4

Expressions & Flow Control

Objectives

- Distinguish between instance and local variables
- Describe how instance variables are initialized
- Identify and correct a Possible reference before assignment compiler error
- Recognize, describe, and use Java operators
- Distinguish between legal and illegal assignments of primitive types
- Identify boolean expressions and their requirements in control constructs

- Recognize assignment compatibility and required casts in fundamental types
- Use `if`, `switch`, `for`, `while`, and `do` constructions and the labeled forms of break and continue as flow control structures in a program

Variables and Scope

- Local variables are
  - Variables which are defined inside a method and are called *local*, *automatic*, *temporary*, or *stack* variables
  - Created when the method is executed and destroyed when the method is exited
  - Variables that must be initialized before they are used or compile-time errors will occur

Variable Scope Example # 1

```
{
        int x = 12;
        /* only x available */
                {
                        int q = 96;
                        /* both x & q available */
                }
        /* only x available */
        /* q is out of scope */
}
```

Variable Scope Example # 2

```
public class ScopeExample {
    private int i=1 ;
    public void firstMethod(){
        int i=4, j=5 ;
        this.i = i + j ;
        secondMethod(7) ;
    }
    public void secondMethod(){
        int j=8 ;
        this.i = i + j ;
    }
}
```

```
public class TestScoping {
  public static void main(String []args){
      ScopeExample scope = new ScopeExample();
      scope.firstMethod() ;
}
```

Initializing Variables

variables defined outside of a method are initialized ***automatically***

boolean	**false**
char	‘\u0000’ **(null)**
byte	**(byte)0**
short	**(short)0**
int	**0**
long	**0L**
float	**0.0f**
double	**0.0d**

Initializing Variables

While variables defined outside of a method are initialized automatically, local variables ***must*** be initialized manually before use:

```
public class doComputation {
  int x = (int) (Math.random() * 100) ;
  int y ;
  int z ;
  if (x > 50){
     y = 9 ;
  }
  z = x + y ;
}
```

Operators

The Java operators are similar in style and function to those of C and C++:

Separator	. [] () ; ,
Associative	**Operators**
R to L	++ -- + - ~ ! (*data type*)
L to R	* / %
L to R	+ -
L to R	<< >> >>>
L to R	< > <= >= instanceof
L to R	== !=

<table>
<tr><th>Associative</th><th>Operators</th></tr>
<tr><td>R to L</td><td>&</td></tr>
<tr><td>L to R</td><td>^</td></tr>
<tr><td>L to R</td><td>|</td></tr>
<tr><td>L to R</td><td>&&</td></tr>
<tr><td>L to R</td><td>||</td></tr>
<tr><td>R to L</td><td>?:</td></tr>
<tr><td>R to L</td><td>= *= /= %= += -= <<=
>>= >>>= &= ^= |=</td></tr>
</table>

Logical Operators

- The `boolean` operators are:

  ! – NOT   & – AND

  | – OR   ^ – XOR

- The short-curcuit `boolean` operators are:

  && – AND || – OR

- You can use these operators as follows:

```
MyDate d ;
if ((d != null) && (d.day > 31)) {
   // do something with d
}
```

Most Java operators are taken from other languages and behave as expected.

Relational and logical operators return a **`boolean`** result. The value **`0`** is not automatically interpreted as **`false`** and non-zero values are not automatically interpreted as **`true`**.

```
int i = 1 ;
if (i) // generates a compile error
if (i != 0) // correct
```


```
MyDate d ;
if ((d != null) && (d.day > 31)) {
    // do something with d
}
```

The second sub-expression is skipped when the first subexpression is false

if ((d != null) **&&** (d.day > 31)) is SAFE

if ((d != null) **&** (d.day > 31)) is NOT safe a=0

```
import java.io.* ;

public class Evaluate {
        public static void main(String[] args) {
                int a=1,b=1,c=3,d=10,y ;
                y = b + ( a==1 || ++b==d || ++d==11 ? b : d ) + 1 ;
                System.out.println("y=") ;
                System.out.println(y) ;
        }
}
```

Bitwise Logical Operators

- The integer *bitwise* operators are:

  ~ – Complement   **&** – AND

  ^ – XOR   **|** – OR

- Byte-sized examples:

~

0	1	0	0	1	1	1	1
1	0	1	1	0	0	0	0

&

0	0	1	0	1	1	0	1
0	1	0	0	1	1	1	1
0	0	0	0	1	1	0	1

	0	0	1	0	1	1	0	1
^	0	1	0	0	1	1	1	1
	0	1	1	0	0	0	1	0

	0	0	1	0	1	1	0	1
|	0	1	0	0	1	1	1	1
	0	1	1	0	1	1	1	1

Right-Shift Operators >> and >>>

- *Arithmetic* or *signed* right-shift (>>) is used as follows:
  - **128 >> 1** returns $128/2^1=64$
  - **256 >> 4** returns $256/2^4=16$
  - **-256 >> 4** returns $-256/2^4=-16$
  - The sign bit is copied during the shift.
- A *logical* or *unsigned* right-shift (>>>) is:
  - Used for bit patterns.
  - The sign bit is not copied during the shift.

1	0	1	0	1	1	0	1	>> 2 =	1	1	1	0	1	0	1	1
1	0	1	0	1	1	0	1	>>> 2 =	0	0	1	0	1	1	0	1

Left-Shift Operator (<<)

- Left-shift (<<) works as follows:

  128 << 1 returns $128*2^1=256$

  16 << 2 returns $16*2^2=64$

Shift Operator Examples

1537	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	1	1	0	1
-1537	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	0	1	1	0	0	1	1
1537 >> 5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0
-1537 >> 5	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	0	1
1537 >>> 5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0
-1537 >>> 5	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	0	1
1537 << 5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	1	1	0	1	0	0	0	0	0
-1537 << 5	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	0	1	1	0	0	1	1	0	0	0	0	0

String Concatenation With +

- The + operator
  - Performs `String` concatenation
  - Produces a new String:

    String salutation = "Dr.";

    String name = "Pete " + " " + "Seymour";

    String title = salutation + " " + name;
- One argument must be a `String` object
- Non-strings are converted to `String` objects <u>automatically</u>

Casting

► If information is lost in an assignment, the programmer must confirm the assignment with a typecast.

► The assignment between `long` and `int` requires an explicit cast.

```
long bigValue = 99L;
int squashed = bigValue; //Wrong, needs a cast
int squashed = (int) bigValue; // OK

int squashed = 99L; //Wrong, needs a cast
int squashed = (int) 99L;  // OK, but...
int squashed = 99; // default integer literal
```

Promotion and Casting of Expressions

- Variables are automatically promoted to a longer form (such as `int` to `long`)

- Expression is *assignment compatible* if the variable type is at least as large (the same number of bits) as the expression type.

```
long bigval = 6 ; // 6 is an int type, OK
int smallval = 99L ; // 99L is a long, illegal

double z = 12.414F; // 12.414F is float, OK
float z1 = 12.414; // 12.414 is double, illegal
```

Assignment compatibility between integer types

to \ from	byte	char	short	int	long
byte	✓	💣	💣	💣	💣
char	💣	✓	💣	💣	💣
short	✓	✓	✓	💣	💣
int	✓	✓	✓	✓	💣
long	✓	✓	✓	✓	✓

✓ **Assignable**

💣 **Cast needed**

```
short a, b, c ;
a = 1 ;
b = 2 ;
c = a + b ;    c = (short) (a + b) ;
```

For binary operators, such as the **+** operator, when the two operands are of primitive numeric types, the result is at least an **int** and has a value calculated by promoting the operands to the result type or promoting the result to the wider type of the operands. This might result in **overflow** or **loss of precision**.

Branching Statements

The `if`, `else` statement syntax:

```
if (boolean expression) {
    statement or block;
}

if (condition is true) {
    statement or block;
} else {
    statement or block;
}
```

Example

```
int count ;
count = getCount() ;
if (count<0) {
    System.out.println("Error: count value is negative.");
} else if (count > getMaxCount()) {
   System.out.println("Error: count value is too big.");
} else {
   System.out.println("There will be"+count+ "people for lunch today.");
}
```

```
if ( x > 5 )
   if ( y > 5 )
      System.out.println( "x and y are > 5" );
else
   System.out.println( “x is <= 5" );
```

```
if ( x > 5 ){
   if ( y > 5 )
      System.out.println( "x and y are > 5" );
}
else
   System.out.println( "x is <= 5" );
```

Branching Statements

The `switch` statement syntax is:

```
switch (expr1) {
   case expr2:
      statements;
   break;
   case expr3:
      statements;
   break;
   default:
      statements;
   break;
}
```

Common Programming Error

```
int i = 1 ;
switch (i) {
    case 1: System.out.println(“bir”) ;
    case 2: System.out.println(“iki”) ;
    break;
        case 3: System.out.println(“uc”) ;
        case 4: System.out.println(“dort”) ;
    default: System.out.println(“on”) ;
    break;
}
```

Looping Statements

The for statement

```
for (init_expr; boolean testexpr; alter_expr) {
     statement or block;
}
```

```
int faktoriyel,i;
for(i=1,faktoriyel=1;i<=9;i++)
   faktoriyel *= i;
System.out.println("9!=" + faktoriyel) ;
```

Looping Statements

The **`while`** loop:

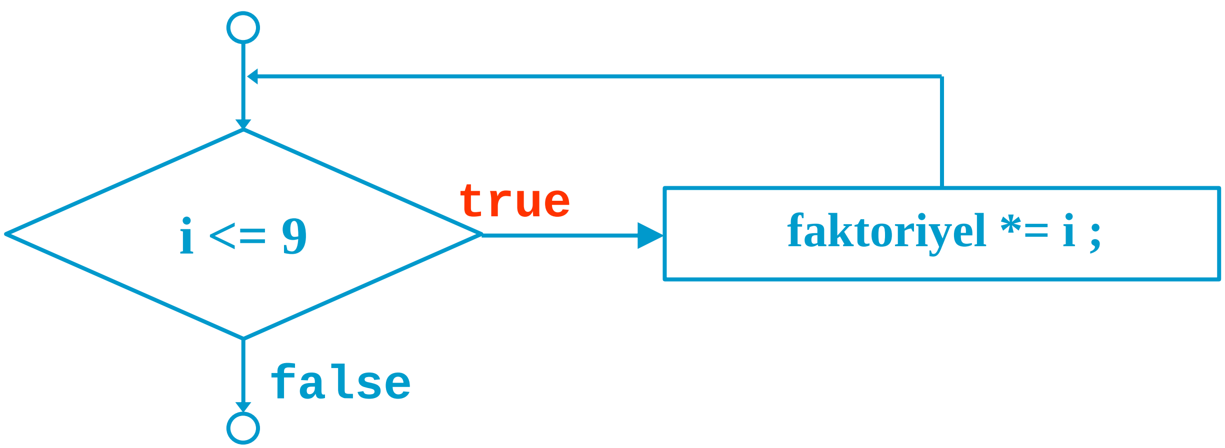
```
while (boolean test) {
      statement or block;
}
```

```
int faktoriyel = 1, i=1;
while ( i <= 9 ){
    faktoriyel *= i;

    i++;

}

System.out.println(“9!=” + faktoriyel);
```



Looping Statements

The **`do/while`** loop:

```
do {
        statement or block;
} while (boolean test);
```

```
int faktoriyel = 1, i=1;
do {
   faktoriyel *= i;
   i++ ;
} while ( i <= 9 );
System.out.println(“9!=” + faktoriyel)
```

Special Loop Flow Control

- `break [`*`label`*`];`
- `continue [`*`label`*`];`
- `label: `*`statement`*`; // where `*`statement`*` should`
  `// be a loop.`

The **break** statement:

```
do {
    statement or block;
    if (condition is true)
      break;
    statement or block;

} while (boolean expression);
```

The **continue** statement:

```
do {
    statement or block;
    if (condition is true)
      continue;
    statement or block;

} while (boolean expression);
```

The **break** statement with a label named **outer**:

```
outer:
do {
statement or block;
   do {
        statement or block;
        if (condition is true)
          break outer;
        statement or block;

   } while (boolean expression);
statement or block;
} while (boolean expression);
Statement;
```

The **continue** statement with a label named **test**:

```
test:
do {
    statement or block;
    do {
        statement or block;
        if (condition is true)
            continue test;
        statement or block;

    } while (boolean expression);
    statement or block;

} while (boolean expression);
```

3# Identifiers, Keywords, and Types

- ►Exercise-1: “Using Loops and Branching Statements”
- ►Exercise-2: “Modifying the withdraw Method”
- ►Exercise-3: “Using Nested Loops”

microsystems

HANDS-ON LAB

An integer number is said to be a perfect number if its factors, including 1 (but not the number itself), sum to the number. For example, 6 is a perfect number because 6=1+2+3. Write a main function that determines all the perfect numbers between 1 and 5000. Print the factors of each perfect number to confirm that the number is indeed perfect.

HANDS-ON LAB

```
import java.io.* ;

public class PerfectNumber {
    public static void main(String[] args) {
            int i,j,sum                    ;

            for(j=2;j<5001;j++){
               sum = 1 ;
               for(i=2 ;i<=j/2;i++)
                  if( (j % i) == 0 ) sum += i ;
               if(sum == j){
                 System.out.print("\n"+j + " is a perfect number, since "+j+"= ");
                 for(i=1;i<=j/2;i++)
                    if ( (j%i) == 0 )
                       System.out.print(i+" ");
               }
            }
  }
}
```