More Programming Constructs -- Introduction

- We can now examine some additional programming concepts and constructs

- Chapter 5 focuses on:
  - internal data representation
  - conversions between one data type and another
  - more operators
  - more selection statements
  - more repetition statements

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Internal Data Representation

- We discussed earlier that every piece of information stored on a computer is represented as binary values
- What is represented by the following binary string?

```
0110000100101 0
```

- You can't tell just from the bit string itself.
- We take specific binary values and apply an interpretation to them

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Representing Integers

- There are four types of integers in Java, each providing a different bits to store the value
- Each has a sign bit. If it is 1, the number is negative; if it is 0, the number is positive

byte | s | 7 bits

short | s | 15 bits

int | s | 31 bits

long | s | 63 bits

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Two's Complement

- Integers are stored in *signed two's complement* format
- A positive value is a straightforward binary number
- A negative value is represented by inverting all of the bits of the corresponding positive value, then adding 1
- To "decode" a negative value, invert all of the bits and add 1
- Using two's complement makes internal arithmetic processing easier

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Two's Complement

- The number 25 is represented in 8 bits (`byte`) as

`00011001`

- To represent -25, first invert all of the bits

`11100110`

then add 1

`11100111`

- Note that the sign bit reversed, indicating the number is negative

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Overflow and Underflow

- Storing numeric values in a fixed storage size can lead to overflow and underflow problems
- *Overflow* occurs when a number grows too large to fit in its allocated space
- *Underflow* occurs when a number shrinks too small to fit in its allocated space
- See `Overflow.java`

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Representing Floating Point Values

- A decimal (base 10) floating point value can be defined by the following equation

$$sign * mantissa * 10^{exponent}$$

- where
  - *sign* is either 1 or -1
  - *mantissa* is a positive value that represents the significant digits of the number
  - *exponent* is a value that indicates how the decimal point is shifted relative to the mantissa

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Representing Floating Point Values

- For example, the number -843.977 can be represented by

$$-1 * 843977 * 10^{-3}$$

- Floating point numbers can be represented in binary the same way, except that the mantissa is a binary number and the base is 2 instead of 10

$$\text{sign} * \text{mantissa} * 2^{\text{exponent}}$$

- Floating point values are stored by storing each of these components in the space allotted

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Representing Characters

- As described earlier, characters are represented according to the Unicode Character Set
- The character set matches a unique number to each character to be represented
- Storing the character is therefore as simple as storing the binary version of the number that represents it
- For example, the character `'z'` has the Unicode value 122, which is represented in 16 bits as

```
0000000001111010
```


Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Representing Characters

- Because they are stored as numbers, Java lets you perform some arithmetic processing on characters
- For example, because 'A' is stored as Unicode value 65, the statement

```
char ch = 'A' + 5;
```

will store the character `'F'` in `ch` (Unicode value 70)

- This relationship is occasionally helpful

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Conversions

- Each data value and variable is associated with a particular data type
- It is sometimes necessary to convert a value of one data type to another
- Not all conversions are possible. For example, boolean values cannot be converted to any other type and vice versa
- Even if a conversion is possible, we need to be careful that information is not lost in the process

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Widening Conversions

- *Widening conversions* are generally safe because they go from a smaller data space to a larger one
- The widening conversions are:

From	To
byte	short, int, long, float, or double
short	int, long, float, or double
char	int, long, float, or double
int	long, float, or double
long	float or double
float	double

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Narrowing Conversions

- *Narrowing conversions* are more dangerous because they usually go from a smaller data space to a larger one
- The narrowing conversions are:

From	To
byte	char
short	byte or char
char	byte or short
int	byte, short, or char
long	byte, short, char, or int
float	byte, short, char, int or long
double	byte, short, char, int, long, or float

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Performing Conversions

- In Java, conversion between one data type and another can occur three ways
- *Assignment conversion* - when a value of one type is assigned to a variable of another type
- *Arithmetic promotion* - occurs automatically when operators modify the types of their operands
- *Casting* - an operator that forces a value to another type

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Assignment Conversion

- For example, if `money` is a `float` variable and `dollars` is an `int` variable (storing 82), then

```
money = dollars;
```

converts the value 82 to 82.0 when it is stored

- The value in `dollars` is not actually changed
- Only widening conversions are permitted through assignment
- Assignment conversion can also take place when passing parameters (which is a form of assignment)

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Arithmetic Promotion

- Certain operators require consistent types for their operands

- For example, if `sum` is a `float` variable and `count` is an `int` variable, then the statement

```
result = sum / count;
```

internally converts the value in `count` to a `float` then performs the division, producing a floating point result

- The value in `count` is not changed

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Casting

- A cast is an operator that is specified by a type name in parentheses
- It is placed in front of the value to be converted
- The following example truncates the fractional part of the floating point value in `money` and stores the integer portion in `dollars`

```
dollars = (int) money;
```

- The value in `money` is not changed
- If a conversion is possible, it can be done through a cast

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

More Operators

- We've seen several operators of various types: arithmetic, equality, relational
- There are many more in Java to make use of:
  - increment and decrement operators
  - logical operators
  - assignement operators
  - the conditional operator

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

The Increment and Decrement Operators

- The *increment operator* (++) adds one to its integer or floating point operand
- The *decrement operator* (--) subtracts one
- The statement

```
count++;
```

is essentially equivalent to

```
count = count + 1;
```


Copyright 1997 by John Lewis and William Loftus. All rights reserved.

The Increment and Decrement Operators

- The increment and decrement operators can be applied in prefix (before the variable) or postfix (after the variable) form
- When used alone in a statement, the prefix and postfix forms are basically equivalent. That is,

```
count++;
```

is equivalent to

```
++count;
```


Copyright 1997 by John Lewis and William Loftus. All rights reserved.

The Increment and Decrement Operators

- When used in a larger expression, the prefix and postfix forms have a different effect
- In both cases the variable is incremented (decremented)
- But the value used in the larger expression depends on the form

Expression	Operation	Value of Expression
`count++`	add 1	old value
`++count`	add 1	new value
`count--`	subtract 1	old value
`--count`	subtract 1	new value

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

The Increment and Decrement Operators

- If `count` currently contains 45, then

```
total = count++;
```

assigns 45 to `total` and 46 to `count`

- If `count` currently contains 45, then

```
total = ++count;
```

assigns the value 46 to both `total` and `count`

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

The Increment and Decrement Operators

- If sum contains 25, then the statement

```
System.out.println (sum++ + "   " + ++sum +
     "   " + sum + "   " + sum--);
```

prints the following result:

```
25   27   27   27
```

and sum contains 26 after the line is complete

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Logical Operators

- There are three logical operators in Java:

Operator	Operation
!	Logical NOT
&&	Logical AND
||	Logical OR

- They all take boolean operands and produce boolean results
- Logical NOT is unary (one operand), but logical AND and OR are binary (two operands)

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Logical NOT

- The logical NOT is also called logical negation or logical complement
- If `a` is true, `!a` is false; if `a` is false, then `!a` is true
- Logical expressions can be shown using *truth tables*

a	!a
false	true
true	false

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Logical AND

- The expression `a && b` is true if both `a` and `b` are true, and false otherwise

- Truth tables show all possible combinations of all terms

`a`	`b`	`a && b`
false	false	false
false	true	false
true	false	false
true	true	true

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Logical OR

- The expression `a || b` is true if `a` or `b` or both are true, and false otherwise

a	b	a || b
false	false	false
false	true	true
true	false	true
true	true	true

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Logical Operators

- Conditions in selection statements and loops can use logical operators to form more complex expressions

```
if (total < MAX && !found)
   System.out.println ("Processing...");
```

- Logical operators have precedence relationships between themselves and other operators

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Logical Operators

- Full expressions can be evaluated using truth tables

`total < MAX`	`found`	`!found`	`total < MAX && !found`
false	false	true	false
false	true	false	false
true	false	true	true
true	true	false	false

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Assignment Operators

- Often we perform an operation on a variable, then store the result back into that variable
- Java provides *assignment operators* that simplify that process
- For example, the statement

```
num += count;
```

is equivalent to

```
num = num + count;
```


Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Assignment Operators

- There are many such assignment operators, always written as *op=* , such as:

Operator	Example	Equivalent To
`+=`	`x += y`	`x = x + y`
`-=`	`x -= y`	`x = x - y`
`*=`	`x *= y`	`x = x * y`
`/=`	`x /= y`	`x = x / y`
`%=`	`x %= y`	`x = x % y`

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Assignment Operators

- The right hand side of an assignment operator can be a complete expression

- The entire right-hand expression is evaluated first, then combined with the additional operation

- Therefore

```
result /= (total-MIN) % num;
```

is equivalent to

```
result = result / ((total-MIN) % num);
```


Copyright 1997 by John Lewis and William Loftus. All rights reserved.

The Conditional Operator

- Java has a conditional operator that evaluates a boolean condition that determines which of two expressions is evaluated
- The result of the chosen expression is the result of the entire conditional operator
- Its syntax is:

```
condition ? expression1 : expression2
```

- If the *condition* is true, *expression1* is evaluated; if it is false, *expression2* is evaluated

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

The Conditional Operator

- It is similar to an if-else statement, except that it is an expression that returns a value
- For example:

```
larger = (num1 > num2) ? num1 : num2;
```

- If `num1` is greater that `num2`, then `num1` is assigned to `larger`; otherwise, `num2` is assigned to `larger`
- The conditional operator is *ternary*, meaning it requires three operands

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

The Conditional Operator

- Another example:

```
System.out.println ("Your change is " + count +
    (count == 1) ? "Dime" : "Dimes");
```

- If `count` equals 1, `"Dime"` is printed, otherwise `"Dimes"` is printed

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

Another Selection Statement

- The `if` and the `if-else` statements are selection statements, allowing us to select which statement to perform next based on some boolean condition
- Another selection construct, called the *switch statement*, provides another way to choose the next action
- The `switch` statement evaluates an expression, then attempts to match the result to one of a series of values
- Execution transfers to statement list associated with the first value that matches

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

The `switch` Statement

- The syntax of the switch statement is:

```
switch (expression) {
    case value1:
        statement-list1
    case value2:
        statement-list2
    case …
}
```


Copyright 1997 by John Lewis and William Loftus. All rights reserved.

The `switch` Statement

- The expression must evaluate to an integral value, such as an integer or character
- The `break` statement is usually used to terminate the statement list of each case, which causes control to jump to the end of the `switch` statement and continue
- A `default` case can be added to the end of the list of cases, and will execute if no other case matches
- See `Vowels.java`

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

More Repetition Constructs

- In addition to `while` loops, Java has two other constructs used to perform repetition:
  - the `do` statement
  - the `for` statement
- Each loop type has its own unique characteristics
- You must choose which loop type to use in each situation

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

The `do` Statement

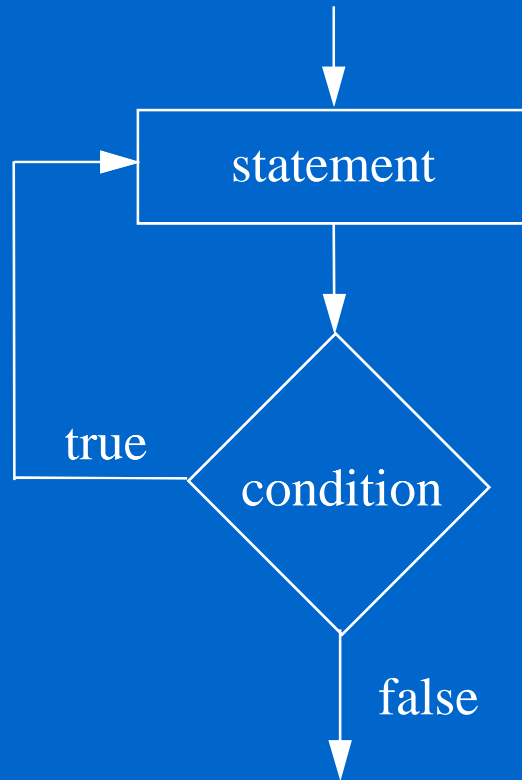
- The *do statement* has the following syntax:

```
do
    statement
while (condition);
```

- The *statement* is executed until the *condition* becomes false
- It is similar to a `while` statement, except that its termination condition is evaluated after the loop body

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

The `do` Statement



Copyright 1997 by John Lewis and William Loftus. All rights reserved.

The `do` Statement

- See `Dice.java`
- The key difference between a `do` loop and a `while` loop is that the body of the `do` loop will execute at least once
- If the condition of a `while` loop is false initially, the body of the loop is never executed
- Another way to put this is that a `while` loop will execute zero or more times and a `do` loop will execute one or more times

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

The `for` Statement

- The syntax of the *for loop* is

```
for (intialization; condition; increment)
   statement;
```

which is equivalent to

```
initialization;
while (condition) {
   statement;
   increment;
}
```

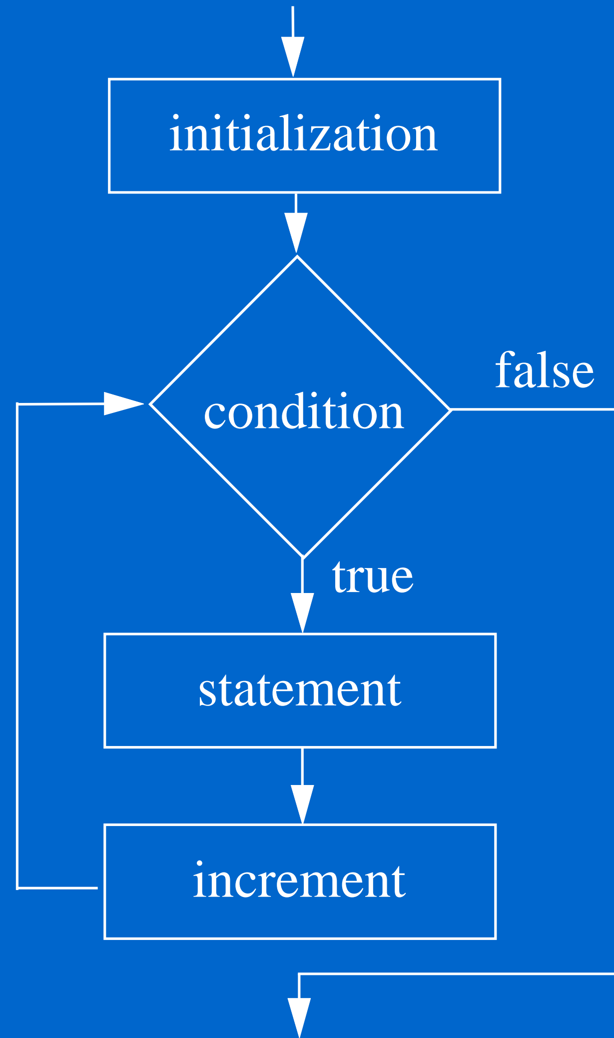

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

The `for` Statement

- Like a `while` loop, the condition of a `for` statement is tested prior to executing the loop body
- Therefore, a `for` loop will execute zero or more times
- It is well suited for executing a specific number of times, known in advance
- Note that the initialization portion is only performed once, but the increment portion is executed after each iteration

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

The `for` Statement



Copyright 1997 by John Lewis and William Loftus. All rights reserved.

The `for` Statement

- Examples:

```
for (int count=1; count < 75; count++)
    System.out.println (count);

for (int num=5; num <= total; num *= 2) {
    sum += num;
    System.out.println (sum);
}
```

- See `Dice2.java`

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

The `for` Statement

- Each expression in the header of a `for` loop is optional
  - If the initialization is left out, no initialization is performed
  - If the condition is left out, it is always considered to be true, and therefore makes an infinite loop
  - If the increment is left out, no increment opertion is performed
- Both semi-colons are always required

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

The `break` and `continue` statements

- The `break` statement, which we used with `switch` statements, can also be used inside a loop
- When the `break` statement is executed, control jumps to the statement after the loop (the condition is not evaluated again)
- A similar construct, the `continue` statement, can also be executed in a loop
- When the `continue` statement is executed, control jumps to the end of the loop and the condition is evaluated

Copyright 1997 by John Lewis and William Loftus. All rights reserved.

The `break` and `continue` Statements

- They can calso be used to jump to a line in your program with a particular *label*
- Jumping from one point in the program to another in an unstructured manner is not good practice
- Therefore, as a rule of thumb, avoid the `break` statement except when needed in `switch` statements, and avoid the `continue` statement altogether

Copyright 1997 by John Lewis and William Loftus. All rights reserved.