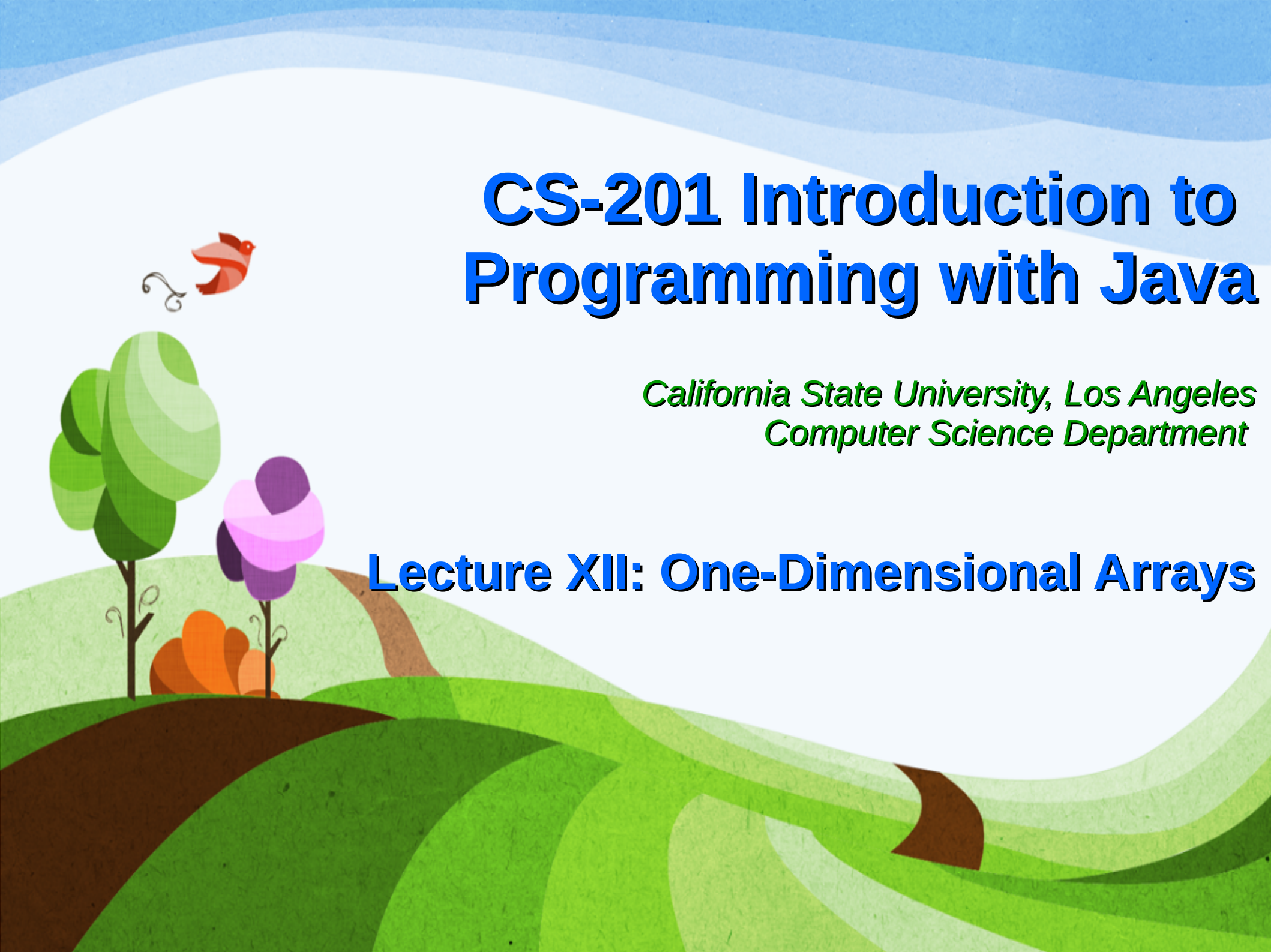
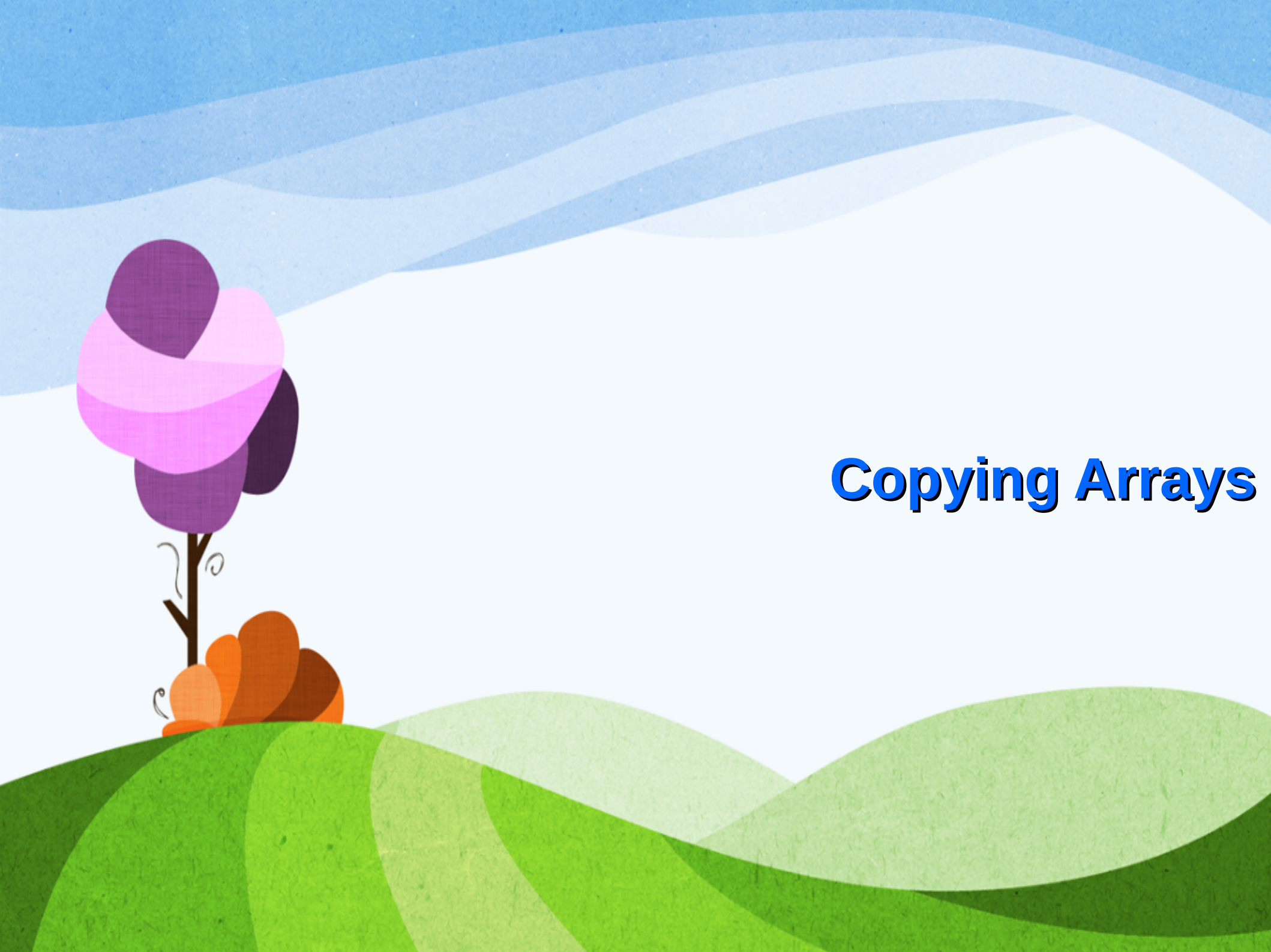


CS-201 Introduction to Programming with Java

California State University, Los Angeles
Computer Science Department

Lecture XII: One-Dimensional Arrays


Copying Arrays

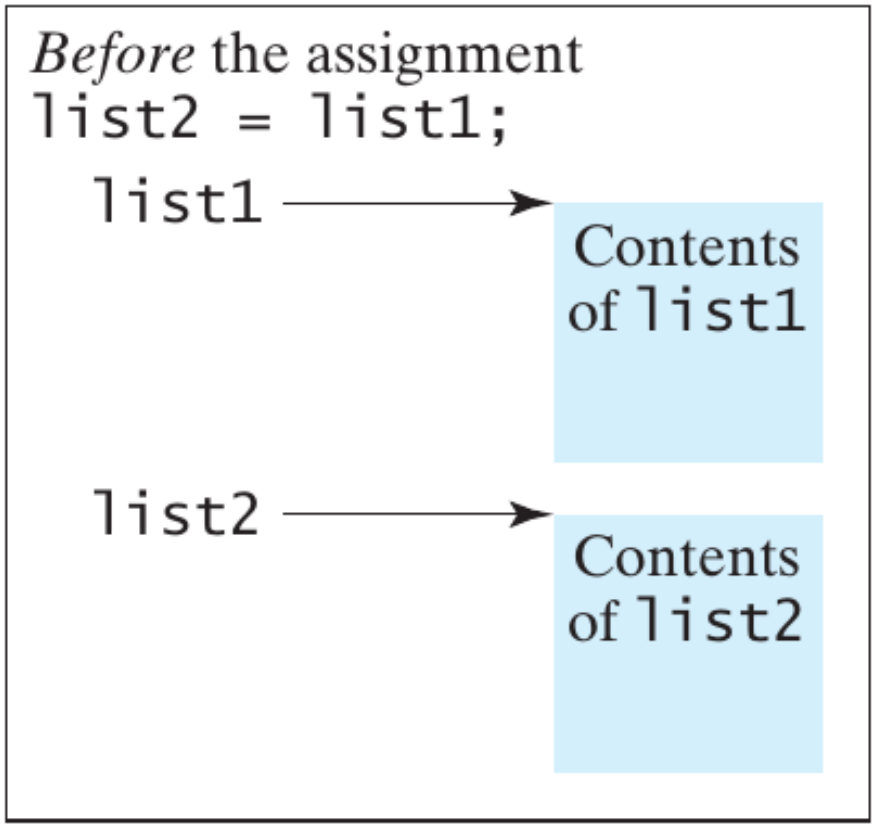
Copying Arrays

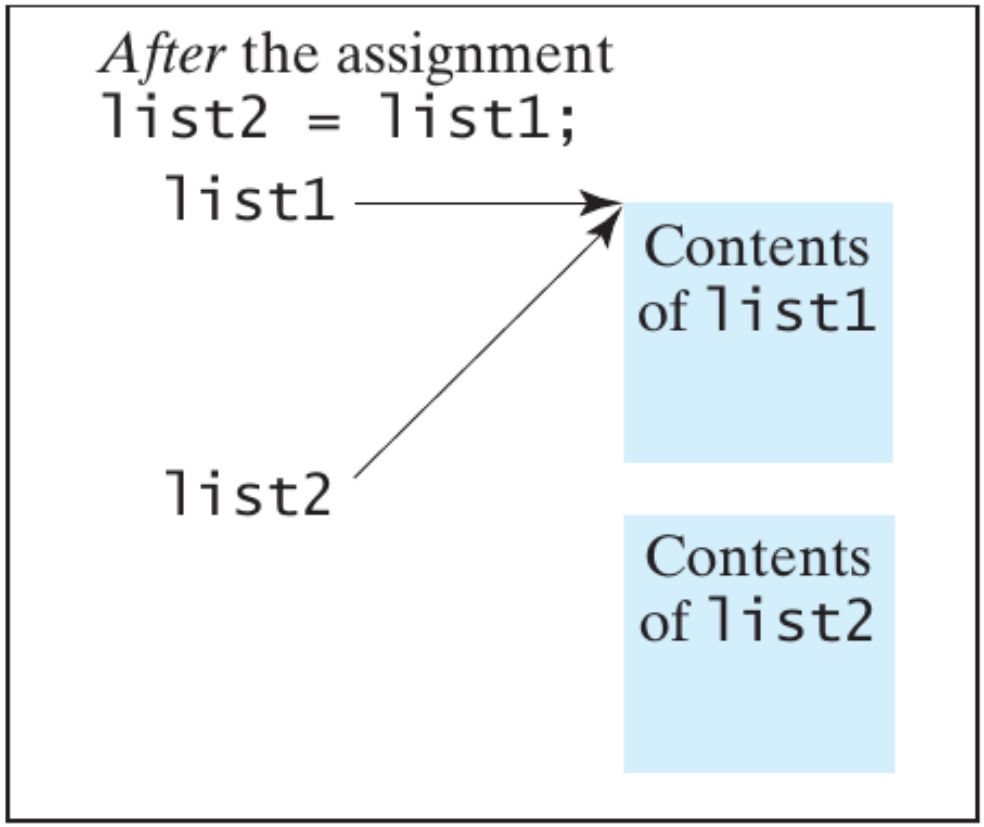
- Duplicating an array is a common occurrence in programming.
- When copying it might be tempting to do the following:

```
list2 = list1;
```

- The statement only copies the reference (memory address) from **`list1`** to **`list2`**
- It **does not** copy the contents of the array referenced by **`list1`** to **`list2`**
- After this statement, **`list1`** and **`list2`** reference to the same array so any changes to the elements in **`list1`** will make the same changes to the elements in **`list2`** and vice versa
- Also the original array that **`list2`** pointed to will now become garbage.

Copying Arrays





Copying Arrays

- In Java, you can use an assignment statement to copy primitive data type variables, **but not arrays**.

- Two ways to copy arrays
  - Using a loop to copy individual elements one by one
  - Use the static **`arraycopy`** method in the **`System`** class

Copying Arrays

- Using a loop:

```
int[] sourceArray = {2, 3, 1, 5, 10};
int[] targetArray = new int[sourceArray.length];

for (int i = 0; i < sourceArrays.length; i++) {
   targetArray[i] = sourceArray[i];
}
```

Copying Arrays

```
arraycopy(sourceArray, src_pos, targetArray, tar_pos, length);
```

- **`src_pos`** and **`tar_pos`**:
  - indicate the starting position in **`sourceArray`** and **`targetArray`**, respectively
- **`length`**:
  - the number of elements copied from **`sourceArray`** to **`targetArray`**
- After the copying, **`sourceArray`** and **`targetArray`** have the **same content but independent memory locations**
- Example:

```
System.arraycopy(sourceArray, 0, targetArray, 0, sourceArray.length);
```

- Note: The **`arraycopy`** method does not allocate memory space for the target array. The target array must have already been created with its memory space allocated.


Variable Length Parameter Lists

Variable Length Parameter Lists

- Suppose we wanted to create a method that processed a different amount of data from one invocation to the next
- For example, let's define a method called average that returns the average of a set of integer parameters

```
// one call to average three values
mean1 = average (42, 69, 37);


// another call to average seven values
mean2 = average (35, 43, 93, 23, 40, 21, 75);
```

Variable Length Parameter Lists

- We could define overloaded versions of the average method
  - Downside: we'd need a separate version of the method for each parameter count.
  - This would mean we would need an infinite number of overloaded methods...yea right...

- Instead, Java provides a convenient way to create ***variable length parameter lists***

Variable Length Parameter Lists

- Using a special syntax in the formal parameter list, we can define a method to accept any number of parameters of the same type.

- For each call, the parameters are **automatically put into an array** for easy processing in the method.

```
      Ellipses:  indicate a variable length parameter list

public double average (int ... list)
{
   //  whatever
}
                                element        array
                                 type          name
```

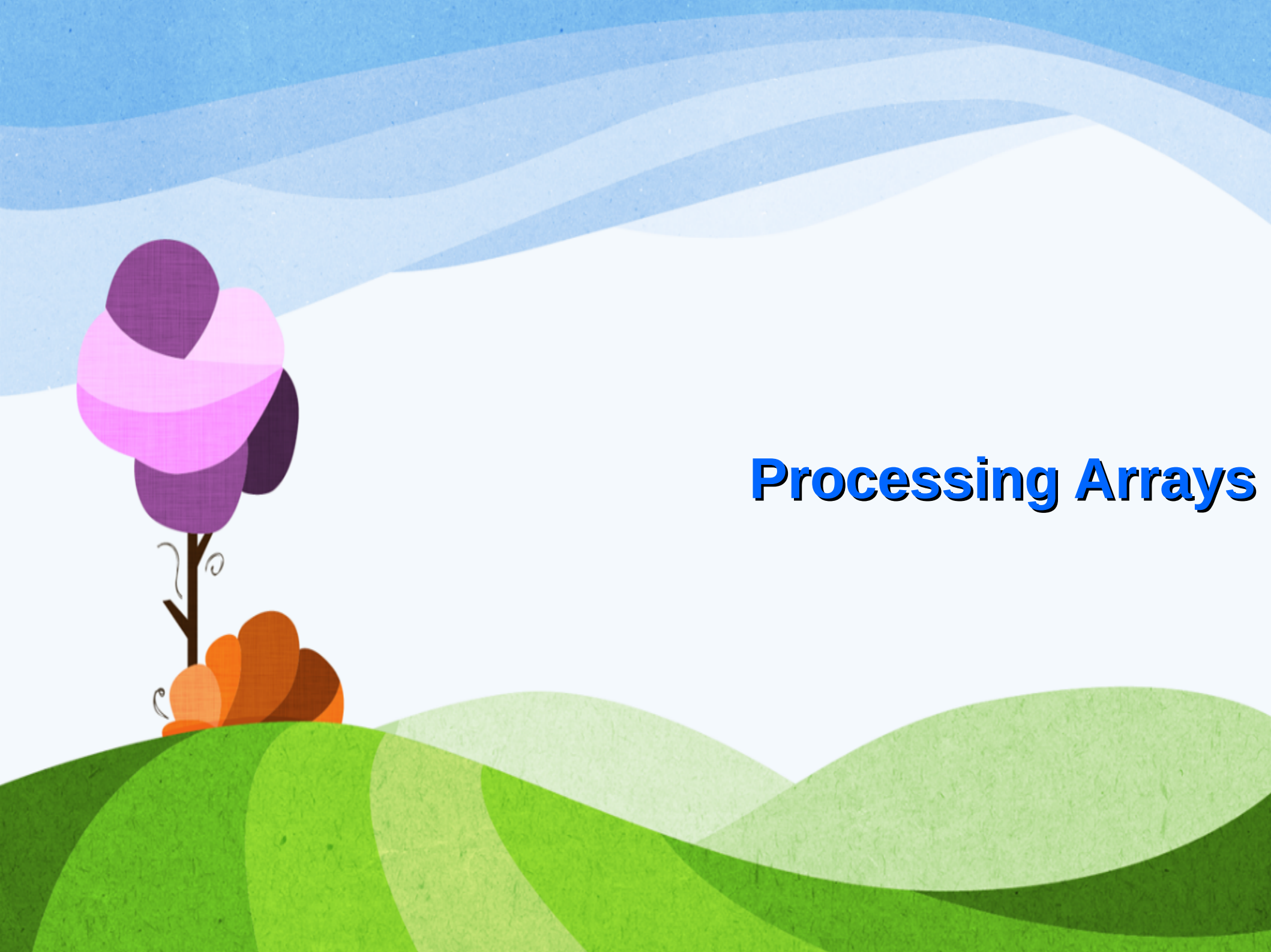
Variable Length Parameter Lists

- A method that accepts a variable length parameter list, can also accept other parameters
- The following method accepts an **`int`**, a **`String`** object, and a variable length parameter list of **`doubles`**

```
public void test(int val, String str, double ... nums) {
    // whatever
}
```

- Rules:
  - The variable length parameter list must appear last in the parameter list
  - You cannot have more than one variable length parameter list in the same method.

- See Code: **`VarArgsDemo.java`**



Processing Arrays

Processing Arrays

- Normally when array elements are processed you will need to use a loop:
  - a **for** loop is the best choice, why? because you always know the size of the array, thus you always know the number of iterations of the loop.

- All elements in an array are the same type so they can generally be processed repeatedly in the same way.

- For the following examples assume:

```
double[] myList = new double[10];
```

Initializing an Array with User Input

1. *Initializing arrays with input values:* The following loop initializes the array `myList` with user input values.

```
java.util.Scanner input = new java.util.Scanner(System.in);
System.out.print("Enter " + myList.length + " values: ");
for (int i = 0; i < myList.length; i++)
  myList[i] = input.nextDouble();
```

Initializing Arrays with Random Values

2. *Initializing arrays with random values:* The following loop initializes the array `myList` with random values between `0.0` and `100.0`, but less than `100.0`.

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

Printing an Array

- NOTE: You CANNOT print an array by simply printing the array reference variable:

```
System.out.println(arrayRefVarName);
```

  - This will only print the memory address of the first index of the array.

3. *Displaying arrays:* To print an array, you have to print each element in the array using a loop like the following:

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

Tip

For an array of the `char[]` type, it can be printed using one print statement. For example, the following code displays `Dallas`:

```
char[] city = {'D', 'a', 'l', 'l', 'a', 's'};
System.out.println(city);
```

Summing All Elements

4. *Summing all elements:* Use a variable named `total` to store the sum. Initially `total` is `0`. Add each element in the array to `total` using a loop like this:

```
double total = 0;
for (int i = 0; i < myList.length; i++) {
  total += myList[i];
}
```

Finding the Largest Element

5. *Finding the largest element:* Use a variable named `max` to store the largest element. Initially `max` is `myList[0]`. To find the largest element in the array `myList`, compare each element with `max`, and update `max` if the element is greater than `max`.

```
double max = myList[0];
for (int i = 1; i < myList.length; i++) {
  if (myList[i] > max) max = myList[i];
}
```

Finding the Smallest Index of the Largest Element

6. *Finding the smallest index of the largest element:* Often you need to locate the largest element in an array. If an array has multiple elements with the same largest value, find the smallest index of such an element. Suppose the array `myList` is {`1`, `5`, `3`, `4`, `5`, `5`}. The largest element is `5` and the smallest index for `5` is `1`. Use a variable named `max` to store the largest element and a variable named `indexOfMax` to denote the index of the largest element. Initially `max` is `myList[0]`, and `indexOfMax` is `0`. Compare each element in `myList` with `max`, and update `max` and `indexOfMax` if the element is greater than `max`.

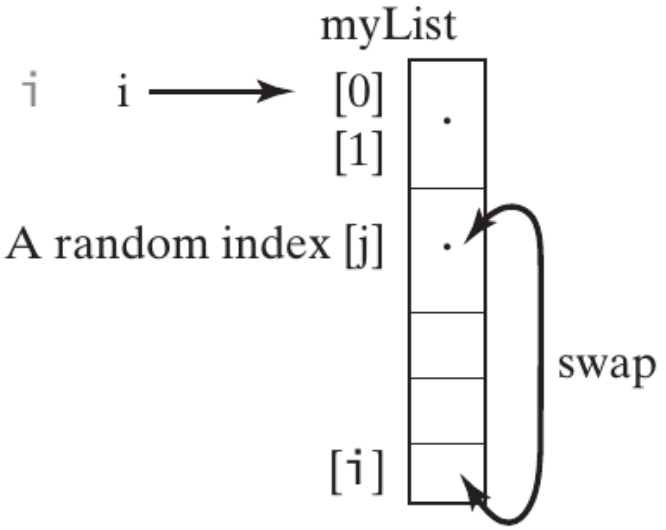
```
double max = myList[0];
int indexOfMax = 0;
for (int i = 1; i < myList.length; i++) {
  if (myList[i] > max) {
    max = myList[i];
    indexOfMax = i;
  }
}
```

Randomly Shuffling an Array

7. *Random shuffling:* In many applications, you need to randomly reorder the elements in an array. This is called *shuffling*. To accomplish this, for each element `myList[i]`, randomly generate an index `j` and swap `myList[i]` with `myList[j]`, as follows:

```
for (int i = myList.length - 1; i > 0; i--) {
  // Generate an index j randomly with 0 <= j <= i
  int j = (int)(Math.random()
    * (i + 1));

  // Swap myList[i] with myList[j]
  double temp = myList[i];
  myList[i] = myList[j];
  myList[j] = temp;
}
```



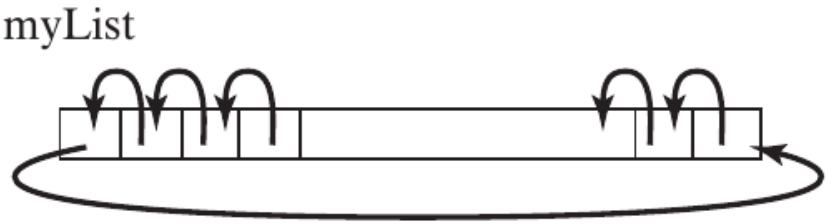
Shifting Elements

8. *Shifting elements:* Sometimes you need to shift the elements left or right. Here is an example of shifting the elements one position to the left and filling the last element with the first element:

```
double temp = myList[0]; // Retain the first element

// Shift elements left
for (int i = 1; i < myList.length; i++) {
  myList[i - 1] = myList[i];
}

// Move the first element to fill in the last position
myList[myList.length - 1] = temp;
```



Simplifying Coding

9. *Simplifying coding:* Arrays can be used to greatly simplify coding for certain tasks. For example, suppose you wish to obtain the English name of a given month by its number. If the month names are stored in an array, the month name for a given month can be accessed simply via the index. The following code prompts the user to enter a month number and displays its month name:

```
String[] months = {"January", "February", ..., "December"};
System.out.print("Enter a month number (1 to 12): ");
int monthNumber = input.nextInt();
System.out.println("The month is " + months[monthNumber - 1]);
```

If you didn't use the **months** array, you would have to determine the month name using a lengthy multi-way **if-else** statement as follows:

```
if (monthNumber == 1)
  System.out.println("The month is January");
else if (monthNumber == 2)
  System.out.println("The month is February");
...
else
  System.out.println("The month is December");
```

Out of Bounds Errors

- Remember, the index of an array must be in the range: **`0 ~ (arrayName.length - 1)`**

- If you try to access an element that is outside this range, your program will crash triggering a special runtime error.

- Runtime errors in Java (and most other programming languages) are called ***exceptions.***

- Accessing an out of bounds array index is called an **`ArrayIndexOutOfBoundsException`**.

Out of Bounds Errors

- When using a loop to process an array, it is very common to make errors in the bounding conditions of your loop triggering the previously mentioned error.

- Loops can fail to process an array when:
  - the loop begins with an index other than 0.
  - when the loop ends with an index other than length-1

Out of Bounds Errors

- Examples:

```
int[] oops = new int[10]; //indexes go from 0 to 9
```

```
for (int i = 1 ; i < oops.length - 1; i++)
```

  - skips the first element 0, skips the last element 9, logic error, does not trigger an exception

```
for (int i = 1 ; i <= oops.length; i++)
```

  - skips the first element 0, tries to access a value at index 10, triggers an **`ArrayIndexOutOfBoundsException`**

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

  - tries to access a value at index 10, triggers an **`ArrayIndexOutOfBoundsException`**

```
for (int i = 0 ; i < oops.length - 1; i++)
```

  - skips the last element 9, logic error, does not trigger an exception

for-each Loops

- These are enhanced for loops
  - added in JDK version 1.5
  - can be used with most Java data structures including arrays
  - allows you to traverse the complete array sequentially ***without*** using an index variable.

- Syntax:

```
for (elementType element: arrayRefVar) {
  //code to process the element
}
```

for-each Loops

- The following code displays all the elements in the array **myList**
- The code can be read as "for each element item in **myList**, do the following."

```
for (double item: myList) {
  System.out.println(item);
}
```

- which is equivalent to the following for loop:

```
for (int i = 0; i < myList.length; i++){
   System.out.println(myList[i]);
}
```

- Note: An index variable is still required if you want to traverse an array in a different order or change the elements in the array.