

# Types

Navigate **Language Fundamentals** topic:

*Types* in Java define the allowed data values and the operations that may be performed on those values. Each expression, field, method parameter, local variable, and method return values, have an associated type. The syntax template for a field declaration in Java is

`[Modifiers] Type FieldName [= Expression];`

Method, method parameter, and local variables have similar syntax: optional modifiers, a required type, and the name of the value being declared.

In java, there are three different kind of *Types*, those are:

- Primitive Type
- Object Reference Type
- An array type

## About Java Types

Java is **strongly typed** in that all expressions and declarations have types (either declared or inferred) and the language only allows programs which adhere to the type constraints. You cannot write a statement or expression in Java which directly invokes an operation on a value that is not allowed by that value's type, nor can you perform assignments that violate the type system: you cannot assign a String value to an integer typed variable. This helps achieve a high level of type safety in Java. Most type errors are caught and detected by the Java compiler. However, some type errors can still occur at runtime because Java supports a cast operation which is a way of changing the type of one expression to another. However, Java performs run time type checking when doing such casts, so an incorrect type cast will cause a runtime exception rather than succeeding silently and allowing data corruption.

Java is also known as a **hybrid language**. While supporting object oriented (OO) programming, Java is not a pure OO language like Smalltalk or Ruby. Instead, Java offers both object types and primitive types. Primitive types are used for boolean, character, and numeric values and operations. This allows relatively good performance when manipulating numeric data, at the expense of flexibility. For example, you cannot subclass the primitive types and add new operations to them.

## Examples of Types

Below are two examples of Java types and a brief description of the allowed values and operations for these types. Additional details on each are available in other modules.

### Example: int

The primitive type **int** represents a signed 32 bit integer value. The allowed data values for int are the integers between -2147483648 to 2147483647 inclusive.

The set of operations that may be performed on int values includes integer arithmetic such as +, -, \*, /, %, comparison operations (==, !=, <, >, <=, >=), assignments (=, ++, --, +=, -=), bit-wise operations such as logical and, logical or, logical xor, negation (&, |, ^, ~), bit shift operations (<<, >>, >>>), conversions to other numeric types and promotion to other integer types.

For example, to declare a **private** integer instance field named length, one would use the declaration

```
private int length;
```

## Example: String

You use **class** and **interface** definition in Java to define new types. Class and interface types are associated with object references also sometime referred to as *Reference types*. An object reference has two main attributes:

- Its **type** associated with a class or an interface
- A java object it references, that is created by instantiating a class

The String class is one such example. String values are a sequence of 0 or more Unicode characters. The **null** reference is another valid value for a String expression.

The operations on a String reference variable are those available for all reference types, such as comparison operations `==`, `!=` and assignment `=`.

The allowed operations on String object, however are the set of methods in the `java.lang.String` class, which includes `length()`, `toString()`, `toLowerCase()`, `toUpperCase()`, `compareTo(String anotherString)` and more... .

In addition, String objects also inherit the set of operations from the base class that String extends from, which is `java.lang.Object`. These operations include methods such as `equals()`, `hashCode()`, `wait()`, `notifyAll()`, and `getClass()`.

```
private String name = "Marry Brown";
```

In the above example the *name* object reference's attributes are:

- **Type is** : String
- The **referenced object** is also : String

Both the `java.lang.String` class methods and `java.lang.Object` class methods are available for the object reference *name*.

```
private Object name = "Marry Brown";
```

In the above example the *name* object reference's attributes are:

- **Type is** : Object
- The **referenced object** is : String

Only the `java.lang.Object` class methods are available for the object reference *name*.

## Array Types

Arrays in Java are represented as a built-in Array object. As with object types, they behave as a reference but have a few differences allowing them to allow easy access to sub elements.

When one declares an array, the data type is changed to include square brackets. (While you can instead place these square brackets next to the variable name, this is not recommended.) To create an array, you will need to use the `new` operator to have it create the Array with the specified number of elements:

```
/* Declares an array named data, and has it assigned to an array with  
25 elements. */  
private int[] data = new int[25];
```

Arrays are described in more detail in the Arrays section.

## Primitive Data Types

The Java primitive data types contain pure values, no operations. It is basically data types similar to what other non-object-oriented languages have.

There are arrays of primitive types in Java; but because they are not objects, primitive values can not be put in a collection.

For this reason object wrappers are defined in JDK 'java.lang.\*' package for all the primitive types.

|                             | Size (bits) | Example                       | Minimum Value | Maximum Value | Wrapper Class |
|-----------------------------|-------------|-------------------------------|---------------|---------------|---------------|
| <b>Integer types</b>        |             |                               |               |               |               |
| <b>byte</b>                 | 8           | byte b = 65;                  | -128          | 127           | Byte          |
| <b>char</b>                 | 16          | char c = 'A';<br>char c = 65; | 0             | $2^{16}-1$    | Character     |
| <b>short</b>                | 16          | short s = 65;                 | $-2^{15}$     | $2^{15}-1$    | Short         |
| <b>int</b>                  | 32          | int i = 65;                   | $-2^{31}$     | $2^{31}-1$    | Integer       |
| <b>long</b>                 | 64          | long l = 65L;                 | $-2^{63}$     | $2^{63}-1$    | Long          |
| <b>Floating-point types</b> |             |                               |               |               |               |
| <b>float</b>                | 32          | float f = 65f;                |               |               | Float         |
| <b>double</b>               | 64          | double d = 65.55;             |               |               | Double        |
| <b>Other</b>                |             |                               |               |               |               |
| <b>boolean</b>              | 1           | boolean b = true;             | --            | --            | Boolean       |
| <b>void</b>                 | --          | --                            | --            | --            | Void          |

The types **short**, **int**, **long**, **float**, and **double** are usually used in arithmetic operations; **byte** and **char** can also be used, but less commonly.

The character type **char** is used for text processing. The type **byte** is commonly used in binary file input output operations.

String objects representing literal character strings in Java, in the java.lang.\* package. java.lang.String is not a primitive type, but instead is a special class built into the Java language. For further info, see String.

## Data Conversion (Casting)

Data conversion (casting) can happen between two primitive types. There are two kinds:

- Implicit : casting operation is not required; the magnitude of the numeric value is always preserved. However, *precision* may be lost when converting from integer to floating point types
- Explicit : casting operation required; the magnitude of the numeric value may not be preserved

Example for implicit casting:

```
int i = 65;
long l = i; // --- int is converted to long, casting is not needed
```

Example for explicit casting:

```
long l = 656666L;
int i = (int) l; // --- long is converted to int, casting is needed
```