Chapter 6

Repetition Statements

CS 180
Prof. Sunil Prabhakar
Department of Computer Science
Purdue University
Objectives

After this week, you should be able to

- Understand looping constructs:
  - while, do-while, for
- Understand nested looping statements
- Implement a generic loop-and-a-half repetition control statement
- Prompt the user for a yes-no reply using the showConfirmDialog method of JOptionPane.
Definition

- Repetition statements control a block of code to be executed for a fixed number of times or until a certain condition is met.

- **Count-controlled repetitions** terminate the execution of the block after it is executed for a fixed number of times.

- **Sentinel-controlled repetitions** terminate the execution of the block after one of the designated conditions called a *sentinel* is encountered.

- Repetition statements are also called **loop statements**.
The **while** Statement

```
int sum = 0, number = 1;
while ( number <= 100 ) {
    sum = sum + number;
    number = number + 1;
}
```

These statements are executed as long as `number` is less than or equal to 100.
```java
while ( <boolean expression> ) {
    <statement>
}
```

**Syntax for the `while` Statement**

- **while**: The `while` keyword is used to start the loop.
- **<boolean expression>**: This is an expression that evaluates to a boolean value. It is the condition that controls whether the loop continues.
- **<statement>**: The block of code that is executed repeatedly as long as the boolean expression is true.

Example:

```java
while ( number <= 100 ) {
    sum    = sum + number;
    number = number + 1;
}
```

**Annual Income Calculation**

The code calculates the sum of the first 100 natural numbers, starting from 1 and incrementing by 1 until the number exceeds 100.
Control Flow of **while**

```
int sum = 0, number = 1

number <= 100 ?

true

sum = sum + number;
number = number + 1;

false

next statement;
```

previous statement;
More Examples

1. Keeps adding the numbers 1, 2, 3, … until the sum becomes larger than 1,000,000.

```c
int sum = 0, number = 1;
while ( sum <= 1000000 ) {
    sum = sum + number;
    number = number + 1;
}
```

2. Computes the product of the first 20 odd integers.

```c
int product = 1, number = 1, count = 20, lastNumber;
lastNumber = 2 * count - 1;
while (number <= lastNumber) {
    product = product * number;
    number = number + 2;
}
```
Finding GCD

Direct Approach

```java
public int gcd_bruteforce(int m, int n) {
    // assume m,n >= 1
    int last = Math.min(m, n);
    int gcd;
    int i = 1;
    while (i <= last) {
        if (m % i == 0 && n % i == 0) {
            gcd = i;
        }
        i++;
    }
    return gcd;
}
```

More Efficient Approach (Euclid)

```java
public int gcd(int m, int n) {
    // it doesn't matter which of n
    // and m is bigger, this method
    // will work fine either way
    int r = n % m;
    while (r != 0) {
        n = m;
        m = r;
        r = n % m;
    }
    return m;
}
```
String inputStr;
int age;

inputStr = JOptionPane.showInputDialog(null,
    "Your Age (between 0 and 130):"); 
age = Integer.parseInt(inputStr);

while (age < 0 || age > 130) {
    JOptionPane.showMessageDialog(null,
        "An invalid age was entered. Please try again.");

    inputStr = JOptionPane.showInputDialog(null,
        "Your Age (between 0 and 130):"); 
age = Integer.parseInt(inputStr);
}
Useful Shorthand Operators

sum = sum + number;  equivalent  sum += number;

<table>
<thead>
<tr>
<th>Operator</th>
<th>Usage</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+=</td>
<td>a += b</td>
<td>a = a + b</td>
</tr>
<tr>
<td>-=</td>
<td>a -= b</td>
<td>a = a - b</td>
</tr>
<tr>
<td>*=</td>
<td>a *= b</td>
<td>a = a * b</td>
</tr>
<tr>
<td>/=</td>
<td>a /= b</td>
<td>a = a / b</td>
</tr>
<tr>
<td>%=</td>
<td>a %= b</td>
<td>a = a % b</td>
</tr>
</tbody>
</table>
Watch Out for Pitfalls

1. Watch out for the off-by-one error (OBOE).
2. Make sure the loop body contains a statement that will eventually cause the loop to terminate.
3. Make sure the loop repeats exactly the correct number of times.
4. If you want to execute the loop body $N$ times, then initialize the counter to 0 and use the test condition counter $< N$ or initialize the counter to 1 and use the test condition counter $\leq N$. 
Loop Pitfall - 1

1

```c
int product = 0;
while ( product < 500000 ) {
    product = product * 5;
}
```

2

```c
int count = 1;
while ( count != 10 ) {
    count = count + 2;
}
```

Infinite Loops
Both loops will not terminate because the boolean expressions will never become false.
Overflow

- An infinite loop often results in an overflow error.
- An **overflow error** occurs when you attempt to assign a value larger than the maximum value the variable can hold.
- In Java, an overflow does not cause program termination. With types `float` and `double`, a value that represents infinity is assigned to the variable. With type `int`, the value “wraps around” and becomes a negative value.
Loop Pitfall - 2

1

```c
float count = 0.0f;

while ( count != 1.0f ) {
    count = count + 0.33333333f;
}    //seven 3s
```

2

```c
float count = 0.0f;

while ( count != 1.0f ) {
    count = count + 0.3333333f;
}   //eight 3s
```

Using Real Numbers
Loop 2 terminates, but Loop 1 does not because only an approximation of a real number can be stored in a computer’s memory.
Loop Pitfall – 2a

1. ```
   int result = 0; double cnt = 1.0;
   while (cnt <= 10.0){
       cnt += 1.0;
       result++;
   }
   System.out.println ( result);
```  
   
Using Real Numbers
Loop 1 prints out 10, as expected, but Loop 2 prints out 11. The value 0.1 cannot be stored precisely in computer memory.

2. ```
   int result = 0; double cnt = 0.0;
   while (cnt <= 1.0){
       cnt += 0.1;
       result++;
   }
   System.out.println ( result);
```
Loop Pitfall - 3

- Goal: Execute the loop body 10 times.

1. `count = 1;
   while ( count < 10 ){
       ...
       count++;
   }`

2. `count = 1;
   while ( count <= 10 ){
       ...
       count++;
   }`

3. `count = 0;
   while ( count <= 10 ){
       ...
       count++;
   }`

4. `count = 0;
   while ( count < 10 ){
       ...
       count++;
   }`

1 and 3 exhibit off-by-one error.
The **do-while** Statement

```c
int sum = 0, number = 1;
do {
    sum += number;
    number++;
} while ( sum <= 1000000 );
```

These statements are executed as long as `sum` is less than or equal to 1,000,000.
Syntax for the do-while Statement

```
do
    <statement>
while ( <boolean expression> ) ;
```

```
do   {
    sum += number;
    number++;
}
while ( sum <= 1000000 ) ;
```
Control Flow of **while**

```
int sum = 0, number = 1
sum += number;
number++;
number <= 100 ? true
next statement;
false
```
The `for` Statement

```java
int i, sum = 0, number;
for (i = 0; i < 20; i++) {
    number = scanner.nextInt();
    sum += number;
}
```

These statements are executed 20 times (i = 0, 1, 2, …, 19).
for statement syntax

for ( <initialization>; <boolean expression>; <increment> )
    <statement>

Initialization: 
Boolean Expression: 
Increment: 

for ( i = 0 ; i < 20 ; i++ )
{
    number = scanner.nextInt();
    sum += number;
}

Statement (loop body):
Control flow of \textit{for}

\begin{verbatim}
for (i = 0; i < 20; i++) {
    number = scanner.nextInt();
    sum += number;
}
\end{verbatim}

\begin{itemize}
    \item \texttt{i=0;}
    \item \texttt{i++;
    \item \texttt{previous statement;}
    \item \texttt{false}
    \item \texttt{i < 20?}
    \item \texttt{true}
    \item \texttt{number = scanner.nextInt();}
    \item \texttt{sum += number;}
    \item \texttt{next statement;}
\end{itemize}
More for Loop Examples

1. `for (int i = 0; i < 100; i += 5)`
   - `i = 0, 5, 10, …, 95`
   - NOTE: Local variable

2. `for (int j = 2; j < 40; j *= 2)`
   - `j = 2, 4, 8, 16, 32`

3. `for (int k = 100; k > 0; k--)`
   - `k = 100, 99, 98, 97, …, 1`
Nesting a `for` statement inside another `for` statement is a commonly used technique in programming.

Let’s generate the following table using a nested-`for` statement.

<table>
<thead>
<tr>
<th>Length (in feet)</th>
<th>5</th>
<th>10</th>
<th>15</th>
<th>20</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>1045</td>
<td>2090</td>
<td>3135</td>
<td>4180</td>
<td>5225</td>
</tr>
<tr>
<td>12</td>
<td>1140</td>
<td>2280</td>
<td>3420</td>
<td>4560</td>
<td>5700</td>
</tr>
<tr>
<td>13</td>
<td>1235</td>
<td>2470</td>
<td>3705</td>
<td>4940</td>
<td>6175</td>
</tr>
<tr>
<td>14</td>
<td>1330</td>
<td>2660</td>
<td>3990</td>
<td>5320</td>
<td>6650</td>
</tr>
<tr>
<td>15</td>
<td>1425</td>
<td>2850</td>
<td>4275</td>
<td>5700</td>
<td>7125</td>
</tr>
<tr>
<td>16</td>
<td>1520</td>
<td>3040</td>
<td>4560</td>
<td>6080</td>
<td>7600</td>
</tr>
<tr>
<td>17</td>
<td>1615</td>
<td>3230</td>
<td>4845</td>
<td>6460</td>
<td>8075</td>
</tr>
<tr>
<td>18</td>
<td>1710</td>
<td>3420</td>
<td>5130</td>
<td>6840</td>
<td>8550</td>
</tr>
<tr>
<td>19</td>
<td>1805</td>
<td>3610</td>
<td>5415</td>
<td>7220</td>
<td>9025</td>
</tr>
<tr>
<td>20</td>
<td>1900</td>
<td>3800</td>
<td>5700</td>
<td>7600</td>
<td>9500</td>
</tr>
</tbody>
</table>

Cost of material at $19 per sq. ft for different lengths and widths (in feet)
Generating the Table

```java
int price;
System.out.println("      5    10     15     20     25");
for (int width = 11; width <=20; width++) {
    System.out.print (width + "   ");
    for (int length = 5; length <=25, length+=5) {
        price = width * length * 19; // $19 per sq. ft.
        System.out.print ("  " + price);
    }
    // finished one row; move on to next row
    System.out.println ("");
}
```

Multiple statements in for loop

The initialization and increment of a for loop can contain multiple statements separated by commas.

```java
int sum;
for (int i=0,sum=0; i<=10; sum+=i,i++){
}
System.out.println("Sum from 1 to 20 is:" + sum);
```
breaking out of a loop

- In some cases, it is necessary to get out of a loop.
- This is achieved using a `break` statement.

```java
int price;
System.out.print("  5    10    15    20    25");
for (int width = 11; width <=20; width++){
    for (int length = 5; length <=25; length+=5){
        price = width * length * 19; // $19 per sq.
        System.out.print (" " + price);
        if (price > 2000)
            break;
    }
    //finished one row; move on to next row
    System.out.println(" ");
}
```
```java
int price;
System.out.print("  5   10    15   20   25");
for (int width = 11; width <= 20; width++) {
    for (int length = 5; length <= 25; length += 5) {
        price = width * length * 19; // $19 per sq. ft.
        System.out.print(" ");
    }
    // finished one row; move on to next row
    System.out.println(" ");
    if (price > 2000) {
        break;
    }
}
```

The code snippet calculates the price for different dimensions of areas, printing the results in a table format. The price calculation is based on a rate of $19 per square foot.
Skipping an iteration

- We can skip the current iteration of a loop using a `continue` statement.
- A continue transfers control to the test statement of the loop.

```java
int price;
System.out.print(" 5 10 20 25");
for (int width = 11; width <= 20; width++){
    for (int length = 5; length <= 25, length+=5){
        if (length == 15)
            continue;
        price = width * length * 19; // $19 per sq. ft.
        System.out.print(" "+price);
    }
    // finished one row; move on to next row
    System.out.println(" ");
}
```
Labeled continue and break

- It is also possible to break or continue any level of a nested loop using labels.

```java
int price;
row:
for (int width = 11; width <= 20, width++ ) {
    for ( int length = 5, length <= 25, length+=5 ) {
        price = width * length * 19; // $19 per sq. ft.
        System.out.print (" "+ price);
        if (price > 2000)
            break row;
    }
    // finished one row; move on to next row
    System.out.println (" ");
}
```

Loop-and-a-Half Repetition Control

- Loop-and-a-half repetition control can be used to test a loop’s terminating condition in the middle of the loop body.

- It is implemented by using reserved words while, if, and break.
Example: Loop-and-a-Half Control

```java
String name;
while (true) {
    name = JOptionPane.showInputDialog(null, "Your name");
    if (name.length() > 0) break;
    JOptionPane.showMessageDialog(null, "Invalid Entry. You must enter at least one character.");
}
```
Pitfalls for Loop-and-a-Half Control

Be aware of two concerns when using the loop-and-a-half control:

- **The danger of an infinite loop.** The boolean expression of the `while` statement is true, which will always evaluate to true. If we forget to include an `if` statement to break out of the loop, it will result in an infinite loop.

- **Multiple exit points.** It is possible, although complex, to write a correct control loop with multiple exit points (`breaks`). It is good practice to enforce the **one-entry one-exit control** flow.
A confirmation dialog can be used to prompt the user to determine whether to continue a repetition or not.

```java
JOptionPane.showConfirmDialog(null,
  /*prompt*/ "Play Another Game?",
  /*dialog title*/ "Confirmation",
  /*button options*/ JOptionPane.YES_NO_OPTION);
```
Example: Confirmation Dialog

```java
boolean keepPlaying = true;
int selection;

while (keepPlaying) {
    //code to play one game comes here
    // . . .

    selection = JOptionPane.showConfirmDialog(null,
                                            "Play Another Game?",
                                            "Confirmation",
                                            JOptionPane.YES_NO_OPTION);

    keepPlaying = (selection == JOptionPane.YES_OPTION);
}
```
We call the space occupied by an output value the **field**. The number of characters allocated to a field is the **field width**. The diagram shows the field width of 6.

From Java 5.0, we can use the **Formatter** class. **System.out** (**PrintStream**) also includes the format method.

Each value occupies six spaces. If the value has three digits, we put three blank spaces in front. If the value has four digits, we put two blank spaces in front, and so forth.
The Formatter Class

- We use the **Formatter** class to format the output.
- First we create an instance of the class

```
Formatter formatter = new Formatter(System.out);
```

- Then we call its format method

```
int num = 467;
formatter.format("%6d", num);
```

- This will output the value with the field width of 6.
The format Method of Formatter

- The general syntax is

\[
\text{format(<control string>, <expr1>, <expr2>, \ldots)}
\]

Example:

```java
int num1 = 34, num2 = 9;
int num3 = num1 + num2;
formatter.format("%3d + %3d = %5d", num1, num2, num3);
```

```
3 4 + 9 = 43
```
The format Method of PrintStream

Instead of using the Formatter class directly, we can achieve the same result by using the format method of PrintStream (System.out)

Formatter formatter = new Formatter(System.out);
formatter.format("%6d", 498);

is equivalent to

System.out.format("%6d", 498);
Control Strings

- Integers
  \%<field width> d

- Real Numbers
  \%<field width> . <decimal places> f

- Strings
  %s

- Date
  %tB %te %tY

For other data types and more formatting options, please consult the Java API for the Formatter class.
Estimating the Execution Time

- In many situations, we would like to know how long it took to execute a piece of code. For example,
  - Execution time of a loop statement that finds the greatest common divisor of two very large numbers, or
  - Execution time of a loop statement to display all prime numbers between 1 and 100 million.

- Execution time can be measured easily by using the Date class.
Using the Date Class

Here's one way to measure the execution time:

```java
Date startTime = new Date();
// code you want to measure the execution time
Date endTime = new Date();
long elapsedTimeInMilliSec = endTime.getTime() - startTime.getTime();
```
Write an application that will play Hi-Lo games with the user. The objective of the game is for the user to guess the computer-generated secret number in the least number of tries. The secret number is an integer between 1 and 100, inclusive. When the user makes a guess, the program replies with HI or LO depending on whether the guess is higher or lower than the secret number. The maximum number of tries allowed for each game is six. The user can play as many games as she wants.
Overall Plan

- Tasks:

  ```
  do {
      Task 1: generate a secret number;
      Task 2: play one game;
  } while (the user wants to play);
  ```
Required Classes

Ch6HiLo  \[\text{main class}\]

JOptionPane

Math  \[\text{standard classes}\]
Development Steps

- We will develop this program in four steps:

1. Start with a skeleton Ch6HiLo class.
2. Add code to the Ch6HiLo class to play a game using a dummy secret number.
3. Add code to the Ch6HiLo class to generate a random number.
4. Finalize the code by tying up loose ends.
Step 1 Design

The topmost control logic of HiLo

1. describe the game rules;
2. prompt the user to play a game or not;

while ( answer is yes ) {

3. generate the secret number;
4. play one game;
5. prompt the user to play another game or not;

}
Step 1 Code

Program source file is too big to list here. From now on, we ask you to view the source files using your Java IDE.

Directory: Chapter6/Step1

Source Files: Ch6HiLo.java
Step 1 Test

- In the testing phase, we run the program and verify confirm that the topmost control loop terminates correctly under different conditions.

- Play the game
  - zero times
  - one time
  - one or more times
Step 2 Design

- Implement the `playGame` method that plays one game of HiLo.
- Use a dummy secret number
  - By using a fix number such as 45 as a dummy secret number, we will be able to test the correctness of the `playGame` method
The Logic of playGame

```java
int guessCount = 0;
do {
    get next guess;
    guessCount++;
    if (guess < secretNumber) {
        print the hint LO;
    } else if (guess > secretNumber) {
        print the hint HI;
    }
} while (guessCount < number of guesses allowed && guess != secretNumber);

if (guess == secretNumber) {
    print the winning message;
} else {
    print the losing message;
}
```
Step 2 Code

Directory: Chapter6/Step2

Source Files: Ch6HiLo.java
Step 2 Test

- We compile and run the program numerous times

- To test getNextGuess, enter
  - a number less than 1
  - a number greater than 100
  - a number between 2 and 99
  - the number 1 and the number 100

- To test playGame, enter
  - a guess less than 45
  - a guess greater than 45
  - 45
  - six wrong guesses
Step 3 Design

- We complete the generateSecretNumber method.
- We want to generate a number between 1 and 100 inclusively.

```java
private void generateSecretNumber() {
    double X = Math.random();

    secretNumber = (int) Math.floor(X * 100) + 1;

    System.out.println("Secret Number: " + secretNumber);  // TEMP
    return secretNumber;
}
```
Step 3 Code

Directory: Chapter6/Step3

Source Files: Ch6HiLo.java
Step 3 Test

- We use a separate test driver to generate 1000 secret numbers.
- We run the program numerous times with different input values and check the results.
- Try both valid and invalid input values and confirm the response is appropriate.
Step 4: Finalize

- Program Completion
  - Finish the describeRules method
  - Remove all temporary statements

- Possible Extensions
  - Allow the user to set her desired min and max for secret numbers
  - Allow the user to set the number of guesses allowed
  - Keep the score—the number of guesses made—while playing games and display the average score when the user quits the program