Module 4

Expressions and Flow Control

Objectives

- Distinguish between instance and local variables
- Describe how to initialize instance variables
- Identify and correct a `Possible reference before assignment` compiler error
- Recognize, describe, and use Java software operators
- Distinguish between legal and illegal assignments of primitive types

Objectives

- 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

Relevance

- What types of variables are useful to programmers?
- Can multiple classes have variables with the same name and, if so, what is their scope?
- What types of control structures are used in other languages? What methods do these languages use to control flow?

Variables and Scope

Local variables are:

- Variables that are defined inside a method and are called *local*, *automatic*, *temporary*, or *stack* variables
- Variables that are created when the method is executed are destroyed when the method is exited

Variable initialization comprises the following:

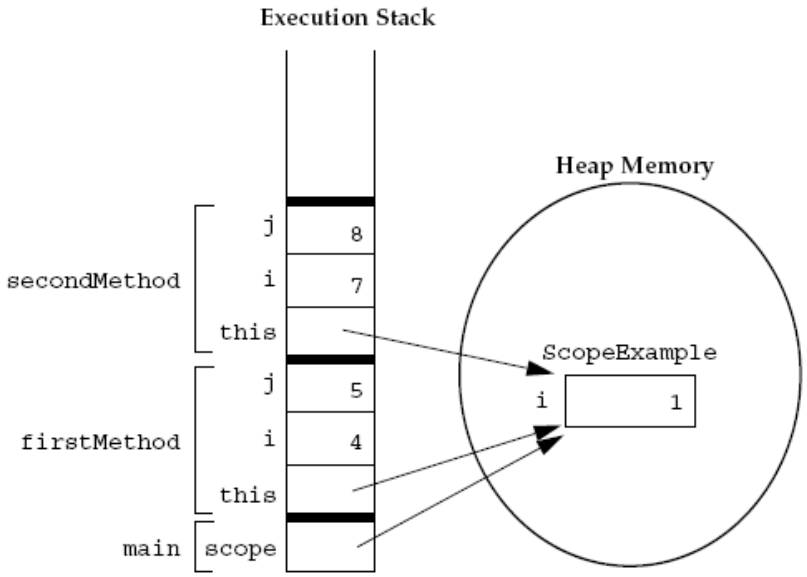
- Local variables require explicit initialization.
- Instance variables are initialized automatically.

Variable Scope Example

```
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 i) {
    int j=8;
    this.i = i + j;
  }
}
```




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

    scope.firstMethod();
  }
}
```

Variable Initialization

Variable	Value
byte	0
short	0
int	0
long	0L
float	0.0F
double	0.0D
char	'\u0000'
boolean	false
All reference types	null

Initialization Before Use Principle

The compiler will verify that local variables have been initialized before used.

```
3     public void doComputation() {
4       int x = (int)(Math.random() * 100);
5       int y;
6       int z;
7       if (x > 50) {
8         y = 9;
9       }
10      z = y + x;  // Possible use before initialization
11    }
```

```
javac TestInitBeforeUse.java
TestInitBeforeUse.java:10: variable y might not have been initialized
    z = y + x;  // Possible use before initialization
        ^
1 error
```

Operator Precedence

Operators	Associative
++ -- + *unary* - *unary* ~ ! (*<data_type>*)	R to L
* / %	L to R
+ -	L to R
<< >> >>>	L to R
< > <= >= instanceof	L to R
== !=	L to R
&	L to R
^	L to R
|	L to R
&&	L to R
||	L to R
<boolean_expr> ? *<expr1>* : *<expr2>*	R to L
= *= /= %= += -= <<= >>= >>>= &= ^= |=	R to L

Logical Operators

- The `boolean` operators are:

```
! - NOT     & - AND
| - OR      ^ - XOR
```

- The short-circuit `boolean` operators are:

```
&& - AND    || - OR
```

- You can use these operators as follows:

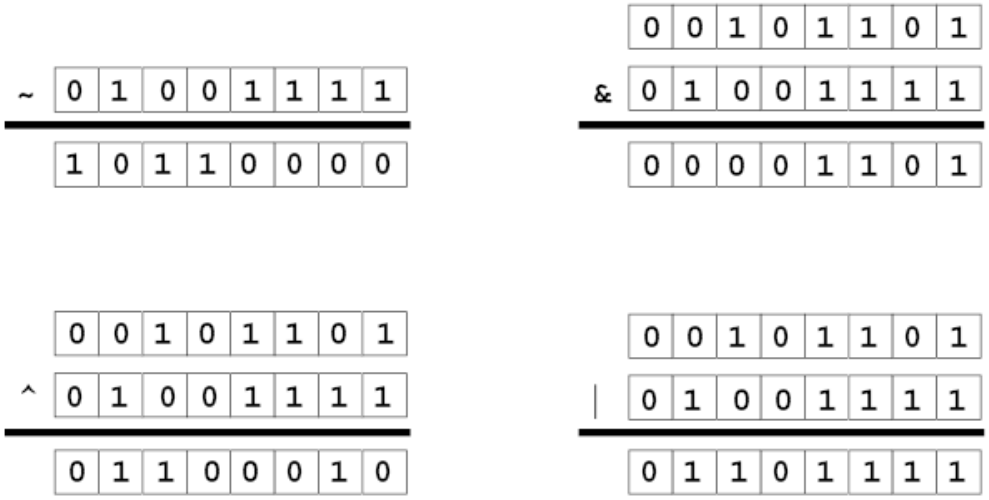
```
MyDate d = reservation.getDepartureDate();
if ( (d != null) && (d.day > 31) {
  // do something with d
}
```

Bitwise Logical Operators

- The integer *bitwise* operators are:

```
~ - Complement   & - AND
^ - XOR          | - OR
```

- Byte-sized examples include:



Right-Shift Operators >> and >>>

- *Arithmetic* or *signed* right shift (>>) operator:
  - Examples are:

```
128 >> 1  returns  128/2^1  =  64
256 >> 4  returns  256/2^4  =  16
-256 >> 4 returns -256/2^4  = -16
```

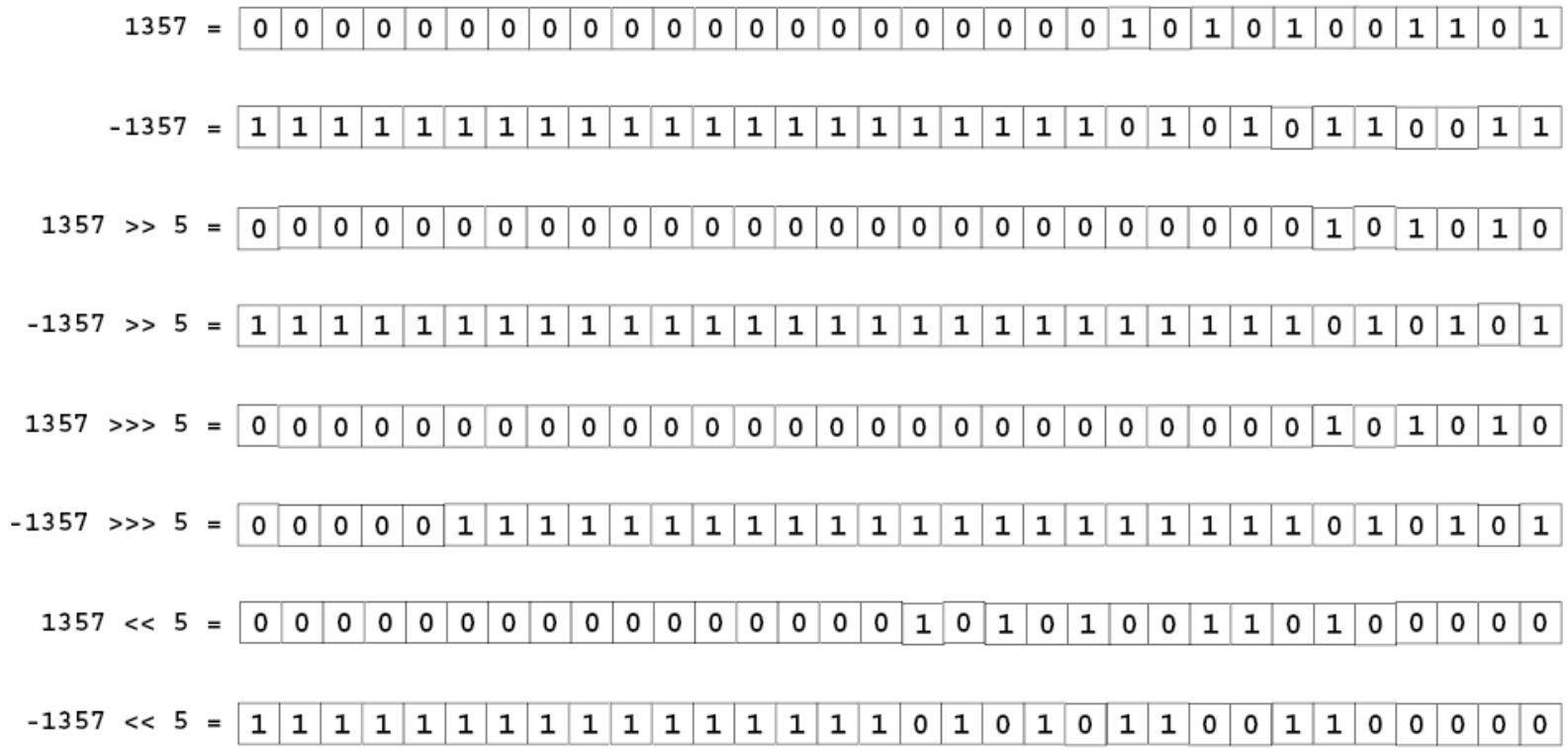
  - The sign bit is copied during the shift.
- *Logical* or *unsigned right-shift* (>>>) operator:
  - This operator is used for bit patterns.
  - The sign bit is not copied during the shift.

Left-Shift Operator <<

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

```
128 << 1 returns 128 * 2^1 = 256
16  << 2 returns  16 * 2^2 = 64
```

Shift Operator Examples



String Concatenation With +

- The + operator works as follows:
  - 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 automatically.

Casting

- If information might be lost in an assignment, the programmer must confirm the assignment with a cast.
- 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 promoted automatically 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
```

Simple `if, else` Statements

The `if` statement syntax:

```
if ( <boolean_expression> )
  <statement_or_block>
```

Example:

```
if ( x < 10 )
  System.out.println("Are you finished yet?");
```

or (*recommended*):

```
if ( x < 10 ) {
  System.out.println("Are you finished yet?");
}
```

Complex `if, else` Statements

The `if-else` statement syntax:

```
if ( <boolean_expression> )
  <statement_or_block>
else
  <statement_or_block>
```

Example:

```
if ( x < 10 ) {
  System.out.println("Are you finished yet?");
} else {
  System.out.println("Keep working...");
}
```

Complex `if, else` Statements

The `if-else-if` statement syntax:

```
if ( <boolean_expression> )
  <statement_or_block>
else if ( <boolean_expression> )
  <statement_or_block>
```

Example:

```
int count = getCount(); // a method defined in the class
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.");
}
```

Switch Statements

The `switch` statement syntax:

```
switch ( <expression> ) {
  case <constant1>:
    <statement_or_block>*
    [break;]
  case <constant2>:
    <statement_or_block>*
    [break;]
  default:
    <statement_or_block>*
    [break;]
}
```

Switch Statements

A `switch` statement example:

```
switch ( carModel ) {
  case DELUXE:
    addAirConditioning();
    addRadio();
    addWheels();
    addEngine();
    break;
  case STANDARD:
    addRadio();
    addWheels();
    addEngine();
    break;
  default:
    addWheels();
    addEngine();
}
```

Switch Statements

This `switch` statement is equivalent to the previous example:

```
switch ( carModel ) {
  case DELUXE:
    addAirConditioning();
  case STANDARD:
    addRadio();
  default:
    addWheels();
    addEngine();
}
```

Without the `break` statements, the execution falls through each subsequent `case` clause.

Looping Statements

The `for` loop:

```
for ( <init_expr>; <test_expr>; <alter_expr> )
  <statement_or_block>
```

Example:

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

or (*recommended*):

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

Looping Statements

The `while` loop:

```
while ( <test_expr> )
  <statement_or_block>
```

Example:

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

Looping Statements

The `do/while` loop:

```
do
  <statement_or_block>
while ( <test_expr> );
```

Example:

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

Special Loop Flow Control

- The **`break`** *`[<label>]`*`;` command
- The **`continue`** *`[<label>]`*`;` command
- The *`<label>`* `:` *`<statement>`* command, where *`<statement>`* should be a loop

The `break` Statement

```
1   do {
2     statement;
3     if ( condition ) {
4       break;
5     }
6     statement;
7   } while ( test_expr );
```

The `continue` Statement

```
1   do {
2     statement;
3     if ( condition ) {
4       continue;
5     }
6     statement;
7   } while ( test_expr );
```

Using `break` Statements with Labels

```
1   outer:
2     do {
3       statement1;
4       do {
5         statement2;
6         if ( condition ) {
7           break outer;
8         }
9         statement3;
10      } while ( test_expr );
11      statement4;
12    } while ( test_expr );
```

Using `continue` Statements with Labels

```
1   test:
2     do {
3       statement1;
4       do {
5         statement2;
6         if ( condition ) {
7           continue test;
8         }
9         statement3;
10      } while ( test_expr );
11      statement4;
12    } while ( test_expr );
```

Declarations and Access Controls

- **access modifiers หรือ access level :** public, protected, private (Default **ไม่ต้องใส่**)
- **class มี** modifers **ได้เพียง** 2 **แบบคือ** public **หรือ** default **จะใช้** protected **หรือ** private **ไม่ได้**
- **class และ method สามารถใช้** strictfp

Declarations and Access Controls

- method สามารถใช้ synchronized
- variable สามารถใช้ transient
- final ใช้ได้ทั้ง class, method และ variable
- abstract ใช้ได้กับ class และ method

Declarations and Access Controls

- Abstract method เป็น private หรือ final ไม่ได้
- local variable มีได้เฉพาะ final
- constant variable ใน interface เป็นได้เฉพาะ public, static และ final เช่น ภาษี
- method ใน interface เป็นได้เฉพาะ public และ abstract
- interface มี constructors ไม่ได้

Operators and Assignments

- shift operator มี >> และ << เลื่อน bit แบบคิดเครื่องหมาย และ >>> ไม่คิดเครื่องหมาย ซ้ายเป็น 0 ตลอด
- ถ้าใช้ <<< จะ compile ไม่ผ่าน
- ผลลัพธ์ของ 8 >> 1 กับ 8 >>> 1 เหมือนกันคือ 4 (เพราะ 0000-1000 เป็น 0000-0100)

Operators and Assignments

- ถ้า -128 คือ 1000-0000
  เมื่อ -128 >> 1 จะได้ -64 คือ 1100-0000 เพราะเวลาคิดจะ complement ได้ 0011-1111 แล้ว + 1 เป็น 0100-0000 = 64
  เมื่อ -128 >> 2 จะได้ -32 คือ 1110-0000 เพราะเวลาคิดจะ complement ได้ 0001-1111 แล้ว + 1 เป็น 0010-0000 = 32

Operators and Assignments

- ถ้า -6 คือ 1111-1010
  เมื่อ -6 << 1 จะได้ -12 คือ 1111-0100 เพราะเวลาคิดจะ complement ได้ 0000-1011 แล้ว + 1 เป็น 0000-1100 = 12
  เมื่อ -6 << 2 จะได้ -24 คือ 1110-1000 เพราะเวลาคิดจะ complement ได้ 0001-0111 แล้ว + 1 เป็น 0001-1000 = 24

Operators and Assignments

- ถ้า -2 คือ 1111-1110
  เมื่อ -2 >>> 1 จะได้ -1 คือ 1111-1111 เพราะเวลาคิดจะ complement ได้ 0000-0000 แล้ว + 1 เป็น 0000-0001 = 1

Operators and Assignments

- ถ้า -127 คือ 1000-0001
  เมื่อ -127 << 1 จะได้ 2 คือ 0000-0010 (เครื่องหมายคือ bit ที่ 8 หายไป)
  เมื่อ -127 >> 1 จะได้ -64 คือ 1100-0000 เพราะเวลาคิดจะ complement ได้ 0011-1111 แล้ว + 1 เป็น 0100-0000 = 64
  เมื่อ -127 >>> 1 จะได้ -64 คือ 0100-0000

Operators and Assignments

- ถ้าใช้ << 33 จะเหมือน << 1 (เพราะ 33 % 32 เหลือ 1)
- การ casting ของ shift operator

  Example :

  wrong : สำหรับ byte ถ้า x = x << 1;

  wrong : สำหรับ byte ถ้า x = (byte)x << 1;

  right : สำหรับ byte ถ้า x = (byte)(x << 1);

Operators and Assignments

- + หมายถึง String Concatenation Operator ซึ่งใช้ได้ทั้งตัวเลข และตัวอักษร

  Example :

  ```
  System.out.println(1 + 1 + "a" + "a" + 1 + 1);
  ```

  ผลลัพธ์คือ 2aa11

Operators and Assignments

- Static blocks — executes once when the class is loaded, in sequence of blocks, before main is executed

  Example :

  ```
  static {
          x = 5;
  }
  ```

  แต่ถ้า int x = 5; จะเป็น local คนละตัวกับ x ที่เป็น field ของ class

Flow Control, Exceptions and Assertions

- **try** เฉย ๆ ไม่ได้ ต้องอยู่กับ **catch** หรือ **finally** อย่างใดอย่างหนึ่ง
- ใช้คำสั่ง **return** ใน catch จะเลิกทำงานต่อเมื่อออกจาก finally
- **finally** จะทำงานเสมอ ไม่ว่าจะเข้า catch หรือไม่ เพราะทำเป็นสิ่งสุดท้ายก่อนออกจาก try

Flow Control, Exceptions and Assertions

- switch รับได้เฉพาะ byte, short, int และ char รวมถึงตัวแปรที่มี data type ข้างต้นที่เป็น final เท่านั้น
- ใน case ต้องมี break เพราะถ้าไม่มีก็จะเลื่อนไปทำ case อื่นๆโดยไม่ตรวจสอบเลย

Flow Control, Exceptions and Assertions

- default ไม่จำเป็นต้องอยู่ท้ายสุด ขอให้มี case และ default อยู่กับ break ครบคู่เท่านั้น แต่ไม่ครบก็ compile และ run ผ่าน
- เมื่อเกิด loop ไม่รู้จบขึ้น สามารถหยุดได้โดยกด Ctrl-Break
- if (true); สามารถ compile แล้ว run ผ่านด้วย แต่ if (1); compile ไม่ผ่าน

Flow Control, Exceptions and Assertions

- **label** ต้องใช้กับ loop เช่น for หรือ while มิฉะนั้นจะเกิด compilation fail
- หลังปิด } ของ try ต้องเป็น catch หรือ finally มิฉะนั้นจะเกิด compilation fail
- **finally** หรือ **catch** ของ try มีซ้ำกันไม่ได้

Flow Control, Exceptions and Assertions

- สามารถ throw ใน try เพื่อแจ้ง error ได้เลย

เช่น

```
class x {
            public static void main(String args[]) {
            try{
                throw new RuntimeException();
            } catch (RuntimeException e) {
               System.out.println("catch");}
            }
    }
```

Flow Control, Exceptions and Assertions

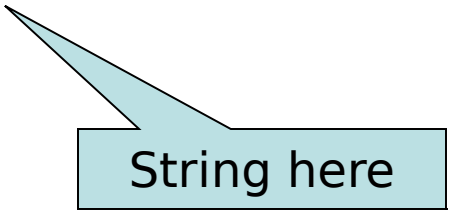
- **การปิดแฟ้ม** เช่น f.close() ต้องทำใน try เพราะทำใน finally จะเกิด compilation fail
- ถ้ามี **throw exception** ที่ไม่กำหนดใน catch จะเลิกงานหลังพบ error ใน try แล้วมาทำ finally จึงจะหยุดทำงาน เพราะ runtime
- Assertion is inappropriate for precondition check in public methods
- Assert statement may not enclosed in try-catch block

Flow Control, Exceptions and Assertions

- assert ใช้ตรวจจับสภาวะไม่พึงประสงค์ ถ้าเป็นเท็จก็จะเป็น runtime พร้อมแสดงข้อความหลัง
  เครื่องหมาย :

```
class X {
   public static void main(String args[]) {
          int j = 1;
          assert(j > 1) : "show here";
   }
}
```

compile: javac -source 1.4 X.java
run : java -ea X
run : java -da X



Flow Control, Exceptions and Assertions

```
boolean x = false;
if (x = true) { x = false; }
```

- คอมไพล์ผ่านและ run ได้ด้วย condition จะเป็น true