Theory: Numeric literals

We've discussed that numbers, strings, characters, and other values we use while writing our code are represented by literals. For example, `13` and `13L` are literals that represent the value of 13, while `true` and `false` literals represent boolean values. Earlier we had a brief overview of the most common literals in Java. Now we're going to consider the group of literals comprising all numbers in detail. They are called **numeric literals**. There are three main types of numeric literals:

1. Integer literals,
2. Floating-point literals,
3. Character literals.

Numeric literals also differ based on the numeral system of the values: there are decimal, hexadecimal, octal and binary literals.

§1. Integer literals

The most common group represents integer numbers, that in turn include `byte`, `short`, `int`, `long` and `char` values. However, when it comes to integer literals, Java has only two types: **int** and **long**. The default type of integer literals is int. Here are some examples:

§1. Integer literals

The most common group represents integer numbers, that in turn include `byte`, `short`, `int`, `long` and `char` values. However, when it comes to integer literals, Java has only two types: **int** and **long**. The default type of integer literals is int. Here are some examples:

```
byte age = 46;
short height = 165;
int max = 2147483647;
```

To store values that are greater than `Integer.MAX_VALUE` we use `long` type, represented by long integer literals. They are indicated by adding `L` to the end of a literal:

```
long bigNum = 2147483648L;
long smallNum = 5L;         // that is a long literal too
```

You can use either `L` or `l` to indicate long literal, but `L` is preferable since `l` is easily confused with `1`.

Note:

1) you can only assign long literals to `long` variables:

```
// can't do that!
int num = 5L; // error
```

1) you can only assign long literals to `long` variables:

```
// can't do that!
int num = 5L; // error
```

2) the range of int literals is equal to the range of `int` values:

```
// can't do that!
long tooBigInt = 2147483648; // error, too big integer literal
```

§2. Different formats of integer literals

So far we've considered only decimal integer literals. However, Java also allows representing hexadecimal, octal and binary numbers.

Hexadecimal numbers consist of 16 digits: `0-9` and the letters `A-F`, case-insensitive, to represent `10-15`. To define hexadecimal numbers in Java, you have to add `0x` prefix to the literal:

```
int hexValue1 = 0x29;           // 16^1 * 2 + 16^0 * 9 = 41
int hexValue2 = 0x4B;           // 16^1 * 4 + 16^0 * 11 = 75
System.out.println(hexValue1); // this prints 41

// next line won't compile:
int hexValue3 = 4B;             // error, use the prefix!
```

Octal literals can have only digits from `0-7`. In Java, numeric literals with leading zeroes are octal numbers:

Octal literals can have only digits from `0-7`. In Java, numeric literals with leading zeroes are octal numbers:

```
int octValue = 027; // = 8^1 * 2 + 8^0 * 7 = 23
int decValue = 27; // = 27
System.out.println(octValue); // this prints 23

// next line won't compile:
int wrongOct = 089; // error, incorrect value
```

Be careful not to confuse decimal and octal literals: if you put a leading zero to the decimal number, it'll be interpreted as an octal and you will get the wrong result.

Finally, the binary literals were introduced to Java 7. You should use the `0b` prefix to define binary numbers in Java. Check out the following example:

```
int binValue = 0b110110;      // = 2^5*1 + 2^4*1 + 2^3*0 + 2^2*1 + 2^1
int decValue = 110110;        // = 110110
System.out.println(binValue); // this prints 54
```

§3. Floating-point literals

To represent floating-point numbers we use floating-point literals. Simple floating-point literals consist of `0-9` digits, a decimal point, and an optional `F`, `f`, `D` or `d` suffix to indicate the type. If no suffix is specified, the type is assumed to be `double`. Here are some examples of assigning:

assumed to be `double`. Here are some examples of assigning:

```
double num1= 123.723; // a double literal
float  num2= 34.0F;   // a float literal

// next line won't compile:
float num3 = 0.05;    // error!
```

In case fractional part is zero, decimal point may be omitted as well as zero.

Here are some more valid formats of floating-point literals:

```
.5F  // same as 0.5F
5F   // same as 5.0
16.  // same as 16.0
0.0  // zero
```

> float, double, and int literals can be assigned to floating-point type instances according to the rules of type casting. This is discussed in the corresponding topic.

Scientific notation is also possible for Java floating-point literals. Then apart from the whole number and its fractional part the power of 10 is indicated. The result of the multiplication of the number and this power of 10 is the resulting floating-point number, while exponent is indicated by an `E` or `e` followed by a positive or negative decimal number:

```
float num1 = 0.05e3F; // = 0.05 * 10^3 = 50
double num2 = 25.255e-4;  // = 25.255 * 10^(-4) = 0.0025255
```

This way you can represent a wider range of values with floating-point literals.

Since Java 6, it is also possible to use the hexadecimal form of floating-point literal. It starts with `0x` prefix as well, but the exponential part is always required, which starts with `p` (or `P`) instead of `e`. `p` implies that a power of 2 is used instead of a power of 10. You are not very likely to use such form, but just in case see how tricky it may look:

```
1  float hexFloat = 0x5.0p0f;   // 5.0 in hexadecimal form. Here 'f' indi
2  double hexDouble = 0xf.8p3; // 15.5 x 2^3 = 124.0 Here 'f' is part of
```

§4. Character literals

Characters might not look like a numeric type at first. However, in Java `char` is actually an integer type, that stores the 16-bit Unicode integer value of each character. That is why character literals can be used as an integer if needed. Check out the example:

```
1  int characterLiteral = 'a'; // this is 97
```

The Unicode value of 'a' is `0x0061`, which is 97 in decimal. Note, that the numeric value of, for example, char '2' is 50. You may find all values in the special tables online.

§5. Using underscore in numeric literals

To increase the readability of numeric literals, it is allowed to use **unders**

Characters might not look like a numeric type at first. However, in Java `char` is actually an integer type, that stores the 16-bit Unicode integer value of each character. That is why character literals can be used as an integer if needed. Check out the example:

```
int characterLiteral = 'a'; // this is 97
```

The Unicode value of 'a' is `0x0061`, which is 97 in decimal. Note, that the numeric value of, for example, char '2' is 50. You may find all values in the special tables online.

§5. Using underscore in numeric literals

To increase the readability of numeric literals, it is allowed to use **underscore** between digits since Java 7. It will be neglected when you do any calculations or print the output. Here are some examples:

```
int million = 1_000_000;
long creditCardNumber = 1234_5678_1234_5678L;
int hex = 0x1FF_A91;
float floatNum = 123_000.500_2f;
```

There are some limitations to using underscores. You cannot use the underscore :

- at the beginning or end of the literal. For example, `_10` will be treated as an identifier instead of numeric literal;
- adjacent to the decimal point, like in `10._025` or `20_.32`;
- before the `F` or `L` prefix. For example, `10002334_L` and `205_F` are

- at the beginning or end of the literal. For example, `_10` will be treated as an identifier instead of numeric literal;
- adjacent to the decimal point, like in `10._025` or `20_.32`;
- before the `F` or `L` prefix. For example, `10002334_L` and `205_F` are incorrect.

§6. Positive and negative values

You can also add optional `+` and `-` sign to the beginning of numeric literals. Plus sign indicates the positive value and can be omitted as default. To indicate a negative value, add the minus `-` sign to any type of numeric literal:

```
-011  // octal for decimal -9
-5.2f // float -5.2
-.2   // double -0.2
-0x5C // hexadecimal for decimal -92
-'z'  // -122
```

These are actually unary operators that perform an operation with a single argument and not a part of the literal.

§7. Conclusion

All numeric values in Java are represented by numeric literals. There are integer and floating-point literals, as well as decimal, hexadecimal, octal and binary representations. You can tell the type by the look of the literal: if it starts with `0b` it is binary, `0x` prefix indicates hexadecimal, leading zero denotes octal. Floating-point literals have a decimal point and an optiona