Honors Computer Programming 1-2

Introduction To Chapter 6
Iteration

Chapter Goals

- To be able to program loops with the `while`, `for`, and `do` statements
- To avoid infinite loops and off-by-one errors
- To understand nested loops
- To implement simulations

`While` Loops

In this chapter we will look at programs that _repeatedly_ execute one or more statements. Suppose we open a bank account with an initial deposit of $10,000. The account earns 5% interest with the interest calculation at the end of each year and then deposited into the bank account. How many years does it take for the balance to reach $20,000?

Year	Balance
0	$10,000
1	$10,500
2	$11,025
3	$11,576.25
...	...

While Loops

In Java, the while statement implements a repetition. A while statement executes a block of code repeatedly. A termination condition controls how often the loop is executed.

The general form of the while statement is:

```
while(condition)
    statement
```

While Loops

In our case we want to know when the bank account has reached a particular balance. While the balance is less we keep adding interest and incrementing the year counter:

```
while (balance < targetBalance)
{
   years++;
   double interest = balance * rate / 100;
   balance = balance + interest;
}
```

termination condition: balance >= targetBalance

increment year counter

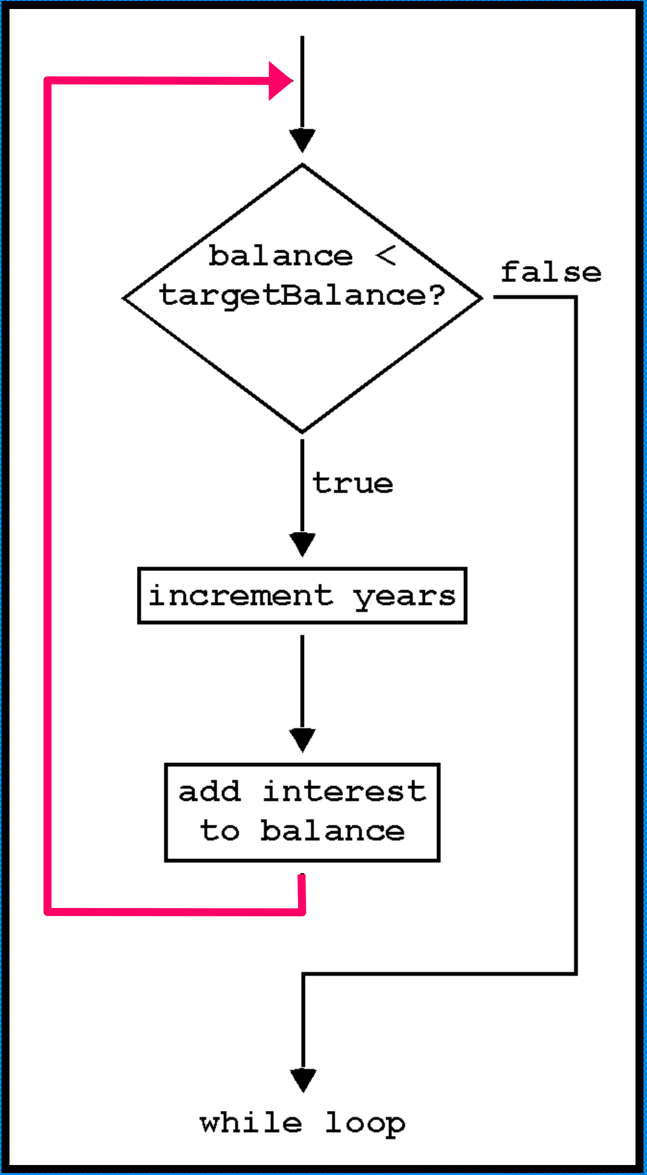
add interest

The complete program that solves our investment problem is on the handout.

While Loops

A while statement is often called a _loop_ .

The flowchart shows that the control loops _backward_ to the _test_ after every _iteration_ .



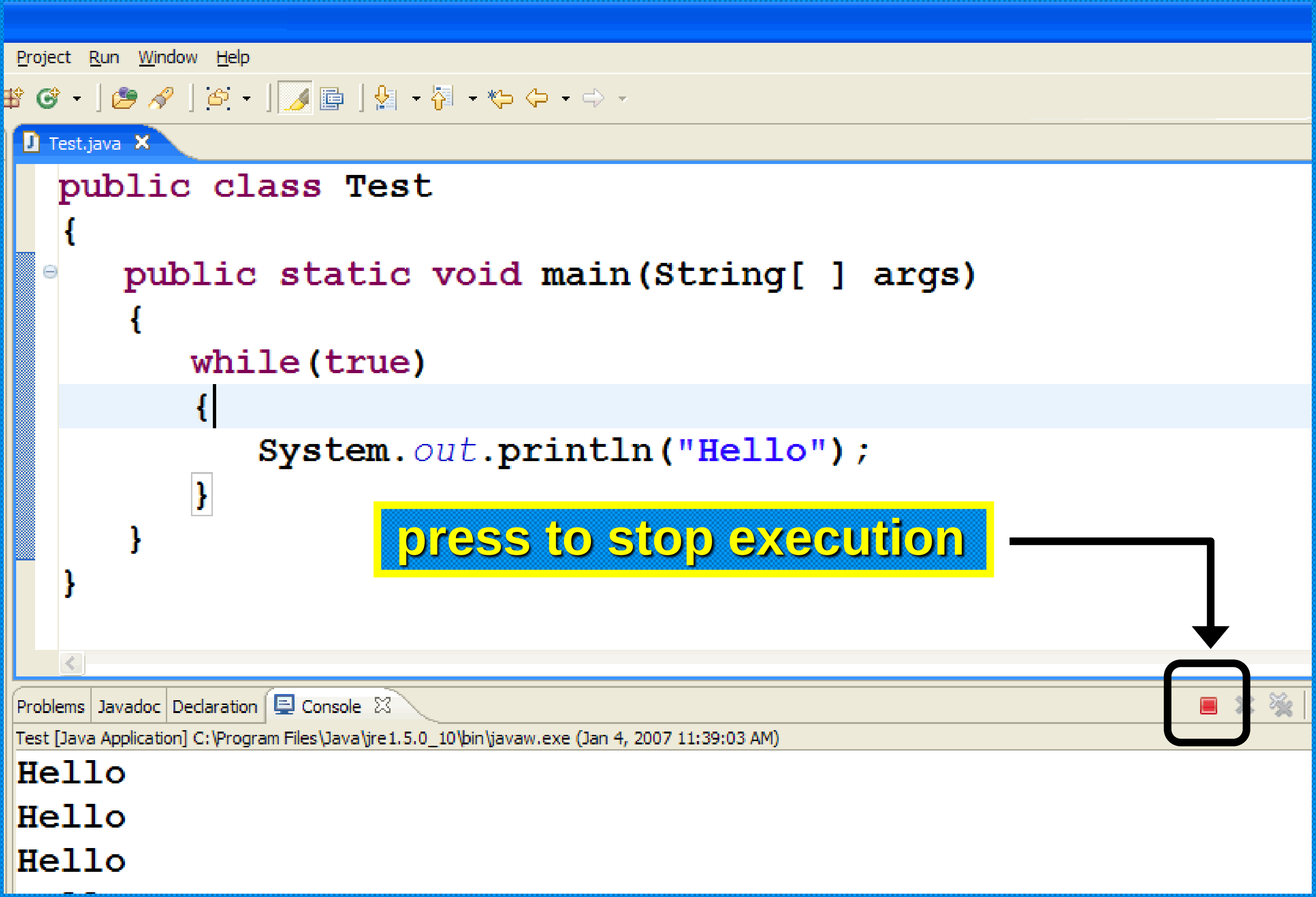
`While` Loops

The `while` loop executes the _body_ over and over without ever _terminating_ . Some programs never exit (examples _ATM machine_ or _telephone switch_) but our programs are not usually of that kind. But even if you can't terminate the loop, you can _exit_ from the method that contains it.

```
while(true)
{
    body
}
```

While Loops Infinite Loop Error

The most annoying loop error is an infinite loop which is a loop that can only be stopped by killing the program or restarting the computer.



```
public class Test
{
    public static void main(String[ ] args)
    {
        while(true)
        {
            System.out.println("Hello");
        }
    }
}
```

press to stop execution

Test [Java Application] C:\Program Files\Java\jre1.5.0_10\bin\javaw.exe (Jan 4, 2007 11:39:03 AM)

```
Hello
Hello
Hello
```

While Loops Infinite Loop Error

A common reason for infinite loops is forgetting to advance the variable that controls the loop:

```
int years = 0;
while (years < 20)
{
   double interest = balance                    00;
   balance = balance + interest;
}
```

years is a loop-control variable

Here the programmer forgot to add a years++ command in the loop. As a result the value of years always stays zero, and the loop never comes to an end .

While Loops do Loops

Sometimes you want the body of a loop to execute _ at least once _ and perform the _ loop test _ after the body was executed. The _ do _ loop serves that purpose.

```
do
   statement
while (condition);
```

While Loops do Loops

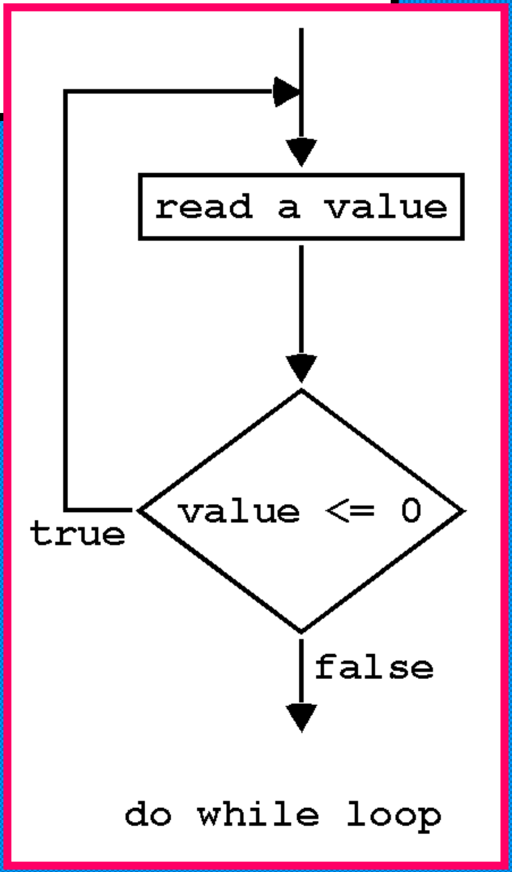
For example, suppose you want to make sure that a user enters a positive number. As long as the user enters a negative number or zero just keep prompting for a correct input.

In this case, a do loop makes sense because you need to get a user input before you can test it .

```
double value;
do
{
   String input =
        JOptionPane.showInputDialog("Enter a positive number");
   value = Double.parseDouble(input);
}
while (value <= 0);
```

While Loops do Loops

```
double value;
do
{
   String input =
        JOptionPane.showInputDialog("Enter a positive number");
   value = Double.parseDouble(input);
}
while (value <= 0);
```



do while loop

While Loops   do Loops

```
double value;
do
{
   String input =
        JOptionPane.showInputDialog("Enter a positive number");
   value = Double.parseDouble(input);
}
while (value <= 0);
```

In practice, this situation is not very common. You can always replace a do loop with a while loop by introducing a boolean control variable.

```
boolean done = false;
while (!done)
{
   String input =
        JOptionPane.showInputDialog("Enter a positive number");
   value = Double.parseDouble(input);
   if (value > 0) done = true;
}
```

For Loops

The most common loop has the form:

```
i = start;
while (i <= end)
{
    ...
   i++;
}
```

Because this form is so common there is a special form for it that emphasizes the patterns

```
for (i = start;  i <= end;  i++)
{
    ...
}
```

You can also declare the loop counter inside the for loop header:

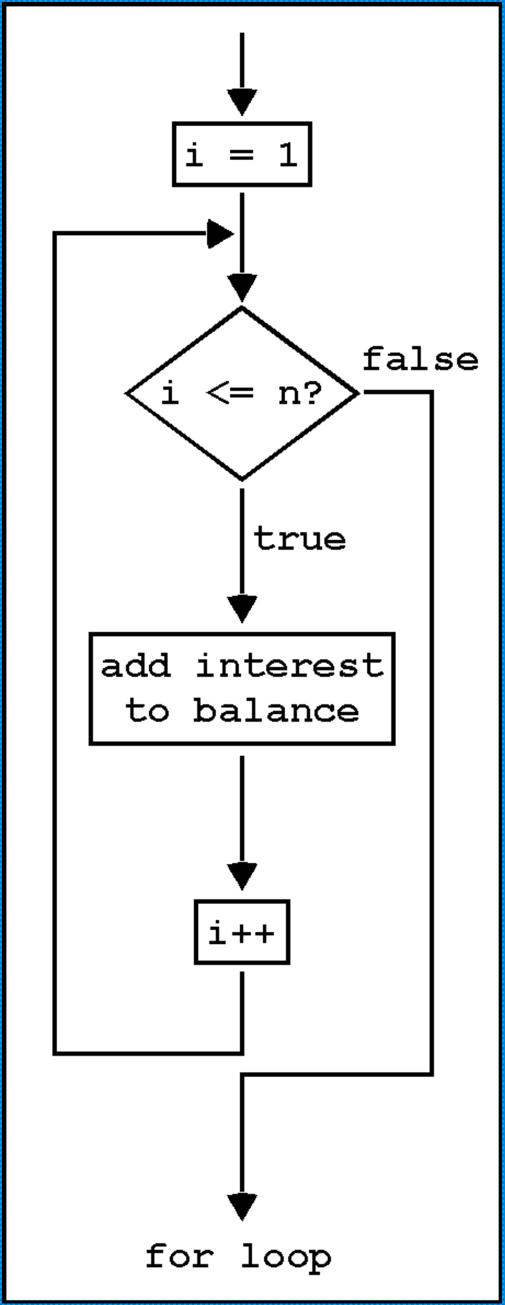
```
for (int i = start;  i <= end;  i++)
{
    ...
}
```

For Loops

Let us use this loop to find out the size of our $10,000 investment if 5% interest is compounded for 20 years. Remember that $500 is added every year.

```
for (int i = 1;  i <= n;  i++)
{
   double interest = balance * rate / 100;
   balance = balance + interest;
}
```

The code for Investment.java and InvestmentTest.java with an additional method waitYears that contains a for loop is shown on the handout.



For Loops

The three slots in the for header can contain any three expressions.

You can count down instead of up :

```
for (years = n;  years > 0;  years--) ...
```

The increment or decrement need not be in steps of one :

```
for (x = -10;  x <= 10;  x = x + 0.5) ...
```

It is possible, but a sign of bad taste , to put unrelated conditions into the loop:

```
for (rate = 5;  years-- > 0;
                        System.out.println(balance)) ... // Bad taste
```

You should stick with for loops that initialize , test , and update a single variable.

For Loops

Use for Loops For Their Intended Purpose Only

A for loop is an _idiom_ for a _while_ loop of a particular form. A _counter_ runs from the _start_ to the _end_ with a _constant_ increment:

```
for (set counter to start;  test whether counter at end;
                                            update counter by increment)
{
   ...
   // counter, start, end, increment not changed here
}
```

If your loop doesn't match this pattern, don't use the _for_ construction.

For Loops

Scope of Variables Defined in a for Loop Header

It is legal in Java to declare a variable in the header of a for loop. Here is the most common form of this syntax:

```
for (int i = 1;  i <= n;  i++)
{
    ...
}

// i no longer defined here
```

The scope of the variables extends to the end of the for loop. Therefore, i is no longer defined when the loop ends. If you need to use the value of the variable beyond the end of the loop, then you need to define it outside the loop.

For Loops

Scope of Variables Defined in a for Loop Header

In the loop header, you can declare multiple variables, as long as they are of the same type and you can include multiple update expressions separated by commas :

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

Many people find it confusing if a for loop controls more than one variable . It is not recommended to use this type of for statement. Instead, make the for loop control a single counter and update the other variable explicitly.

```
int j = 10;
for (int i = 0;  i <= 10;  i++)
{
    ...
    j--;
}
```

For Loops   A Semicolon Too Many

What does the loop at the right print?

This loop is supposed to compute

1 + 2 + . . . + 10 which is 55.

```
int i;
sum = 0;
for (i = 1;  i <= 10;  i++);
   sum = sum + i;
System.out.println(sum);
```

But actually, the loop prints 11 . Did you spot the semicolon at the end of the for loop? The loop really is a loop with an empty body .

```
for (i = 1;  i <= 10;  i++)
   ;
```

The loop does nothing 10 times and when finished, sum = 0 and i = 11 . Then the statement

```
sum = sum + i;
```

makes sum = 11 .

Nested Loops

```
[]
[][]
[][][]
[][][][]
[][][][][]
[][][][][][]
[][][][][][][]
```

Suppose you need to print the triangle shape shown:

You have to generate a number of rows as shown at the right.

```
for (int i = 1;  i <= width;  i++)
{
   // make a triangle row
   ...
}
```

How do you make a triangle row? Use another loop for the squares in that row. Then add a newline at the end of the row. The `i`th row has `i` symbols so the loop counter goes from 1 to `i` .

```
for (int j = 1;  j <= i;  j++)
   r = r + "[]";
r = r + "\n";
```

Nested Loops

```
[]
[] []
[] [] []
[] [] [] []
[] [] [] [] []
[] [] [] [] [] []
[] [] [] [] [] [] []
```

Putting these two loops together yields two nested loops as shown.

```
for (int i = 1;  i <= width;  i++)
{
   for (int j = 1;  j <= i;  j++)
     r = r + "[ ] ";
   r = r + "\n";
}
```

The complete program is shown on the handout.

Processing Input

Suppose you want to process a set of values. For reading input, you can use the _showInputDialog_ method of the JOptionPane class. Or you can use the _nextInt_ method to read an int , the _nextDouble_ method to read a double , the _next_ method to read a word , or the _nextLine_ method to read a line of text all from the Scanner class.

Processing Input

The loop shown at the right reads through input data. This loop is a little different from earlier examples because the test condition is a variable _done_ . That variable stays _false_ until you reach the end of _input data_ ; then it is set to _true_ . The next time the loop starts at the top, done is _true_ and the loop _exits_ .

```
boolean done = false;
while (!done)
{
    String input = read input;
    if (end of input indicated)
       done = true;
    else
    {
       process input
    }
}
```

Processing Input

There is a reason for using a variable. The test for loop termination occurs in the middle of the loop, not at the top or the bottom. You must first try to read input before you can test whether you have reached the end of input .

```
boolean done = false;
while (!done)
{
   String input = read input;
   if (end of input indicated)
      done = true;
   else
   {
      process input
   }
}
```

Processing Input

Let's write a program that analyzes a set of values. This will use a class DataSet. You add values to a DataSet object with the add method. The getAverage method returns the average of all added data and the getMaximum method returns the largest.

The fi lout.

```
publ
{
   ..
   //
   pu
   {
                                        = x;

   }

   ..
}
```

```
public class DataSet
{
   . . .
   // gets the largest of the added data
   public double getMaximum( )
   {
      return maximum;
   }

   . . .
}
   . . .
}
```

Processing Input

The method of exiting the loop using the `boolean` variable `done` is called the "Loop and a Half" method since loop exit is in the middle of the loop. Another technique of exiting a loop that is preferred by some programmers involves the use of the `break` statement.

Processing Input

The `break` statement was used in chapter 5 to exit a `switch` statement. A `break` can also be used to exit a `while` , `for` , or `do` loop.

In this example, the `break` statement is used to terminate the loop when the end of input is reached.

```
while (true)
{
   String input = JOptionPane.showInputDialog
                                 ("Enter value, Cancel to quit");
   if (input == null)  // leave loop in the middle
      break;
   double x = Double.parseDouble(input);
   data.add(x);
}
```

when input is `null`, the `break` statement exits the loop

Processing Input

Reading Data from the Console

Reading from the console is done with the Scanner class.

The code shown on the handout is a modified version of the input test with input from the console.

```
boolean done = false;
while (!done)
{
   System.out.print("Enter value, Q to quit: ");
   String input = console.next( );
   if (input.equalsIgnoreCase("Q"))
      done = true;
   else
   {
      double x = Double.parseDouble(input);
      data.add(x);
   }
}
```

prompt

stop the loop

Note that there is a prompt to the user inside the while loop.

The loop continues to run until done is changed to true.

Processing Input   Reading Data from a File

The loop needs to be modified when reading an _unknown_ number of data values from a _file_ .   We will not use a _`boolean`_ variable to control the loop.   Instead, we will use the _`hasNext`_ method or the _`hasNextInt`_ method of the `Scanner` class.

Code for the input test has been modified so that an unknown number of data items can be read from a file.   The code is shown on the handout.

Note that when reading data from a file, no _prompts_ are needed.
And loop exit will eventually occur at the _beginning_ of the loop.

Processing Input String Tokenization

Sometimes it is convenient to have an input line that contains multiple items of input data. Suppose an input line contains two numbers: "5.5 10000" . You can't convert the string "5.5 10000" to a number but you can break the string into a sequence of strings, each of which represents a separate input item. There is a special class `StringTokenizer` that can break up a string into items, or as they are called tokens . By default, the string tokenizer uses whitespace (spaces , tabs , newlines) as delimiters. For example, the string `"5.5 10000"` will be decomposed into two tokens `"5.5"` and `"10000"` .

Processing Input String Tokenization

To tokenize a string, you need to construct a `StringTokenizer` object and supply the string to be broken up in the constructor :

```
StringTokenizer tokenizer = new StringTokenizer(input);
```

Then keep calling the `nextToken` method to get the next token.

Processing Input String Tokenization

The loop at the right shows the proper technique. It uses the hasMoreTokens method to ensure that there are still tokens to be processed.

```
while (tokenizer.hasMoreTokens( ))
{
    String token = tokenizer.nextToken( );
    do something with token
}
```

If you want to use another separator, such as a comma to separate the individual values, you need to specify a second argument when you construct the Tokenizer object:

```
StringTokenizer tokenizer = new StringTokenizer(input, ",");
```

Here is a modified version of the input test using the tokenizers on the handout.

Traversing the Characters in a String

The _charAt_ method of the String class returns an individual character as a value of type _char_. Recall that string positions are numbered from _0_. The pattern for transversing a string is shown below.

```
for (int i = 0;  i < s.length( );  i++)
{
   char ch = s.charAt(i);
   ...  // do something with ch
}
```

Traversing the Characters in a String

Suppose you want to count the number of vowels in a string.
The loop below carries out the task.

```
int vowelCount = 0;
String vowels = "aeiouy";
for (int i = 0;  i < s.length( );  i++)
{
   char ch = Character.toLowerCase(s.charAt(i));
   if (vowels.indexOf(ch) >= 0)
      vowelCount++;
}
```

Here we use the `indexOf` method of the `String` class. The call

`str.indexOf(ch);` returns the first occurrence of `ch` in `str` or

`-1` if `ch` doesn't occur in `str`.

Symmetric and Asymmetric Bounds

It is easy to write a loop with `i` going from 1 to n:

```
for (i = 1;  i <= n;  i++) ...
```

The values for i are bounded by the relation `1 ≤ i ≤ n` . Because there are ≤ comparisons on both bounds, the bounds are called symmetric .

When traversing the characters of a string, the bounds are asymmetric :

```
for (i = 0;  i < s.length( );  i++) ...
```

The values of i are bounded by `0 ≤ i < s.length( )` with a ≤ on the left and a < on the right. That is appropriate because `s.length( )` is not a valid position.

Random Numbers and Simulations

In a _simulation_ you generate _random_ events and evaluate their outcomes. The _`Random`_ class of the Java library implements a random number generator which produces numbers that appear to be completely random. To generate random numbers, you construct an object of the _`Random`_ class and then apply one of the methods shown in the chart.

Method	Returns
`nextInt(n)`	a random integer between the integers 0 (inclusive) and n (exclusive)
`nextDouble(n)`	a random floating-point number between 0 (inclusive) and n (exclusive)

Random Numbers and Simulations

For example, you can simulate the cast of a die as shown.

```
Random generator = new Random( );
int d = 1 + generator.nextInt(6);
```

The call `generator.nextInt(6)` gives you a random number between 0 and 5 . Add 1 to obtain a number between 1 and 6 .

The program on the handout is a dice program to give you a feeling of how to use random numbers.