Introduction to Java (cs2514)

Lecture 6: Designing Classes (Continued)

M. R. C. van Dongen

February 3, 2017

Introduction

- Study the `for` and `while` statements.
- Study *invariants*: comments about object relationships.
- We shall study the linear search algorithm.
- Implement a simplified battleship-like game.
- Using the specifications as our input, we shall
  - Write prep code,
  - Write test code (Sort of.), and
  - Write real code.

The for Statement

- Mainly used for *bounded iteration*.

Java

```
for (⟨initialisation⟩; ⟨condition⟩; ⟨update⟩) {
    ⟨stuff⟩
}
```

1. The statement starts by carrying out ⟨initialisation⟩.
2. Carries out ⟨stuff⟩ while ⟨condition⟩ holds.
3. After each iteration ⟨update⟩ is carried out.

The for Statement

- □ Mainly used for *bounded iteration*.

Java

```
for (⟨initialisation⟩; !⟨done⟩; ⟨update⟩) {
    ⟨stuff⟩
}
```

1. The statement starts by carrying out ⟨initialisation⟩.
2. Carries out ⟨stuff⟩ while not ⟨done⟩.
3. After each iteration ⟨update⟩ is carried out.

The for Statement

Java

```
int digit; // Declare induction variable.
for (digit = 0; digit <= 1; digit++) {
    System.out.print( "Next binary digit is " );
    System.out.println( digit );
}
```

The for Statement

Java

```
int digit; // Declare induction variable.
for (digit = 0; digit <= 1; digit++) {
    System.out.print( "Next binary digit is " );
    System.out.println( digit );
}
```

The for Statement

Java

```
int digit; // Declare induction variable.
for (digit = 0; digit <= 1; digit++) {
    System.out.print( "Next binary digit is " );
    System.out.println( digit );
}
```

The for Statement

Java

```
int digit; // Declare induction variable.
for (digit = 0; digit <= 1; digit++) {
    System.out.print( "Next binary digit is " );
    System.out.println( digit );
}
```

The for Statement

Java

```
int digit; // Declare induction variable.
for (digit = 0; digit <= 1; digit++) {
    System.out.print( "Next binary digit is " );
    System.out.println( digit );
}
```

The for Statement

Java

```
int digit; // Declare induction variable.
for (digit = 0; digit <= 1; digit++) {
    System.out.print( "Next binary digit is " );
    System.out.println( digit );
}
```

The for Statement

Java

```
int digit; // Declare induction variable.
for (digit = 0; digit <= 1; digit++) {
    System.out.print( "Next binary digit is " );
    System.out.println( digit );
}
```

The for Statement

Java

```
int digit; // Declare induction variable.
for (digit = 0; digit <= 1; digit++) {
    System.out.print( "Next binary digit is " );
    System.out.println( digit );
}
```

Don't Try This at Home

```
...
int helper;
for (helper = 0; helper <= 1; helper++ ) {
    System.out.println( "Next binary digit is " + helper );
}
...
for (helper = 0; helper <= 1; helper++ ) {
    System.out.println( "Next binary digit is " + helper );
}
```

Don’t Try This at Home

```
int helper;
int helper;
for (helper = 0; helper <= 1; helper++ ) {
    System.out.println( "Next binary digit is " + helper );
}
...
for (helper = 0; helper <= 1; helper++ ) {
    System.out.println( "Next binary digit is " + helper );
}
```

Don’t Try This at Home

```
int helper;
int helper;
for (helper = 0; helper <= 1; helper++ ) {
    System.out.println( "Next binary digit is " + helper );
}
...
for (helper = 0; helper <= 1; helper++ ) {
    System.out.println( "Next binary digit is " + helper );
}
```

The for Statement: Keep Yer Variables Local

Java

```
for (int digit = 0; digit <= 1; digit++) {
    System.out.print( "Next binary digit is " + digit );
}
```

The while Statement

- Mainly used for *unbounded iteration*.

Java

```
while (⟨condition⟩) {
    ⟨stuff⟩
}
```

- This carries out ⟨stuff⟩ while ⟨condition⟩ holds.

The while Statement

Java

```java
final double initialBalance = 10000.0;
final double targetBalance = 20000.0;
final double interestRate = 5.00;

double balance = initialBalance;
int years = 0;
while (balance < targetBalance) {
    years++;
    final double interest = balance * interestRate / 100.0;
    balance = balance + interest;
}

System.out.println( "initial balance: " + initialBalance );
System.out.println( "target balance: " + targetBalance );
System.out.println( "years: " + years );
System.out.println( "balance: " + balance );
```

Introduction to Java

M. R. C. van Dongen

Introduction

The for Statement

The while Statement

The do-while Statement

Invariants

Linear Search

Specifications

Class Development

Prep Code

Test Code

Real Code

Debugging

Question Time

For Next Monday

Acknowledgements

References

About this Document

The do-while Statement

Java

```
do {
    ⟨statement⟩
} while (⟨condition⟩);
```

Java

```
⟨statement⟩
while (⟨condition⟩) {
    ⟨statement⟩
}
```

The do-while Statement

Java

```
do {
    ⟨statement⟩
} while (⟨condition⟩);
```

Java

```
⟨statement⟩
while (⟨condition⟩) {
    ⟨statement⟩
}
```

The do-while Statement

Java

```
do {
    ⟨statement⟩
} while (⟨condition⟩);
```

Java

```
⟨statement⟩
while (⟨condition⟩) {
    ⟨statement⟩
}
```

The do-while Statement

⟨condition⟩ is true

Java

```
do {
    ⟨statement⟩
} while (⟨condition⟩);
```

Java

```
⟨statement⟩
while (⟨condition⟩) {
    ⟨statement⟩
}
```

The do-while Statement

Java

```
do {
    ⟨statement⟩
} while (⟨condition⟩);
```

Java

```
⟨statement⟩
while (⟨condition⟩) {
    ⟨statement⟩
}
```

The do-while Statement

⟨condition⟩ is true

Java

```
do {
    ⟨statement⟩
} while (⟨condition⟩);
```

Java

```
⟨statement⟩
while (⟨condition⟩) {
    ⟨statement⟩
}
```

The do-while Statement

Java

```
do {
    ⟨statement⟩
} while (⟨condition⟩);
```

Java

```
⟨statement⟩
while (⟨condition⟩) {
    ⟨statement⟩
}
```

The do-while Statement

⟨condition⟩ is false

Java

```
do {
    ⟨statement⟩
} while (⟨condition⟩);
```

Java

```
⟨statement⟩
while (⟨condition⟩) {
    ⟨statement⟩
}
```

The do-while Statement

Done

Java

```
do {
    ⟨statement⟩
} while (⟨condition⟩);
```

Java

```
⟨statement⟩
while (⟨condition⟩) {
    ⟨statement⟩
}
```

Adding Numbers

Java

```
int i, sum;

i = 0;
sum = 0;
while (i < 100) {
  i = i + 1;
  sum = sum + i;
} // sum == 1 + 2 + ... + 100
```

Invariants

- *Invariants* relate the values of the variables in your program.

  Concretize: Makes relationships explicit (documentation).

  - This helps when writing the program.

  Correctness: They may help you prove the program is correct.

  Maintenance: They help you maintain your program.

- Good programmers state invariants as comments in programs.

Not Meaningful

Don’t Try This at Home

```
// variable declaration.
int x;

// assign zero to x.
x = 0;

// add two to x.
x = x + 2;

// increment x.
x++;
```

Useful Relationship

Java

```
if (⟨condition⟩) {
    // ⟨condition⟩
    ⋮
} else {
    //
    ⋮
}
```

Useful Relationship

Java

```
if (⟨condition⟩) {
    // ⟨condition⟩
    ⋮
} else {
    // ! ⟨condition⟩
    ⋮
}
```

Another Useful Relationship

Assuming Conditions are Side-Effect Free

Java

```
// ⟨condition⟩_1
while (⟨condition⟩_2) {
    ⋮
    // ⟨condition⟩_1
}
    //
```

Another Useful Relationship

Assuming Conditions are Side-Effect Free

Java

```
// ⟨condition⟩_1
while (⟨condition⟩_2) {
    ⋮
    // ⟨condition⟩_1
}
    //                 ! ⟨condition⟩_2
```

Another Useful Relationship

Assuming Conditions are Side-Effect Free

Java

```
// ⟨condition⟩_1
while (⟨condition⟩_2) {
    ⋮
    // ⟨condition⟩_1
}
    // ⟨condition⟩_1
```

Another Useful Relationship

Assuming Conditions are Side-Effect Free

Java

```
// ⟨condition⟩_1
while (⟨condition⟩_2) {
    ⋮
    // ⟨condition⟩_1
}
    // ⟨condition⟩_1 && ! ⟨condition⟩_2
```

Developing the Invariant

Java

```
int i, sum;

i = 0;
sum = 0;
while (i < 100) {
  i = i + 1;
  sum = sum + i;
}     // i >= 100
```

Developing the Invariant

Java

```
int i, sum;

i = 0;
sum = 0;
while (i < 100) {
  i = i + 1;
  sum = sum + i;
}     // i >= 100             && sum == 0 + 1 + ... + i
```

Developing the Invariant

Java

```
int i, sum;

i = 0;
sum = 0;
while (i < 100) {
  i = i + 1;
  sum = sum + i;
}     // i >= 100             && sum == 0 + 1 + ... + i
                  // i == 100 && sum == 0 + 1 + ... + i
```

Developing the Invariant

Java

```
int i, sum;

i = 0;
sum = 0;
while (i < 100) {
  i = i + 1;
  sum = sum + i;
}     // i >= 100             && sum == 0 + 1 + ... + i
                  // i == 100 && sum == 0 + 1 + ... + i
                              // sum == 0 + 1 + ... + 100
```

Developing the Invariant

Java

```
int i, sum;

i = 0;
sum = 0;
while (i < 100) {
  i = i + 1;
  sum = sum + i;
}     // i >= 100 && i <= 100 && sum == 0 + 1 + ... + i
                  // i == 100 && sum == 0 + 1 + ... + i
                              // sum == 0 + 1 + ... + 100
```

Developing the Invariant

Java

```
int i, sum;

i = 0;
sum = 0;          // i <= 100 && sum == 0 + 1 + ... + i
while (i < 100) {
  i = i + 1;
  sum = sum + i;  // i <= 100 && sum == 0 + 1 + ... + i
}     // i >= 100 && i <= 100 && sum == 0 + 1 + ... + i
                  // i == 100 && sum == 0 + 1 + ... + i
                              // sum == 0 + 1 + ... + 100
```

Developing the Invariant

Java

```
int i, sum;

i = 0;
sum = 0;          // i <= 100 && sum == 0 + 1 + ... + i
while (i < 100) {
  i = i + 1;      // i <= 100 && sum == 0 + 1 + ... + i-1
  sum = sum + i;  // i <= 100 && sum == 0 + 1 + ... + i
}     // i >= 100 && i <= 100 && sum == 0 + 1 + ... + i
                  // i == 100 && sum == 0 + 1 + ... + i
                              // sum == 0 + 1 + ... + 100
```

Developing the Invariant

Java

```
int i, sum;

i = 0;
sum = 0;          // i <= 100 && sum == 0 + 1 + ... + i
while (i < 100) { // i <  100 && sum == 0 + 1 + ... + i
  i = i + 1;      // i <= 100 && sum == 0 + 1 + ... + i-1
  sum = sum + i;  // i <= 100 && sum == 0 + 1 + ... + i
}     // i >= 100 && i <= 100 && sum == 0 + 1 + ... + i
                  // i == 100 && sum == 0 + 1 + ... + i
                              // sum == 0 + 1 + ... + 100
```

Linear Search

Java

```
int index = 0;

while (index < array.length && !satisfies( array[ index ] )) {
    index ++;

}
```

Linear Search

Java

```
int index = 0;
// index <= array.length and
// !satisfies( array[ prev ] ) for 0 <= prev < index
while (index < array.length && !satisfies( array[ index ] )) {
    index ++;
    // index <= array.length and
    // !satisfies( array[ prev ] ) for 0 <= prev < index.
}
// index <= array.length and
// (!satisfies( array[ prev ] ) for 0 <= prev < index) and
// (index >= array.length || satisfies( array[ index ] ))
```

Linear Search

Java

```
int index = 0;
// index <= array.length and
// !satisfies( array[ prev ] ) for 0 <= prev < index
while (index < array.length && !satisfies( array[ index ] )) {
    index ++;
    // index <= array.length and
    // !satisfies( array[ prev ] ) for 0 <= prev < index.
}
// index <= array.length and
// (!satisfies( array[ prev ] ) for 0 <= prev < index) and
// (index >= array.length || satisfies( array[ index ] ))
```

Linear Search

Java

```
int index = 0;
// index <= array.length and
// !satisfies( array[ prev ] ) for 0 <= prev < index
while (index < array.length && !satisfies( array[ index ] )) {
    index ++;
    // index <= array.length and
    // !satisfies( array[ prev ] ) for 0 <= prev < index.
}
// index <= array.length and
// (!satisfies( array[ prev ] ) for 0 <= prev < index) and
// (index >= array.length || satisfies( array[ index ] ))
```

Linear Search

Distinguishing Cases: `index < array.length || index == array.length`

Java

```java
int index = 0;
// index <= array.length and
// !satisfies( array[ prev ] ) for 0 <= prev < index
while (index < array.length && !satisfies( array[ index ] )) {
    index ++;
    // index <= array.length and
    // !satisfies( array[ prev ] ) for 0 <= prev < index.
}
// index <= array.length and
// (!satisfies( array[ prev ] ) for 0 <= prev < index) and
// (index >= array.length || satisfies( array[ index ] ))
```

Linear Search

Distinguishing Cases: index < array.length || index == array.length

Java

```
int index = 0;
// index <= array.length and
// !satisfies( array[ prev ] ) for 0 <= prev < index
while (index < array.length && !satisfies( array[ index ] )) {
    index ++;
    // index <= array.length and
    // !satisfies( array[ prev ] ) for 0 <= prev < index.
}
// index <= array.length and
// (!satisfies( array[ prev ] ) for 0 <= prev < index) and
// (index >= array.length || satisfies( array[ index ] ))
```

Linear Search

Distinguishing Cases: index < array.length || index == array.length

Java

```
int index = 0;
// index <= array.length and
// !satisfies( array[ prev ] ) for 0 <= prev < index
while (index < array.length && !satisfies( array[ index ] )) {
    index ++;
    // index <= array.length and
    // !satisfies( array[ prev ] ) for 0 <= prev < index.
}
// index <= array.length and
// (!satisfies( array[ prev ] ) for 0 <= prev < index) and
// (index >= array.length || satisfies( array[ index ] ))
```

Battleship: The Specifications

- Implement a Battleship-style game called Sink-a-dot-Com.
- Game is played on 7 × 7 grid.
- We're sinking "dot.coms" instead of ships.
- Initially there are three dot.coms.
- Each dot.com occupies three cells on the grid.
- The program randomly places the dot.coms on the grid.
- While there are dot.coms left:
  1. The program prompts the user to guess a cell.
  2. The program reads in the user's guess.
  3. The program checks the cell against the dot.com positions.
  4. Finally, the program takes an appropriate action:
     - If the guess is a kill then the dot.com is deleted.
     - If the guess is a hit then the cell is deleted.
     - Otherwise, the program reports a miss.

Simplified Version

- We have only one dot.com.
- We represent it as a 3-valued `int` array.
- The values are location cell numbers.
- The location cells are consecutive numbers between 1 and 7.
- User now guesses location cells.
- If the user guesses right we announce a hit.
- If there are three hits the game ends.
- Otherwise we continue.

Developing the `SimpleDotCom` Class

1. Figure out what the class is supposed to do.
2. List the instance variables and methods.
3. Write *prep code* for the methods.
4. Write *test code* for the methods.
   - Helps clarify what the methods need to to.
   - Helps design the method API.
   - Test code acts as documentation/contract.
   - By writing test code early, we can use it straight away.
5. Write *real code* for the methods: write the class.
6. Debug and reimplement as required.

Figure Out what the Class is Supposed to Do

1. **Game starts:**
   - □ Create a random DotCom.
   - □ Generate random cell locations.
   - □ For example: | 1 | 2 | 3 |.
2. **Game play begins:**
   - □ User starts guessing.

Unix Session

```
$ java SimpleDotComGame
Enter a number: 2
hit
Enter a number: 3
hit
Enter a number: 4
miss
Enter a number: 1
kill
```

3. **Game finishes:**

Unix Session

```
You took 4 guesses
```

List the Instance Variables and Methods

SimpleDotCom

`int[] locationCells`
`int hits`

`String checkYourself( final String guess )`
`void setLocationCells( final int[] loc )`

`locationCells`: Stores the location cell numbers.

`hits`: Counts the number of hits.

`checkYourself`: Checks guess and returns program's answer.

`setLocationCells`: Initialises `locationCells` with random cells.

Write Prep Code

PseudoCode

```
public String checkYourself( final String guess ) {
    final int cell = ⟨convert guess to int⟩;
    final boolean found = ⟨find cell in locationCells⟩;
    ⟨increment hits if found⟩;
    return ⟨use found and hits and return result as String⟩;
}
```

PseudoCode

```
private void setLocationCells( ) {
    final int cell = ⟨generate first cell number⟩;
    ⟨set locationCells to {cell, cell+1, cell+2}⟩;
}
```

Write Test Code

- Read the Book.
- Better to use `JUnit`.
  - Not covered in CS2514.

Write Real Code: checkYourself

- `final int cell = ⟨convert guess to int⟩`

Java

```
    final int cell = Integer.parseInt( guess );
```

Write Real Code: checkYourself

- final int cell = ⟨convert guess to int⟩

Java

```
final int cell = Integer.parseInt( guess );
```

- final boolean found = ⟨find cell in locationCells⟩

Java

```
final boolean found = findLocation( cell );
```

Write Real Code: checkYourself

□ final int cell = ⟨convert guess to int⟩

Java

```
final int cell = Integer.parseInt( guess );
```

□ final boolean found = ⟨find cell in locationCells⟩

Java

```
final boolean found = findLocation( cell );
```

□ ⟨increment hits if found⟩

Java

```
hits += (found ? 1 : 0);
```

Write Real Code: checkYourself

- □ final int cell = ⟨convert guess to int⟩

Java

```
final int cell = Integer.parseInt( guess );
```

- □ final boolean found = ⟨find cell in locationCells⟩

Java

```
final boolean found = findLocation( cell );
```

- □ ⟨increment hits if found⟩

Java

```
hits += (found ? 1 : 0);
```

- □ ⟨use found and hits and return result as String⟩

Java

```
return getResultAsString( found );
```

Let’s See: checkYourself

Java

```
public String checkYourself( final String guess ) {
    final int cell = Integer.parseInt( guess );
    final boolean found = findLocation( cell );
    hits += (found ? 1 : 0);
    return getResultAsString( found );
}
```

Write Real Code: setLocationCells

- final int cell = ⟨generate first cell number⟩

Java

```
final int maxStartValue = MAX_CELL_VALUE - CELLS_IN_DOT_COM + 1;
final int cell = 1 + rand.nextInt( maxStartValue );
```

Write Real Code: setLocationCells

- □ final int cell = ⟨generate first cell number⟩

Java

```
final int maxStartValue = MAX_CELL_VALUE - CELLS_IN_DOT_COM + 1;
final int cell = 1 + rand.nextInt( maxStartValue );
```

- □ ⟨set locationCells to {cell, cell+1, cell+2}⟩

Java

```
for (int position = 0; position != CELLS_IN_DOT_COM; position ++) {
    locationCells[ position ] = cell ++;
}
```

Let’s See: fillLocationCells( )

Java

```
private void setLocationCells( ) {
    final int maxStartValue = MAX_CELL_VALUE - CELLS_IN_DOT_COM;
    final int cell = rand.nextInt( maxStartValue + 1 );
    for (int position = 0; position != CELLS_IN_DOT_COM; position ++) {
        locationCells[ position ] = cell ++;
    }
}
```

Write Real Code: findLocation

Java

```java
private boolean findLocation( final int cell ) {
    int position = 0;
    boolean found = false;
    while ((position != locationCells.length) && !found) {
        found = locationCells[ position ++ ] == cell;
    }
    return found;
}
```

Write Real Code: findLocation

Alternative Implementation

Java

```
private boolean findLocation( final int cell ) {
    return (loationCells[ 0 ] <= cell)
                   && (cell <= locationCells[ locationCells.length - 1 ]);
}
```

Write Real Code: findLocation

Alternative Implementation

Java

```java
private boolean findLocation( final int cell ) {
    final int difference = cell - locationCells[ 0 ];
    return (0 <= difference) && (difference < locationCells.length);
}
```

Write Real Code: getResultAsString

Java

```
private static final String MISS_MESSAGE = "miss";
private static final String KILL_MESSAGE = "kill";
private static final String HIT_MESSAGE  = "hit";

private String getResultAsString( final boolean found ) {
    final String result;

    if (!found) {
        result = MISS_MESSAGE;
    } else if (hits == CELLS_IN_DOT_COM) {
        result = KILL_MESSAGE;
    } else {
        result = HIT_MESSAGE;
    }

    return result;
}
```

Debug and Reimplement as Required

Java

```java
private final Random rand = new Random( 0 );

public static void main( String[] args ) {
    final SimpleDotCom dotCom = new SimpleDotCom( );
    System.out.println( dotCom.checkYourself( "0" ) );
    System.out.println( dotCom.checkYourself( "1" ) );
    System.out.println( dotCom.checkYourself( "2" ) );
    System.out.println( dotCom.checkYourself( "3" ) );
    System.out.println( dotCom.checkYourself( "4" ) );
}
```

- We get: miss miss hit hit kill.

Debug and Reimplement as Required

Java

```
public static void main( String[] args ) {
    final SimpleDotCom dotCom = new SimpleDotCom( );
    System.out.println( dotCom.checkYourself( "3" ) );
    System.out.println( dotCom.checkYourself( "4" ) );
    System.out.println( dotCom.checkYourself( "4" ) );
}
```

- We get: `hit hit kill`.

Debug and Reimplement as Required

Java

```
public static void main( String[] args ) {
    final SimpleDotCom dotCom = new SimpleDotCom( );
    System.out.println( dotCom.checkYourself( "3" ) );
    System.out.println( dotCom.checkYourself( "4" ) );
    System.out.println( dotCom.checkYourself( "4" ) );
}
```

- We get: `hit hit kill`.
- Noooooooooooooooo.

Debug and Reimplement as Required

Java

```
public static void main( String[] args ) {
    final SimpleDotCom dotCom = new SimpleDotCom( );
    System.out.println( dotCom.checkYourself( "3" ) );
    System.out.println( dotCom.checkYourself( "4" ) );
    System.out.println( dotCom.checkYourself( "4" ) );
}
```

- We get: `hit hit kill`.
- Noooooooooooooo.
- We may have found a bug.

Question Time

Questions Anybody?

For Next Monday

- Study Chapter 5.
- Locate the bug in the program and fix it.
- The book mentions packages:
  - They are not examinable;
  - Do *not* use them for assignments.

Acknowledgements

- This lecture is partially based on
  - [Sierra, and Bates 2004].

Bibliography I

Sierra, Kathy, and Bert Bates [2004]. *Head First Java*. O'Reilly. ISBN: 978-0-596-00712-6.

About this Document

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