What we will cover:
Method Abstraction
Stepwise Refinement
Top-Down Design
Top-Down or Bottom-Up Implementation
Arrays
Declaring Array Variables and Creating Arrays
Array Initialization
Default Values and Array Size
Processing Arrays

Method Abstraction
In computing, we deal with abstraction at many levels. Abstraction is a very important concept when programming. You should always think about your solutions in general terms to avoid making them overly specific to a particular problem. When looking at method abstraction, the goal is to keep the use of a method separate from its implementation. A client—anyone using the method—should be able to call the method without knowing how it is implemented. This concept is known as black box implementation. In science and engineering, a black box is an object that is considered based only on its input and output. The internal workings are not necessarily known. You should write your methods as black boxes where the method signature is a contract between you and anyone who might use your code. This method of programming is known as encapsulation or information hiding, a topic that will come up again when looking at object orientation.

Stepwise Refinement
Stepwise refinement is also known as divide-and-conquer. Note that all of our work involving methods has been in an attempt to decompose a larger problem into smaller and smaller problems (sub problems) and then solve those pieces individually. Note the similarity between this and the suggestion of using initial commenting early in the semester to denote where code should be written.

Top-Down Design
Now that we have the ability to write and call methods, you should start designing programs using a top-down approach. Start with the overall problem and start breaking it down into smaller pieces. Top-down design is something you have already been doing for most of the semester. For instance, our examples and projects have usually consisted of three major problems: getting input, performing some calculation, and displaying output. We can then take those three problems and break them down into smaller problems if necessary. For Project 5, the calculation portion broke down into calculating distance and determining if the point was in the circle.

Top-Down or Bottom-Up Implementation
The implementation of your methods can start at the top and work down, or at the bottom and work up. You can use method stubs (a signature and return statement, if necessary) to work as placeholders while you work on your code. The main characteristic of both is that you work incrementally; completing a piece of code and making sure it works before moving on to another. Top-down implementation starts by identifying all methods needed for the program. Coding starts at the main method and works on each method that is called from there. Stubs are used to return “dummy” values until that method is reached in implementation.

See the design and implementation of PrintCalendar.java from your book

Arrays
An array is a data structure that can store a fixed number of sequential elements of the same data type. The structure consists of a pointer to a contiguous block of memory where the block is divided into sub blocks each of which can store

---

1 Merriam-Webster defines abstract as: to consider apart from application to or association with a particular instance.
one element of the appropriate type. Consider our previous examples where we have collected numbers and presented information about those numbers (average, min, max). Without arrays, we could create multiple variables and hold those values, but that would be tedious and messy. Consider the following example:

```
import java.util.Scanner;

public class ArrayAverage {
    public static void main(String[] args) {
        int numElements = 10;
        double[] numbers = new double[numElements];
        double sum = 0.0, average = 0.0;

        Scanner input = new Scanner(System.in);
        for (int i = 0; i < numElements; i++) {
            System.out.print("Enter a number (" + (i + 1) + " of " + numElements + "): ");
            numbers[i] = input.nextDouble();
            sum += numbers[i];
        }

        average = sum / numElements;
        System.out.println("The average was: " + average);
    }
}
```

This code produces the following output:

```
Enter a number (1 of 10): 2
Enter a number (2 of 10): 8
Enter a number (3 of 10): 9
Enter a number (4 of 10): 5
Enter a number (5 of 10): 43
Enter a number (6 of 10): 3
Enter a number (7 of 10): 7
Enter a number (8 of 10): 2
Enter a number (9 of 10): 1
Enter a number (10 of 10): 78
The average was: 15.8
```

We store 10 elements in a double array and calculate their average. The array we are using is a single-dimensional array. You should think of arrays as collections of variables with the same type. In our example above, we have an array of ten double values after user input.

The array name corresponds to the starting location of the first element of the array. Each element of the array is accessed by the array name and the element’s position in the array. For example, `numbers[3]` accesses the 4th element in the array `numbers` which has the value seven. The array elements have indexes running from 0 to n-1 (where n is the size of the array). You can see this by using the debugger to view the array while the program is running.

<table>
<thead>
<tr>
<th>numbers</th>
<th>numbers[0]</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>numbers[1]</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>numbers[2]</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>numbers[3]</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>numbers[4]</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>numbers[5]</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>numbers[6]</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>numbers[7]</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>numbers[8]</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>numbers[9]</td>
<td>78</td>
</tr>
</tbody>
</table>
Each element of the array requires enough memory to hold a value of the array type, for example, doubles require eight bytes. Each element is stored in enough successive memory locations as required to hold one object of that type. Our array of ten double values would require 80 bytes (ignoring overhead for the array object).

**Declaring Array Variables and Creating Arrays**

The syntax for an array declaration in Java is:

\[
\langle \text{data type} \rangle [\ ] \text{arrayName};
\]

For example, to declare an array of strings and an array of integers:

```java
String[] studentNames;
int[] studentScores;
```

To allocate the space to hold 25 names and scores:

```java
studentNames = new String[25];
studentScores = new int[25];
```

Declaring the arrays (or in general any object) does not allocate any space in memory for the array (unlike declaring variables of a built-in type). To allocate the memory for an array the `new` operator must be used. The declaration of the array and the allocation of space for the array elements can be done in one step.

```java
String[] studentNames = new String[25];
int[] studentScores = new int[25];
```

**Initializing Arrays**

Individual elements of an array can be assigned values just like variables of built in types, for example:

```java
studentNames[0] = "Chris Williams";
studentScores[0] = 92;
```

For “small” arrays (arrays that contain few elements) a list of initial values can be provided comma separated and in curly braces. Using this syntax, the `new` operator is not needed as the compiler calls it automatically when it sees the initialization list, for example:

```java
int[] numbers  = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

Creates an array of ten integers and gives the appropriate initial values.

**Default Values and Array Size**

Arrays of numerical data types in Java are automatically initialized to zero, arrays of character types are initialized to `\u0000`, and arrays of booleans to false. You can obtain the size of an array (in integer form) by using the syntax `arrayName.length`. Using the above array, `numbers.length` will be 10.

```java
int[] numbers  = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
System.out.print("Length: " + numbers.length);
```

Prints:

Length: 10

**Processing Arrays**

We almost always use a `for` loop to process arrays (for both input and output). Because we already know the number of elements in the array, the `for` loop is the most logical choice to process arrays. In addition, since we are working with large collections of the same data type, we often want to process that whole collection in a certain way. We could initialize our array using a loop like the following example:

```java
String temp;
int[] numbers = new int[10];
for (int index = 0; index < 10; index++) {
```
temp = JOptionPane.showInputDialog(null, "Enter a number: ");
numbers[index] = Integer.parseInt(temp);
}

Notice that we should rewrite this for loop using the length property of the array we are working with instead of a literal value and use our standard i variable for the loop counter:

String temp;
int[] numbers = new int[10];

for (int i = 0; i < numbers.length; i++) {
    temp = JOptionPane.showInputDialog(null, "Enter a number: ");
    numbers[i] = Integer.parseInt(temp);
}

For test data, we could randomly initialize an array of numbers:

for (int i = 0; i < numbers.length; i++) {
    numbers[i] = (int) Math.random() * 100;
}

To print the values inside an array:

for (int i = 0; i < numbers.length; i++) {
    System.out.print(numbers[i] + " ");
}

Summing the value inside an array:

for (int i = 0; i < numbers.length; i++) {
    sum += numbers[i];
}

Finding the maximum value inside an array:

for (int i = 0; i < numbers.length; i++) {
    if (numbers[i] > max) {
        max = numbers[i];
    }
}

Class Exercise 23 - Array Practice