

Keywords in Java

In Java, a keyword is a reserved word that has a specific meaning and purpose in the programming language. These keywords cannot be used as identifiers (such as variable names or method names) in the code because they are already predefined with a particular function in the Java language. Some examples of keywords in Java include:

public	if	switch	try	throws	interface	boolean	int
class	else	case	catch	new	extends	char	long
static	for	break	finally	this	implements	byte	float
void	while	return	throw	super	import	short	double

Java syntax requires all code, including keywords, to be case-sensitive. Therefore, an "public" with all lowercase letters is not equivalent to "Public" with a capital letter, as they have different meanings. It's important to note that "public" is a keyword in Java.

Java Primitive Data Types

All the values in Java are divided into two categories: **reference types** and **primitive types**. We can start with primitive types and operators as they are most basic and natural entry into Java

➤ OVERVIEW

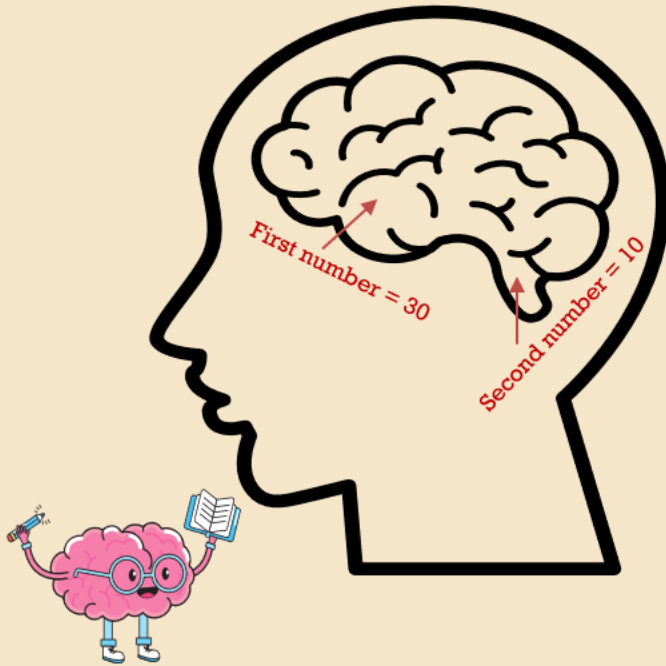
Java supports eight basic data types known as primitive types. The primitive types include a boolean type, a character type, four integer types, and two floating-point types. The four integer types and the two floating-point types differ in the number of bits that represent them and therefore in the range of numbers they can represent/support.

➤ DATA TYPES

- **boolean**
- **int**
- **char**
- **long**
- **byte**
- **float**
- **short**
- **double**

Why we need Primitive Data Types ?

Think a scenario where your friend asked to subtract two number. He told the first number as 30 and the second number as 10



You will store the number details inside your brain memory and perform subtraction. If asked, you can perform addition or multiplication etc. based on the same numbers

Just like how we store data inside brain memory, computers also has to store them inside it's memory. To help around this, we have Primitive data types in Java

firstNumber	30
secondNumber	10

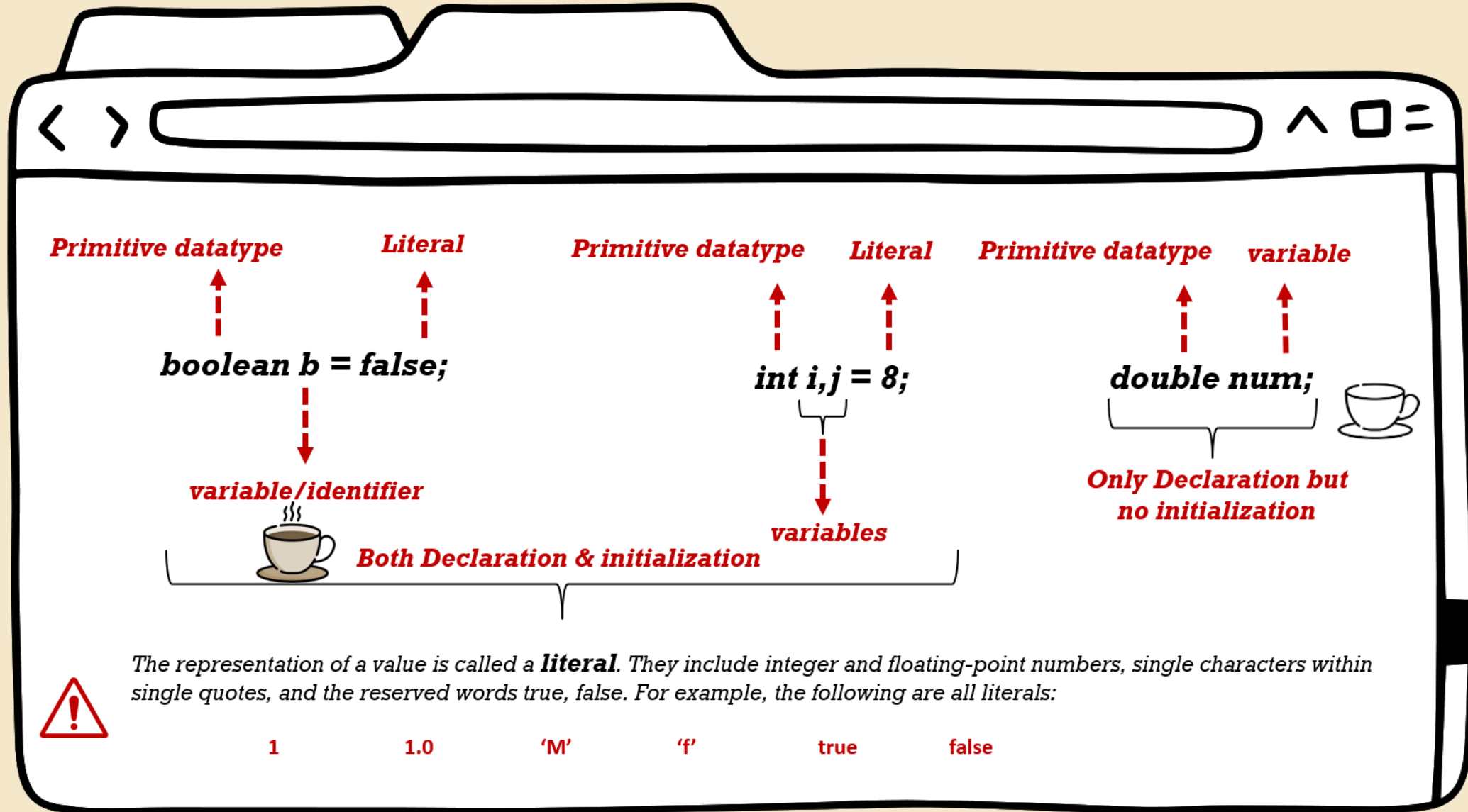
Since computers can't think themselves, we need to follow various syntaxes, rules while trying to store data using primitive data types

type	default	size	range
boolean	false	1 bit	NA
char	\u0000	16 bits / 2 Bytes	\u0000 to \uFFFF
byte	0	8 bits / 1 Byte	-128 to 127
short	0	16 bits / 2 Bytes	-32768 to 32767
int	0	32 bits / 4 Bytes	-2147483648 to 2147483647
long	0	64 bits / 8 Bytes	-9223372036854775808 to 9223372036854775807
float	0.0	32 bits / 4 Bytes	1.4E-45 to 3.4028235E38
double	0.0	64 bits / 8 Bytes	4.9E-324 to 1.7976931348623157E308

Java Primitive Data Types

SYNTAX TO DECLARE **PRIMITIVE DATA TYPES** ?

eazy
bytes





HOW TO NAME A **VARIABLE** ?

A **variable** or **identifier** is simply a name given to a location where we want to store a value that is supported by a given primitive data type. We can also give an identifier to a class, a method within a class, method parameters etc. which we are going to discuss in coming lectures.

For variables, the Java naming convention is to always start with a lowercase letter and then capitalize the first letter of every subsequent word. Variables in Java are not allowed to contain white space, so variables made from compound words are to be written with a **camel case syntax**.

Examples: firstName, totalAmnt, isValid,
numOfWords

Variable names are **case-sensitive** and can be formed using alphabets, numbers, "\$", "_". They can get started with a letter, the dollar sign "\$", or the underscore character "_". The convention, however, is to always begin your variable names with a letter, not "\$" or "_".

When choosing a name for your variables, use full words instead of cryptic abbreviations. Doing so will make your code easier to read and understand. In many cases it will also make your code self-documenting; fields named amount, speed, and gear, for example, are much more intuitive than abbreviated versions, such as a, s, and g. Also keep in mind that the name you choose **must not be a keyword or reserved word**.

Examples of valid & invalid identifiers ?

Identifier name	Valid ?	Reason
firstName1	Yes	Contains letters and number
\$personName	Yes	Can start with \$, _ and a letter
7firstName	No	Can not start with a number
age_of_person	Yes	Contains letters and allowed special char _
person name	No	Contains space in between the name of the variable
personName*	No	Contains * which is not allowed. Only \$ and _ are allowed
\$	Yes	Only \$ is a valid allowed name
-	No	Since JDK 9, the use of an underscore as a standalone identifier is no longer permitted
FirSTName	Yes	Though it is not following camelCase, it is still valid
654	No	Using all numbers is not allowed

In Java programs, three words—true, false, and null—might seem like keywords, but they aren't. Instead, true and false function as Boolean literals, while null serves as a null literal. It's important to note that despite not being keywords, true, false, and null cannot be employed as identifiers in Java.

Case Styles in programming

In programming, naming conventions are crucial as they help in writing more readable and maintainable code. One aspect of these conventions is case styles, which determine how to format variable, function, method and class names. Case styles refer to the way you use capitalization and spacing to differentiate between words in a name. To create a variable in your program with a name that consists of more than one word, you must combine the words into a single string and remove any spaces between them. Failure to do so may cause your program to crash. Most programming languages have specific naming conventions for combining words in variable names. For example, you can't create a variable like below,

number of players = 11

camelCase

Camel case involves starting with a lowercase letter for the first word and then capitalizing the first letter of each subsequent word. This means that each new word after the first has a capital letter at the beginning. It is commonly used in JavaScript and Java programming

**numberOfPlayers,
firstName,numOfPeople,
emailAddress**

PascalCase

Pascal case and camel case are similar, with the only distinction being that pascal case mandates the first letter of the first word to be capitalized. In pascal case, every word begins with an uppercase letter, as opposed to camel case, where the first word begins with a lowercase letter. It is commonly used in C# and .NET programming.

**NumberOfPlayers,
FirstName,NumOfPeople,
EmailAddress**

snake_case

Snake case involves using an underscore character (`_`) to separate each word in a variable name. This naming convention is commonly used to declare constants in programming languages when capitalized. It is commonly used in Java, Python and Ruby programming.


**number_of_players,
first_name, num_of_people,
email_address,
NUMBER_OF_PLAYERS**

kebab-case

Kebab case is quite similar to snake case, except that it uses a dash character, `-` , to separate each word in a variable name instead of an underscore. This naming convention requires all words to be in lowercase and separated by a dash. Kebab case is commonly used in URLs and CSS.

**number-of-players,
first-name, num-of-people,
email-address**

The **boolean** type



The **boolean** type represents truth values. This type has only two possible values **true** & **false**. They represent the two boolean states such as on or off, yes or no, true or false etc.

Java reserves the words **true** and **false** to represent these two Boolean values.

A boolean variable is typically used in **control flow statements**, which we are going to discuss in the coming sections.

Sample declaration of boolean variables & values,


```
boolean hasChildren;  
boolean isValid = true;  
boolean isMajor, hasPassport = true;
```

Sample code snippet in jshell,

```
jshell> boolean isValid = true;  
isValid ==> true  
  
jshell> System.out.println(isValid);  
true  
  
jshell> █
```

Using the prefix "is" for a boolean variable name is a common practice among developers, as it helps to create a name that appears like a question and makes the code easier to read and understand. However, there are other prefixes that can be used for boolean variable names that are also acceptable and valid.

The **char** type



The **char** data type is used to store a single character. The character must be surrounded by single quotes, like 'M'

The char data type is a single 16-bit Unicode character. Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive). The char data type is used to store characters.

Unicode is a universal international standard character encoding that is capable of representing most of the world's written languages.

Sample declaration of char variables & values,

```
char num = '2';  
char x = 'M';  
char x1 = '\u0032';  
char y;  
char z = 456;
```

Sample code snippet in jshell,

```
jshell> char x='M';  
x ==> 'M'  
  
jshell> System.out.println(x);  
M  
  
jshell> █
```

The **char** type

If you know a Unicode value of a character that starts with `\u`, you can directly mention the same. Internally the Unicode value will get converted to correspondent character. For example, the below will assign '2' to `x1`.

```
char x1 = '\u0032';
```

In addition, Java supports a number of other escape sequences that make it easy both to represent commonly used nonprinting ASCII characters, such as newline, tab space and to escape certain punctuation characters that have special meaning in Java. For example:

```
char tab = '\t', empty = '\000',  
backslash = '\\';
```

The `char` type is an unsigned integer that can hold a value (called a **code point**) from 0 to 65,535 inclusive. It represents a Unicode character, which means there are 65,536 Unicode characters. For example, if you try to sum 2 `char` variables, `char` value is represented by its code point.

```
char x = 'M';  
char x1 = '2';  
System.out.println(x+x1);
```

The `Character` class defines a number of useful static methods for working with characters, including `isDigit()`, `isLowerCase()`, and `toUpperCase()` etc.



Few sample escape sequences & their representation value,

`\b` - Backspace, `\t` - Horizontal tab, `\n` - Newline, `\f` - Form feed, `\r` - Carriage return, `\"` - Double quote, `\'` - Single quote, `\\` - Backslash

The **Integer** types

Integer types stores whole numbers, positive or negative (such as 166 or -613), without decimals. Valid types are **byte**, **short**, **int** and **long**. Which type you should use, depends on the size of numeric value.

Since there is overlap between int type literals and long type literals, an integer literal of type long always ends with **L** (or lowercase **l**).

We can always fetch the min and max values supported by these data types using **MIN_VALUE** & **MAX_VALUE** static variables available in the respective classes like Byte, Short, Integer & Long

Sample declaration of Integer variables & values,

```
byte b1 = 127, b2 = 1;  
short s = 3244;  
int x = 10;  
long num = 456L;
```

Sample code snippet in jshell,

```
jshell> long num = 456L;  
num ==> 456  
  
jshell> System.out.println(num);  
456  
  
jshell> █
```

The **Integer** types

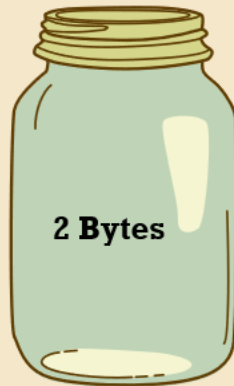
eazy
bytes

Size & Range of numeric values need to be considered while choosing one of the interger data types.

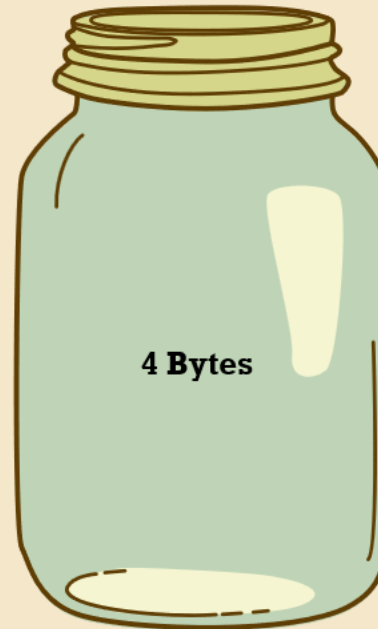
byte



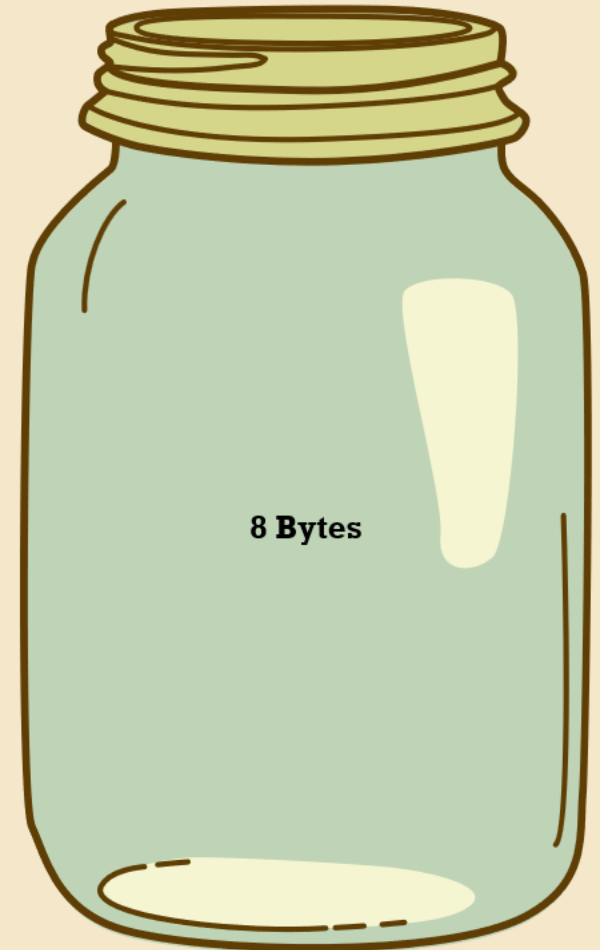
short



int



long



Importance of **L** suffix

In Java, the L suffix is required to indicate a long literal.

By default, integer literals are of the int type, which can represent values between -2,147,483,648 and 2,147,483,647. If you want to represent a value that is outside this range, you need to use a long literal instead.

A long literal is a whole number that ends with the letter L (lowercase or uppercase) or l (lowercase only). For example:

```
long longValue = 1234567890123L;
```

In the example above, the L suffix is required to indicate that the value 1234567890123 should be treated as a long literal, rather than an int literal.

Without the L suffix, the compiler would interpret the value as an int literal and generate a compiler error, since the value is outside the range of the int data type.

Overflow and Underflow

In Java, overflow and underflow refer to situations where the result of an arithmetic operation exceeds the range of the data type being used.

Java compiler can detect the overflow or underflow scenarios if we try to assign a value directly that is either greater than max value or less than min value.

```
int overFlowNum = 2147483648; // Will not compile due to overflow  
int underFlowNum = -2147483649; // Will not compile due to underflow
```



Overflow occurs when the result of an arithmetic operation exceeds the maximum value that can be represented by the data type. For example, if an int data type is used to store a value of 2,147,483,647 and 1 is added to it, the result will overflow and wrap around to the minimum value of -2,147,483,648.

```
int overFlowNum = 2147483647 + 1; // Will compile & result in overflow
```

Overflow and Underflow


Underflow occurs when the result of an arithmetic operation is less than the minimum value that can be represented by the data type. For example, if an int data type is used to store a value of -2,147,483,648 and 1 is subtracted from it, the result will underflow and wrap around to the maximum value of 2,147,483,647.

```
int underFlowNum = -2147483648 - 1; // Will compile & result in underflow
```

Since the compiler does not evaluate the expressions during compile time, it cannot detect these scenarios and will not issue any warnings or errors related to overflow or underflow. It is up to the programmer to ensure that the values being used in the arithmetic expressions are within the acceptable range for the data type being used, and to handle any potential overflow or underflow scenarios in the code.

This can be achieved through various techniques such as using larger data types to accommodate larger values, validating user input, and implementing checks to detect potential overflow or underflow scenarios.

The **Floating point** types



Real or decimal numbers in Java are represented by the **float** and **double** data types.

Using **float** we can store upto **7 decimal digits** where as with **double** we can store up to **15 decimal digits**.

We can always fetch the min and max values supported by these data types using **MIN_VALUE & MAX_VALUE** static variables available in the respective classes like Float & Double

To indicate that you would like it to be treated as a float type, for all the float literals you need to add either **f** or **F**.

Otherwise the real number will be considered as double. Optionally, we can use **d** or **D** suffix for double values.

Sample declaration of Floating point variables & values,

```
float f = 33.3f;  
float d = 24.8F;  
double a = 4343.3;
```

Sample code snippet in jshell,

```
[jshell]> float f = 33.3f;  
f ==> 33.3  
  
[jshell]> double a = 4343.3;  
a ==> 4343.3
```



*float & double data types should never be used for precise values, such as currency. For that, you will need to use the **java.math.BigDecimal** class instead.*

The **Floating point** types

float



“
Decimal precision
need to be considered
while choosing one of the
floating data types.
”

double



In general, double is a better choice than float if you need higher precision, a wider range of values, greater accuracy.

The **Floting** point types



The float data type in Java includes two zero representations: 0.0F and -0.0F. However, when it comes to comparisons, +0.0F and -0.0F are considered equal. Additionally, the float data type introduces two infinities: positive infinity and negative infinity. For instance, dividing 1.5F by 0.0F results in positive infinity, while dividing 1.5F by -0.0F yields negative infinity.

Certain operations on float can produce undefined results. For instance, dividing 0.0F by 0.0F leads to an indeterminate outcome. Such indeterminate results are denoted by a special value in the float data type known as NaN (Not-a-Number).

To handle these special values, Java provides a Float class (note the uppercase F) that defines three constants representing positive infinity (POSITIVE_INFINITY), negative infinity (NEGATIVE_INFINITY), and NaN for the float data type.




The double data type in Java includes two zero representations: 0.0D and -0.0D. However, when it comes to comparisons, 0.0D and -0.0D are considered equal. Additionally, the double data type introduces two infinities: positive infinity and negative infinity. For instance, dividing 1.5D by 0.0D results in positive infinity, while dividing 1.5D by -0.0D yields negative infinity.


Certain operations on double can produce undefined results. For instance, dividing 0.0D by 0.0D leads to an indeterminate outcome. Such indeterminate results are denoted by a special value in the float data type known as NaN (Not-a-Number).

To handle these special values, Java provides a Double class (note the uppercase D) that defines three constants representing positive infinity (POSITIVE_INFINITY), negative infinity (NEGATIVE_INFINITY), and NaN for the double data type.

The **Floating point** types



Don't compare float and double using == operator, Instead use > or < you may be wondering why?
Well, if you use == with float/double as loop terminating condition, then this may result in an endless loop because a float/double may not be precisely equal to sometimes as Java uses binary floating-point calculation, which results in approximation.



By default, the result of an integer, the calculation is int, and a floating-point calculation is double in Java. If an expression involves both double and int, then the result will always be double, as shown in the below example

```
double myDoubleNum = 10.0;
```

```
int myIntNum = 9;
```

```
int result = myDoubleNum * myIntNum; // compile time error, need casting because result is double
```

```
double result = myDoubleNum * myIntNum; // Valid statement
```

USING **UNDERSCORE** IN NUMERIC LITERALS

FROM JAVA 7



We can use the underscore(_) inside the numeric values to improve readability of the code. However the compiler will remove them internally while processing the numeric values.

For example if our code contains numbers with many digits, we can use an underscore character to separate digits in groups, similar to how we would use a punctuation mark like a comma in mathematics.



```
int num = 1_00_00_000;
```



But please note that below restrictions while using the underscore inside your numeric values,

- It is not allowed at the beginning or end of a number
- It is not allowed adjacent to a decimal point
- It is not allowed prior to an L or F suffix that we use to indicate long/float numbers
- It is not allowed where a string of digits is expected.



```
int num = _6, 6_, 6._0, 6_.0, 6_54_00_L, 6_54.00_F
```

Other formats supported by integer & floating data types

“

Apart from Decimal number format, we can also define values in the following formats by using integer and floating primitive data types,

Octal number format

Hexadecimal number format

Binary number format

”

Octal number format

If an literal starts with a zero and has at least two digits, it is treated as an octal number. In the following line of code, a decimal value of 25 (031 in octal) is assigned to the variable myNumber:

```
// octal number 031 will be 25 in decimal  
int myOctalNumber = 031;
```

Hexadecimal number format

Every literal in the hexadecimal number format commences with either 0x or 0X, signifying zero followed by an uppercase or lowercase 'X'. It is imperative that these literals consist of at least one hexadecimal digit. The hexadecimal number format employs 16 digits, encompassing 0–9 and A–F (or a–f), with the case of letters A–F being insignificant.

```
// hexa number 0x453f will be 17727.0 in decimal  
float myHexaNumber = 0x453f;
```

Binary number format

A literal can also be expressed in the binary number format. Every integer literal in binary starts with either 0b or 0B, denoting zero followed by an uppercase or lowercase 'B'.

```
// binary number 0B011 will be 3 in decimal  
byte myBinaryNumber = 0B011;
```

Sample example of Octal to Decimal

1 Identify Octal Digits:
An octal number contains digits from 0 to 7. Identify each digit in the octal number.

2 Write Down Powers of 8:
Starting from the rightmost digit, write down powers of 8 for each position. The rightmost digit corresponds to 8^0 , the next digit to 8^1 , and so on.

3 Multiply and Sum:
Multiply each octal digit by the corresponding power of 8 and sum the results.

4 Calculate Decimal Value:
The sum obtained in step 3 is the decimal equivalent of the octal number.

Here's an example to illustrate the process to convert the octal number 346_8 or 0346 into decimal,

$$\begin{aligned} & (6 \times 8^0) + (4 \times 8^1) + (3 \times 8^2) \\ &= (6 \times 1) + (4 \times 8) + (3 \times 64) \\ &= 6 + 32 + 192 \\ &= 230 \end{aligned}$$

So, 346_8 in octal is equal to 230_{10} in decimal.

Output sample from Java:

```
jshell> int myOctalNumber = 0346;  
myOctalNumber ==> 230
```


Sample example of Hexa to Decimal

1

Identify Hex Digits:

A hexadecimal number contains digits from 0 to 9 and letters A to F (or a to f) representing values 10 to 15. Identify each digit in the hexadecimal number.

2

Write Down Powers of 16:

Starting from the rightmost digit, write down powers of 16 for each position. The rightmost digit corresponds to 16^0 , the next digit to 16^1 , and so on.

3

Convert Hex Letters to Decimal:

If there are letters (A-F or a-f) in the hex number, replace them with their decimal equivalents: A=10, B=11, C=12, D=13, E=14, F=15.

4

Multiply and Sum to calculate decimal value:

Multiply each hex digit (including the converted letters) by the corresponding power of 16 and sum the results to get the decimal value.

Here's an example to illustrate the process to convert the hexa number $1A3_{16}$ or $0X1A3$ into decimal,

$$\begin{aligned} & (3 \times 16^0) + (10 \times 16^1) + (1 \times 16^2) \\ &= 3 + (10 \times 16) + (1 \times 256) \\ &= 3 + 160 + 256 \\ &= 419 \end{aligned}$$

So, $1A3_{16}$ in hexa is equal to 419_{10} in decimal.

Output sample from Java:

```
jshell> int myHexaNumber = 0X1A3;  
myHexaNumber ==> 419
```


Sample example of Binary to Decimal

1

Identify Binary Digits:

A binary number contains only digits 0 and 1. Identify each digit in the binary number.

2

Write Down Powers of 2:

Starting from the rightmost digit, write down powers of 2 for each position. The rightmost digit corresponds to 2^0 , the next digit to 2^1 , and so on.

3

Multiply and Sum to calculate decimal value:

Multiply each binary digit by the corresponding power of 2 and sum the results to get the decimal value.

Here's an example to illustrate the process to convert the octal number 1101_2 or $0B1101$ into decimal,

$$\begin{aligned} &(1 \times 2^0) + (0 \times 2^1) + (1 \times 2^2) + (1 \times 2^3) \\ &= 1 + 0 + 4 + 8 \\ &= 13 \end{aligned}$$

So, 1101_2 in binary is equal to 13_{10} in decimal.

Output sample from Java:

```
jshell> int myBinaryNumber = 0B1101;  
myBinaryNumber ==> 13
```

Type Casting in Java

eazy
bytes

When we assign a value of one primitive data type to another type, we call it as **type casting**.

In Java, there are two types of casting:

Widening Casting (automatically/implicit) - converting a smaller type to a larger type size. It is done automatically & safe because there is no chance to lose data.

byte -> short -> char -> int -> long -> float -> double

Narrowing Casting (manually/explicit) - converting a larger type to a smaller size type. It should be done manually by the programmer. If we do not perform casting then the compiler reports a compile-time error.

double -> float -> long -> int -> char -> short -> byte

Type Casting in Java

eazy
bytes

```
int myIntNum = 16;  
double myDoubleNum = myIntNum; // Automatic casting from int to double  
System.out.println(myIntNum);    // Outputs 16  
System.out.println(myDoubleNum); // Outputs 16.0
```

Example of
implicit
casting



boolean to other data types casting is not possible as the supported values are different

```
jshell> int myIntNum = 16;  
myIntNum ==> 16  
  
jshell> double myDoubleNum = myIntNum;  
myDoubleNum ==> 16.0  
  
jshell> System.out.println(myIntNum);  
16  
  
jshell> System.out.println(myDoubleNum);  
16.0  
  
jshell> █
```

Type Casting in Java

eazy
bytes

Example of
explicit
casting

```
double myDoublePiNum = 3.14d;  
int myIntPiNum = (int) myDoublePiNum; // Manual casting: double to int  
System.out.println(myDoublePiNum); // Outputs 3.14  
System.out.println(myIntPiNum); // Outputs 3
```

Casting Syntax

RequiredDatatype Variable1 = (TargetType) Variable2 ;

```
jshell> double myDoublePiNum = 3.14d;  
myDoublePiNum ==> 3.14  
  
jshell> int myIntPiNum = (int) myDoublePiNum;  
myIntPiNum ==> 3  
  
jshell> System.out.println(myDoublePiNum);  
3.14  
  
jshell> System.out.println(myIntPiNum);  
3  
  
jshell> █
```

Type Casting in Java

eazy
bytes

More
examples



```
int num1 = 13;  
long num2 = 2147483648L; // number outside int range  
num1 = num2; // compile-time error. long to int assignment is not allowed in Java  
num1 = (int) num2; // No error. But result in overflow
```

```
int num1 = 6;  
long num2 = 36L;  
num1 = num2; // compile-time error. Even if num2 value 36 is in the range of int
```

```
int num1 = 2000;  
float num2 = 2314.23F;  
num1 = num2; // compile-time error. Cannot assign float to int  
num1 = (int) num2; // OK. num1 will store 2314
```

```
char charValue = 'A';  
int intValue = charValue; // intValue holds 65 because casting a char to an int  
gives the ASCII value of the character.
```

```
int num1 = 1438764585;  
float num2 = num1;  
int num3 = (int) num2;
```

You anticipate that the values held in num1 and num3 should be identical. Nevertheless, they differ due to the inability to precisely represent the value stored in num1 within the floating-point format of the float variable num2. Not all floating-point numbers can be exactly represented in binary format, which accounts for the disparity between num1 and num3.



The **String** types

More to come...

In Java, a **String** is a sequence of characters. It is an object that represents a series of characters and provides a wide range of methods to manipulate and work with strings.

For example, the string **"Java"** consists of the four characters J, a, v, a

We have just described the primitive value types of the Java language. All the other value types in Java including String belong to a category of **reference types**.

String literals are the most common way to create text in Java. They are created by enclosing a series of characters within **double quotes**.

Any reference class can refer to a null literal. It represents a reference value that does not point to any object. In the case of a String type, it looks like below:

String input = null;

Sample declaration & initialization of String,

String input = "Hello World !";

Sample code snippet in jshell,

```
jshell> String input = "Hello World !";  
input ==> "Hello World !"  
  
jshell> System.out.println(input);  
Hello World !  
  
jshell> █
```



A reference type is so-called because, in the code, we do not deal with values of this type directly. A value of a reference type is more complex than a primitive-type value. It is called an object and requires more complex memory allocation, so a reference-type variable contains a memory reference. It points (refers) to the memory area where the object resides, hence the name. We will discuss more about Class & Objects in coming lectures