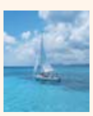
Java Summer 18

Repetition Statements



Objectives

After you have read and studied this chapter, you should be able to

- Implement repetition control in a program using while statements.
- Implement repetition control in a program using do-while statements.
- Implement a generic loop-and-a-half repetition control statement
- Implement repetition control in a program using for statements.
- Nest a loop repetition statement inside another repetition statement.
- Choose the appropriate repetition control statement for a given task
- (Optional) Write simple recursive methods

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 values called a *sentinel* is encountered.
- Repetition statements are called loop statements also.

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.

Syntax for the while Statement

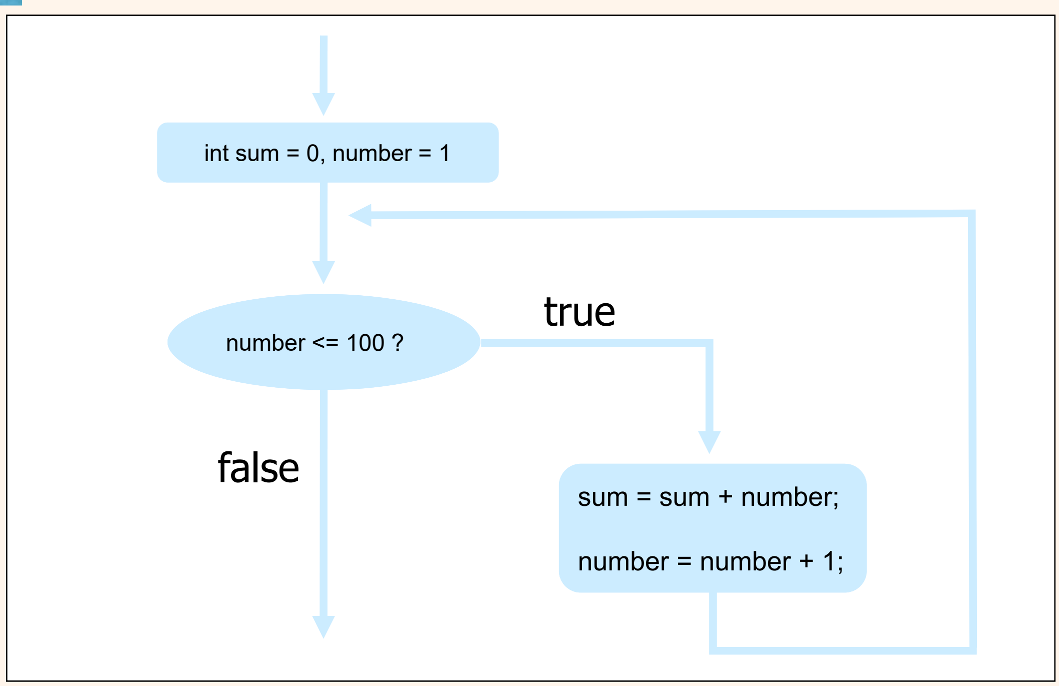
```
while ( <boolean expression> )

        <statement>
```

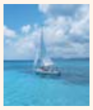
Boolean Expression

Statement (loop body)

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

   number = number + 1;
}
```


Control Flow of while





More Examples

1

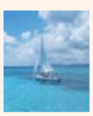
```
int sum = 0, number = 1;

while ( sum <= 1000000 ) {
    sum    =  sum + number;
    number = number + 1;
}
```

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

2

```
int product =  1, number = 1,
    count   = 20, lastNumber;

lastNumber = 2 * count - 1;

while (number <= lastNumber) {
    product = product * number;
    number  = number + 2;
}
```

Computes the product of the first 20 odd integers.



Finding GCD

```
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;
}
```

Direct Approach

```
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

   //assume m,n >= 1

   int r = n % m;

   while (r !=0) {

      n = m;

      m = r;

      r = n % m;
   }

   return m;
}
```

More Efficient Approach

Example: Testing Input Data

Priming Read

```
int  age;
Scanner scanner = new Scanner(System.in);

System.out.print("Your Age (between 0 and 130): ");
age      = scanner.nextInt( );

while (age < 0 || age > 130) {

    System.out.println(
                "An invalid age was entered. Please try again.");

    System.out.print("Your Age (between 0 and 130): ");

    age = scanner.nextInt( );
}
```


Useful Shorthand Operators

`sum = sum + number;` is equivalent to `sum += number;`

Operator	Usage	Meaning
`+=`	`a += b;`	`a = a + b;`
`-=`	`a -= b;`	`a = a – b;`
`*=`	`a *= b;`	`a = a * b;`
`/=`	`a /= b;`	`a = a / b;`
`%=`	`a %= b;`	`a = a % b;`

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 <= N.

Loop Pitfall - 1

1

```
int product = 0;

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

2

```
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

```
float count = 0.0f;

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

2

```
float count = 0.0f;

while ( count != 1.0f ) {
    count = count + 0.33333333f;
}             //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 memory.

Loop Pitfall – 2a

1

```
int result = 0; double cnt = 1.0;
while (cnt <= 10.0){
    cnt += 1.0;
    result++;
}
System.out.println(result);
```

10

2

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

11

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.

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

```
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

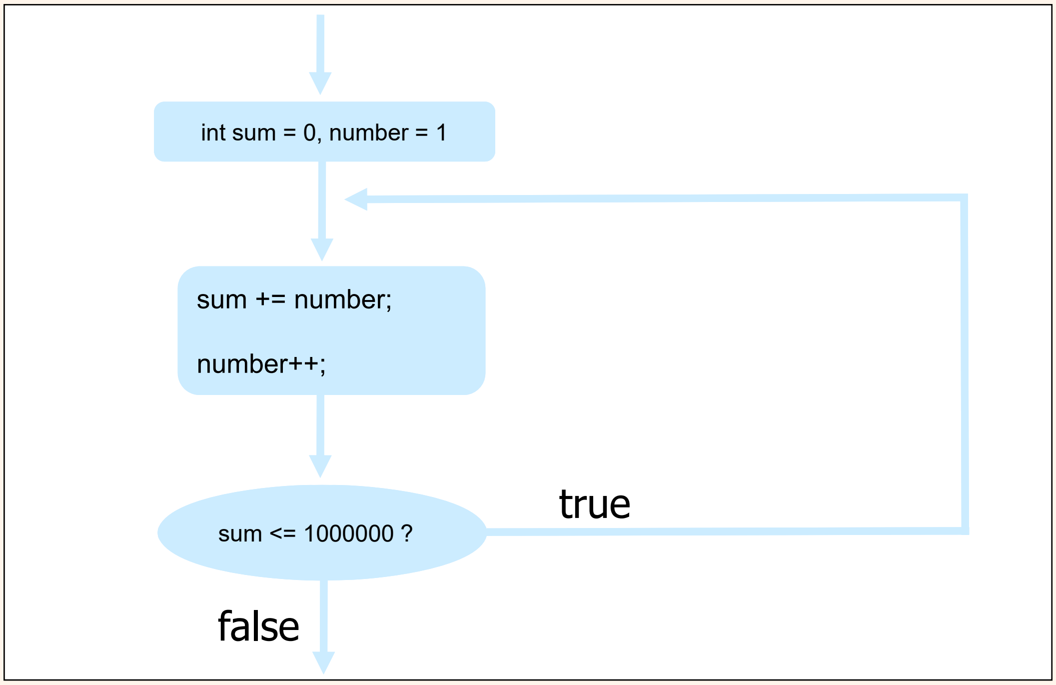
```
do
      <statement>

while ( <boolean expression> ) ;
```

```
do   {

    sum += number;
    number++;

}   while (  sum <= 1000000   );
```

Statement (loop body)

Boolean Expression

Control Flow of do-while



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

```
String name;
Scanner scanner = new Scanner(System.in);

while (true){
    System.out.print("Your name“);
    name = scanner.next( );

    if (name.length() > 0) break;

    System.out.println("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.

The for Statement

```
int i, sum = 0, number;

for (i = 0; i < 20; i++) {

    number = scanner.nextInt( );
    sum += number;

}
```

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

Syntax for the for Statement

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

                 <statement>
```

Initialization

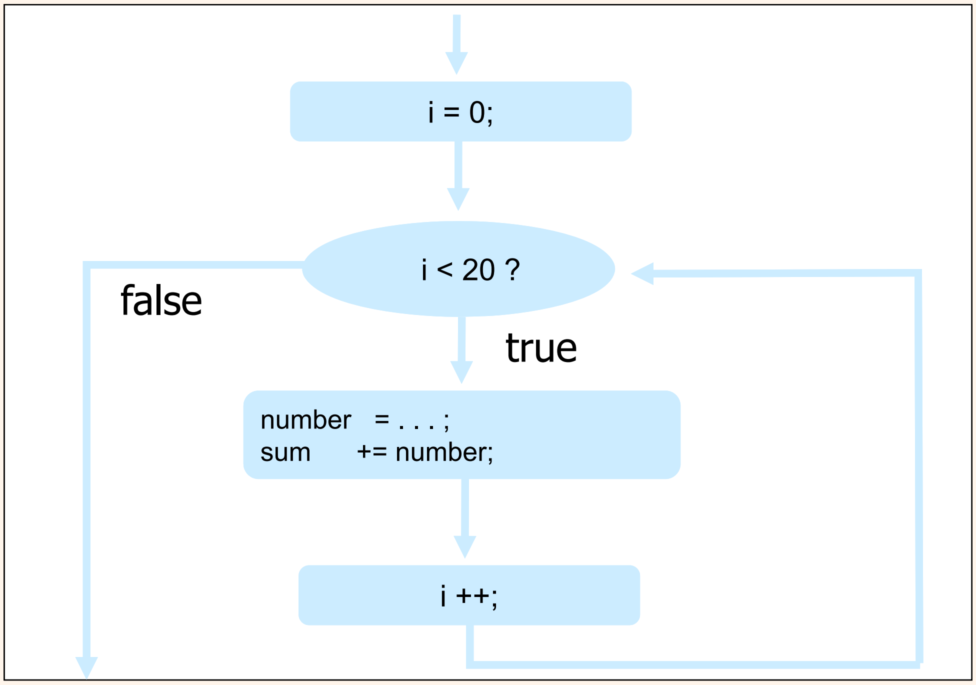
Boolean Expression

Increment

```
for (  i = 0  ;  i < 20  ;  i++  ) {

    number = scanner.nextInt();
    sum += number;
}
```

Statement (loop body)

Control Flow of for



More for Loop Examples

1

```
for (int i = 0; i < 100; i += 5)
```

i = 0, 5, 10, … , 95

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

The Nested-for Statement

- Nesting a for statement inside another for statement is commonly used technique in programming.
- Let's generate the following table using nested-for statement.

C:\WINNT\System32\cmd.exe

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

Generating the Table

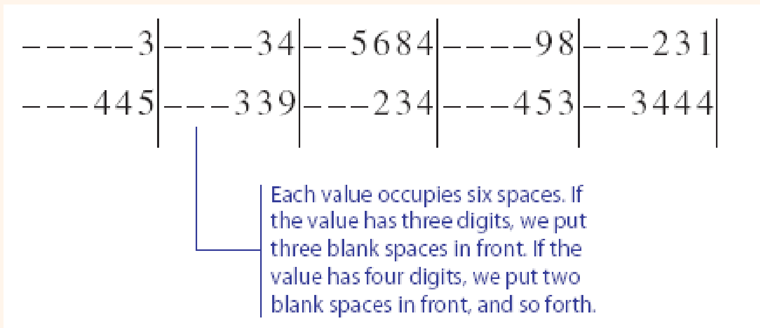
OUTER

INNER

```
int price;
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);
   }
   //finished one row; move on to next row
   System.out.println("");
}
```

Formatting Output

- 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.



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.

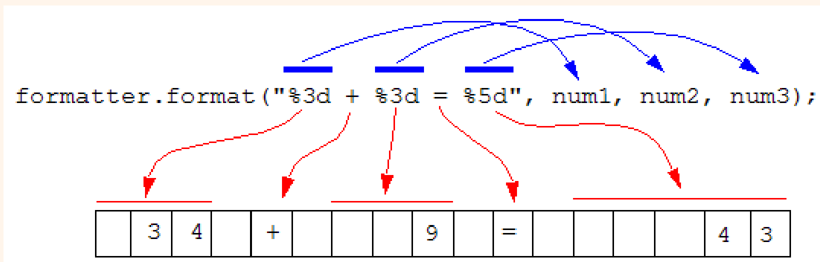
The format Method of Formatter

- The general syntax is

```
format(<control string>, <expr1>, <expr2>, . . . )
```

Example:

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



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

- 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

```
Date startTime = new Date();

//code you want to measure the execution time

Date endTime = new Date();

long elapsedTimeInMilliSec =
          endTime.getTime() – startTime.getTime();
```


Problem Statement

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 );
```


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 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

```
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 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.

```
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 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