



## CHAPTER 4

# Flow of Control

Shin-Jie Lee (李信杰)

Associate Professor

Computer and Network Center

Department of Computer Science and Information Engineering

National Cheng Kung University



成功大學  
National Cheng Kung University



# Flow of Control

- ❑ As in most programming languages, *flow of control* in Java refers to its *branching* and *looping* mechanisms
- ❑ Java has several branching mechanisms: **if-else**, **if**, and **switch** statements
- ❑ Java has three types of loop statements: the **while**, **do-while**, and **for** statements
- ❑ Most branching and looping statements are controlled by Boolean expressions
  - A Boolean expression evaluates to either **true** or **false**
  - The primitive type **boolean** may only take the values **true** or **false**



## Branching with an **if-else** Statement

- ❑ An **if-else** statement chooses between two alternative statements based on the value of a Boolean expression

```
if (Boolean_Expression)  
    Yes_Statement  
else  
    No_Statement
```

- The **Boolean\_Expression** must be enclosed in parentheses
- If the **Boolean\_Expression** is **true**, then the **Yes\_Statement** is executed
- If the **Boolean\_Expression** is false, then the **No\_Statement** is executed



# Compound Statements

---

- ❑ Each **Yes\_Statement** and **No\_Statement** branch of an **if-else** can be made up of a single statement or many statements
- ❑ *Compound Statement*: A branch statement that is made up of a list of statements
  - A compound statement must always be enclosed in a pair of braces (**{ }**)
  - A compound statement can be used anywhere that a single statement can be used



# Compound Statements

---

```
if (myScore > your Score)
{
    System.out.println("I win!");
    wager = wager + 100;
}
else
{
    System.out.println
        ("I wish these were golf scores.");
    wager = 0;
}
```



# Omitting the `else` Part

- ❑ The `else` part may be omitted to obtain what is often called an `if` statement

```
if (Boolean_Expression)
    Action_Statement
```

- If the `Boolean_Expression` is true, then the `Action_Statement` is executed
- The `Action_Statement` can be a single or compound statement
- Otherwise, nothing happens, and the program goes on to the next statement

```
if (weight > ideal)
    calorieIntake = calorieIntake - 500;
```



# Nested Statements

---

- **if-else** statements and **if** statements both contain smaller statements within them
  - For example, single or compound statements
- In fact, any statement at all can be used as a subpart of an **if-else** or **if** statement, including another **if-else** or **if** statement
  - Each level of a nested **if-else** or **if** should be indented further than the previous level
  - Exception: *multiway if-else* statements



# Lab

---

```
public class SelectionStatements
{
    public static void main(String[] args){
        int number = 24;
        if(number % 2 == 0)
            System.out.print("The condition evaluated to true!");
        else
            System.out.print("The condition evaluated to false!");
    }
}
```





# Multiway **if-else** Statements

---

- ❑ The multiway **if-else** statement is simply a normal **if-else** statement that nests another **if-else** statement at every **else** branch
  - It is indented differently from other nested statements
  - All of the **Boolean\_Expressions** are aligned with one another, and their corresponding actions are also aligned with one another
  - The **Boolean\_Expressions** are evaluated in order until one that evaluates to **true** is found
  - The final **else** is optional



# Multiway if-else Statement

---

```
if (Boolean_Expression)
    Statement_1
else if (Boolean_Expression)
    Statement_2
    :
else if (Boolean_Expression_n)
    Statement_n
else
    Statement_For_All_Other_Possibilities
```



# Lab

---

```
public class MultiwayIfElse {  
  
    public static void main(String[] args) {  
        int a = 10;  
  
        if(a>0){  
            System.out.println("a>0");  
        }  
        else if(a>1){  
            System.out.println("a>1");  
        }  
        else if(a>2){  
            System.out.println("a>2");  
        }  
    }  
}
```



# The `switch` Statement

---

- ❑ The `switch` statement is the only other kind of Java statement that implements *multiway* branching
  - When a `switch` statement is evaluated, one of a number of different branches is executed
  - The choice of which branch to execute is determined by a *controlling expression* enclosed in parentheses after the keyword `switch`
    - The controlling expression must evaluate to a `char`, `int`, `short`, or `byte`



# The `switch` Statement

---

- ❑ Each branch statement in a `switch` statement starts with the reserved word `case`, followed by a *constant* called a *case label*, followed by a colon, and then a sequence of statements
  - Each case label must be of the same type as the controlling expression
  - Case labels need not be listed in order or span a complete interval, but each one may appear only once
  - Each sequence of statements may be followed by a `break` statement ( `break ;` )



# The `switch` Statement

- ❑ There can also be a section labeled `default`:
  - The `default` section is optional, and is usually last
  - Even if the case labels cover all possible outcomes in a given `switch` statement, it is still a good practice to include a `default` section
    - It can be used to output an error message, for example
- ❑ When the controlling expression is evaluated, the code for the case label whose value matches the controlling expression is executed
  - If no case label matches, then the only statements executed are those following the `default` label (if there is one)



# The `switch` Statement

- ❑ The `switch` statement ends when it executes a `break` statement, or when the end of the `switch` statement is reached
  - When the computer executes the statements after a case label, it continues until a `break` statement is reached
  - If the `break` statement is omitted, then after executing the code for one case, the computer will go on to execute the code for the next case
  - If the `break` statement is omitted inadvertently, the compiler will not issue an error message



# The switch Statement

---

```
int a=4;
switch (a)
{
    case 1:
        System.out.println("1");
        break;
    case 2:
        System.out.println("2");
        break;
    case 3:
        System.out.println("3");
        break;
    default:
        System.out.println("default");
        break;
}
```





# Lab

---

```
char myChar = 'b';  
switch(myChar) {  
    case 'a':  
        System.out.println("It is an A");  
        break;  
    case 'b':  
        System.out.println("It is a B");  
        break;  
    default:  
        System.out.println("Default case");  
        break;  
}
```



# The Conditional Operator

- ❑ The *conditional operator* is a notational variant on certain forms of the **if-else** statement
  - Also called the *ternary operator* or *arithmetic if*
  - The following examples are equivalent:

```
if (n1 > n2)    max = n1;  
else           max = n2;
```

vs.

```
max = (n1 > n2) ? n1 : n2;
```

- The expression to the right of the assignment operator is a *conditional operator expression*
- If the Boolean expression is true, then the expression evaluates to the value of the first expression (**n1**), otherwise it evaluates to the value of the second expression (**n2**)



# Lab

---

```
int grade1 = 65;
```

```
int grade2 = 50;
```

```
System.out.println( grade1 >= 60 ? "Passed." : "Failed." );
```

```
System.out.println( grade2 >= 60 ? "Passed." : "Failed." );
```



# Boolean Expressions

---

- ❑ A Boolean expression is an expression that is either **true** or **false**
- ❑ The simplest Boolean expressions compare the value of two expressions

**time < limit**

**yourScore == myScore**

- Note that Java uses two equal signs (**==**) to perform equality testing: A single equal sign (**=**) is used only for assignment
- A Boolean expression does not need to be enclosed in parentheses, unless it is used in an **if-else** statement



# Java Comparison Operators

Display 3.3 Java Comparison Operators

MATH NOTATION	NAME	JAVA NOTATION	JAVA EXAMPLES
=	Equal to	==	<code>x + 7 == 2*y</code> <code>answer == 'y'</code>
≠	Not equal to	!=	<code>score != 0</code> <code>answer != 'y'</code>
>	Greater than	>	<code>time &gt; limit</code>
≥	Greater than or equal to	>=	<code>age &gt;= 21</code>
<	Less than	<	<code>pressure &lt; max</code>
≤	Less than or equal to	<=	<code>time &lt;=limit</code>



## Pitfall: Using == with Strings

- ❑ The equality comparison operator (==) can correctly test two values of a *primitive* type
- ❑ However, when applied to two *objects* such as objects of the **String** class, == tests to see if they are stored in the same memory location, not whether or not they have the same value
- ❑ In order to test two strings to see if they have equal values, use the method **equals**, or **equalsIgnoreCase**  
`string1.equals(string2)`  
`string1.equalsIgnoreCase(string2)`



# Lab

---

```
String a = new String("Java");  
String b = new String("Java");  
System.out.println(a);  
System.out.println(b);  
  
if(a==b)  
    System.out.println("a==b");  
if(a.equals(b))  
    System.out.println("a equals to b");
```



# Building Boolean Expressions

---

- ❑ When two Boolean expressions are combined using the *"and"* (**&&**) operator, the entire expression is true provided both expressions are true
  - Otherwise the expression is false
- ❑ When two Boolean expressions are combined using the *"or"* (**||**) operator, the entire expression is true as long as one of the expressions is true
  - The expression is false only if both expressions are false
- ❑ Any Boolean expression can be negated using the **!** operator
  - Place the expression in parentheses and place the **!** operator in front of it
- ❑ Unlike mathematical notation, strings of inequalities must be joined by **&&**
  - Use **(min < result) && (result < max)** rather than **min < result < max**





# Evaluating Boolean Expressions

---

- ❑ Even though Boolean expressions are used to control branch and loop statements, Boolean expressions can exist independently as well
  - A Boolean variable can be given the value of a Boolean expression by using an assignment statement
- ❑ A Boolean expression can be evaluated in the same way that an arithmetic expression is evaluated
  - The only difference is that arithmetic expressions produce a number as a result, while Boolean expressions produce either **true** or **false** as their result

```
boolean madeIt = (time < limit) && (limit < max);
```



# Truth Tables

## Display 3.5 Truth Tables



AND		
<i>Exp_1</i>	<i>Exp_2</i>	<i>Exp_1</i> && <i>Exp_2</i>
true	true	true
true	false	false
false	true	false
false	false	false
OR		
<i>Exp_1</i>	<i>Exp_2</i>	<i>Exp_1</i>    <i>Exp_2</i>
true	true	true
true	false	true
false	true	true
false	false	false

NOT	
<i>Exp</i>	! ( <i>Exp</i> )
true	false
false	true



# Lab

---

```
boolean b = !( ( 6 < 5) && (4 < 3));  
System.out.println(b);
```



# Short-Circuit and Complete Evaluation

- ❑ Java can take a shortcut when the evaluation of the first part of a Boolean expression produces a result that evaluation of the second part cannot change
- ❑ This is called *short-circuit evaluation* or *lazy evaluation*
  - For example, when evaluating two Boolean subexpressions joined by `&&`, if the first subexpression evaluates to `false`, then the entire expression will evaluate to `false`, no matter the value of the second subexpression
  - In like manner, when evaluating two Boolean subexpressions joined by `||`, if the first subexpression evaluates to `true`, then the entire expression will evaluate to `true`



# Short-Circuit and Complete Evaluation

- ❑ There are times when using short-circuit evaluation can prevent a *runtime error*

- In the following example, if the number of **kids** is equal to zero, then the second subexpression will not be evaluated, thus preventing a *divide by zero error*
- Note that reversing the order of the subexpressions will not prevent this

```
if ((kids !=0) && ((toys/kids) >=2)) . . .
```

- ❑ Sometimes it is preferable to always evaluate both expressions, i.e., request complete evaluation

- In this case, use the **&** and **|** operators instead of **&&** and **||**



# Lab

---

```
int a = 1;
int b = 0;
if ((b !=0) && ((a/b) >=2))
System.out.println("bingo");

if ((b !=0) & ((a/b) >=2))
System.out.println("bingo again");
```



# Precedence and Associativity Rules

---

- ❑ Boolean and arithmetic expressions need not be fully parenthesized
- ❑ If some or all of the parentheses are omitted, Java will follow *precedence* and *associativity* rules (summarized in the following table) to determine the order of operations
  - If one operator occurs higher in the table than another, it has *higher precedence*, and is grouped with its operands before the operator of lower precedence
  - If two operators have the same precedence, then *associativity rules* determine which is grouped first



Display 3.6 Precedence and Associativity Rules

# Precedence and Associativity Rules

*Highest  
Precedence*



*Lowest  
Precedence*

PRECEDENCE	ASSOCIATIVITY
From highest at top to lowest at bottom. Operators in the same group have equal precedence.	
Dot operator, array indexing, and method invocation., [ ], ( )	Left to right
++ (postfix, as in x++), -- (postfix)	Right to left
The unary operators: +, -, ++ (prefix, as in ++x), -- (prefix), and !	Right to left
Type casts (Type)	Right to left
The binary operators *, /, %	Left to right
The binary operators +, -	Left to right
The binary operators <, >, <=, >=	Left to right
The binary operators ==, !=	Left to right
The binary operator &	Left to right
The binary operator	Left to right
The binary operator &&	Left to right
The binary operator	Left to right
The ternary operator (conditional operator) ? :	Right to left
The assignment operators =, *=, /=, %=, +=, -=, &=,  =	Right to left





# Evaluating Expressions

---

- ❑ In general, parentheses in an expression help to document the programmer's intent
  - Instead of relying on precedence and associativity rules, it is best to include most parentheses, except where the intended meaning is obvious
- ❑ *Binding*: The association of operands with their operators
  - A fully parenthesized expression accomplishes binding for all the operators in an expression
- ❑ *Side Effects*: When, in addition to returning a value, an expression changes something, such as the value of a variable
  - The *assignment*, *increment*, and *decrement* operators all produce side effects



# Rules for Evaluating Expressions

---

- ❑ Perform binding
  - Determine the equivalent fully parenthesized expression using the precedence and associativity rules
- ❑ Proceeding left to right, evaluate whatever subexpressions can be immediately evaluated
  - These subexpressions will be operands or method arguments, e.g., numeric constants or variables
- ❑ Evaluate each outer operation and method invocation as soon as all of its operands (i.e., arguments) have been evaluated



# Loops

---

- ❑ *Loops* in Java are similar to those in other high-level languages
- ❑ Java has three types of loop statements: the **while**, the **do-while**, and the **for** statements
  - The code that is repeated in a loop is called the *body* of the loop
  - Each repetition of the loop body is called an *iteration* of the loop



# while statement

---

- ❑ A **while** statement is used to repeat a portion of code (i.e., the loop body) based on the evaluation of a Boolean expression
  - The Boolean expression is checked *before* the loop body is executed
    - When false, the loop body is not executed at all
  - Before the execution of each following iteration of the loop body, the Boolean expression is checked again
    - If true, the loop body is executed again
    - If false, the loop statement ends
  - The loop body can consist of a single statement, or multiple statements enclosed in a pair of braces (**{ }**)



# while Syntax

---

```
while (Boolean_Expression)
    Statement
```

Or

```
while (Boolean_Expression)
{
    Statement_1
    Statement_2
    :
    Statement_Last
}
```



# Lab

---

```
int x = 3;  
System.out.println("The value of x is:" + x);  
while(x < 10)  
{  
    x+=3;  
}  
System.out.println("The value of x is:" + x);
```



# do-while Statement

---

- ❑ A **do-while** statement is used to execute a portion of code (i.e., the loop body), and then repeat it based on the evaluation of a Boolean expression
  - The loop body is executed at least once
    - The Boolean expression is checked *after* the loop body is executed
  - The Boolean expression is checked after each iteration of the loop body
    - If true, the loop body is executed again
    - If false, the loop statement ends
    - Don't forget to put a semicolon after the Boolean expression
  - Like the while statement, the loop body can consist of a single statement, or multiple statements enclosed in a pair of braces ( {  
})



# do-while Syntax

```
do  
    Statement  
while (Boolean_Expression);
```

Or

```
do  
{  
    Statement_1  
    Statement_2  
    ⋮  
    Statement_Last  
}while (Boolean_Expression);
```





# Lab

---

```
int evenNumber = 2;  
do  
{  
    System.out.println(evenNumber);  
    evenNumber += 2;  
}while(evenNumber <= 10);
```



# Algorithms and Pseudocode

---

- ❑ The hard part of solving a problem with a computer program is not dealing with the syntax rules of a programming language
- ❑ Rather, coming up with the underlying solution method is the most difficult part
- ❑ An *algorithm* is a set of precise instructions that lead to a solution
  - An algorithm is normally written in *pseudocode*, which is a mixture of programming language and a human language, like English
  - Pseudocode must be precise and clear enough so that a good programmer can convert it to syntactically correct code
  - However, pseudocode is much less rigid than code: One needn't worry about the fine points of syntax or declaring variables, for example



# The **for** Statement

---

- ❑ The **for** statement is most commonly used to step through an integer variable in equal increments
- ❑ It begins with the keyword **for**, followed by three expressions in parentheses that describe what to do with one or more *controlling variables*
  - The first expression tells how the **control variable or variables are initialized or declared** and *initialized* before the first iteration
  - The second expression determines when the loop should *end*, based on the evaluation of a Boolean expression *before* each iteration
  - The third expression tells how the control variable or variables are **updated after** each iteration of the loop body



# The `for` Statement Syntax

---

```
for (Initializing; Boolean_Expression; Update)  
    Body
```

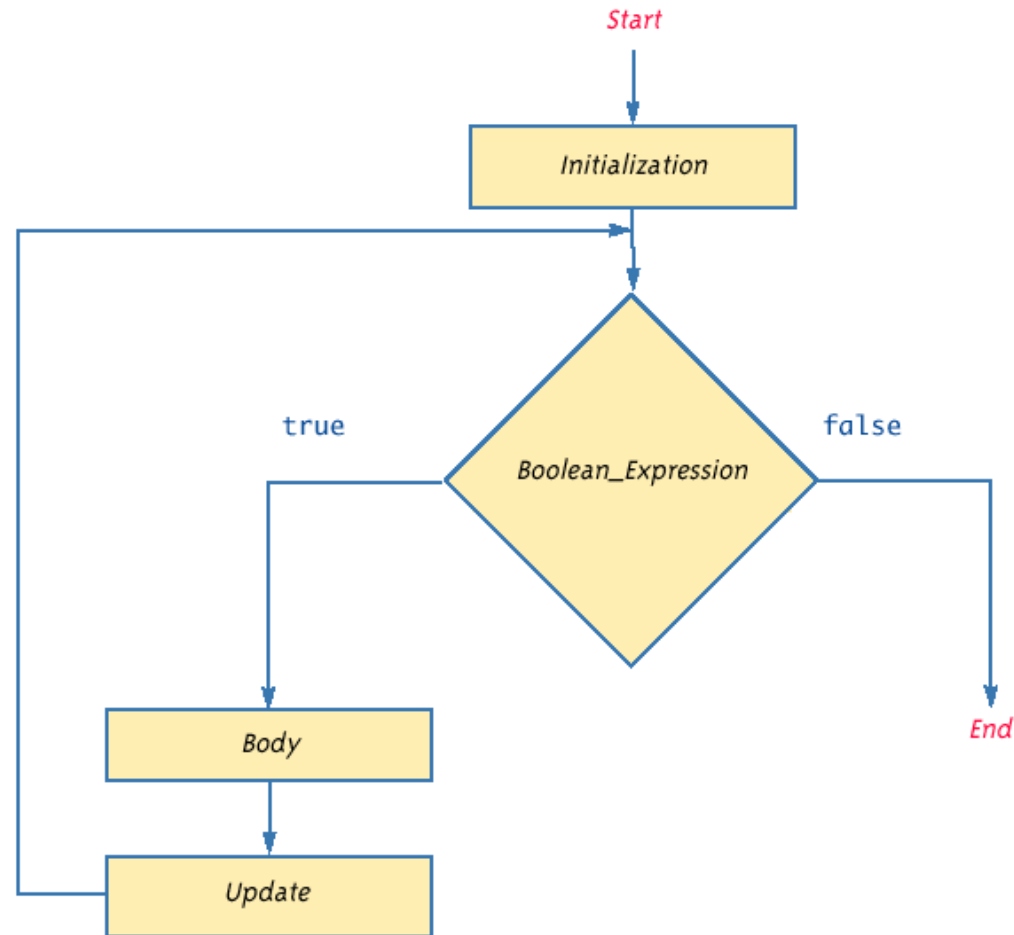
- ❑ The **Body** may consist of a single statement or a list of statements enclosed in a pair of braces (`{ }`)
- ❑ Note that the three control expressions are separated by two, not three, semicolons
- ❑ Note that there is no semicolon after the closing parenthesis at the beginning of the loop



# Semantics of the `for` Statement

Display 3.9 Semantics of the `for` Statement

`for` (*Initialization*; *Boolean\_Expression*; *Update*)  
    *Body*





# for Statement Syntax and Alternate Semantics

---

## Display 3.10 for Statement Syntax and Alternate Semantics (Part 1 of 2)

---

### for STATEMENT SYNTAX:

#### SYNTAX:

```
for (Initialization; Boolean_Expression; Update)  
    Body
```

#### EXAMPLE:

```
for (number = 100; number >= 0; number--)  
    System.out.println(number  
        + " bottles of beer on the shelf.");
```



# for Statement Syntax and Alternate Semantics

Display 3.10 for Statement Syntax and Alternate Semantics (Part 2 of 2)

## EQUIVALENT while LOOP:

### EQUIVALENT SYNTAX:

```
Initialization;  
while (Boolean_Expression)  
{  
    Body  
    Update;  
}
```

### EQUIVALENT EXAMPLE:

```
number = 100;  
while (number >= 0)  
{  
    System.out.println(number  
        + " bottles of beer on the shelf.");  
  
    number--;  
}
```

### SAMPLE DIALOGUE

```
100 bottles of beer on the shelf.  
99 bottles of beer on the shelf.  
.  
.  
.  
0 bottles of beer on the shelf.
```



# Lab

---

```
int sum = 0;
for(int i=1; i <= 3; i++)
{
    sum += i;
}
System.out.println("The total is: " + sum);
```





# Lab

---

```
int evenSum = 0;
int oddSum = 0;
//loop through the numbers
for(int i=1; i <= 5; i++)
{
    if(i % 2 == 0)
    {
        //even number
        evenSum += i;
    }
    else
    {
        oddSum += i;
    }
}
System.out.println("Even sum = " + evenSum);
System.out.println("Odd sum = " + oddSum);
```



# Infinite Loops

---

- ❑ A **while**, **do-while**, or **for** loop should be designed so that the value tested in the Boolean expression is changed in a way that eventually makes it false, and terminates the loop
- ❑ If the Boolean expression remains true, then the loop will run forever, resulting in an *infinite loop*
  - Loops that check for equality or inequality (**==** or **!=**) are especially prone to this error and should be avoided if possible



# Nested Loops

---

- ❑ Loops can be *nested*, just like other Java structures
  - When nested, the inner loop iterates from beginning to end for each single iteration of the outer loop



# Lab

---

```
for (int row = 1; row <=3; row++)  
{  
    for (int column = 1; column <=2; column++){  
        System.out.print("<" + row + ", " + column + ">");  
    }  
    System.out.println();  
}
```



# The **break** and **continue** Statements

---

- ❑ The **break** statement consists of the keyword **break** followed by a semicolon
  - When executed, the **break** statement ends the nearest enclosing switch or loop statement
- ❑ The **continue** statement consists of the keyword **continue** followed by a semicolon
  - When executed, the **continue** statement ends the current loop body iteration of the nearest enclosing loop statement
  - Note that in a **for** loop, the **continue** statement transfers control to the *update* expression
- ❑ When loop statements are nested, remember that any **break** or **continue** statement applies to the innermost, containing loop statement



# The Labeled **break** Statement

---

- ❑ There is a type of **break** statement that, when used in nested loops, can end any containing loop, not just the innermost loop
- ❑ If an enclosing loop statement is labeled with an *Identifier*, then the following version of the break statement will exit the labeled loop, even if it is not the innermost enclosing loop:  
**break someIdentifier;**
- ❑ To label a loop, simply precede it with an *Identifier* and a colon:  
**someIdentifier:**



# Lab

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

```
loop1: for(int i=0;i<=10;i++){  
    System.out.println("i="+i);  
    for(int j=0;j<=10;j++){  
        System.out.println("j="+j);  
        break loop1;  
    }  
}
```



# The `exit` Statement

- ❑ A `break` statement will end a loop or switch statement, but will not end the program
- ❑ The `exit` statement will immediately end the program as soon as it is invoked:  
`System.exit(0);`
- ❑ The `exit` statement takes one integer argument
  - By tradition, a zero argument is used to indicate a normal ending of the program





# Loop Bugs

---

- ❑ The two most common kinds of loop errors are unintended *infinite loops* and *off-by-one errors*
  - An off-by-one error is when a loop repeats the loop body one time too many or too few
    - This usually results from a carelessly designed Boolean test expression
  - Use of **==** in the controlling Boolean expression can lead to an infinite loop or an off-by-one error
    - This sort of testing works only for characters and integers, and should never be used for floating-point



# Tracing Variables

- ❑ *Tracing variables* involves watching one or more variables change value while a program is running
- ❑ This can make it easier to discover errors in a program and debug them
- ❑ Many *IDEs* (*Integrated Development Environments*) have a built-in utility that allows variables to be traced without making any changes to the program
- ❑ Another way to trace variables is to simply insert temporary output statements in a program

```
System.out.println("n = " + n); // Tracing n
```

- When the error is found and corrected, the trace statements can simply be commented out



# Generating Random Numbers

---

❑ The Random class can be used to generate pseudo-random numbers

➤ Not truly random, but uniform distribution based on a mathematical function and good enough in most cases

❑ Add the following import

```
import java.util.Random;
```

❑ Create an object of type Random

```
Random rnd = new Random();
```



# Generating Random Numbers

---

- ❑ To generate random numbers use the `nextInt()` method to get a random number from 0 to  $n-1$

```
int i = rnd.nextInt(10);    // Random number from 0 to 9
```

- ❑ Use the `nextDouble()` method to get a random number from 0 to 1 (always less than 1)

```
double d = rnd.nextDouble();    //  $0 \leq d < 1$ 
```



# Simulating a Coin Flip

Display 3.11

```
1 import java.util.Random;
2 public class CoinFlipDemo
3 {
4     public static void main(String[] args)
5     {
6         Random randomGenerator = new Random();
7         int counter = 1;
8
9         while (counter <= 5)
10        {
11            System.out.print("Flip number " + counter + ": ");
12            int coinFlip = randomGenerator.nextInt(2);
13            if (coinFlip == 1)
14                System.out.println("Heads");
15            else
16                System.out.println("Tails");
17            counter++;
18        }
19    }
20 }
```

Sample Dialogue (output will vary)

```
Flip number 1: Heads
Flip number 2: Tails
Flip number 3: Heads
Flip number 4: Heads
Flip number 5: Tails
```



# Lab

---

```
import java.util.Random;
```

```
Random rand = new Random();  
for(int i=0;i<100;i++){  
    int x = rand.nextInt(40);  
    System.out.println(x);  
}
```



# Lab

---

6.1 What will be the result of attempting to compile and run the following class?

(d) 

```
public class IfTest {  
    public static void main(String[] args) {  
        if (true)  
        if (false)  
        System.out.println("a");  
        else  
        System.out.println("b");  
    }  
}
```

Select the one correct answer.

- (a) The code will fail to compile because the syntax of the if statement is incorrect.
- (b) The code will fail to compile because the compiler will not be able to determine which if statement the else clause belongs to.
- (c) The code will compile correctly and display the letter a, when run.
- (d) The code will compile correctly and display the letter b, when run.
- (e) The code will compile correctly, but will not display any output.



# Lab

6.9 What will be the result of attempting to compile and run the following code?

(e)

```
class MyClass {  
    public static void main(String[] args) {  
        boolean b = false;  
        int i = 1;  
        do {  
            i++;  
            b = ! b;  
        } while (b);  
        System.out.println(i);  
    }  
}
```

Select the one correct answer.

- (a) The code will fail to compile because b is an invalid conditional expression for the do-while statement.
- (b) The code will fail to compile because the assignment b = ! b is not allowed.
- (c) The code will compile without error and will print 1, when run.
- (d) The code will compile without error and will print 2, when run.
- (e) The code will compile without error and will print 3, when run.





# Lab

---

Q: An if selection statement executes if and only if:

- (a) the Boolean condition evaluates to false.
- (b) the Boolean condition evaluates to true.
- (c) the Boolean condition is short-circuited.
- (d) none of the above.



# Lab

---

Q: A compound statement is enclosed between:

(a) [ ]

(b) { }

(c) ( )

(d) < >



# Lab

---

Q: The controlling expression for a switch statement includes all of the following types except:

(a)char

(b)int

(c)byte

(d)double



# Lab

---

Q: The OR operator in Java is represented by:

(a)!

(b)&&

(c)||

(d)None of the above



## Lab

---

Q: The negation operator in Java is represented by:

(a)!

(b)&&

(c)||

(d)None of the above



# Lab

---

Q: The looping mechanism that always executes at least once is the \_\_\_\_\_ statement.

(a)if...else

(b)do...while

(c)While

(d)for



# Lab

---

Q: A mixture of programming language and human language is known as:

- (a) Algorithms
- (b) Recipes
- (c) Directions
- (d) Pseudocode



# Lab

---

Q: When the number of repetitions are known in advance, you should use a \_\_\_\_\_ statement.

(a)while

(b)do...while

(c)For

(d)None of the above





# Lab

---

Q: A \_\_\_\_\_ statement terminates the current iteration of a loop.

(a) Break

(b) Continue

(c) Switch

(d) Assert



# Lab

---

□ Please write a program to calculate the factorial value of a number

**Console:**

Please input a number:5

5!=120



# Lab

---

- Please write a program to calculate the values of the following equation:

$$y = \sum_{i=1}^n x^2$$

**Console:**

Please input a value for x:2

Please input a value for n:3

y=12



# Lab

□ Please write a program to calculate the values of the following equation:  $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

```
public class MathTest {  
  
    public static void main(String[] args) {  
  
        double a = 1;  
        double b = -3;  
        double c = 2;  
  
        double x = (-1*b + Math.sqrt(Math.pow(b, 2) - 4*a*c))/ 2*a;  
  
        System.out.println(x);  
    }  
}
```



## Reference

---

- ❑ “Absolute Java”. Walter Savitch and Kenrick Mock. Addison-Wesley; 5 edition. 2012
- ❑ “Java How to Program”. Paul Deitel and Harvey Deitel. Prentice Hall; 9 edition. 2011.
- ❑ “A Programmers Guide To Java SCJP Certification: A Comprehensive Primer 3rd Edition”. Khalid Mughal, Rolf Rasmussen. Addison-Wesley Professional. 2008