Acknowledgements

- This lecture corresponds to[Horstmann 2013, 7.3–7.4].

About this Document

- This document was created with `pdflatex`.
- The LaTeX document class is `beamer`.