Theory: Type casting

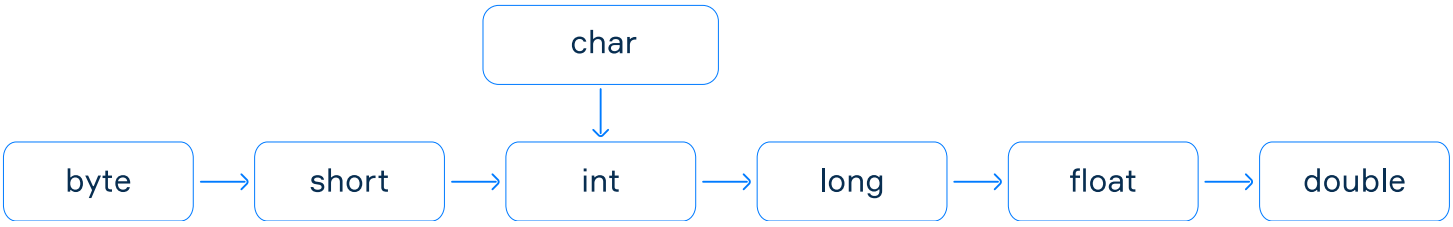
13 minutes 0 / 5 problems solved

Skip this topic Start practicing

Suppose, you need to assign a value of one type to a variable of another type. To do that, your program needs to cast the source type to the target type. Java provides two kinds of casting for primitive types: **implicit** and **explicit**. The first one is performed automatically by the java compiler when it is possible, and the second one – only by a programmer.

Implicit casting

The compiler automatically performs **implicit casting** when the target type is wider than the source type. The picture below illustrates the direction of this casting. Any value of a given type can be assigned to the one on the right implicitly.



The direction of implicit primitive type castings

Normally, there is no loss of information when the target type is wider than the source type, for example when we cast `int` to `long`. But it is not possible to automatically cast in the backward order (e.g. from `long` to `int` or from `double` to `float`).

> **Note,** there is no `boolean` type on the picture above, because it is impossible to cast this type to any other and vice versa.

Here are several examples of implicit castings:

- from `int` to `long`:

```
int num = 100;
long bigNum = num; // 100L
```

- from `long` to `double`:

```
long bigNum = 100_000_000L;
double bigFraction = bigNum; // 100000000.0
```

- from `short` to `int`:

```
short shortNum = 100;
int num = shortNum; // 100
```

- from `char` to `int`:

```
char ch = '?';
int code = ch; // 63
```

In some cases, implicit type casting may be a bit lossy. When we convert an `int` to `float`, or a `long` to `float` or to `double`, we may lose some less significant bits of the value, which will result in the loss of precision. However, the result of this conversion will be a correctly rounded version of the integer value, which will be in the overall range of the target type. To understand that, check out the example:

```
long bigLong =  1_200_000_002L;
float bigFloat = bigLong; // 1.2E9 (= 1_200_000_000)
```

When we convert a `char` to an `int` in Java we actually get the ASCII value for a given character. ASCII value is an integer representation of English alphabet letters (both uppercase and lowercase), digits, and other symbols. Here you can find some of the standard symbols in ASCII.

```
char character = 'a';
char upperCase = 'A';

int ascii1 = character; // this is 97
int ascii2 = upperCase; // this is 65
```

> Strictly speaking, Java uses Unicode Character Representations (UTF-16), which is a superset of ASCII and includes a by far larger set of symbols. However, the numbers 0–127 have the same values in ASCII and Unicode.

As you can see, implicit casting works absolutely transparently.

Explicit casting

The considered **implicit casting** does not work when the target type is narrower than the source type. But programmers can apply **explicit casting** to a source type to get another type they want. It may lose information about the overall magnitude of a numeric value and may also lose precision.

To perform explicit casting, a programmer must write the target type in parentheses before the source.

```
(targetType) source
```

Any possible casting not presented in the picture above needs such approach, for example `double` to `int`, and `long` to `char`.

Examples:

```
double d = 2.00003;

// it loses the fractional part
long l =  (long) d; // 2

// requires explicit casting because long is wider than int
int i = (int) l; // 2

// requires explicit casting because the result is long (indicated by L)
int val = (int) (3 + 2L); // 5

// casting from a long literal to char
char ch = (char) 55L; // '7'
```

However, the explicit casting may truncate the value, because `long` and `double` can store a much larger number than `int`.

```
long bigNum = 100_000_000_000_000L;
int n = (int) bigNum; // 276447232
```

Oops! The value has been truncated. This problem is known as **type overflow**. The same problem may occur when casting `int` to `short` or `byte`. Let's see what happens exactly.

As you remember, in Java `long` is a 64-bit number, while `int` is 32-bit. When converting `long` to `int` the program just takes the last 32 bits to represent the new number. If the `long` contains a number less than or equal to `Integer.MAX_VALUE` you can convert it by casting without losing information. Otherwise, the result will be quite meaningless, although determined. That is why you shouldn't perform casting from a larger type to a smaller type unless you are absolutely sure that it is necessary, and that truncation will not interfere with your program.

Explicit casting also works when implicit casting is enough.

```
int num = 10;
long bigNum = (long) num; // redundant casting
```

But this is redundant and should not be used to avoid unnecessary constructs in your code.

> **Note**, that despite the power of the explicit casting it is still impossible to cast something to and from the `boolean` type.