Primitive Types

Primitive types are the most basic data types available within the Java language. There are 8: **`boolean`**, **`byte`**, **`char`**, **`short`**, **`int`**, **`long`**, **`float`** and **`double`**. These types serve as the building blocks of data manipulation in Java. Such types serve only one purpose — containing pure, simple values of a kind. Because these data types are defined into the Java type system by default, they come with a number of operations predefined. You can not define a new operation for such primitive types. In the Java type system, there are three further categories of primitives:

- Numeric primitives: **`short`**, **`int`**, **`long`**, **`float`** and **`double`**. These primitive data types hold only numeric data. Operations associated with such data types are those of simple arithmetic (addition, subtraction, etc.) or of comparisons (is greater than, is equal to, etc.)
- Textual primitives: **`byte`** and **`char`**. These primitive data types hold characters (that can be Unicode alphabets or even numbers). Operations associated with such types are those of textual manipulation (comparing two words, joining characters to make words, etc.). However, **`byte`** and **`char`** can also support arithmetic operations.
- Boolean and null primitives: **`boolean`** and **`null`**.

All the primitive types have a fixed size. Thus, the primitive types are limited to a range of values. A smaller primitive type (**`byte`**) can contain less values than a bigger one (**`long`**).

Category	Types	Size (bits)	Minimum Value	Maximum Value	Precision	Example
Integer	`byte`	8	-128	127	From +127 to -128	`byte b = 65;`
	`char`	16	0	2^{16}-1	All Unicode characters[1]	`char c = 'A';` `char c = 65;`
	`short`	16	-2^{15}	2^{15}-1	From +32,767 to -32,768	`short s = 65;`
	`int`	32	-2^{31}	2^{31}-1	From +2,147,483,647 to -2,147,483,648	`int i = 65;`
	`long`	64	-2^{63}	2^{63}-1	From +9,223,372,036,854,775,807 to -9,223,372,036,854,775,808	`long l = 65L;`
Floating-point	`float`	32	2^{-149}	$(2-2^{-23})\cdot 2^{127}$	From 3.402,823,5 E+38 to 1.4 E-45	`float f = 65f;`
	`double`	64	2^{-1074}	$(2-2^{-52})\cdot 2^{1023}$	From 1.797,693,134,862,315,7 E+308 to 4.9 E-324	`double d = 65.55;`
Other	`boolean`	--	--	--	false, true	`boolean b = true;`
	`void`	--	--	--	--	--

Integer primitive types silently overflow:

Code section 3.52: Several operators.

```
1 int i = Integer.MAX_VALUE;
2 System.out.println(i);
3 i = i + 1;
4 System.out.println(i);
5 System.out.println(Integer.MIN_VALUE);
```

Console for Code section 3.52

```
2147483647
-2147483648
-2147483648
```

As Java is strongly typed, you can't assign a floating point number (a number with a decimal point) to an integer variable:

Code section 3.53: Setting a floating point number as a value to an `int` (integer) type.

```
1 int age;
2 age = 10.5;
```

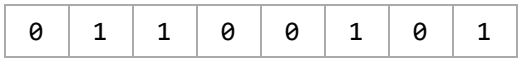
A primitive type should be set by an appropriate value. The primitive types can be initialized with a literal. Most of the literals are primitive type values, except String Literals, which are instance of the `String` class.

Numbers in computer science

Programming may not be as trivial or boring as just crunching huge numbers any more. However, huge chunks of code written in any programming language today, let alone Java, obsessively deal with numbers, be it churning out huge prime numbers,[2] or just calculating a cost of emission from your scooter. In 1965, Gemini V space mission escaped a near-fatal accident caused by a programming error.[3] Again in 1979, a computer program overestimated the ability of five nuclear reactors to withstand earthquakes; the plants shut down temporarily.[4] There is one thing common to both these programming errors: the subject data, being computed at the time the errors occurred, was numeric. Out of past experience, Java came bundled with revised type checking for numeric data and put significant emphasis on correctly identifying different types of it. You must recognise the importance of numeric data when it comes to programming.

Numbers are stored in memory using a binary system. The memory is like a grid of cells:

Each cell can contain a *binary digit* (shortened to *bit*), that is to say, zero or one:



Actually, each cell **does** contain a binary digit, as one bit is roughly equivalent to `1` and an empty cell in the memory signifies `0`. A single binary digit can only hold two possible values: a zero or a one.

Multiple bits held together can hold multiple permutations — 2 bits can hold 4 possible values, 3 can hold 8, and so on. For instance, the maximum number 8 bits can hold (`11111111` in binary) is `255` in the decimal system. So, the numbers from 0 to 255 can fit within 8 bits.

Memory state									*Gives*
0	0	0	0	0	0	0	0	→	0
0	0	0	0	0	0	0	1	→	1
0	0	0	0	0	0	1	0	→	2
0	0	0	0	0	0	1	1	→	3
...								...	
1	1	1	1	1	1	1	1	→	255

It is all good, but this way, we can only host positive numbers (or unsigned integers). They are called *unsigned integers*. Unsigned integers are whole number values that are all positive and do not attribute to negative values. For this very reason, we would ask one of the 8 bits to hold information about the sign of the number (positive or negative). This leaves us with just 7 bits to actually count out a number. The maximum number that these 7 bits can hold (1111111) is 127 in the decimal system.

Positive numbers

Memory state									*Gives*
0	0	0	0	0	0	0	0	→	0
0	0	0	0	0	0	0	1	→	1
0	0	0	0	0	0	1	0	→	2
0	0	0	0	0	0	1	1	→	3
...	...							...	
0	1	1	1	1	1	1	1	→	127

Negative numbers

Memory state									*Gives*
1	0	0	0	0	0	0	0	→	-128
1	0	0	0	0	0	0	1	→	-127
1	0	0	0	0	0	1	0	→	-126
1	0	0	0	0	0	1	1	→	-125
...	...							...	
1	1	1	1	1	1	1	1	→	-1

Altogether, using this method, 8 bits can hold numbers ranging from -128 to 127 (including zero) — a total of 256 numbers. Not a bad pay-off one might presume. The opposite to an unsigned integer is a *signed integer* that have the capability of holding both positive and negative values.

But, what about larger numbers. You would need significantly more bits to hold larger numbers. That's where Java's numeric types come into play. Java has multiple numeric types — their size dependent on the number of bits that are at play.

In Java, numbers are dealt with using data types specially formulated to host numeric data. But before we dive into these types, we must first set some concepts in stone. Just like you did in high school (or even primary school), numbers in Java are placed in clearly distinct groups and systems. As you'd already know by now, number systems includes groups like the *integer* numbers (0, 1, 2 ... ∞); *negative integers* (0, -1, -2 ... -∞) or even *real* and *rational* numbers (value of Pi, ¾, 0.333~, etcetera). Java simply tends to place these numbers in two distinct groups, **integers** (-∞ ... 0 ... ∞) and **floating point** numbers (any number with decimal points or fractional representation). For the moment, we would only look into integer values as they are easier to understand and work with.

Integer types in Java

With what we have learned so far, we will identify the different types of signed integer values that can be created and manipulated in Java. Following is a table of the most basic numeric types: integers. As we have discussed earlier, the data types in Java for integers caters to both positive and negative values and hence are **signed numeric types**. The size in bits for a numeric type determines what its minimum and maximum value would be. If in doubt, one can always calculate these values.

Lets see how this new found knowledge of the basic integer types in Java fits into the picture. Say, you want to numerically manipulate the days in a year — all 365 days. What type would you use? Since the data type `byte` only goes up to 127, would you risk giving it a value greater than its allowed maximum. Such decisions might save you from dreaded errors that might occur out of the programmed code. A much more sensible choice for such a numeric operation might be a `short`. Now, why couldn't they make just one data type to hold all kinds of numbers? Let's explore why.

When you tell a program you need to use an integer, say even a `byte`, the Java program allocates a space in the memory. It allocates whole 8 bits of memory. Where it wouldn't seem to matter for today's memory modules that have place for almost a dozen trillion such bits, it matters in other cases. Once allocated that part of the memory gets used and can only be claimed back after the operation is finished. Consider a complicated Java program where the only data type you'd be using would be `long` integers. What happens when there's no space for more memory allocation jobs? Ever heard of the Stack Overflow errors. That's exactly what happens — your memory gets completely used up and fast. So, choose your data types with extreme caution.

Enough talk, let's see how you can create a numeric type. A numeric type begins with the type's name (`short`, `int`, etc.) and then provides with a name for the allocated space in the memory. Following is how it's done. Say, we need to create a variable to hold the number of days in a year.

Code section 3.54: Days in a year.

```
1 short daysInYear = 365;
```

Here, `daysInYear` is the name of the variable that holds `365` as its value, while `short` is the data type for that particular value. Other uses of integer data types in Java might see you write code such as this given below:

Code section 3.55: Integer data types in Java.

```
1 byte maxByte = 127;
2 short maxShort = 32767;
3 int maxInt = 2147483647;
4 long maxLong = 9223372036854775807L;
```

Integer numbers and floating point numbers

The data types that one can use for integer numbers are `byte`, `short`, `int` and `long` but when it comes to floating point numbers, we use `float` or `double`. Now that we know that, we can modify the code in the code section 3.53 as:

Code section 3.56: Correct floating point declaration and assignment.

```
1 double age = 10.5;
```

Why not `float`, you say? If we'd used a `float`, we would have to append the number with a `f` as a suffix, so `10.5` should be `10.5f` as in:

Code section 3.57: The correct way to define floating point numbers of type `float`.

```
1 float age = 10.5f;
```

Floating-point math never throws exceptions. Dividing a non-zero value by `0` equals `infinity`. Dividing a non-infinite value by `infinity` equals `0`.

Test your knowledge

Question 3.7: Consider the following code:

Question 3.7: Primitive type assignments.

```
5 ...
6
7 a = false;
8 b = 3.2;
9 c = 35;
```

```
10 d = -93485L;
11 e = 'q';
```

These are five variables. There are a `long`, a `byte`, a `char`, a `double` and a `boolean`. Retrieve the type of each one.

Answer

Answer 3.7: Primitive type assignments and declarations.

```
1 boolean a;
2 double b;
3 byte c;
4 long d;
5 char e;
6
7 a = false;
8 b = 3.2;
9 c = 35;
10 d = -93485L;
11 e = 'q';
```

- a can only be the `boolean` because only a boolean can handle boolean values.
- e can only be the `char` because only a char can contain a character.
- b can only be the `double` because only a double can contain a decimal number here.
- d is the `long` because a `byte` can not contain such a low value.
- c is the remaining one so it is the `byte`.

Data conversion (casting)

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

- 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

Code section 3.58: Implicit casting (int is converted to long, casting **is not needed).**

```
1 int  i = 65;
2 long l = i;
```

Code section 3.59: Explicit casting (long is converted to int, casting is needed).

```
1 long l = 656666L;
2 int  i = (int) l;
```

The following table shows the conversions between primitive types, it shows the casting operation for explicit conversions:

	from `byte`	from `char`	from `short`	from `int`	from `long`	from `float`	from `double`	from `boolean`
to `byte`	-	(byte)	(byte)	(byte)	(byte)	(byte)	(byte)	N/A
to `char`		-	(char)	(char)	(char)	(char)	(char)	N/A
to `short`		(short)	-	(short)	(short)	(short)	(short)	N/A
to `int`				-	(int)	(int)	(int)	N/A
to `long`					-	(long)	(long)	N/A
to `float`						-	(float)	N/A
to `double`							-	N/A
to `boolean`	N/A	N/A	N/A	N/A	N/A	N/A	N/A	-

Unlike C, C++ and similar languages, Java can't represent `false` as `0` or `null` and can't represent `true` as non-zero. Java can't cast from boolean to a non-boolean primitive data type, or vice versa.

For non primitive types:

	to Integer	to Float	to Double	to String	to Array
Integer	-	(float)x	(double)x x.doubleValue()	x.toString() Float.toString(x)	new int[] {x}
Float	java.text.DecimalFormat("#").format(x)	-	(double)x	x.toString()	new float[] {x}
Double	java.text.DecimalFormat("#").format(x)	java.text.DecimalFormat("#").format(x)	-	x.toString()	new double[] {x}
String	Integer.parseInt(x)	Float.parseFloat(x)	Double.parseDouble(x)	-	new String[] {x}
Array	x[0]	x[0]	x[0]	Arrays.toString(x)	-

Notes

1. According to "STR01-J. Do not assume that a Java char fully represents a Unicode code point". Carnegie Mellon University - Software Engineering Institute. https://wiki.sei.cmu.edu/confluence/display/java/STR01-J.+Do+not+assume+that+a+Java+char+fully+represents+a+Unicode+code+point. Retrieved 27 Nov 2018., not all Unicode characters fit into a 16 bit representation
2. As of edit (11 December 2013), the Great Internet Mersenne Prime Search project has so far identified the largest prime number as being 17,425,170 digits long. Prime numbers are valuable to cryptologists as the bigger the number, the securer they can make their data encryption logic using that particular number.
3. Gemini 5 landed 130 kilometers short of its planned Pacific Ocean landing point due to a software error. The Earth's rotation rate had been programmed as one revolution per solar day instead of the correct value, one revolution per sidereal day.
4. A program used in their design used an arithmetic sum of variables when it should have used the sum of their absolute values. (Evars Witt, "*The Little Computer and the Big Problem*", AP Newswire, 16 March 1979. See also Peter Neumann, "*An Editorial on Software Correctness and the Social Process*" Software Engineering Notes, Volume 4(2), April 1979, page 3)

Retrieved from "https://en.wikibooks.org/w/index.php?title=Java_Programming/Primitive_Types&oldid=3701227"

This page was last edited on 18 June 2020, at 19:56.

Text is available under the Creative Commons Attribution-ShareAlike License.; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy.