



Iteration Loops

Copyright Sheridan College
May not be published or modified without permission.

Motivations

Suppose that you need to print a string (e.g., "Welcome to Java!") a hundred times. It would be tedious to have to write the following statement a hundred times:

```
System.out.println("Welcome to Java!");
```

So, how do you solve this problem?



Opening Problem

Problem:

100
times

```
System.out.println("Welcome to Java!");  
System.out.println("Welcome to Java!");  
System.out.println("Welcome to Java!");  
System.out.println("Welcome to Java!");  
System.out.println("Welcome to Java!");  
System.out.println("Welcome to Java!");
```

...

...

...

```
System.out.println("Welcome to Java!");  
System.out.println("Welcome to Java!");  
System.out.println("Welcome to Java!");
```



Introducing while Loops

```
int count = 0;
while (count < 100) {
    System.out.println("Welcome to Java");
    count++;
}
```

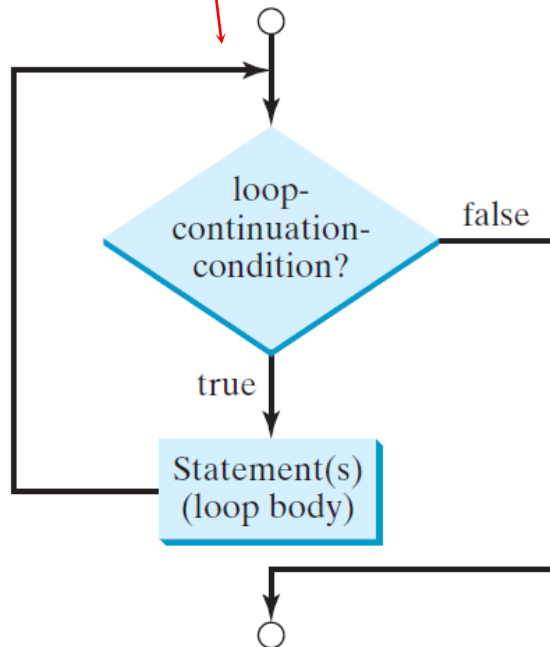
The general format of a while loop can be found at the start of Chapter 5.2. Note that if you have more than one statement in the body of your loop, you will need to enclose the statements in braces:

```
while (loop-continuation-condition)
{
    // statement 1
    // statement 2
    // etc...
}
```

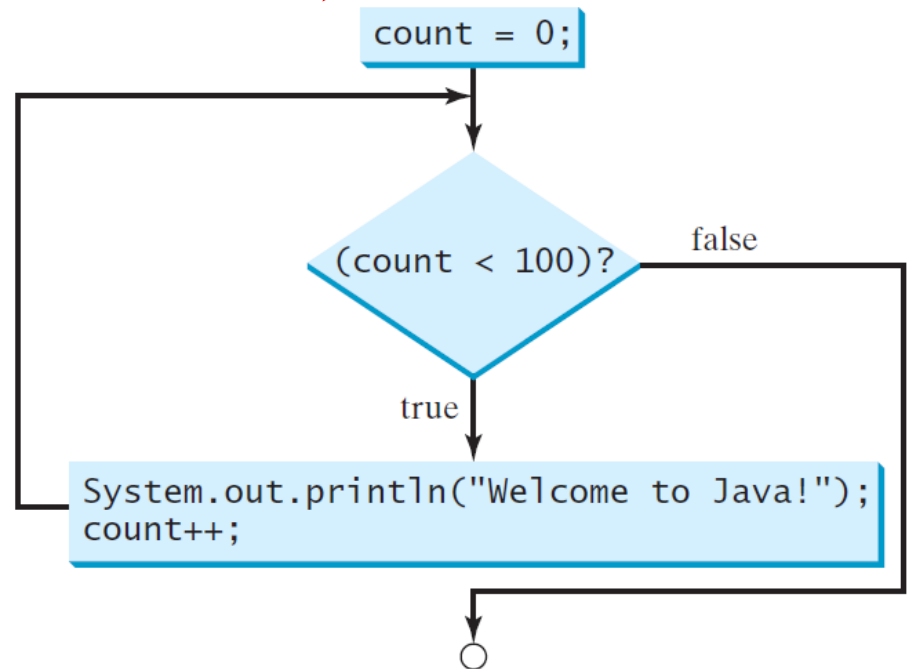


while Loop Flow Chart

```
while (loop-continuation-condition) {  
    // loop-body;  
    Statement(s);  
}
```



```
int count = 0;  
while (count < 100) {  
    System.out.println("Welcome to Java!");  
    count++;  
}
```



In a while-loop, the loop-continuation-condition is a valid condition that evaluates to true or false. As long as this condition evaluates to true, the loop's body will execute.

When the loop condition becomes false, the loop will terminate, and program flow will move to any statements that appear after the loop. For example, the loop below iterates 4 times:

```
public class BasicLoop {  
    public static void main(String[] args) {  
        int count = 1;  
        while (count <= 4) {  
            System.out.println(count + ": Hello!");  
            count++;  
        }  
        System.out.println(count + ": Done!")  
    }  
}
```

The output of this loop would be:

```
1: Hello!  
2: Hello!  
3: Hello!  
4: Hello!  
5: Done!
```

Note that the loop iterates 4 times for the values 1, 2, 3, and 4 of the **count** variable.

Inside the last iteration, the variable gets incremented to 5.

At this point, the loop condition (`count <= 4`) evaluates to false and the loop terminates.

Then the last print statement executes; it prints the value of **count** followed by ": Done!".

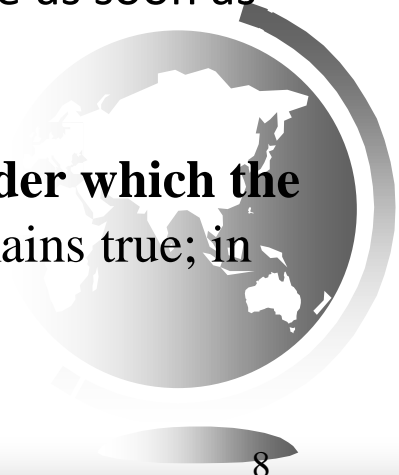


Example: The condition is tested at the top of the loop. This implies that a while loop's code block might never execute.

```
public class NeverExecutes {  
  
    public static void main(String[] args) {  
  
        int x = 5;  
        while (x < 5) {  
            System.out.println(x);  
            x--;  
        }  
    }  
}
```

Since x starts with a value of 5, the while loop's condition is false as soon as the loop begins, so the code block is never entered.

Note: the condition on a while loop specifies **the conditions under which the loop continues**. The loop continues as long as the condition remains true; in other words, the loop terminates when the condition is false.



Trace while Loop

```
int count = 0;
```

Initialize count

```
while (count < 2) {
```

```
    System.out.println("Welcome to Java!");
```

```
    count++;
```

```
}
```



Trace while Loop, cont.

```
int count = 0;
```

```
while (count < 2) {
```

```
    System.out.println("Welcome to Java!");
```

```
    count++;
```

```
}
```

(count < 2) is true



Trace while Loop, cont.

```
int count = 0;
```

```
while (count < 2) {
```

```
    System.out.println("Welcome to Java!");
```

```
    count++;
```

```
}
```

Print Welcome to Java



Trace while Loop, cont.

```
int count = 0;
```

```
while (count < 2) {
```

```
    System.out.println("Welcome to Java!");
```

```
    count++;
```

```
}
```

Increase count by 1
count is 1 now



Trace while Loop, cont.

```
int count = 0;
```

```
while (count < 2) {
```

```
    System.out.println("Welcome to Java!");
```

```
    count++;
```

```
}
```

(count < 2) is still true since count
is 1



Trace while Loop, cont.

```
int count = 0;
```

```
while (count < 2) {
```

```
    System.out.println("Welcome to Java!");
```

```
    count++;
```

```
}
```

Print Welcome to Java



Trace while Loop, cont.

```
int count = 0;
```

```
while (count < 2) {
```

```
    System.out.println("Welcome to Java!");
```

```
    count++;
```

```
}
```

Increase count by 1
count is 2 now



Trace while Loop, cont.

```
int count = 0;
```

```
while (count < 2) {
```

```
    System.out.println("Welcome to Java!");
```

```
    count++;
```

```
}
```

(count < 2) is false since count is 2
now



Trace while Loop

```
int count = 0;
while (count < 2) {
    System.out.println("Welcome to Java!");
    count++;
}
```

The loop exits. Execute the next statement after the loop.



Exercises 1:

For what values do each of these loops terminate?

- a. `while (recCount != 10)`
- b. `while (userInput.equals(""))`
- c. `while (x >= 5)`
- d. `while (x < 5)`
- e. `while ((x < 5) && (y > 10))`



1. For what values do each of these loops terminate?

a.while (recCount != 10)

when recCount contains the value 10)

b.while (userInput.equals(""))

**when the userInput String object contains characters
(when it isn't empty)**

c.while (x >= 5)

when x's value becomes less than 5

d.while (x < 5)

when x's value becomes 5 or more

e.while ((x < 5) && (y > 10))

**when x's value becomes 5 or more OR when y's value
becomes 10 or less**



Exercises 2:

What do you think is the output of this loop if the user enters the value 4?

```
import java.util.Scanner;

public class LoopThis {

    public static void main(String[] args) {

        System.out.print("Enter a value to count to:");
        int number = keysIn.nextInt();
        int counter = 1;
        while (counter <= number) {
            System.out.print(counter + " ");
            counter++;
        }
        System.out.println("\nDone!");
    }
}
```

The trace chart below shows the rest of the variable values and the output of this code

Statement Executed	Value of counter	Value of number	counter <= number?	Output Shown
System.out.print("Enter a value to count to:");				
int number = keyIn.nextInt();		4		
int counter = 1;	1			
while (counter <= number) {			true	
System.out.print(counter + " ");				1
counter++;	2			
while (counter <= number) {			true	
System.out.print(counter + " ");				1 2
counter++;	3			
while (counter <= number) {			true	
System.out.print(counter + " ");				1 2 3
counter++;	4			
while (counter <= number) {			true	
System.out.print(counter + " ");				1 2 3 4
counter++;	5			
while (counter <= number) {			false	
System.out.println("\nDone!");				1 2 3 4 Done!



Exercises 3:

What is the output of the following code segment?

```
int counter = 1;
int sum = 0;
while (counter <= 5)
{
    sum += counter++;
}
System.out.println(sum);
```



Exercises 4:

What's wrong with the following program?

```
public class LoopProblem
{
    public static void main(String[] args)
    {
        // print from 1 to 10 by 2's
        int counter = 1;
        while (counter != 10)
        {
            System.out.println(counter);
            counter += 2;
        }
    }
}
```



Solution:

You have an endless loop because counter will never be equal to 10.



Problem: Repeat Addition Until Correct

Recall that Listing 3.1 AdditionQuiz.java gives a program that prompts the user to enter an answer for a question on addition of two single digits.

Using a loop, you can now rewrite the program to let the user enter a new answer until it is correct.

IMPORTANT NOTE: If you cannot run the buttons, see www.cs.armstrong.edu/liang/javaslidenote.doc.

RepeatAdditionQuiz

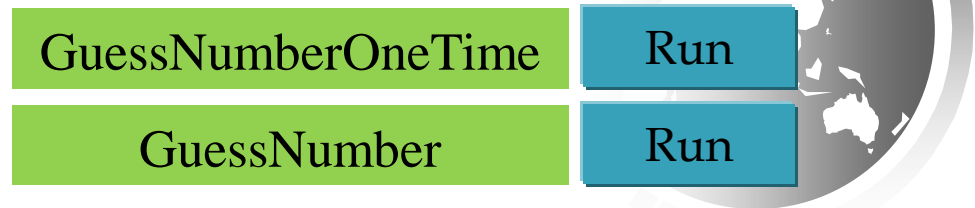
Run



Problem: Guessing Numbers

Write a program that randomly generates an integer between 0 and 100, inclusive. The program prompts the user to enter a number continuously until the number matches the randomly generated number. For each user input, the program tells the user whether the input is too low or too high, so the user can choose the next input intelligently.

Here is a sample run:



Problem: An Advanced Math Learning Tool

The Math subtraction learning tool program generates just one question for each run. You can use a loop to generate questions repeatedly. This example gives a program that generates five questions and reports the number of the correct answers after a student answers all five questions.

SubtractionQuizLoop

Run



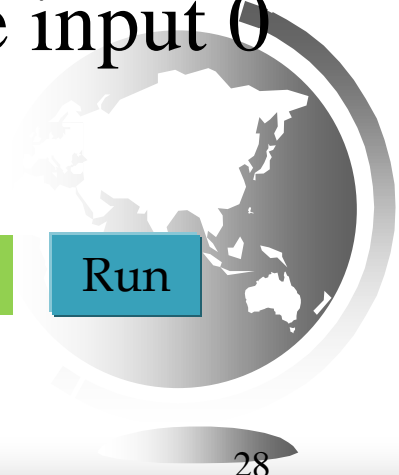
Ending a Loop with a Sentinel Value

Often the number of times a loop is executed is not predetermined. You may use an input value to signify the end of the loop. Such a value is known as a *sentinel value*.

Write a program that reads and calculates the sum of an unspecified number of integers. The input 0 signifies the end of the input.

SentinelValue

Run



Caution

Don't use floating-point values for equality checking in a loop control. Since floating-point values are approximations for some values, using them could result in imprecise counter values and inaccurate results.

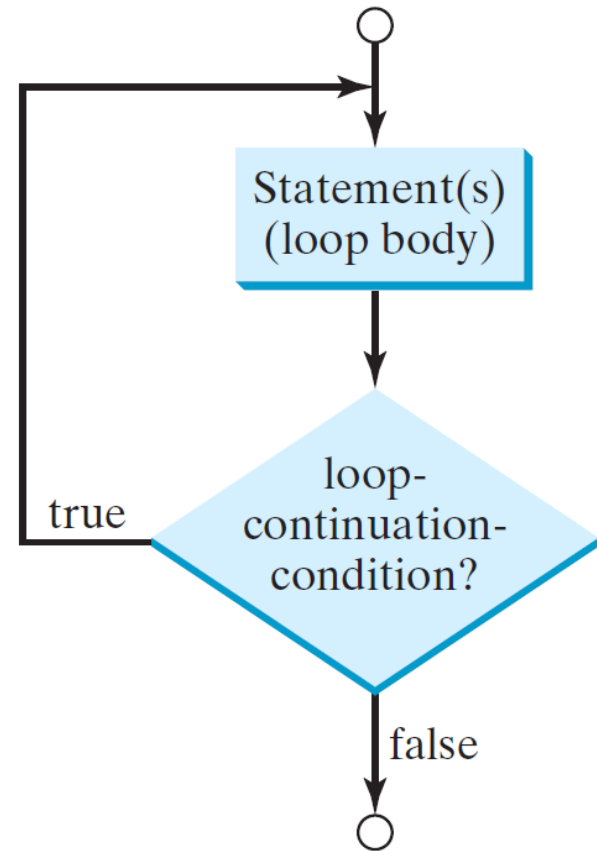
Consider the following code for computing $1 + 0.9 + 0.8 + \dots + 0.1$:

```
double item = 1; double sum = 0;
while (item != 0) { // No guarantee item will be 0
    sum += item;
    item -= 0.1;
}
System.out.println(sum);
```



do-while Loop

```
do {  
    // Loop body;  
    Statement(s) ;  
} while (loop-continuation-condition) ;
```



Do-While loops are covered in chapter 5.6.

A do-loop is often referred to as a **post-test loop** or **bottom-checking loop** because the condition that checks for loop continuation is at the bottom of the loop, instead of the top.

Note the syntax carefully - there is a semi-colon after the while condition. This is because the braces of a do-while loop enclose the loop body, but not the **while** clause in the loop. Since the **while** clause appears after the closing brace, it requires a line-terminator (semi-colon) so that Java knows

The key difference is that a while loop, a top-checking or pre-test loop, checks the condition of the loop **before** executing the loop body.

The do-while loop, a bottom-checking or post-test loop, checks the condition of the loop **after** executing the loop body.

What does this mean? It means that the body of a while loop might never execute, whereas the body of a do-while loop is always guaranteed to execute at least once. See the following examples:



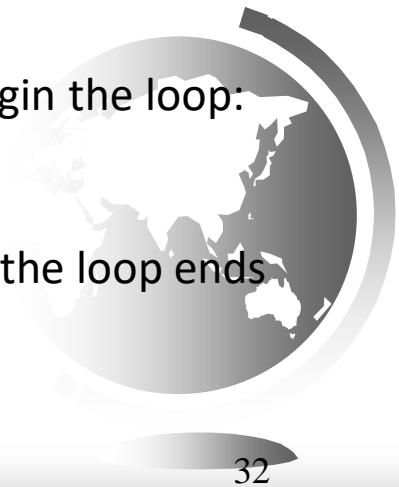
Example A: a while loop

What is the output of the following program?

```
public class LoopProblem {  
    public static void main(String[] args) {  
  
        int counter = 1;  
        while (counter > 1) {  
            System.out.println(counter);  
            counter--;  
        }  
    }  
}
```

In the program above, we first initialize counter to 1. Then we begin the loop: check the condition **counter >= 1**.

The condition is false, because counter is 1, and 1 is not >= 1, so the loop ends right away and no output is printed.



Example B: a do-while loop

What is the output of the following program?

```
public class LoopProblem {  
    public static void main(String[] args) {  
  
        int counter = 1;  
        do {  
            System.out.println(counter);  
            counter--;  
        } while (counter > 1);  
    }  
}
```

This summarizes what we said about the difference between pre-test loops and post-test loops:

The body of a pre-test loop might never execute if the condition is false right away, whereas the body of a post-test loop is always guaranteed to execute at least once.



1. What's the output of each of the following code segments:

Example 1:

```
int count = 4;
do
{
    System.out.println(count);
    count--;
} while (count > 0);
```

Example 2:

```
int count = 9;
do
{
    System.out.println(count);
    count--;
} while (count <= 10 && count > 4);
System.out.println(count);
```

Example 3:

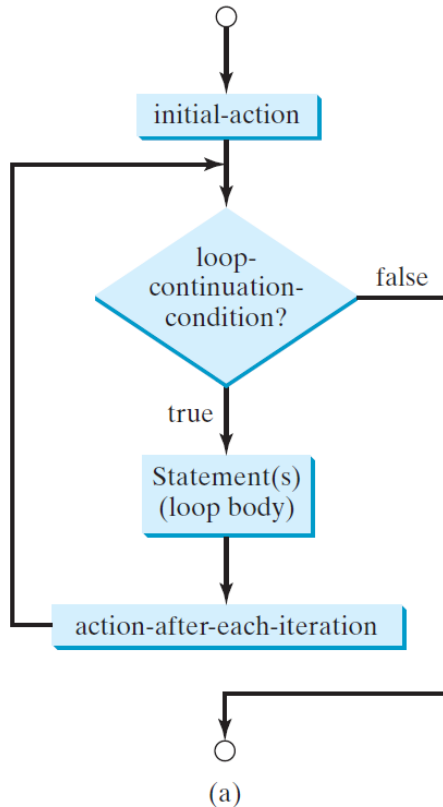
```
int count = 1;
do
{
    System.out.println(count);
    count--;
} while (count > 1);
```

Example 1:	Example 2:	Example 3:
4	9	1
3	8	
2	7	
1	6	
	5	
	4	

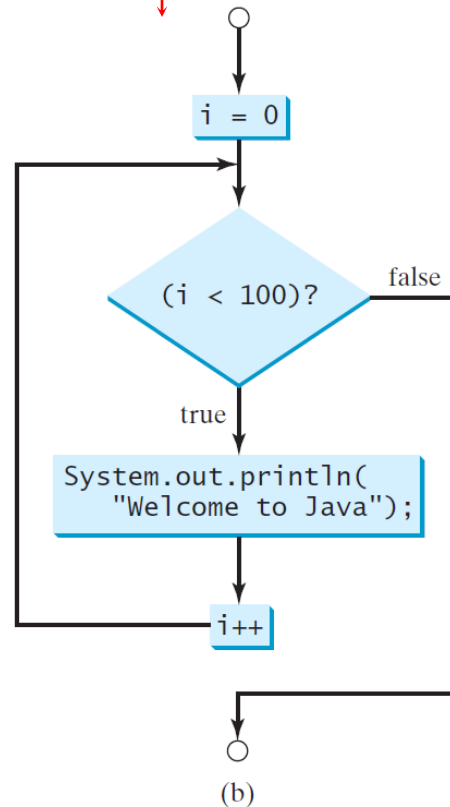


for Loops

```
for (initial-action; loop-  
    continuation-condition; action-  
    after-each-iteration) {  
    // loop body;  
    Statement(s);  
}
```



```
int i;  
for (i = 0; i < 100; i++) {  
    System.out.println(  
        "Welcome to Java!");  
}
```



Trace for Loop

```
int i;
```

Declare i

```
for (i = 0; i < 2; i++) {  
    System.out.println(  
        "Welcome to Java!");  
}
```



Trace for Loop, cont.

```
int i;  
for (i = 0; i < 2; i++) {  
    System.out.println(  
        "Welcome to Java!");  
}
```

Execute initializer
i is now 0



Trace for Loop, cont.

```
int i;  
for (i = 0; i < 2; i++) {  
    System.out.println( "Welcome to Java!");  
}
```

(i < 2) is true
since i is 0



Trace for Loop, cont.

```
int i;  
for (i = 0; i < 2; i++) {  
    System.out.println("Welcome to Java!");  
}
```

Print Welcome to Java

System.out.println("Welcome to Java!");



Trace for Loop, cont.

```
int i;  
for (i = 0; i < 2; i++) {  
    System.out.println("Welcome to Java!");  
}
```

Execute adjustment statement
i now is 1



Trace for Loop, cont.

```
int i;  
for (i = 0; i < 2; i++) {  
    System.out.println("Welcome to Java!");  
}
```

(i < 2) is still true
since i is 1



Trace for Loop, cont.

```
int i;  
for (i = 0; i < 2; i++) {  
    System.out.println("Welcome to Java!");  
}
```

Print Welcome to Java

System.out.println("Welcome to Java!");



Trace for Loop, cont.

```
int i;  
for (i = 0; i < 2; i++) {  
    System.out.println("Welcome to Java!");  
}
```

Execute adjustment statement
i now is 2



Trace for Loop, cont.

```
int i;  
for (i = 0; i < 2; i++) {  
    System.out.println("Welcome to Java!");  
}
```

(i < 2) is false
since i is 2



Trace for Loop, cont.

```
int i;  
for (i = 0; i < 2; i++) {  
    System.out.println("Welcome to Java");  
}
```

Exit the loop. Execute the next statement after the loop



Note

The initial-action in a for loop can be a list of zero or more comma-separated expressions. The action-after-each-iteration in a for loop can be a list of zero or more comma-separated statements. Therefore, the following two for loops are correct. They are rarely used in practice, however.

```
for (int i = 1; i < 100; System.out.println(i++));
```

```
for (int i = 0, j = 0; (i + j < 10); i++, j++) {
```

```
// Do something
```

```
}
```



Note

If the loop-continuation-condition in a for loop is omitted, it is implicitly true. Thus the statement given below in (a), which is an infinite loop, is correct. Nevertheless, it is better to use the equivalent loop in (b) to avoid confusion:

```
for ( ; ; ) {  
    // Do something  
}
```

(a)

Equivalent

```
while (true) {  
    // Do something  
}
```


(b)

Caution

Adding a semicolon at the end of the for clause before the loop body is a common mistake, as shown below:

```
for (int i=0; i<10; i++) ;  
{  
    System.out.println("i is " + i);  
}
```

Logic Error



Caution, cont.

Similarly, the following loop is also wrong:

```
int i=0;
while (i < 10); ← Logic Error
{
    System.out.println("i is " + i);
    i++;
}
```

In the case of the do loop, the following semicolon is needed to end the loop.

```
int i=0;
do {
    System.out.println("i is " + i);
    i++;
} while (i<10); ← Correct
```



Which Loop to Use?

The three forms of loop statements, while, do-while, and for, are expressively equivalent; that is, you can write a loop in any of these three forms. For example, a while loop in (a) in the following figure can always be converted into the following for loop in (b):

```
while (loop-continuation-condition) {  
    // Loop body  
}
```

(a)

Equivalent

```
for ( ; loop-continuation-condition; )  
    // Loop body  
}
```

(b)

A for loop in (a) in the following figure can generally be converted into the following while loop in (b) except in certain special cases (see Review Question 3.19 for one of them):

```
for (initial-action;  
     loop-continuation-condition;  
     action-after-each-iteration) {  
    // Loop body;  
}
```

(a)

Equivalent

```
initial-action;  
while (loop-continuation-condition) {  
    // Loop body;  
    action-after-each-iteration;  
}
```

(b)

Recommendations

Use the one that is most intuitive and comfortable for you. In general, a for loop may be used if the number of repetitions is known, as, for example, when you need to print a message 100 times. A while loop may be used if the number of repetitions is not known, as in the case of reading the numbers until the input is 0. A do-while loop can be used to replace a while loop if the loop body has to be executed before testing the continuation condition.



Nested Loops

Problem: Write a program that uses nested for loops to print a multiplication table.

MultiplicationTable

Run



Minimizing Numerical Errors

Numeric errors involving floating-point numbers are inevitable. This section discusses how to minimize such errors through an example.

Here is an example that sums a series that starts with 0.01 and ends with 1.0. The numbers in the series will increment by 0.01, as follows: $0.01 + 0.02 + 0.03$ and so on.



TestSum

Run

Problem:

Finding the Greatest Common Divisor

Problem: Write a program that prompts the user to enter two positive integers and finds their greatest common divisor.

Solution: Suppose you enter two integers 4 and 2, their greatest common divisor is 2. Suppose you enter two integers 16 and 24, their greatest common divisor is 8. So, how do you find the greatest common divisor? Let the two input integers be $n1$ and $n2$. You know number 1 is a common divisor, but it may not be the greatest common divisor. So you can check whether k (for $k = 2, 3, 4$, and so on) is a common divisor for $n1$ and $n2$, until k is greater than $n1$ or $n2$.



GreatestCommonDivisor

Run

Problem: Predicting the Future Tuition

Problem: Suppose that the tuition for a university is \$10,000 this year and tuition increases 7% every year. In how many years will the tuition be doubled?

FutureTuition

Run



Problem: Predicating the Future Tuition

```
double tuition = 10000; int year = 0 // Year 0
tuition = tuition * 1.07; year++;      // Year 1
tuition = tuition * 1.07; year++;      // Year 2
tuition = tuition * 1.07; year++;      // Year 3
...
```



Case Study: *Converting Decimals to Hexadecimals*

Hexadecimals are often used in computer systems programming (see Appendix F for an introduction to number systems). How do you convert a decimal number to a hexadecimal number? To convert a decimal number d to a hexadecimal number is to find the hexadecimal digits $h_n, h_{n-1}, h_{n-2}, \dots, h_2, h_1$, and h_0 such that

$$d = h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \dots + h_2 \times 16^2 + h_1 \times 16^1 + h_0 \times 16^0$$

These hexadecimal digits can be found by successively dividing d by 16 until the quotient is 0. The remainders are $h_0, h_1, h_2, \dots, h_{n-2}, h_{n-1}$, and h_n .



Dec2Hex



Run

Using break and continue

Examples for using the break and continue keywords:

□ TestBreak.java

TestBreak

Run

□ TestContinue.java


TestContinue

Run



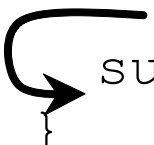
break

```
public class TestBreak {  
    public static void main(String[] args) {  
        int sum = 0;  
        int number = 0;  
  
        while (number < 20) {  
            number++;  
            sum += number;  
            if (sum >= 100)  
                break;  
        }  
        System.out.println("The number is " + number);  
        System.out.println("The sum is " + sum);  
    }  
}
```



continue

```
public class TestContinue {  
    public static void main(String[] args) {  
        int sum = 0;  
        int number = 0;  
  
        while (number < 20) {  
            number++;  
            if (number == 10 || number == 11)  
                continue;  
            sum += number;  
        }  
  
        System.out.println("The sum is " + sum);  
    }  
}
```



Guessing Number Problem

Here is a program for guessing a number. You can rewrite it using a break statement.

GuessNumberUsingBreak

Run



Problem: Displaying Prime Numbers

Problem: Write a program that displays the first 50 prime numbers in five lines, each of which contains 10 numbers. An integer greater than 1 is *prime* if its only positive divisor is 1 or itself. For example, 2, 3, 5, and 7 are prime numbers, but 4, 6, 8, and 9 are not.

Solution: The problem can be broken into the following tasks:

- For number = 2, 3, 4, 5, 6, ..., test whether the number is prime.
- Determine whether a given number is prime.
- Count the prime numbers.
- Print each prime number, and print 10 numbers per line.

PrimeNumber

Run

