Control Flow

COMS W1007
Introduction to Computer Science

Christopher Conway
3 June 2003
Overflow from Last Time: Why Types?

Assembly code is typeless. You can take any 32 bits in memory, say "this is an integer" and add it to any other 32 bits in memory. It is up to the programmer to keep track of what he is keeping where.
Why Types?

If we added \( i \) and \( f \) on the previous slide, we would hope to get \( 7.5 \), or \( 7 \), or \( 8 \). But if we treat the bits of \( x \) like an \textit{int}, we get:

\[
i + x = 1085276162
\]

Java won’t let us make that kind of mistake. Type checking finds some of our ugliest mistakes at compile time.
Char: A Special Integer

In some ways char behaves like an integer. We can do some useful arithmetic with char values.

```c
int n = '7' - '0' ;
    /* n = 7 */
```

```c
char c = 'R' - ('A' - 'a') ;
    /* c = 'r' */
```

Note: ‘0-9’ form a sequence in the Unicode character set, as do ‘a-z’ and ‘A-Z’.
Bitwise NOT

The bitwise NOT operator ‘~’ inverts every bit of an integer value.

\[
\begin{align*}
\sim 0xFF &= 0x00 \\
\sim 0x01 &= 0xFE \\
\sim 0x74 &= 0x8B
\end{align*}
\]
The Conditional Operator

The conditional operator is *ternary*—it takes three operands. The first operand is a boolean value, the second is the result of the expression if the boolean is true, the third is the result if the boolean is false.

```java
boolean a = true, b = false;

int i = a ? 0 : 1; /* i = 0 */
int j = b ? 5 : 2; /* j = 2 */

int k = i > j ? i - j : j - i;
/* k = j - i = 2 */
```
Conversions

In general, a variable can be implicitly converted to any type that is *wider* (i.e., has more bits). Integers can always be converted to floating-point. Floating-point numbers must be explicitly cast to integers.

```java
byte b = 0; int i = 1; double x = 6.2;

i = b;    /* OK: i=0 */
i = (int) x; /* OK: i=6 */
x = i;    /* OK: x=6.0 */
```
Type Casting

The type cast operator ‘(type)’ has very high precedence. Carelessness can lead to premature truncation.

```java
double x = 5.5;
int i = 4;

int a = (int) x * i;
/* a=20 */

int b = (int)( x * i );
/* b=22 */
```
Operator Precedence

We’ve now talked about 45 of Java’s 48 operators. I can’t possibly fit a full precedence chart on this slide. See Appendix D in the book for a full listing.

You should learn and trust Java’s operator precedence rules. Parenthesize when necessary and when it makes your code clearer, but try to avoid over-paranthesization.

\[ a = b \times c + d \times e \]

Is better than

\[ a = ((b) \times (c)) + ((d) \times (e)) \]
Symbolic Constants

Sometimes we want to give a name to a value that will not change. The `final` keyword allows us to assign a variable a value exactly once.

```java
final double PI = 3.141592653589793;
final int MAX_VALUE = 55;

PI += 1;
/* Error: can't modify PI */
```
If-Then

Sometimes we only want to execute certain statements if a condition holds. The \texttt{if} statement evaluates a boolean expression and only executes the next statement if the expression is \texttt{true}.

\begin{verbatim}
if( a >= b )
    max = a ;

if( a < b )
    max = b ;
\end{verbatim}

We use indentation to make clear which statement is affected by the \texttt{if}. 
If-Then-Else

If we also want to do something when the condition is false, we can use an optional `else`. The previous example becomes:

```plaintext
if( a >= b )
    max = a ;
else
    max = b ;
```

“max = b” gets executed whenever “a >= b” is not true.
If-Then-Else vs. ?: 

The previous example could also be written:

```c
int max = a >= b ? a : b;
```

The conditional operator is an expression that has a value. **if-then-else** is a control-flow statement—it changes the execution path, but it does not have a value.

If the only thing an **if-then-else** does is set a variable, we can usually rewrite it using the conditional operator. Whether we choose to do so is a matter of style and preference.
The If-Then-Else Ambiguity

What happens if we have two if statements and only one else? Which if does it apply to?

```
n = 0 ;
if( a > b )
  if( a > c )
    n = 1 ;
else
  n = 2 ;
```

What happens if a <= b? What if a > b && a <= c?
The If-Then-Else Ambiguity: 2

We resolve this ambiguity by associating every `else` with its nearest `if`.

```plaintext
n = 0 ;
if( a > b )
    if( a > c )
        n = 1 ;
    else
        n = 2 ;
```

"n = 2" is only executed if `a > b && a <= c`. 
Blocks

If we want to execute more than one statement as the result of a conditional, we can put a list of statements inside curly braces to form a block.

```plaintext
define(a, b, c)
    a = a + b
    c = c + b
    return a, c
```

```plaintext
if( num >= 0 ) {
    total += num ;
    pos++ ;
} else {
    total -= num ;
    neg++ ;
}
```
Blocks: 2

Blocks help resolve the if–then–else ambiguity:

```plaintext
n = 0;
if (a > b) {
    if (a > c)
        n = 1;
    else
        n = 1;
}
```

Now there is no question which if the else belongs to.
Scope

A *scope* is the area of a program where a variable declaration is visible (i.e., where you can use the variable). An identifier can be associated with only one variable in each scope.

Variables declared at the top-level of the `main` method have *method scope*.

```
int i = 0 ;
if( a < b ) {
    int i = 1 ;
    /* Error: i is already defined */
    ...
}
```
Block Scope

Every block has its own scope. Variables with method scope are visible within the block, but variables declared inside the block are not visible outside it. They have block scope.

```java
if( a < b ) {
    int i = 1 ;
    ...
} else {
    int i = 2 ;
    /* OK: i above is out of scope */
    ...
}
i++ ; /* Error: i is out of scope */
```
Else If

We can chain a series of mutually exclusive conditions using else if.

```java
if( n == 0 ) {
    ...
} else if( n == 1 ) {
    ...
} else if( n == 2 ) {
    ...
} else if( n == 3 ) {
    ...
} else {
    ...
}
```

The final else executes only if every if expression fails.
The Switch Statement

When the execution path is governed by the value of an integer expression, the `switch` statement gives us a more compact syntax that cascading `if-then-else` statements.

```java
switch( n ) {
    case 0: ... 
    case 1: ... 
    case 2: ... 
    case 3: ... 
    default: ... 
}
```
The Switch Statement: 2

- Each case label must be an integer or character constant.

- If the value of the switch expression matches a case label, control transfers to the statement following the label. Execution ends when a `break` statement is encountered.

- If no case label matches, control transfers to the statement following the default label. If no case label matches and there is no default label, the entire `switch` statement is skipped.
The Break Statement

Execution of a case is terminated by a break statement, not the next case. This can lead to subtle bugs:

```c
int n = 0;
switch(a) {
    case 0: n = 10; break;
    case 1: n = 20; break;
    case 2: n = 30; /* No break here */
    case 3: n = 40; break;
    /* If a==2, n=40 */
}
```
This “fall-through” behavior isn’t just a source of bugs. It can be useful.

```java
switch( digit ) { 
    case '0': case '1': case '2': case '3':
    case '4': case '5': case '6': case '7':
    case '8': case '9':
        val = digit - '0' ; break ;

    case 'A': case 'B': case 'C':
    case 'D': case 'E': case 'F':
        val = digit - 'A' + 10 ; break ;
}
```
The While Statement

If we want to repeat a statement or block an indefinite number of times, we can use the `while` statement. `while` evaluates a boolean expression and repeats its body until the expression becomes `false`.

```java
while( i < length ) {
    ...
    i++ ;
}
while( keep_playing ) { ... }
```
Infinite Loops

If the expression that controls a loop never becomes \texttt{false}, the program will enter an \textit{infinite loop}. Here’s a trivial example:

\begin{verbatim}
    while( true ) { ... }
\end{verbatim}

If the program enters an infinite loop, the user will have to forcibly quit it (Ctrl-C does this in Unix). Be careful that your loop conditions are falsifiable and any event you are waiting for can actually occur.
The Do-While Statement

If the boolean expression controlling a `while` statement is false the first time, the body of the `while` never executes. If you want the body to execute at least once, you can use `do-while`.

```java
boolean cond = false;
do {
    /* statement body */
    ...
} while( cond ) ;
```

Note: there must be a semicolon after the `while` part.
The For Statement

If we want to loop over a range of values, we can use the `for` statement. Here’s a loop that counts from 1 to 10:

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

The `for` statement is slightly more complicated than other statements we’ve seen so far. Let’s look at it components one at a time.
For: Initialization

The first component of the `for` statement is the *initialization expression*. It may contain any expression at all, but we usually just use it to set up a loop control variable.

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

Notice that the declaration of `i` is within the scope of the statement. This prevents the outer scope from getting cluttered with lots of little temporary variables.
For: Loop Condition

The second component of the for statement is the loop condition. It must be a boolean expression. The loop executes until the loop condition becomes false.

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

The loop condition is checked before every iteration of the loop, including the first.
For: Increment Expression

The last component of the for statement is the increment expression. It may contain any expression at all, but it should always modify the loop control variable.

```java
for( int i=1 ; i<=10 ; i++ )
    System.out.println("i="+i) ;
```
For: Examples

You don’t have to count by one:

```java
for( int i=0 ; i < 10 ; i+=2 )
    System.out.println("i="+i) ;
```

You don’t have to count up:

```java
for( int i=10 ; i > 0 ; i-- )
    System.out.println("i="+i) ;
```

And you can have more than one counter:

```java
for( int i=0, j=10 ; j > 0 ; i++, j-- )
    System.out.println("i="+i+" j="+j) ;
```
Nested Loops

A loop can occur inside another loop. The inner loop will execute completely in every iteration of the outer loop.

```c
int count_i = 0;
int count_j = 0;
for( int i=0 ; i < 10 ; i++ ) {
    count_i++;
    for( int j=0 ; j < 10 ; j++ )
        count_j++;
}
/* count_i=10, count_j=100 */