The statements inside your source files are generally executed from top to bottom, in the order that they appear. *Control flow statements*, however, break up the flow of execution by employing decision making, looping, and branching, enabling your program to *conditionally* execute particular blocks of code. This section describes the decision-making statements (if, if-else, switch), the looping statements (for, while, do-while), and the branching statements (break, continue, return) supported by the Java programming language.

Outline

1. The if and if-else statements
2. The switch statement
3. The while and do-while statements
4. The for statement
5. Branching statements
6. Algorithms

1. The if and if-else Statements

The if Statement

The if statement is the most basic of all the control flow statements. It tells your program to execute a certain section of code *only if* a particular test evaluates to true. For example, suppose we want to model a Bicycle, and have a Boolean isMoving which is evaluated to true whenever the bicycle is moving. We also have an int currentSpeed which indicates the current speed of the bike. If we want to allow the brakes to decrease the bicycle's speed *only if* the bicycle is already in motion, one possible implementation could be as follows:

```
// the "if" clause: bicycle must be moving
if (isMoving){
    // the "then" clause: decrease current speed
    currentSpeed--;
}
```

If this test evaluates to false (meaning that the bicycle is not in motion), control jumps to the end of the if statement.

In addition, the opening and closing braces are optional, provided that the "then" clause contains only one statement:

```
// same as above, but without braces
if (isMoving)
    currentSpeed--;
```

Deciding when to omit the braces is a matter of personal taste. Omitting them can make the code more brittle. If a second statement is later added to the "then" clause, a common mistake would be forgetting to add the newly required braces. The compiler cannot catch this sort of error; you'll just get the wrong results.

The if-else Statement

The if-else statement provides a secondary path of execution when an "if" clause evaluates to false. You could use an if-else statement in the previous example to take some action if the brakes are applied when the bicycle is not in motion. In this case, the action is to simply print an error message stating that the bicycle has already stopped.

```
if (isMoving) {
    currentSpeed--;
} else {
    System.err.println("The bicycle has already stopped!");
}
```

The following program, IfElseDemo, assigns a grade based on the value of a test score: an A for a score of 90% or above, a B for a score of 80% or above, and so on.

```
public class IfElseDemo {
  public static void main(String[] args) {

    int testscore = 76;
    char grade;

    if (testscore >= 90) {
      grade = 'A';
    } else if (testscore >= 80) {
      grade = 'B';
    } else if (testscore >= 70) {
      grade = 'C';
    } else if (testscore >= 60) {
      grade = 'D';
    } else {
      grade = 'F';
    }
```

```
        System.out.println("Grade = " + grade);
    }
}
```

The output from the program is:

```
    Grade = C
```

You may have noticed that the value of testscore can satisfy more than one expression in the compound statement: 76 >= 70 and 76 >= 60. However, once a condition is satisfied, the appropriate statements are executed (grade = 'C';) and the remaining conditions are not evaluated.

2. The Switch statement

Unlike if and if-else statements, the switch statement can have a number of possible execution paths. A switch works with the byte, short, char, and int primitive data types. It also works with *enumerated types* (see below), String class, and a few special classes that wrap certain primitive types: Character, Byte, Short, and Integer.

The following code example, SwitchDemo, declares an int named month whose value represents a month. The code displays the name of the month, based on the value of month, using the switch statement.

```
public class SwitchDemo {
    public static void main(String[] args) {


        int month = 8;
        String monthString;
        switch (month) {
            case 1:  monthString = "January";
                     break;
            case 2:  monthString = "February";
                     break;
            case 3:  monthString = "March";
                     break;
            case 4:  monthString = "April";
                     break;
            case 5:  monthString = "May";
                     break;
```

```
            case 6:  monthString = "June";
                     break;
            case 7:  monthString = "July";
                     break;
            case 8:  monthString = "August";
                     break;
            case 9:  monthString = "September";
                     break;
            case 10: monthString = "October";
                     break;
            case 11: monthString = "November";
                     break;
            case 12: monthString = "December";
                     break;
            default: monthString = "Invalid month";
                     break;
        }
        System.out.println(monthString);
    }
}
```

In this case, August is printed to standard output.

The body of a switch statement is known as a *switch block*. A statement in the switch block can be labeled with one or more case or default labels. The switch statement evaluates its expression, then executes all statements that follow the matching case label.

You could also display the name of the month with if-else statements:

```
int month = 8;
if (month == 1) {
    System.out.println("January");
} else if (month == 2) {
    System.out.println("February");
}
...  // and so on
```

Deciding whether to use if-else statements or a switch statement is based on readability and the expression that the statement is testing. An if-else statement can test expressions based on ranges of values or conditions, whereas a switch statement tests expressions based only on a single integer, enumerated value, or String object.

Another point of interest is the break statement. Each break statement terminates the enclosing switch statement. Control flow continues with the first statement following the switch block. The break statements are necessary because without them, statements in switch blocks *fall through*: All statements after the matching case label are executed in sequence, regardless of the expression of subsequent case labels, until a break statement is encountered. The program SwitchDemoFallThrough shows statements in a switch block that fall through. The program displays the month corresponding to the integer month and the months that follow in the year:

```
public class SwitchDemoFallThrough {


   public static void main(String[] args) {

      int month = 8;


      switch (month) {
         case 1:  System.out.println("January");
         case 2:  System.out.println("February");
         case 3:  System.out.println("March");
         case 4:  System.out.println("April");
         case 5:  System.out.println("May");
         case 6:  System.out.println("June");
         case 7:  System.out.println("July");
         case 8:  System.out.println("August");
         case 9:  System.out.println("September");
         case 10: System.out.println("October");
         case 11: System.out.println("November");
         case 12: System.out.println("December");
                  break;
         default: break;
      }
   }
}
```

This is the output from the code:

August

September

October

November

December

Technically, the final break is not required because flow falls out of the switch statement. Using a break is recommended so that modifying the code is easier and less error prone.
The default section handles all values that are not explicitly handled by one of the case sections.

The following code example, SwitchDemo2, shows how a statement can have multiple case labels. The code example calculates the number of days in a particular month:

```
public class SwitchDemo2 {

   public static void main(String[] args) {


      int month = 2;

      int year = 2000;

      int numDays = 0;


      switch (month) {

         case 1: case 3: case 5:

         case 7: case 8: case 10:

         case 12:

            numDays = 31;

            break;

         case 4: case 6:

         case 9: case 11:

            numDays = 30;

            break;

         case 2:

            if (((year % 4 == 0) &&

                !(year % 100 == 0))

                || (year % 400 == 0))
```

```
            numDays = 29;
        else
            numDays = 28;
        break;
      default:
        System.out.println("Invalid month.");
        break;
    }
    System.out.println("Number of Days = "
                + numDays);
  }
}
```

This is the output from the code:

```
Number of Days = 28
```

Using Strings in switch Statements

In Java SE 7 and later, you can use a String object in the switch statement's expression. The following code example, StringSwitchDemo, displays the number of the month based on the value of the String named month:

```
public class StringSwitchDemo {
        public static void main(String[] args) {
                int monthNumber = 0;
                String month = "August";
                switch (month.toLowerCase()) {
                        case "january":
                                monthNumber = 1;
                                break;
                        case "february":
                                monthNumber = 2;
                                break;
                        case "march":
                                monthNumber = 3;
                                break;
```

```
        case "april":
                monthNumber = 4;
                break;
        case "may":
                monthNumber = 5;
                break;
        case "june":
                monthNumber = 6;
                break;
        case "july":
                monthNumber = 7;
                break;
        case "august":
                monthNumber = 8;
                break;
        case "september":
                monthNumber = 9;
                break;
        case "october":
                monthNumber = 10;
                break;
        case "november":
                monthNumber = 11;
                break;
        case "december":
                monthNumber = 12;
                break;
        default:
                monthNumber = 0;
                break;
}
```

```
            if (monthNumber == 0) {
                System.out.println("Invalid month");
            } else {
                System.out.println(monthNumber);
            }
        }
}
```

The output from this code is 8.

The String in the switch expression is compared with the expressions associated with each case label as if the String.equals method were being used. In order for the StringSwitchDemo example to accept any month regardless of case, month is converted to lowercase (with the toLowerCase method), and all the strings associated with the case labels are in lowercase.

Enums

The switch statement is often used in conjunction with a reference type called an enum.

An *enum type* is a special data type that enables for a variable to be a set of predefined constants. The variable must be equal to one of the values that have been predefined for it. Common examples include compass directions (values of NORTH, SOUTH, EAST, and WEST) and the days of the week.

Because they are constants, the names of an enum type's fields are in uppercase letters.

In the Java programming language, you define an enum type by using the enum keyword. For example, you would specify a days-of-the-week enum type as:

```
public enum Day {
  SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
  THURSDAY, FRIDAY, SATURDAY
}
```

You should use enum types any time you need to represent a fixed set of constants. That includes natural enum types such as the planets in our solar system and data sets where you know all possible values at compile time—for example, the choices on a menu, command line flags, and so on.

Here is some code that shows you how to use the Day enum defined above:

```
public class EnumTest {
        public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
        }
```

```
    public static void main(String[] args) {
        Day day = Day.SATURDAY;
        switch (day) {
            case MONDAY:
                System.out.println("Mondays are bad.");
                break;
            case FRIDAY:
                System.out.println("Fridays are better.");
                break;
            case SATURDAY: case SUNDAY:
                System.out.println("Weekends are best.");
                break;
            default:
                System.out.println("Midweek days are so-so.");
                break;
        }
    }
}
```

The output is:

Weekends are best.

3. The while and do-while Statements

The while statement continually executes a block of statements while a particular condition is true. Its syntax can be expressed as:

```
while (expression) {
    statement(s)
}
```

The while statement evaluates *expression*, which must return a boolean value. If the expression evaluates to true, the while statement executes the *statement*(s) in the while block.
The while statement continues testing the expression and executing its block until the expression evaluates to false. Using the while statement to print the values from 1 through 10 can be accomplished as in the following WhileDemo program:

```
public class WhileDemo {
   public static void main(String[] args){
      int count = 1;
      while (count < 11) {
         System.out.println("Count is: " + count);
         count++;
      }
   }
}
```

You can implement an infinite loop using the while statement as follows:

```
while (true){
   // your code goes here
}
```

The Java programming language also provides a do-while statement, which can be expressed as follows:

```
do {
    statement(s)
} while (expression);
```

The difference between do-while and while is that do-while evaluates its expression at the bottom of the loop instead of the top. Therefore, the statements within the do block are always executed at least once, as shown in the following DoWhileDemo program:

```
public class DoWhileDemo {
   public static void main(String[] args){
      int count = 1;
      do {
         System.out.println("Count is: " + count);
         count++;
      } while (count < 11);
   }
}
```

4. The for Statement

The for statement provides a compact way to iterate over a range of values. Programmers often refer to it as the "for loop" because of the way in which it repeatedly loops until a particular condition is satisfied. The general form of the for statement can be expressed as follows:

```
for (initialization; termination;
    increment) {
   statement(s)
}
```

When using this version of the for statement, keep in mind that:

- The *initialization* expression initializes the loop; it's executed once, as the loop begins.
- When the *termination* expression evaluates to false, the loop terminates.
- The *increment* expression is invoked after each iteration through the loop; it is perfectly acceptable for this expression to increment *or* decrement a value.

The following program, ForDemo, uses the general form of the for statement to print the numbers 1 through 10 to standard output:

```
public class ForDemo {
   public static void main(String[] args){
      for(int i=1; i<11; i++){
         System.out.println("Count is: " + i);
      }
   }
}
```

The output of this program is:

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
```

Count is: 10

Notice how the code declares a variable within the initialization expression. The scope of this variable extends from its declaration to the end of the block governed by the for statement, so it can be used in the termination and increment expressions as well. If the variable that controls a for statement is not needed outside of the loop, it's best to declare the variable in the initialization expression. The names i, j, and k are often used to control for loops; declaring them within the initialization expression limits their life span and reduces errors.

The three expressions of the for loop are optional; an infinite loop can be created as follows:

```
// infinite loop
for ( ; ; ) {


   // your code goes here
}
```

The for statement also has another form designed for iteration through arrays. This form is sometimes referred to as the *enhanced for* statement, or the *for-each* statement. It can be used to make your loops more compact and easy to read. To demonstrate, consider the following array, which holds the numbers 1 through 10:

```
int[] numbers = {1,2,3,4,5,6,7,8,9,10};
```

The following program, EnhancedForDemo, uses the enhanced for to loop through the array:

```
public class EnhancedForDemo {
   public static void main(String[] args){
      int[] numbers =
         {1,2,3,4,5,6,7,8,9,10};
      for (int item : numbers) {
         System.out.println("Count is: " + item);
      }
   }
}
```

In this example, the variable item holds the current value from the numbers array. The output from this program is the same as before:

Count is: 1

Count is: 2

Count is: 3

Count is: 4

Count is: 5

Count is: 6

Count is: 7

Count is: 8

Count is: 9

Count is: 10

We recommend using this form of the for statement instead of the general form whenever possible.

5. Branching Statements

The break Statement

The break statement has two forms: labeled and unlabeled. You saw the unlabeled form in the previous discussion of the switch statement. You can also use an unlabeled break to terminate a for, while, or do-while loop, as shown in the following BreakDemo program:

```
public class BreakDemo {
   public static void main(String[] args) {


      int[] arrayOfInts =
         { 32, 87, 3, 589,
           12, 1076, 2000,
           8, 622, 127 };
      int searchfor = 12;


      int i;
      boolean foundIt = false;


      for (i = 0; i < arrayOfInts.length; i++) {
         if (arrayOfInts[i] == searchfor) {
            foundIt = true;
            break;
         }
      }
```

```
        if (foundIt) {
            System.out.println("Found " + searchfor + " at index " + i);
        } else {
            System.out.println(searchfor + " not in the array");
        }
    }
}
```

This program searches for the number 12 in an array. The break statement, shown in boldface, terminates the for loop when that value is found. Control flow then transfers to the statement after the for loop. This program's output is:

Found 12 at index 4

An unlabeled break statement terminates the innermost switch, for, while, or do-while statement, but a labeled break terminates an outer statement. The following program, BreakWithLabelDemo, is similar to the previous program, but uses nested for loops to search for a value in a two-dimensional array. When the value is found, a labeled break terminates the outer for loop (labeled "search"):

```
public class BreakWithLabelDemo {
    public static void main(String[] args) {

        int[][] arrayOfInts = {
            { 32, 87, 3, 589 },
            { 12, 1076, 2000, 8 },
            { 622, 127, 77, 955 }
        };
        int searchfor = 12;

        int i;
        int j = 0;
        boolean foundIt = false;

    search:
        for (i = 0; i < arrayOfInts.length; i++) {
            for (j = 0; j < arrayOfInts[i].length;
                 j++) {
```

```
            if (arrayOfInts[i][j] == searchfor) {
                foundIt = true;
                break search;
            }
        }
    }

    if (foundIt) {
        System.out.println("Found " + searchfor + " at " + i + ", " + j);
    } else {
        System.out.println(searchfor + " not in the array");
    }
  }
}
```

This is the output of the program.

```
Found 12 at 1, 0
```

The break statement terminates the labeled statement; it does not transfer the flow of control to the label. Control flow is transferred to the statement immediately following the labeled (terminated) statement.

The continue Statement

The continue statement skips the current iteration of a for, while , or do-while loop. The unlabeled form skips to the end of the innermost loop's body and evaluates the boolean expression that controls the loop. The following program, ContinueDemo, steps through a String, counting the occurences of the letter "p". If the current character is not a p, the continue statement skips the rest of the loop and proceeds to the next character. If it *is* a "p", the program increments the letter count.

```
public class ContinueDemo {
  public static void main(String[] args) {

    String searchMe = "peter piper picked a " + "peck of pickled peppers";
    int max = searchMe.length();
    int numPs = 0;

    for (int i = 0; i < max; i++) {
```

```
        // interested only in p's
        if (searchMe.charAt(i) != 'p')
            continue;

        // process p's
        numPs++;
      }
      System.out.println("Found " + numPs + " p's in the string.");
   }
}
```

Here is the output of this program:

```
Found 9 p's in the string.
```

To see this effect more clearly, try removing the continue statement and recompiling. When you run the program again, the count will be wrong, saying that it found 35 p's instead of 9.

A labeled continue statement skips the current iteration of an outer loop marked with the given label. The following example program, ContinueWithLabelDemo, uses nested loops to search for a substring within another string. Two nested loops are required: one to iterate over the substring and one to iterate over the string being searched. The following program, ContinueWithLabelDemo, uses the labeled form of continue to skip an iteration in the outer loop.

```
public class ContinueWithLabelDemo {
   public static void main(String[] args) {

      String searchMe = "Look for a substring in me";
      String substring = "sub";
      boolean foundIt = false;

      int max = searchMe.length() -
                substring.length();

   test:
      for (int i = 0; i <= max; i++) {
         int n = substring.length();
         int j = i;
```

```
        int k = 0;
        while (n-- != 0) {
          if (searchMe.charAt(j++) != substring.charAt(k++)) {
            continue test;
          }
        }
        foundIt = true;
          break test;
      }
      System.out.println(foundIt ? "Found it" : "Didn't find it");
  }
}
```

Here is the output from this program.

Found it

The return Statement

The last of the branching statements is the return statement. We will discuss it in detail in Lecture 3.

6. Algorithms

Loops, conditional behaviour and arrays are often combined to produce useful effects, for example searching an array for a value or sorting array elements in to order.

Search for a value in an array

The most straightforward search algorithms operate by iterating over the array contents and testing each element for some property of interest. A sequential search might be performed as follows (where we search for the int quarry in the Array array):

```
search:
  for (int i=0; i<array.length; i++){
      if (array[i]==quarry){
        System.out.println(i);
        break search;
      }
  }
```

Search for a value in an ordered array

This search algorithm is simple to express, a more efficient search can be conducted on an ordered array. If the elements in the array are in order the algoritm will be able to determine for any given

element in the array, not only if it matches the quarry but also if the quarry is above or below the element in the list. This gives rise to the binary search algorithm.

Binary search operates by finding the mid point of the list, finding out if that element is the element being searched for and if so returning that position. If the element is not equal to the quarry the algorithm can determine whether to continue searching in the top half or the bottom half of the list and therefore dramatically reduce the number of comparsion operations required before finding the element. Binary search may be implemented either iteratively or recusively. An iterative implementation is as follows (where we search for an int needle in an Array hayStack):

```
  int lower = 0;
  int upper = hayStack.length-1;
  int find = -1;
  search:
  while (lower <= upper){

    int mid = (lower + upper) / 2;
    if (hayStack[mid] == needle){
      find = mid;
      break search;
    }else if (hayStack[mid] < needle){
      lower = mid + 1;
    }else {
      upper = mid - 1;
    }
  }
 System.out.println(find);
```

Bubble sort

Binary search operates on sorted arrays and a number of algorithms exist to sort an array into order. For sort operations the elements in the array must have some ordering principle, for example int values can be ordered by magnitude.

One sort algorithm commonly encountered on computer science courses is bubble sort. It compares unfavourably to the best sorting algorithms for efficiency but it is relatively simple to express.

The bubble sort algorithm works by repeatedly looking at element pairs in an array and running a swap if the first element is larger than the second. By looping through the array an iteration of bubble sort is guaranteed to fill the final position in the list with the largest value in the list.

By running this bubbling up process for every element in the list then the resulting list will be ordered.

```
   for (int i=0;i<array.length;i++){

      for (int j=1;j<array.length-i;j++){

         if (array[j-1]>array[j]){

            int tmp = array[j];

            array[j] = array[j-1];
```

```
            array[j-1] = tmp;
        }
    }
}
```