DigiPen
INSTITUTE OF TECHNOLOGY

Chapter 8 Arrays

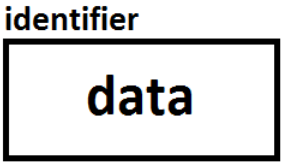
Java AP

Primitive Data Types VS Reference Data Types ?

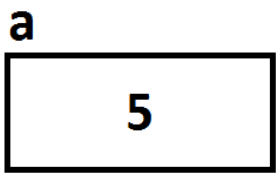
Before we cover arrays we need to cover the concept of objects. In Java we have two kind of data types: primitive data types and object data types.

A primitive type is predefined by the language and is named by a reserved keyword.

PrimitiveType identifier = data;

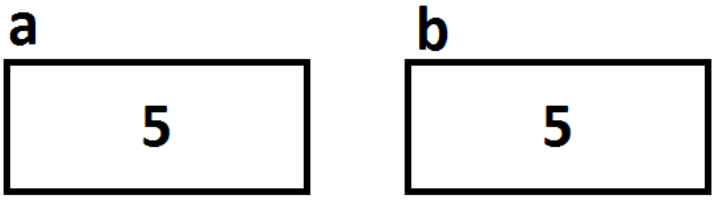


for example: *int a = 5;*



Primitive values do not share state with other primitive values.

for example: *int a = 5;*
int b = a;



The eight primitive data types supported by the Java programming language are: *byte*, *short*, *int*, *long*, *float*, *double*, *boolean* and *char*.

Example

```
public class Main
{
   public static void main(String[] args)
   {
      int a = 5;
      int b = a;

      System.out.println("a = " + a);
      System.out.println("b = " + b);

      a = 6;
      System.out.println("a = " + a);
      System.out.println("b = " + b);

      b = 7;
      System.out.println("a = " + a);
      System.out.println("b = " + b);
   }
}
```

Output

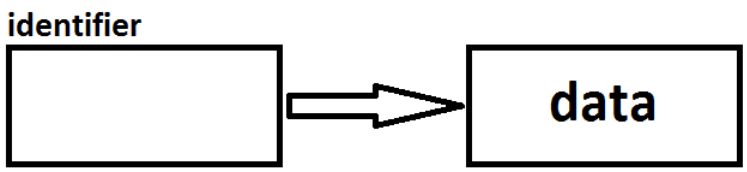
```
a = 5
b = 5
a = 6
b = 5
a = 6
b = 7
```

As you can see, even though we said **"b = a"** the two variables stayed separate. Changing the value of **a** doesn't affect **b** and vice versa.

Note: ***In addition to the eight primitive data types listed above, the Java programming language also provides special support for strings (java.lang.String). String is not a primitive data type, it is an immutable object (which means that once created, their values cannot be changed). Immutable objects can be thought of as primitive types since they behave a lot like one.***
We will cover the String object in much more details in a future chapter.

However, a reference data type (a.k.a object) is only a means to access the data.

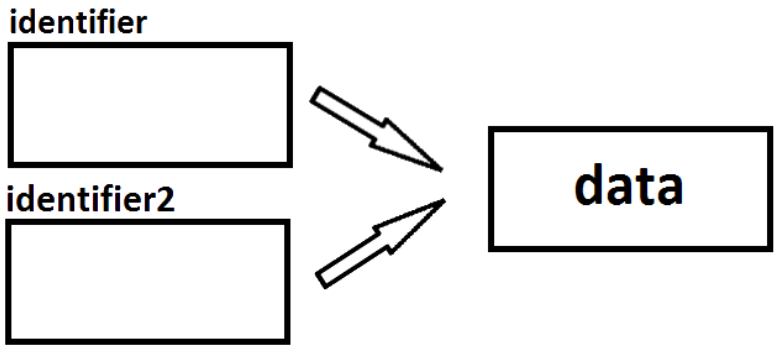
ReferenceType identifier = data;



Suppose the following declaration is made:

ReferenceType identifier = data;

ReferenceType identifier2 = identifier;



Both variables are referencing the same data, so if any of them change the data then it is changed for both. The following concept is going to be very important when learning about arrays since an array is a reference data type.

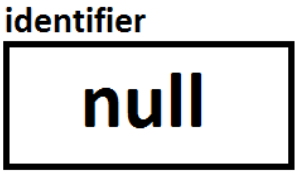
Having two references for the same data is known as *aliasing*. Aliasing can cause unintended problems for the programmer.

More detailed examples will be shown later in the chapter.

The Null Reference

Any reference type variable that is declared without being initialized is called a *null reference* or *null pointer*. In other words, a null reference is when the reference variable is created but is pointing to nothing.

ReferenceType identifier;



You can always check if the reference variable is initialized or not.

```
if ( identifier != null) /* true means identifier is initialized */
```

Note: An attempt to access data through a null reference will cause your program to terminate with a NullPointerException.

What's an Array?

An array is an ***aggregate*** data structure. This means that it consists of multiple values, all of which are the same type. Contrast this to ***scalar*** data types like **float** and **int**, which are single values. Each value in an array is called an element.

Because all of the values in an array have the same type, an array is called a homogeneous data structure. To declare an array, you must specify an additional operator **[ ]** between the type and the identifier.
The general form is:

type [ ] identifier;
or
type identifier [ ];

for example:

int [ ] a;
int a [ ];

Creating an Array

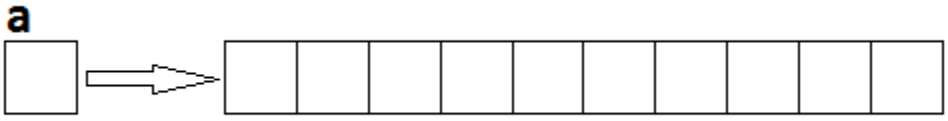
The above form is not enough to create an array since the compiler doesn't know yet how many elements the programmer wants in the array. So in order to fully create an array, you should use the following form:

type [ ] identifier = new type [size]; (where size is the number of elements in the array)
or
type identifier [ ] = new type [size];

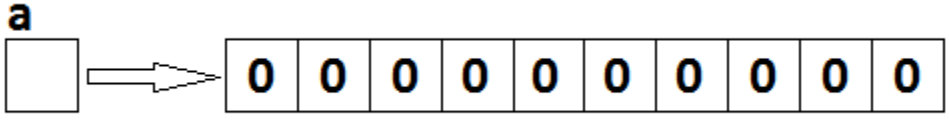
for example:

int [ ] a = new int [10];
or
int a [ ] = new int [10];

Visually, we can think of the array in memory like this:



When arrays are declared, the elements are automatically initialized to zero for the primitive data types (int and double), to false for boolean variables, or to null for object references.



PS: *All reference type variables are initialized using the **new** operator. The **new** operator returns the address of the newly constructed data so that the variable can reference it.*

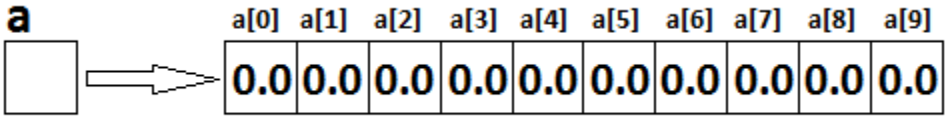
new int [10] *is creating 10 consecutive integers and returning the address of the first one*

int [] a = ... *is capturing that returned address and storing it in '**a**' so that we can access the 10 integers later through '**a**'.*

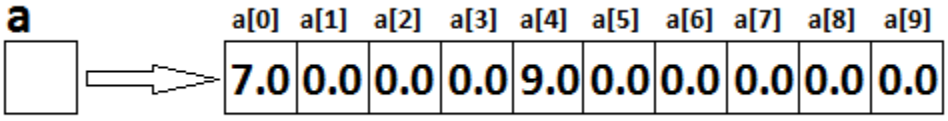
Accessing Elements of an Array

When creating an array, The name for the array applies to the entire array; all of the elements.
Each individual element is anonymous, so we can't access elements by name. We access them as *offsets* into the array. The first element has an offset of 0 (offset is also called index).

double [ ] a = new double [10];



a[0] = 7.0;
a[4] = 9.0;



Trying to access elements outside of the array, **a[-1]** or **a[10]**, will crash the application with an *Array Index Out Of Bounds Exception*.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
        at Main.main(Main.java:6)
```

Initializer List

Arrays whose values are known can be declared with an *initializer list*. The *initializer list* allows the programmer to create and initialize an array in one line.

Example:

```
public class Main
{
   public static void main(String[] args)
   {
      int a[] = {5, 7, 12, -3, 16};

      System.out.println("The value in element 0 is " + a[0]);
      System.out.println("The value in element 1 is " + a[1]);
      System.out.println("The value in element 2 is " + a[2]);
      System.out.println("The value in element 3 is " + a[3]);
      System.out.println("The value in element 4 is " + a[4]);
   }
}
```

Output:

```
The value in element 0 is 5
The value in element 1 is 7
The value in element 2 is 12
The value in element 3 is -3
The value in element 4 is 16
```

The size of array **a** in the above example is 5.
This construction is the one case where ***new*** is not required to create an array.

Other ways to use the *initializer list* which leads to the same exact output:

```
public class Main
{
   public static void main(String[] args)
   {
      int a[] = new int []{5, 7, 12, -3, 16};
      int b[] = new int []{1};
      int c[] = new int []{};
      int d[] = new int [5]{1,2,3,4,5};
   }
}
```

- Array **a** will be created, its size is 5 and the elements will contain the values according to the order given.
- Array **b** will be created, its size is 1, and the first and only element will contain the value 1
- Array **c** will be created and its size will be 0. At this point, the array **c** is useless.
- Creating array **d** in that matter will lead to an error

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
    Cannot define dimension expressions when an array initializer is provided

    at Main.main(Main.java:8)
```

Looping Through an Array

Since array elements are accessed by offsets, when programmers need to apply the same commands on all the elements it is highly recommended to use any sort of a loop.
Let's start with a simple example, initializing all the elements of an array using a loop then multiplying all the elements by 5.

Example:

```
public class Main
{
   public static void main(String[] args)
   {
      int a[] = new int [10];

      /* Initializing the array */
      for(int i = 0; i < 10; ++i)
      {
         a[i] = i * 2;
      }

      /* Printing the array */
      for(int j = 0; j < 10; ++ j)
      {
         System.out.println("The value in element " + j + " is " + a[j]);
      }

      System.out.format("%n%n");

      /* Multiplying all elements by 5 */
      for(int k = 0; k < 10; ++k)
      {
         a[k] *= 5;
      }

      /* Printing the array */
      for(int j = 0; j < 10; ++ j)
      {
         System.out.println("The value in element " + j + " is " + a[j]);
      }
   }
}
```

Output:

```
The value in element 0 is 0
The value in element 1 is 2
The value in element 2 is 4
The value in element 3 is 6
The value in element 4 is 8
The value in element 5 is 10
The value in element 6 is 12
The value in element 7 is 14
The value in element 8 is 16
The value in element 9 is 18


The value in element 0 is 0
The value in element 1 is 10
The value in element 2 is 20
The value in element 3 is 30
The value in element 4 is 40
The value in element 5 is 50
The value in element 6 is 60
The value in element 7 is 70
The value in element 8 is 80
The value in element 9 is 90
```

If it wasn't for loops, the above code would've been so tedious to write. Imagine the following:

```
public class Main
{
   public static void main(String[] args)
   {
      int a[] = new int [10];

      /* Initializing the array */
      a[0] = 0;    a[1] = 2;    a[2] = 4;    a[3] = 6;    a[4] = 8;
      a[5] = 10;  a[6] = 12; a[7] = 14;  a[8] = 16;  a[9] = 18;

            /* Printing the array */
      System.out.println("The value in element " + 0 + " is " + a[0]);
      System.out.println("The value in element " + 1 + " is " + a[1]);
      System.out.println("The value in element " + 2 + " is " + a[2]);
      System.out.println("The value in element " + 3 + " is " + a[3]);
      System.out.println("The value in element " + 4 + " is " + a[4]);
      System.out.println("The value in element " + 5 + " is " + a[5]);
      System.out.println("The value in element " + 6 + " is " + a[6]);
      System.out.println("The value in element " + 7 + " is " + a[7]);
      System.out.println("The value in element " + 8 + " is " + a[8]);
      System.out.println("The value in element " + 9 + " is " + a[9]);
```

```
      .
      .
      .

   }
}
```

What if the array is made out of 1000 elements! Definitely, using loops makes the code clearer, more elegant and has less chance of unintended errors.

Common beginner's mistake

Many beginner programmers make the mistake of using the **<=** operator in the loop instead of the **<** operator when accessing the array elements which, depending on the loop's condition, might lead to accessing elements out of the bounds of the array. Always remember, for an N elements array, the first element's index is 0 and the last is N - 1

Example:

```
public class Main
{
   public static void main(String[] args)
   {
      int a[] = new int [10];

      for(int i = 0; i <= 10; ++i)
      {
         System.out.println("The value in element " + i + " is " + a[i]);
      }
   }
}
```

Output:

```
The value in element 0 is 0
The value in element 1 is 0
The value in element 2 is 0
The value in element 3 is 0
The value in element 4 is 0
The value in element 5 is 0
The value in element 6 is 0
The value in element 7 is 0
The value in element 8 is 0
The value in element 9 is 0
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
        at Main.main(Main.java:9)
```

The *length* Property

As we mentioned earlier in the chapter, the array type in Java is an object. Objects usually contain properties *(something that we will cover in a lot of depth in future chapters)*. In here, we will cover the ***length*** property which is a variable inside the array object that provides to the programmer the amount of elements found in an array. In other words, it is the size of the array.

The dot (.) operator is used to access object's properties.

Example:

```
public class Main
{
        public static void main(String[] args)
        {
           int a[] = new int [10];

           System.out.println("The size of the array is: " + a.length);
        }
}
```

Output:

```
The size of the array is: 10
```

The ***length*** property allows programmers to write more general code when using arrays. No matter how big the array is, the code can stay the same.

Example: Size 10

```
public class Main
{
   public static void main(String[] args)
   {
      int a[] = new int [10];

      for(int i = 0; i < a.length; ++i)
      {
         System.out.println("The value in element " + i + " is " + a[i]);
      }
   }
}
```

Output:

```
The value in element 0 is 0
The value in element 1 is 0
The value in element 2 is 0
The value in element 3 is 0
The value in element 4 is 0
The value in element 5 is 0
The value in element 6 is 0
The value in element 7 is 0
The value in element 8 is 0
The value in element 9 is 0
```

Example: Size 20

```
public class Main
{
   public static void main(String[] args)
   {
      int a[] = new int [20];

      for(int i = 0; i < a.length; ++i)
      {
         System.out.println("The value in element " + i + " is " + a[i]);
      }
   }
}
```

Output:

```
The value in element 0 is 0
The value in element 1 is 0
The value in element 2 is 0
The value in element 3 is 0
The value in element 4 is 0
The value in element 5 is 0
The value in element 6 is 0
The value in element 7 is 0
The value in element 8 is 0
The value in element 9 is 0
The value in element 10 is 0
The value in element 11 is 0
The value in element 12 is 0
The value in element 13 is 0
The value in element 14 is 0
The value in element 15 is 0
The value in element 16 is 0
The value in element 17 is 0
The value in element 18 is 0
The value in element 19 is 0
```

Assigning Arrays

As mentioned before, arrays are objects. Let's see what happens when an array is assigned to another array.

Example:

```
public class Main
{
   public static void main(String[] args)
   {
      int a[] = {10, 20, 30};
      int b[] = a;

          for(int i = 0; i < a.length; ++i)
          {
             System.out.println("The value of a[" + i + "] is " + a[i]);
          }

          for(int j = 0; j < b.length; ++j)
     {
        System.out.println("The value of b[" + j + "] is " + b[j]);
     }
   }
}
```

Output:

```
The value of a[0] is 10
The value of a[1] is 20
The value of a[2] is 30
The value of b[0] is 10
The value of b[1] is 20
The value of b[2] is 30
```

The values are the same, but the big question is: When we assigned **a** to **b**, did the code make a new copy of array **a** and assign it to array **b**? or are they both referencing the same elements?

If you understand how reference data types work, you might have the answer for the above question. But, I would like to show you another example before answering.

Example:

```
public class Main
{
   public static void main(String[] args)
   {
      int a[] = {10, 20, 30};
      int b[] = a;

       for(int i = 0; i < a.length; ++i)
       {
            System.out.println("The value of a[" + i + "] is " + a[i]);
      }

      for(int j = 0; j < b.length; ++j)
      {
            System.out.println("The value of b[" + j + "] is " + b[j]);
      }

      System.out.format("%n%n");

      a[0] = 12;    b[1] = 15;

      for(int i = 0; i < a.length; ++i)
      {
            System.out.println("The value of a[" + i + "] is " + a[i]);
      }

      for(int j = 0; j < b.length; ++j)
      {
            System.out.println("The value of b[" + j + "] is " + b[j]);
      }
   }
}
```

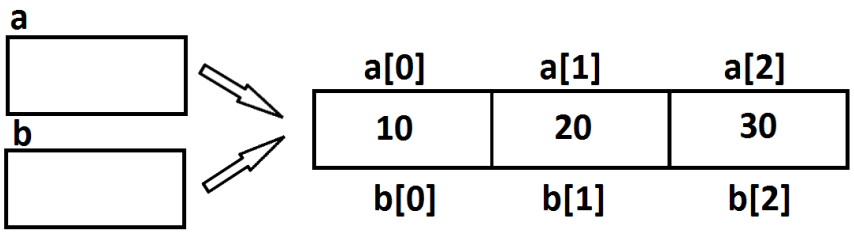
Output:

```
The value of a[0] is 10
The value of a[1] is 20
The value of a[2] is 30
The value of b[0] is 10
The value of b[1] is 20
The value of b[2] is 30


The value of a[0] is 12
The value of a[1] is 15
The value of a[2] is 30
The value of b[0] is 12
The value of b[1] is 15
The value of b[2] is 30
```

As you can see, changing values in **a** will change the values in **b** and vice versa. So array **a** and **b** are both referencing the same set of data.

```
int a[] = {10, 20, 30};
int b[] = a;
```



Arrays As Parameters

Since arrays are treated as objects, passing an array as parameter means passing its object reference to the function. It assures that no copy of the array is made. This is important since making a copy at every function call can be very memory consuming.

Example:

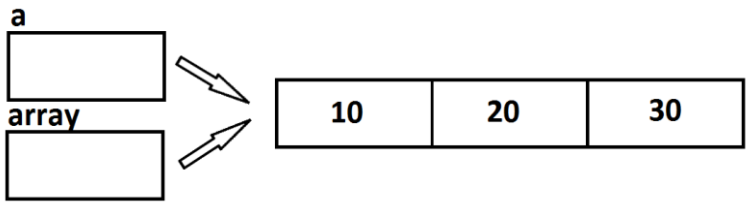
```
public class Main
{
   public static void printArray(int array[])
   {
      for(int i = 0; i < array.length; ++i)
      {
         System.out.print(array[i] + " ");
      }
   }

   public static void main(String[] args)
   {
      int a[] = {10, 20, 30};
      printArray(a);
   }
}
```

Output:

10 20 30

When calling the printArray function with **a** as a parameter, the printArray function will create the **array** object and make it point to the same address as **a.**



By doing that, the printArray function now has access to the original data created in main and can even modify it.

Example:

```
public class Main
{
   public static void printArray(int array[])
   {
      for(int i = 0; i < array.length; ++i)
      {
         System.out.print(array[i] + " ");
      }

   }

   public static void multiplyArray(int array[], int scalarValue)
   {
      for(int i = 0; i < array.length; ++i)
      {
         array[i] *= scalarValue;
      }

   }

   public static void main(String[] args)
   {
      int a[] = {10, 20, 30};
      printArray(a);
      System.out.println();
      multiplyArray(a, 5);
      printArray(a);
   }
}
```

Output:

```
10 20 30
50 100 150
```

For Each Loop

When dealing with arrays, there is another form of loop that will allow you to iterate over all elements in an array without worrying about indices or size of the array. This new form of loop is called ***for-each*** and it has the following form:

```
for (type name : array)
{
   statements...
}
```

Example:

```
public class Main
{
   public static void main(String[] args)
   {
      int array[] = {10, 20, 30};

      for (int value : array)
      {
         System.out.print(value + " ");
      }
   }
}
```

Output:

```
10 20 30
```

The above example shows how the ***for-each*** loop goes through all the elements and in this case prints the values. The difference though is that the variable value will each time contain the value of the next element in the array.

Iteration1 : ***value*** will contain ***array[0]*** content.
Iteration2 : ***value*** will contain ***array[1]*** content.
Iteration3 : ***value*** will contain ***array[2]*** content.

One important thing to know, you cannot change or remove any elements of an array with a ***for-each***.

Example:

```
public class Main
{
   public static void main(String[] args)
   {
      int array[] = {10, 20, 30};

      for (int value  : array)
      {
         System.out.print(value  + " ");
         value++;
      }

      System.out.println();

      for (int value  : array)
      {
         System.out.print(value  + " ");
      }
   }
}
```

Output:

```
10 20 30
10 20 30
```