### Review: Creating Arrays

> Like any Java construct, arrays are **declared** and **instantiated**

#### Array Declaration

```java
TypeName[] arrayVariableName;
```

```java
int[] intArray;
String[] strArray;
Oval[] ovalArray;
```

**Note:** size must be a *fixed*, non-negative integer

#### Array Instantiation

```java
arrayVariableName = new TypeName[arraySize];
```

```java
intArray = new int[20];
strArray = new String[100];
ovalArray = new Oval[1000];
```

---

### Default Data Values

> If we do not set particular values for array contents when we initialize, they are set to the **defaults** for their type

```java
int[] intArray = new int[5];
String[] strArray = new String[5];
Oval[] ovalArray = new Oval[5];
```

This works the same way as for individual variables:

1. **primitives** (int, double, boolean, etc.) *all have* default values and can be used as such (0, 0.0, false)
2. **reference types** (String, Window, Oval, etc.) *do not* have any fixed useful value (i.e., they are all null)

---

### Review: Array Initialization

> One way to add data to an array is to **explicitly** input it

> Done at point that array is declared and initialized

```java
int[] intArray = { 0, 15, 30, 45, 60 };
String[] strArray = { “Hello”, “There” };
Oval[] ovalArray = { new Oval(0, 0, 10, 10) };
```

**Important Note:** each array now has **fixed size** (5, 2, or 1)

> Note: If we *don’t do this*, then the array that is originally created contains only **default values** of data

---

### Array Declaration

```java
TypeName[] arrayVariableName;
```

```java
int[] intArray;
String[] strArray;
Oval[] ovalArray;
```

**Note:** size must be a *fixed*, non-negative integer

### Array Instantiation

```java
arrayVariableName = new TypeName[arraySize];
```

```java
intArray = new int[20];
strArray = new String[100];
ovalArray = new Oval[1000];
```
Typical Array Creation Procedure

- Even if default values exist, they're often not what we actually want.
- Where no such defaults exist, we must supply actual objects to avoid running into null pointer errors when we run the code.
- Typical procedure is therefore to declare and instantiate an empty version, and then use a loop to fill it.

```java
String[] strArray = new String[5];
for (int i = 0; i < strArray.length; i++)
    strArray[i] = "";

Oval[] ovalArray = new Oval[5];
for (int i = 0; i < ovalArray.length; i++)
    ovalArray[i] = new Oval(0, 0, 0, 0);
```

Arrays and Methods

- Just like any other Java construct, an array can be either the input or output of a method.
- A nice way to return multiple objects of the same type from a method.

```java
private String[] arrayOp(String[] inArray)
{
    String[] outArray = new String[inArray.length];
    for (int i = 0; i < inArray.length; i++)
        outArray[i] = inArray[i];
    return outArray;
}
```

We produce and return a new output array, like the old one, only reversed.

Example: Reversing an Array in Place

- How can we reverse the array without using extra memory space for a 2nd copy?

```java
array arr = [0] [1] [2] ... [l-3] [l-2] [l-1]
arr.length = l
```

1. Swap arr[1] & arr[l-(l+1)]
4. Swap arr[0] & arr[l-1]

Coding a Reverse Method

```java
private void reverseInput(String[] inArray)
{
    for (int i = 0; i < (inArray.length / 2); i++)
    {
        int opposite = inArray.length - (i + 1);
        // Swap [i] and [opposite]
    }
}
```

Question One: How many times do we need to loop?
Answer: Half-way

Question Two: What to swap?
Answer: A[i] & A[length - (i + 1)]

Question Three: How to swap?
Swapping Elements

An important thing to remember: Java only does one thing at a time!

What happens if we run these instructions?


Start:

\[ A[0] = A[1]; \]

A[0] = A[1 - 1];
A[1 - 1] = A[0];

Arrays, Reference Types, and Methods

Note: if you input an array and change its elements in a method, the original will be changed in the process.

Arrays therefore behave like reference type objects: changing the input changes the original.

Not true for primitives: changing input does not affect original.

private void reverseInput(String[] inArray) {
    for (int i = 0; i < (inArray.length / 2); i++) {
        int opposite = inArray.length - (i + 1);
        String temp = inArray[i];
        inArray[i] = inArray[opposite];
        inArray[opposite] = temp;
    }
}

General Procedure for Swapping

If we want to swap two variables, X and Y:

Use a temporary variable.

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>temp</th>
</tr>
</thead>
<tbody>
<tr>
<td>valX</td>
<td>valY</td>
<td>--</td>
</tr>
<tr>
<td>temp = X;</td>
<td>valX</td>
<td>valY</td>
</tr>
<tr>
<td>X = Y;</td>
<td>valY</td>
<td>valX</td>
</tr>
<tr>
<td>Y = temp;</td>
<td>valY</td>
<td>valX</td>
</tr>
</tbody>
</table>

Arrays, Reference Types, and Methods

Java uses what is called pass-by-value semantics for method calls.

When we call a method with input parameters, the value of that parameter is copied and passed along, not the original.

For primitive types, the values are actual numbers, characters, booleans, etc.

Changing the value of the input parameter doesn’t change the original value back at the calling location, it only changes the copied value.

private void method1() {
    int i = 5;
    method2(i);
    System.out.println(i);
}

private void method2(int num) {
    num = num + 5;
    System.out.println(num);
}

Prints: 10
Java uses what is called pass-by-value semantics for method calls. When we call a method with input parameters, the value of that parameter is copied and passed along, not the original.

For reference types, the values are memory references (addresses).

Thus, if we go to that address and change things, we will change the original object (without actually changing its address/reference).

```java
private void method1() {
    Oval o = new Oval(0, 0, 5, 5);
    method2(o);
    System.out.println(o.getWidth());
}
```

```java
private void method2(Oval ov) {
    ov.setSize(10, 10);
    System.out.println(ov.getWidth());
}
```

Prints: 10
Prints: 5

One thing to watch out for: for reference types, the thing we pass as input is a copy of the reference (address).

Changing the copied address itself does not change the original.

We can only change the original object by actually going to the address and changing things about the object that is there.

```java
private void method1() {
    Oval o = new Oval(0, 0, 5, 5);
    method2(o);
    System.out.println(o.getWidth());
}
```

```java
private void method2(Oval ov) {
    ov = new Oval(0, 0, 10, 10);
    System.out.println(ov.getWidth());
}
```

Prints: 10
Prints: 10

Upcoming Weeks:

- **Week of March 16:**
  - Spring Break

- **Week of March 23:**
  - Classes cancelled

- **Weeks of March 30 and April 6:**
  - Watch your email for announcements from me

- **“Office” Hours:** Wing 212
  - Watch your email for announcements from me