Fundamentals of Java

What is a Variable?

- In Java, a variable is a name given to a memory location. It is the basic unit of storage in a program. The value stored in a variable can be changed during program execution. Variables in Java are only a name given to a memory location. All the operations done on the variable affect that memory location. Software objects are conceptually similar to real-world objects: they too consist of state and related behavior. An object stores its state in *fields* (variables in some programming languages) and exposes its behavior through *methods* (functions in some programming languages).

Declaring Variable

- To declare a variable, we specify its type followed by a name. For example, int age; declares an integer variable called age.

Assigning a Value

- After declaring a variable, we can assign a value to it using the assignment operator “=”. For example, age = 25; assigns the value 25 to the age variable.

- We can also declare and assign a value in a single line, like int score = 95;

Printing a Variable

- We can display the value of a variable using the System.out.println() method.

- For example, System.out.println("Age: " + age); will print the value of the age variable along with the text "Age: ".

Example:

```java
public class Example {
    public static void main(String[] args) {
        int age;        // Declare an integer variable
        age = 25;       // Assign a value to it
        System.out.println("Age: " + age);  // Print the value
    }
}
```

In this simple example, we declared an integer variable called age, assigned the value 25 to it, and then printed its value, resulting in the output "**Age: 25**".

Variables (Literals) Naming Conventions

Every programming language has its own set of rules and conventions for the kinds of names that you're allowed to use, and the Java programming language is no different. The rules and conventions for naming your variables can be summarised as follows:

- Variable names are case-sensitive. A variable's name can be any legal identifier — an unlimited-length sequence of Unicode letters and digits, beginning 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 **"_"** or digits.

- Additionally, the dollar sign character, by convention, is never used at all. In some situations, auto-generated names will contain the dollar sign, but your variable names should always avoid using it.

- A similar convention exists for the underscore character; while it's technically legal to begin your variable's name with **"_"**, this practice is discouraged. White space is not permitted.

- Subsequent characters may be letters, digits, dollar signs, or underscore characters. Conventions (and common sense) apply to this rule as well. 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 `cadence`, `speed`, and `gear`, for example, are much more intuitive than abbreviated versions, such as `s`**,** `c`**,** and `g`. Also, keep in mind that the name you choose must not be a keyword or reserved word.

- If the name you choose consists of only one word, spell that word in all lowercase letters. If it consists of more than one word, capitalize the first letter of each subsequent word (also known as **Camel Case**). The names `gearRatio` and `currentGear` are prime examples of this convention. If your variable stores a constant value, such as `static final int NUM_GEARS = 6`, the convention changes slightly, capitalizing every letter and separating subsequent words with the underscore character. By convention, the underscore character is never used elsewhere.

Data Types

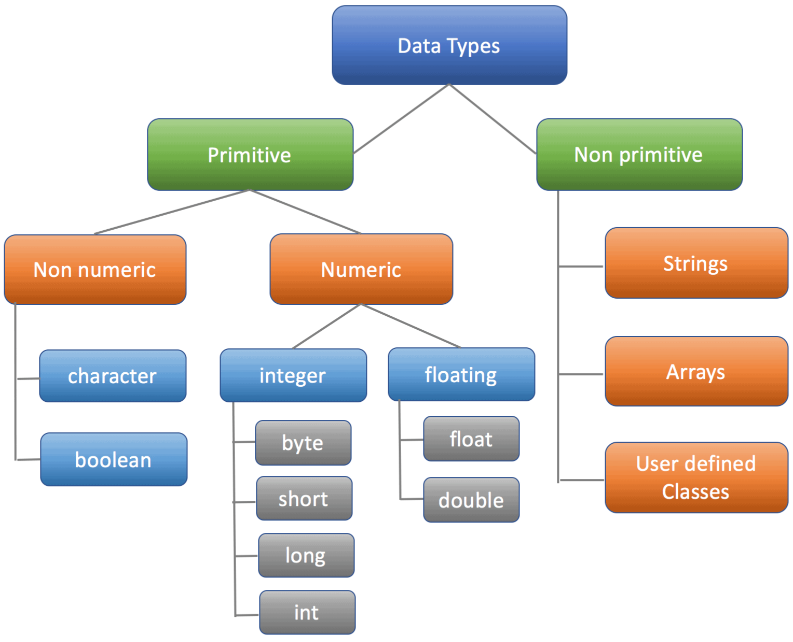
- **Declaration:** The Java programming language is statically typed, which means that all variables must first be declared before they can be used. This involves stating the variable's type and name, as you've already seen:

  ```
  int gear = 1;
  ```

  Doing so tells your program that a field named **"gear"** exists, holds numerical data, and has an initial value of **"1"**. A variable's data type determines the values it may contain, plus the operations that may be performed on it. In addition to `int`, the Java programming language supports seven other *primitive data types*.

  A primitive type is predefined by the language and is named by a reserved keyword.

  Primitive values do not share a state with other primitive values.



  The eight primitive data types supported by the Java programming language are:

- **byte**: The `byte` data type is an 8-bit signed two's complement integer. It has a minimum value of **-128** and a maximum value of **127** (inclusive). The `byte` data type can be useful for saving memory in large arrays, where memory savings actually matter. They can also be used in place of `int` where their limits help to clarify your code.

- **short**: The `short` data type is a 16-bit signed two's complement integer. It has a minimum value of **-32,768** and a maximum value of **32,767** (inclusive). As with `byte`, the same guidelines apply: you can use a `short` to save memory in large arrays, in situations where the memory savings actually matters.

- **int**: By default, the `int` data type is a **32-bit** signed two's complement integer, which has a minimum value of **-2^{31}** and a maximum value of **2^{31}-1**.
  - In Java SE 8 and later, you can use the `int` data type to represent an unsigned 32-bit integer, which has a minimum value of 0 and a maximum value of **2^{32}-1**. Use the Integer class to use `int` data type as an unsigned integer.

- **long**: The `long` data type is a 64-bit two's complement integer. The signed long has a minimum value of **-2^{63}** and a maximum value of **2^{63}-1**. In Java SE 8 and later, you can use the `long` data type to represent an unsigned 64-bit long, which has a minimum value of 0 and a maximum value of **2^{64}-1**. Use this data type when you need a range of values wider than those provided by `int`.

- **float**: The `float` data type is a single-precision **32-bit** floating point. As with the recommendations for `byte` and `short`, use a `float` (instead of `double`) if you need to save memory in large arrays of floating point numbers.

- **double**: The `double` data type is a double-precision **64-bit** floating point.

- **boolean**: The `boolean` data type has only two possible values: `true` and `false`. Use this data type for simple flags that track true/false conditions. This data type represents one bit of information, but its "size" isn't something that's precisely defined.

- **char**: The `char` data type is a single **16-bit Unicode** character. It has a minimum value of `'\u0000'` (or 0) and a maximum value of `'\uffff'` (or 65,535 inclusive).

- In addition to the eight primitive data types listed above, the Java programming language also provides special support for character strings via the java.lang.String class.

- Enclosing your character string within double quotes will automatically create a new String object; for example, String s = **"this is a string";** .

- String objects are immutable, which means that once created, their values cannot be changed. The String class is not technically a primitive data type.

- For the primitive data types, mention the associated wrapper classes too (Byte, Short, Integer, Long, Float, Double, Boolean, Character, etc.).

Literal Types

You may have noticed that the new keyword isn't used when initializing a variable of a primitive type. Primitive types are special data types built into the language; they are not objects created from a class. A *literal* is the source code representation of a fixed value; literals are represented directly in your code without requiring computation.

As shown below, it's possible to assign a literal to a variable of a primitive type:

```
boolean result = true;
char capitalC = 'C';
byte b = 100;
short s = 10000;
int i = 100000;
```

Integer Literals:

- An integer literal is of type long if it ends with the letter `L` or `l`; otherwise, it is of type `int`. It is recommended that you use the uppercase letter L because the lowercase letter l is hard to distinguish from the digit 1.

- Values of the integral types byte, short, int, and long can be created from int literals. Values of type long that exceed the range of int can be created from long literals. Integer literals can be expressed by these number systems:

  - **Decimal**: Base 10, whose digits consist of the numbers 0 through 9; this is the number system you use every day
  - **Hexadecimal**: Base 16, whose digits consist of the numbers 0 through 9 and the letters A through F
  - **Binary**: Base 2, whose digits consists of the numbers 0 and 1 (you can create binary literals in Java SE 7 and later)

- For general-purpose programming, the decimal system is likely to be the only number system you'll ever use. However, if you need to use another number system, the following example shows the correct syntax. The prefix 0x indicates hexadecimal and 0b indicate binary:

```
// The number 26, in decimal
int decVal = 26;
//  The number 26, in hexadecimal
int hexVal = 0x1a;
// The number 26, in binary
int binVal = 0b11010;
```

Floating Point Literals

- A floating-point literal is of type `float` if it ends with the letter `F` or `f`; otherwise its type is `double` and it can optionally end with the letter `D` or `d`.

- The floating point types (`float` and `double`) can also be expressed using E or e (for scientific notation), F or f (32-bit float literal), and D or d (64-bit double literal; this is the default and by convention is omitted).

```
double d1 = 123.4;
// same value as d1, but in scientific notation
double d2 = 1.234e2;
float f1  = 123.4f;
```

Characters and String Literals

- Literals of types `char` and `String` may contain any Unicode (UTF-16) characters. If your editor and file system allow it, you can use such characters directly in your code. If not, you can use a "Unicode escape" such as `'\u0108'` (capital C with circumflex), or `"S\u00ED Se\u00F1or"` (Sí Señor in Spanish).

- Always use 'single quotes' for `char` literals and "double quotes" for `String` literals. Unicode escape sequences may be used elsewhere in a program (such as in field names, for example), not just in `char` or `String` literals.

- The Java programming language also supports a few special escape sequences for `char` and `String` literals: `\b` (backspace), `\t` (tab), `\n` (line feed), `\f` (form feed), `\r` (carriage return), `\"` (double quote), `\'` (single quote), and `\\` (backslash).

- There's also a special `null` literal that can be used as a value for any reference type. `null` may be assigned to any variable, except variables of primitive types. There's little you can do with a `null` value beyond testing for its presence. Therefore, `null` is often used in programs as a marker to indicate that some object is unavailable.

- Finally, there's also a special kind of literal called a *class literal*, formed by taking a type name and appending "`.class`"; for example, `String.class`. This refers to the object (of type `Class`) that represents the type itself.

- ASCII Table (Unicode 15.0 Character Code Charts).

Using Underscore Characters in Numeric Literals

In Java SE 7 and later, any number of underscore characters (_) can appear anywhere between digits in a numerical literal. This feature enables you, for example. to separate groups of digits in numeric literals, which can improve the readability of your code. For instance, if your code contains numbers with many digits, you can use an underscore character to separate digits into groups of three, similar to how you would use a punctuation mark like a comma, or a space, as a separator.

The following example shows other ways you can use the underscore in numeric literals:

```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
float pi =  3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```

You can place underscores only between digits; you cannot place underscores in the following places:

- At the beginning or end of a number.
- Adjacent to a decimal point in a floating point literal.
- Prior to an `F` or `L` suffix.
- In positions where a string of digits is expected.

The following examples demonstrate valid and invalid underscore placements (which are highlighted) in numeric literals:

```
// Invalid: cannot put underscores
// adjacent to a decimal point
float pi1 = 3_.1415F;

// Invalid: cannot put underscores
// adjacent to a decimal point
float pi2 = 3._1415F;

// Invalid: cannot put underscores
// prior to an L suffix
```

```java
long socialSecurityNumber1 = 999_99_9999_L;

// OK (decimal literal)
int x1 = 5_2;

// Invalid: cannot put underscores
// At the end of a literal
int x2 = 52_;

// OK (decimal literal)
int x3 = 5_______2;

// Invalid: cannot put underscores
// in the 0x radix prefix
int x4 = 0_x52;

// Invalid: cannot put underscores
// at the beginning of a number
int x5 = 0x_52;

// OK (hexadecimal literal)
int x6 = 0x5_2;

// Invalid: cannot put underscores
// at the end of a number
int x7 = 0x52_;
```

String Literals

Strings, which are widely used in Java programming, is a sequence of characters. In the Java programming language, strings are objects. The Java platform provides the `String` class to create and manipulate strings.

Creating Strings

The most direct way to create a string is to write:

```
String greeting = "Hello world!";
```

In this case, "Hello world!" is a *string literal*—a series of characters in your code that is enclosed in double quotes. Whenever it encounters a string literal in your code, the compiler creates a `String` object with its value—in this case, `Hello world!`.

As with any other object, you can create `String` objects by using the `new` keyword and a constructor. The `String` class has thirteen constructors that allow you to provide the initial value of the string using different sources, such as an array of characters:

```
char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };
String helloString = new String(helloArray);
System.out.println(helloString);
```

The last line of this code snippet displays `hello`.

Java Wrapper Classes

In Java, data can be categorized into two types: primitive and non-primitive (objects). While primitive data types like int, float, and boolean are efficient for simple data storage, they lack the flexibility of objects. This is where Java wrapper classes come to the rescue. Wrapper classes provide a way to represent primitive data types as objects, enabling us to use them in object-oriented contexts and collections. In this beginner-friendly tutorial, we'll explore Java wrapper classes and how to use them effectively.

What Are Wrapper Classes?

Java wrapper classes are a set of classes that wrap primitive data types into objects. There's a wrapper class for each of the eight primitive data types:

1. Byte: Wraps byte.
2. Short: Wraps short.
3. Integer: Wraps int.
4. Long: Wraps long.
5. Float: Wraps float.
6. Double: Wraps double.
7. Character: Wraps char.
8. Boolean: Wraps boolean.

Why Use Wrapper Classes?

1. Compatibility: Many Java libraries and APIs require objects. Wrapper classes make it possible to use primitive data types where objects are needed.

2. Null Values: Wrapper classes allow you to assign null values to variables, which isn't possible with primitives.

3. Utility Methods: They provide useful methods for conversions, comparisons, and other operations.

Creating Wrapper Objects

You can create wrapper objects using constructors or static factory methods. For example:

```
Integer intObj = new Integer(42); // Using constructor
Double doubleObj = Double.valueOf(3.14); // Using valueOf()
method
Boolean boolObj = Boolean.TRUE; // Using a constant
```

Auto-Boxing and Auto-Unboxing

Java simplifies the process of converting between primitive types and wrapper objects. With auto-boxing, you can assign primitive values to wrapper objects without explicit conversion, and auto-unboxing lets you extract primitive values from wrappers. For instance:

```
Integer num = 10; // Auto-boxing (primitive to wrapper)
int value = num; // Auto-unboxing (wrapper to primitive)
```

Common Operations

Wrapper classes provide methods for common tasks:

- `toString(): Convert to a string.`
- `parseInt(): Parse a string to an int.`
- `equals(): Compare two wrapper objects.`
- `compareTo(): Compare two wrapper objects.`

In conclusion, Java wrapper classes bridge the gap between primitive and object-oriented programming. They offer flexibility and compatibility while simplifying data manipulation and usage in various Java contexts. Learning to use wrapper classes effectively will enhance your Java programming skills.