Chapter 3: Controlling Program Flow

3.1 Using Java operators

All operators produce a value from their operands. In addition, an operator can change the value of an operand. This is called a *side effect*.

Almost all operators work only with primitives. The exceptions are '=', '==' and '!=', which work with all objects. In addition, the **String** class supports '+' and '+='.

Programmers often forget the precedence rules, so you should use parentheses to make the order of evaluation explicit.

(1) Assignment

When you assign "from one object to another" you're actually copying a reference from one place to another. This means that if you say **C = D** for objects, you end up with both **C** and **D** pointing to the object that, originally, only **D** pointed to. ---- Aliasing

Sample : Assignment.java PassObject.java

(2) Mathematical operators

Sample : MathOps.java

The **prt( )** method prints a **String**, the **pInt( )** prints a **String** followed by an **int**.

To generate numbers, the program first creates a **Random** object. Because no arguments are passed during creation, Java uses the current time as a seed for the random number generator.

"Java=C++--" ---- suggests that Java is C++ with the unnecessary hard parts removed.

Though many parts of Java are simpler, it isn't *that* much easier than C++.

(3) Relational operators

Testing object equivalence:

Sample : Equivalence.java EqualsMethod.java EqualsMethod2.java

The default behavior of **equals( )** is to compare references. So unless you *override* **equals( )** in your new class you won't get the desired behavior.

Most of the Java library classes implement **equals( )** so that it compares the contents of objects instead of their references.

(4) Logical operators

You can't use a non-**boolean** as if it were a **boolean** in a logical expression as you can in C and C++.

Sample : Bool.java

A **boolean** value is automatically converted to an appropriate text form if it's used where a **String** is expected.

Short-circuiting:

The expression will be evaluated only *until* the truth or falsehood of the entire expression can be unambiguously determined. As a result, all the parts of a logical expression might not be evaluated.

Sample : ShortCircuit.java

You can get a potential performance increase if all the parts of a logical expression do not need to be evaluated.

(5) Shift operators

The shift operators also manipulate bits. They can be used solely with primitive, integral types.

<<, >>, >>>

The signed right shift >> uses *sign extension*: + → 0, - → 1

Java has also added the unsigned right shift **>>>,** which uses *zero extension*. This operator does not exist in C or C++.

If you shift a **char**, **byte,** or **short**, it will be promoted to **int** before the shift takes place, and the result will be an **int**. If you're operating on a **long**, you'll get a **long** result.

Sample : BitManipulation.java

Note : The high bit represents the sign: 0 means positive and 1 means negative. *Signed two's complement*

(6) String operator +

Operator overloading was added to C++ to allow the C++ programmer to add meanings to almost any operator.

Unfortunately, operator overloading turns out to be a fairly complicated feature. Java programmers cannot implement their own overloaded operators as C++ programmers can.

If an expression begins with a **String**, then all operands that follow must be **String**s.

```
int x = 0, y = 1, z = 2;
String sString = "x, y, z ";
System.out.println(sString + x + y + z);
```

(7) Common pitfalls

One of the pitfalls is trying to get away without parentheses when you are uncertain about how an expression will evaluate.

Sample : CommonPitfalls.java

(8) Casting operators

Java will automatically change one type of data into another when appropriate. Casting allows you to make this type conversion explicit, or to force it when it wouldn't normally happen.

```
void casts() {
  int i = 200;
  long l = (long)i;
  long l2 = (long)200;
}
```

It's possible to perform a cast on a numeric value as well as on a variable. The above cast is superfluous, since the compiler will automatically promote an **int** value to a **long** when necessary.

In Java, casting is safe, with the exception : *narrowing conversion*. Here the compiler forces you to do a cast.

With a *widening conversion* an explicit cast is not needed because the new type will hold the information from the old type.

Java allows you to cast any primitive type to any other primitive type, except for **boolean.** Class types do not allow casting.

Literals :

We may add some extra information associated with the literal value to let the compiler know the exact type.

Sample : Literals.java

There is no literal representation for binary numbers in C, C++ or Java.

Promotion :

If you perform any mathematical or bitwise operations on primitive data types that are smaller than an **int** (**char**, **byte,** or **short**), those values will be promoted to **int** before performing the operations, and the resulting value will be **int**.

So if you want to assign back into the smaller type, you must use a cast.

In general, the largest data type in an expression is the one that determines the size of the result.

(9) Java has no "sizeof"

The most compelling need for **sizeof( )** in C and C++ is portability.

Java does not need a **sizeof( )** operator because all the data types are the same size on all machines. You do not need to think about portability on this level.

(10) Precedence

Mnemonic	Operator type	Operators
Ulcer	Unary	+ - ++ --
Addicts	Arithmetic (and shift)	* / % + - << >>
Really	Relational	> < >= <= == !=
Like	Logical (and bitwise)	&& || & | ^
C	Conditional (ternary)	A > B ? X : Y
A Lot	Assignment	= (and compound assignment like *=)

The following example shows which primitive data types can be used with particular operators.

Sample : AllOps.java

boolean is quite limited.

Each arithmetic operation on char, byte or short results in an **int** result, which must be explicitly cast back to the original type to assign back to that type . ---- Promotion

If you multiply two **int**s that are big enough, you'll overflow the result. And you get no errors or warnings from the compiler, and no exceptions at run-time.

Compound assignments do *not* require casts for **char**, **byte,** or **short**, even though they are performing promotions that have the same results as the direct arithmetic operations.

With the exception of **boolean**, any primitive type can be cast to any other primitive type.

3.2 Execution control

Java uses all of C's execution control statements.

(1) true and false

If you want to use a non-**boolean** in a **boolean** test, such as **if(a)**, you must first convert it to a **boolean** value using a conditional expression, such as **if(a != 0)**.

(2) Iteration

while do-while for

A **for** loop performs initialization before the first iteration. Then it performs conditional testing and, at the end of each iteration, some form of "stepping".

```
public class ListCharacters {
  public static void main(String[] args) {
  for( char c = 0; c < 128; c++)
    if (c != 26 )  // ANSI Clear screen
      System.out.println(
        "value: " + (int)c +
        " character: " + c);
  }
} ///:~
```

Note : The variable **c** is defined at the point where it is used, inside the control expression of the **for** loop. The scope of **c** is the expression controlled by the **for**.

In Java and C++ you can spread your variable declarations throughout the block, defining them at the point that you need them.

You can define multiple variables within a **for** statement, but they must be of the same type.

```
for(int i = 0, j = 1;
    i < 10 && j != 11;
    i++, j++)
  /* body of for loop */;
```

The ability to define variables in the control expression is limited to the for loop.

About "goto" :

If you read the assembly code generated by virtually any compiler, you'll see that program control contains many jumps.

However, a **goto** is a jump at the source-code level.

The problem is not the use of **goto**, but the overuse of **goto**—in rare situations **goto** is actually the best way to structure control flow.

Although **goto** is a reserved word in Java, it is not used in the language. Java does have something like a jump tied in with the **break** and **continue** keywords.

A label is an identifier followed by a colon. ---- label1:

The *only* place a label is useful in Java is right before an iteration statement. And the sole reason to put a label before an iteration is if you're going to nest another iteration or a switch inside it.

When used with a label, "break and continue" will interrupt the loops up to where the label exists.

```
label1:
outer-iteration {
  inner-iteration {
    //...
    break; // 1
    //...
    continue;  // 2
    //...
    continue label1; // 3
    //...
    break label1;  // 4
  }
}
```

In case 3, the **continue label1** breaks out of the inner *and* the outer iteration. Then it does in fact continue the iteration, but starting at the outer iteration.

In case 4, the **break label1** also breaks out to **label1**, but it does not re-enter the iteration.

Sample : LabeledFor.java

Note : In this sample, the "*break* and *continue outer*" statements skip the increment.

If not for the ***break outer*** statement, there would be no way to get out of the outer loop from within an inner loop.

In the cases where breaking out of a loop will also exit the method, you can simply use a **return**.

Labels and gotos make programs difficult to analyze statically, since they introduce cycles in the program execution graph.

Java labels don't suffer from this problem, since they are constrained in their placement and can't be used to transfer control in an random manner.

(3) switch

The **switch** statement is a clean way to implement multi-way selection. but it requires a selector that evaluates to an integral value such as **int** or **char**.

Sample : VAnsC.java

Math.random( ) produces a **double.**

Casting from a **float** or **double** to an integral value always truncates.

Sample : RandomBounds.java

Consider that there are about 2^{62} different double fractions between 0 and 1.

0.0 *is* included in the output of **Math.random( )**. ---- [0,1)