Arrays in Java

```
class ArrayDemo {
   public static void main(String[] args) {
      int[] anArray;              // declares an array of integers
      anArray = new int[10];      // allocates memory for 10 integers
      anArray[0] = 100;           // initialize first element

      anArray[1] = 200;           // initialize second element

      System.out.println("Element at index 0: "  + anArray[0]);
      System.out.println("Element at index 1: " + anArray[1]);
      // and so forth
   }
}
```

An array declaration has two components: the array's type and the array's name. An array's type is written as type[], where type is the data type of the contained elements; the brackets are special symbols indicating that this variable holds an array. The size of the array is not part of its type.

- As with variables of other types, the declaration does not actually create an array; it simply tells the compiler that this variable will hold an array of the specified type.
- anArray = new int[10];   actually create the array assigned it to the variable.
- You can also place the brackets after the array's name: float anArrayOfFloats[];

However, Java convention discourages this form; the brackets identify the array type and should appear with the type designation.

- Alternatively, you can use the shortcut syntax to create and initialize an array:
  int[] anArray = { 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000 };
  Here the length of the array is determined by the number of values provided between braces and separated by commas.
- You can also declare an array of arrays (also known as a *multidimensional* array) by using two or more sets of brackets, such as String[][] names. Each element, therefore, must be accessed by a corresponding number of index values.
- In the Java programming language, a multidimensional array is an array whose components are themselves arrays. This is unlike arrays in C or Fortran. A consequence of this is that the rows are allowed to vary in length, as shown in the following MultiDimArrayDemo program:

```
class MultiDimArrayDemo {
   public static void main(String[] args) {
      String[][] names = {   {"Mr. ", "Mrs. ", "Ms. "},    {"Smith", "Jones"}    };
      System.out.println(names[0][0] + names[1][0]);      // Mr. Smith
      System.out.println(names[0][2] + names[1][1]);      // Ms. Jones
   }
}
```

The built-in length property determines the size of any array.

System.out.println(anArray.length);

Copying Arrays

The System class has an arraycopy method that you can use to efficiently copy data from one array into another:
public static void **arraycopy**(Object src, int srcPos, Object dest, int destPos, int length)

The two Object arguments specify the array to copy from and the array to copy to. The three int arguments specify the starting position in the source array, the starting position in the destination array, and the number of array elements to copy.

```
class ArrayCopyDemo {
   public static void main(String[] args) {
      char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e', 'i', 'n', 'a', 't', 'e', 'd' };
      char[] copyTo = new char[7];

      System.arraycopy(copyFrom, 2, copyTo, 0, 7);
      System.out.println(new String(copyTo));
   }
}
```

The output from this program is:
caffein

Array Manipulations

For your convenience, Java SE provides several methods for performing array manipulations (common tasks, such as copying, sorting and searching arrays) in the **java.util.Arrays** class. For instance, the previous example can be modified to use the copyOfRange method of the java.util.Arrays class, as you can see in the ArrayCopyOfDemo example. The difference is that using the copyOfRange method does not require you to create the destination array before calling the method, because the destination array is returned by the method:

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

      char[] copyFrom = {'d', 'e', 'c', 'a', 'f', 'f', 'e',  'i', 'n', 'a', 't', 'e', 'd'};
      char[] copyTo = java.util.Arrays.copyOfRange(copyFrom, 2, 9);
      System.out.println(new String(copyTo));
   }
}
```

As you can see, the output from this program is the same (caffein), although it requires fewer lines of code

Some other useful operations provided by methods in the java.util.Arrays class, are:

- Searching an array for a specific value to get the index at which it is placed (the binarySearch method).
- Comparing two arrays to determine if they are equal or not (the equals method).
- Filling an array to place a specific value at each index (the fill method).
- Sorting an array into ascending order. This can be done either sequentially, using the sort method, or concurrently, using the parallelSort method introduced in Java SE 8. Parallel sorting of large arrays on multiprocessor systems is faster than sequential array sorting.

Java - Character Class

Normally, when we work with characters, we use primitive data types char.
Example

```
char ch = 'a';          char uniChar = '\u039A';      // Unicode for uppercase Greek omega character

char[] charArray ={ 'a', 'b', 'c', 'd', 'e' };             // an array of chars
```

Java also provides wrapper class for each primitive type. **Character** is the wrapper class for primitive data type char.

The Character class offers a number of useful class (i.e., static) methods for manipulating characters. You can create a Character object with the Character constructor –

```
Character ch = new Character('a');
```

The Java compiler will also create a Character object for you under some circumstances. For example, if you pass a primitive char into a method that expects an object, the compiler automatically converts the char to a Character for you. This feature is called **autoboxing** or **unboxing**, if the conversion goes the other way.

Example

```
Character ch = 'a';    // Here following primitive char 'a' is boxed into the Character object ch

char c = test('x');     // Here primitive 'x' is boxed for method test return is unboxed to char 'c'
```

Character Methods
Following is the list of the important instance methods that all the subclasses of the Character class implement –

Sr.No.	Method & Description
1	isLetter() : Determines whether the specified char value is a letter.
2	isDigit() : Determines whether the specified char value is a digit.
3	isWhitespace() : Determines whether the specified char value is white space.
4	isUpperCase() : Determines whether the specified char value is uppercase.
5	isLowerCase() : Determines whether the specified char value is lowercase.
6	toUpperCase() :Returns the uppercase form of the specified char value.
7	toLowerCase() : Returns the lowercase form of the specified char value.
8	toString() : Returns a String object representing the specified character value that is, a one-character string.

For a complete list of methods, please refer to the java.lang.Character API specification.

Reference Types in Java

- All primitive types have well-defined standard sizes, so all primitive values can be stored in a fixed amount of memory. Java handles values of the primitive types directly, or by value.
- But classes and array types are composite types; objects and arrays contain other values, so they do not have a standard size. For this reason, Java does not manipulate objects and arrays directly.
- Java manipulates objects and arrays with their *references* (memory address at which the object or array is stored). Because Java handles objects and arrays by reference, classes and array types are known as reference types. In contrast,
- A reference to an object or an array is simply some fixed-size value (generally 4 byte integer) that refers to the object or array in some way. When you assign an object or array to a variable, you are actually setting the variable to hold a reference to that object or array. Similarly, when you pass an object or array to a method, what really happens is that the method is given a reference to the object or array through which it can manipulate the object or array.
- Like C++ , Java does not support the & address-of operator or the * and –> dereference operators. In Java, primitive types are always handled exclusively by value, and objects and arrays are always handled exclusively by reference. Furthermore, unlike pointers in C and C++, references in Java are entirely opaque: they cannot be converted to or from integers, and they cannot be incremented or decremented.
- Java programs cannot manipulate references in any way. Despite this, there are significant differences between the behavior of primitive types and reference types in two important areas: the way values are copied and the way they are compared for equality.

Copying Objects

Consider the following code that manipulates a primitive int value:

```
int x = 42;
int y = x;

Point p = new Point(1.0, 2.0);
Point q = p;
```

Inside the Java VM, there are two independent copies of the 32-bit integer 42.

Here the variable q holds a copy of the reference held in the variable p. There is still only one copy of the Point object in the VM, but there are now two copies of the reference to that object.

Now if we execute following lines :

```
System.out.println(p.x);  // Print out the X coordinate of p: 1.0
q.x = 13.0;               // Now change the X coordinate of q
System.out.println(p.x);  // Print out p.x again; this time it is 13.0
```

This behavior is not specific to objects; the same thing happens with arrays.

Since the variables p and q hold references to the same object, either variable can be used to make changes to the object, and changes are visible through the other variable as well.

On the other hand, consider what happens if we modify the method so that the parameter is a reference type:

```
void changeReference(Point p) {
  while(p.x > 0)
    System.out.println(p.x--);
}
```

When this method is invoked, it is passed a private copy of a reference to a Point object and can use this reference to change the Point object.

Consider the following:

```
Point q = new Point(3.0, 4.5);   // A point with an X coordinate of 3
changeReference(q);              // Prints 3,2,1 and modifies the Point
System.out.println(q.x);         // The X coordinate of q is now 0!
```

- Now both the variable q and the method parameter p hold references to the same object.
- The method can use its reference to change the contents of the object.
- However, it cannot change the contents of the variable q. In other words, it cannot change the fact that the variable q refers to that object.

Comparing Objects

When working with reference types, there are two kinds of equality:

1. equality of reference and
2. equality of object.

The method for testing whether one object is equivalent to another is named **equals**(). To test two objects for equivalence, pass one of them to the equals() method of the other:

```
String letter = "o";          String s = "hello";
String t = "hell" + letter;        // These two String objects contain exactly the same text.
if (s.equals(t))                   // And the equals() method tells us so.
  System.out.println("equivalent");
```

?
- All objects inherit an equals() method (from Object, but the default implementation simply uses = = to test for equality of references, not equivalence of content.
- A class that wants to allow objects to be compared for equivalence can define its own version of the equals() method. Our Point class does not do this, but the String class does, as indicated by the code above.

The null Reference : The default value for all reference types is null. The null value is unique in that it can be assigned to a variable of any reference type whatsoever.

Reference Type Conversions

Values of certain primitive types can be converted to values of other types. Widening conversions are performed automatically by the Java interpreter, as necessary. Narrowing conversions, however, can result in lost data, so the interpreter does not perform them unless explicitly directed to do so with a cast.

Java does not allow any kind of conversion from primitive types to reference types or vice versa. Java does allow widening and narrowing conversions among certain reference types as discussed below.

Every Java class *extends* some other class, known as its *superclass*. There is a special class named Object that serves as the root of the class hierarchy in Java.

With this simple understanding of the class hierarchy, we can return to the rules of reference type conversion:

- An object cannot be converted to an unrelated type. The Java compiler does not allow you to convert a String to a Point, for example, even if you use a cast operator.
- An object can be converted to the type of a superclass. This is a widening conversion, so no cast is required. For example, a String value can be assigned to a variable of type Object or passed to a method where an Object parameter is expected.
- An object can be converted to the type of a subclass, but this is a narrowing conversion and requires a cast. The Java compiler provisionally allows this kind of conversion, but the Java interpreter checks at runtime to make sure it is valid. If it is not, the interpreter throws a ClassCastException. For example, if we assign a String object to a variable of type Object, we can later cast the value of that variable back to type String:

```
Object o = "string";    // Widening conversion from String to Object
// Later in the program...
String s = (String) o;  // Narrowing conversion from Object to String
```

- All array types are distinct, so an array of one type cannot be converted to an array of another type, even if the individual elements could be converted. For example, although a byte can be widened to an int, a byte[] cannot be converted to an int[], even with an explicit cast.

- Arrays do not have a type hierarchy, but all arrays are considered instances of Object, so any array can be converted to an Object value through a widening conversion. A narrowing conversion with a cast can convert such an object value back to an array. For example:

```
Object o = new int[] {1,2,3};  // Widening conversion from array to Object
// Later in the program...
int[] a = (int[]) o;                 // Narrowing conversion back to array type
```