# Type casting
# and
# Non-Primitive Data Type

# Java Type Casting

Type casting is when you assign a value of one primitive data type to another type.

In Java, there are two types of casting:

- **Widening Casting** (automatically) - converting a smaller type to a larger type size
  `byte` -> `short` -> `char` -> `int` -> `long` -> `float` -> `double`

- **Narrowing Casting** (manually) - converting a larger type to a smaller size type
  `double` -> `float` -> `long` -> `int` -> `char` -> `short` -> `byte`

# Widening Casting

Widening casting is done automatically when passing a smaller size type to a larger size type:

## Example

```
public class MyClass {
  public static void main(String[] args) {
    int myInt = 9;
    double myDouble = myInt; // Automatic casting: int to double

    System.out.println(myInt);      // Outputs 9
    System.out.println(myDouble);   // Outputs 9.0
  }
}
```

# Narrowing Casting

Narrowing casting must be done manually by placing the type in parentheses in front of the value:

## Example

```
public class MyClass {
  public static void main(String[] args) {
    double myDouble = 9.78;
    int myInt = (int) myDouble; // Manual casting: double to int

    System.out.println(myDouble);   // Outputs 9.78
    System.out.println(myInt);      // Outputs 9
  }
}
```

If not casting, souce code become uncompilable becasuse incompatible type

# Java Data Types

a variable in Java must be a specified data type:

## Example

```
int myNum = 5;               // Integer (whole number)
float myFloatNum = 5.99f;    // Floating point number
char myLetter = 'D';         // Character
boolean myBool = true;       // Boolean
String myText = "Hello";     // String
```

Data types are divided into two groups:

- Primitive data types - includes `byte`, `short`, `int`, `long`, `float`, `double`, `boolean` and `char`
- Non-primitive data types - such as String, Arrays

# Numbers

Primitive number types are divided into two groups:

**Integer types** stores whole numbers, positive or negative (such as 123 or -456), without decimals. Valid types are `byte`, `short`, `int` and `long`. Which type you should use, depends on the numeric value.

**Floating point types** represents numbers with a fractional part, containing one or more decimals. There are two types: `float` and `double`.

# Primitive Data Types

A primitive data type specifies the size and type of variable values, and it has no additional methods.

There are eight primitive data types in Java:

| Data Type | Size | Description |
|---|---|---|
| byte | 1 byte | Stores whole numbers from -128 to 127 |
| short | 2 bytes | Stores whole numbers from -32,768 to 32,767 |
| int | 4 bytes | Stores whole numbers from -2,147,483,648 to 2,147,483,647 |
| long | 8 bytes | Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| float | 4 bytes | Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits |
| double | 8 bytes | Stores fractional numbers. Sufficient for storing 15 decimal digits |
| boolean | 1 bit | Stores true or false values |
| char | 2 bytes | Stores a single character/letter or ASCII values |

## Float

The `float` data type can store fractional numbers from 3.4e−038 to 3.4e+038. Note that you should end the value with an "f":

### Example

```
float myNum = 5.75f;
System.out.println(myNum);
```

Use float or double?

The precision of a floating point value is how many digits the value can have after the decimal point.

-precision of float is only 7 decimal digits

-precision of double variables is about 15 digits.

-it is safer to use double for most calculations.

## Double

The `double` data type can store fractional numbers from 1.7e−308 to 1.7e+308. Note that you should end the value with a "d":

### Example

```
double myNum = 19.99d;
System.out.println(myNum);
```

# Non-Primitive Data Types

Non-primitive data types are called reference types because they refer to objects.

- The main difference between primitive and non-primitive data types are:
  - Primitive types are predefined (already defined) in Java. Non-primitive types are created by the programmer and is not defined by Java (except for String).
  - Non-primitive types can be used to call methods to perform certain operations, while primitive types cannot.
  - A primitive type has always a value, while non-primitive types can be null.
  - The size of a primitive type depends on the data type, while non-primitive types have all the same size.

# Array

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To declare an array, define the variable type with **square brackets**:

```
String[] cars;
```

```
dataType[] arrayRefVar;   // preferred way.
or
dataType arrayRefVar[];  // works but not preferred way.
```

We have now declared a variable that holds an array of strings. To insert values to it, we can use an array literal - place the values in a comma-separated list, inside curly braces:

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

To create an array of integers, you could write:

```
int[] myNum = {10, 20, 30, 40};
```

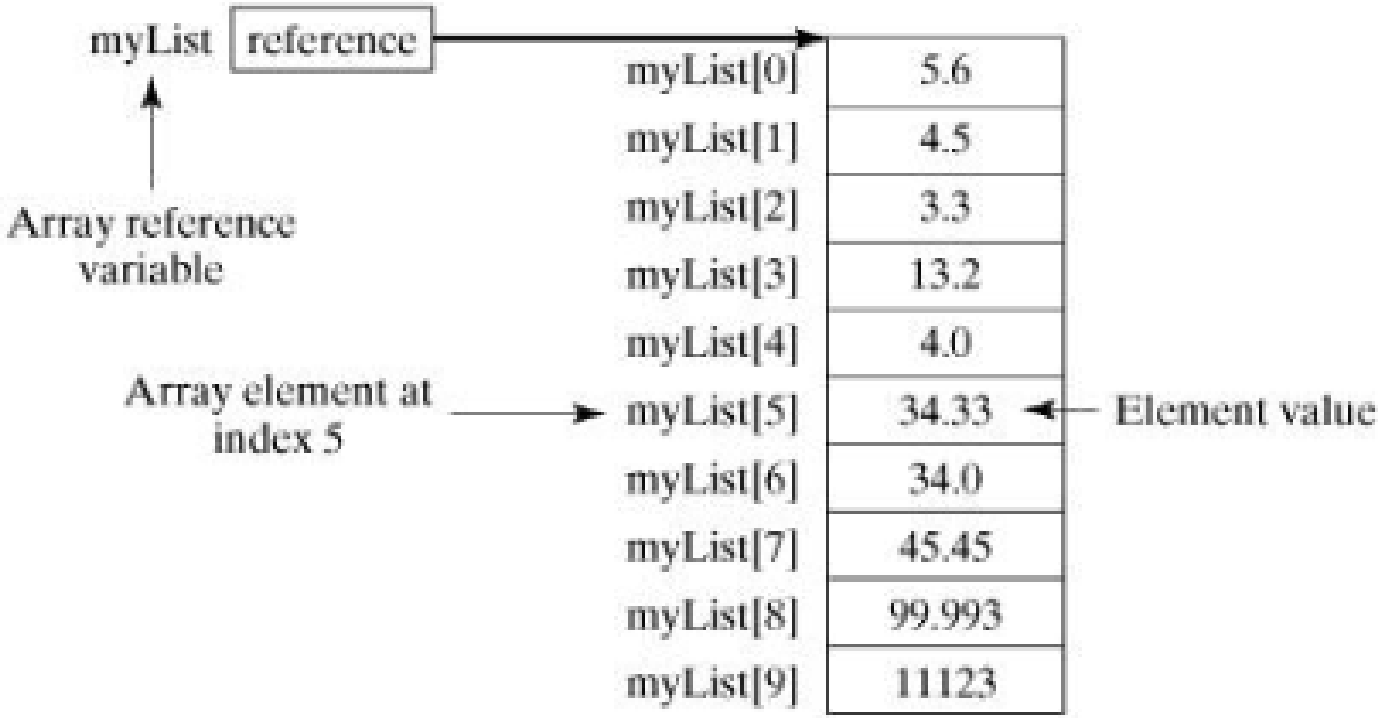
The array elements are accessed through the **index**. Array indices are 0-based; that is, they start from 0 to **arrayRefVar.length-1**.

## Example

Following statement declares an array variable, myList, creates an array of 10 elements of double type and assigns its reference to myList –

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

Following picture represents array myList. Here, myList holds ten double values and the indices are from 0 to 9.



# Access the Elements of an Array

You access an array element by referring to the index number.

This statement accesses the value of the first element in cars:

## Example

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
System.out.println(cars[0]);
// Outputs Volvo
```

# Change an Array Element

To change the value of a specific element, refer to the index number:

## Example

```
cars[0] = "Opel";
```

## Example

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
cars[0] = "Opel";
System.out.println(cars[0]);
// Now outputs Opel instead of Volvo
```

## Array Length

To find out how many elements an array has, use the `length` property:

### Example

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
System.out.println(cars.length);
// Outputs 4
```

# Loop Through an Array

You can loop through the array elements with the `for` loop, and use the `length` property to specify how many times the loop should run.

The following example outputs all elements in the **cars** array:

## Example

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (int i = 0; i < cars.length; i++) {
  System.out.println(cars[i]);
}
```

# Loop Through an Array with For-Each

There is also a "**for-each**" loop, which is used exclusively to loop through elements in arrays:

## Syntax

```
for (type variable : arrayname) {
  ...
}
```

- for each String element (called i - as in index) in cars, print out the value of i.

The following example outputs all elements in the **cars** array, using a "**for-each**" loop:

## Example

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (String i : cars) {
  System.out.println(i);
}
```

- compare the for loop and for-each loop, we will see that the for-each method is easier to write,
- it does not require a counter (using the length property), and it is more readable.

# Multidimensional Arrays

A multidimensional array is an array containing one or more arrays.

To create a two-dimensional array, add each array within its own set of **curly braces**:

## Example

```
int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };
```

```
int twoDimensionalArray[][] = new int[6][6];
twoDimensionalArray[0][0] = 100;
int threeDimensionalArray[][][] = new int[2][2][2];
threeDimensionalArray[0][0][0] = 200;
int varDimensionArray[][] = {{0,0},{1,1,1},{2,2,2,2}};
varDimensionArray[0][0] = 300;
```

**myNumbers** is now an array with two arrays as its elements.

To access the elements of the **myNumbers** array, specify two indexes: one for the array, and one for the element inside that array. This example accesses the third element (2) in the second array (1) of myNumbers:

## Example

```
int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };
int x = myNumbers[1][2];
System.out.println(x); // Outputs 7
```

# Loop in multidimensional array

```
public class MyClass {
  public static void main(String[] args) {
    int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };
    for (int i = 0; i < myNumbers.length; ++i) {
      for(int j = 0; j < myNumbers[i].length; ++j) {
        System.out.println(myNumbers[i][j]);
      }
    }
  }
}
```

- Non-primitive types can be used to call methods to perform certain operations

# Passing Arrays to Methods

Just as you can pass primitive type values to methods, you can also pass arrays to methods. For example, the following method displays the elements in an **int** array −

Example

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

You can invoke it by passing an array. For example, the following statement invokes the printArray method to display 3, 1, 2, 6, 4, and 2 −

Example

```
printArray(new int[]{3, 1, 2, 6, 4, 2});
```

## Returning an Array from a Method

A method may also return an array. For example, the following method returns an array that is the reversal of another array –

Example

```
public static int[] reverse(int[] list) {
   int[] result = new int[list.length];

   for (int i = 0, j = result.length - 1; i < list.length; i++, j--) {
      result[j] = list[i];
   }
   return result;
}
```

# String

## Java Strings

Strings are used for storing text.

A `String` variable contains a collection of characters surrounded by double quotes:

### Example

Create a variable of type `String` and assign it a value:

```
String greeting = "Hello";
```

## String Length

A String in Java is actually an object, which contain methods that can perform certain operations on strings. For example, the length of a string can be found with the `length()` method:

### Example

```
String txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
System.out.println("The length of the txt string is: " + txt.length());
```

# More String Methods

There are many string methods available, for example `toUpperCase()` and `toLowerCase()`:

## Example

```
String txt = "Hello World";
System.out.println(txt.toUpperCase());   // Outputs "HELLO WORLD"
System.out.println(txt.toLowerCase());   // Outputs "hello world"
```

# Finding a Character in a String

The `indexOf()` method returns the **index** (the position) of the first occurrence of a specified text in a string (including whitespace):

## Example

```
String txt = "Please locate where 'locate' occurs!";
System.out.println(txt.indexOf("locate")); // Outputs 7
```

# String Concatenation

The `+` operator can be used between strings to combine them. This is called **concatenation**:

## Example

```
String firstName = "John";
String lastName = "Doe";
System.out.println(firstName + " " + lastName);
```

- added an empty text (" ") to create a space between firstName and lastName on print.

You can also use the `concat()` method to concatenate two strings:

## Example

```
String firstName = "John ";
String lastName = "Doe";
System.out.println(firstName.concat(lastName));
```

- use the concat() method to concatenate two strings give same result

# Adding Numbers and Strings

If you add two numbers, the result will be a number:

## Example

```
int x = 10;
int y = 20;
int z = x + y;       // z will be 30 (an integer/number)
```

WARNING!

Java uses the + operator for both addition and concatenation.

Numbers are added. Strings are concatenated.

If you add a number and a string, the result will be a string concatenation:

## Example

```
String x = "10";
int y = 20;
String z = x + y;    // z will be 1020 (a String)
```

# Special Characters

Because strings must be written within quotes, Java will misunderstand this string, and generate an error:

```
String txt = "We are the so-called "Vikings" from the north.";
```

The solution to avoid this problem, is to use the **backslash escape character**.

The backslash (`\`) escape character turns special characters into string characters:

| Escape character | Result | Description |
|---|---|---|
| \' | ' | Single quote |
| \" | " | Double quote |
| \\ | \ | Backslash |

The sequence `\"` inserts a double quote in a string:

## Example

```
String txt = "We are the so-called \"Vikings\" from the north.";
```