A variable used to hold a string value in Java has the type `String`. The string data itself, when typed in quotes in code, is called a *string literal*; Java should interpret it as literal string data, and not as code.

`String` is a class

Each of the Java programs you have seen so far was defined as a single class containing some methods. **A class in Java has two purposes:**

1. A class is a collection of methods that you may call;
2. A class defines a custom data type, and can be used to make objects of that class.

The `String` class serves both purposes: it provides some methods that are useful for `Strings`, and it defines a `String` data type that you can use to make string objects.

Let's look at the first use first. The `String` class is contained somewhere in a Java file called `String.java`. It contains within it several static methods. You can call a static method of a class by using the format `Classname.methodname()`.

Usually, before using the static methods of a class, you need to **import** that class, which we will see how to do later, but `String` is special and built-in.

The static method `String.valueOf()`

As an example of a static method in the `String` class, let's look at the method `valueOf`. It takes a single parameter, which may be a double, int, or boolean, for example. It then constructs and returns a string representation of that parameter. Let's do it:

```
package com.github.akarazhev.jacademy.jprog.basics;

public final class StringExampleOne {

    public static void main(final String[] args) {
        String myFavoriteNumber;
        final int x = 42;

        myFavoriteNumber = String.valueOf(x);
        System.out.println("My favorite number is " + myFavoriteNumber + ".");
    }
}
```

Strings are objects

A class in Java provides the code for methods, but also allows you to create what is called an *object* of that class. Among other things, an object is a structure in memory that stores values. A `String` object stores data: the value of each character in the string, as well as the length of the string. Some things you can do with objects are:

1. Declare a variable to store a reference to an object of a class: `String s`.
2. Create a new object of that type: `s = "Rumplestiltskin";` Strings are somewhat special in Java; other types of objects are created using special method calls.
3. Read or modify data from the object, using dot notation to get at *instance variables*. In Java, the String method has no directly useful instance variables, so we'll see examples of this with other classes.
4. Call non-static methods on an object; these methods have access to the data within an object: `System.out.println(s.toUpperCase());`

```
package com.github.akarazhev.jacademy.jprog.basics;

public final class StringsAsObjects {

    public static void main(final String[] args) {
        String s = "Rumplestiltskin";
        System.out.println(s.length());
        System.out.println(s.toUpperCase());
        System.out.println("The character at index 5 is: " + s.charAt(5));
    }
}
```

Static vs non-static methods of the String class

The Java code for the String class is stored in some file `String.java` that the Java compiler has access to, and contains code for both static and non-static method definitions.

To access static methods of the `String` class, recall that you use both the name of the class and the method name: `String.valueOf(5)`. You need the class name because there might be other methods named `valueOf()` out there. (Indeed, there is a class called `Integer` that also has a `valueOf` method.)

Static methods do not need access to a particular string object's data. The static `valueOf` method takes a parameter that is not a String, and creates a String. Static methods are just like functions in Javascript, C, or Python. Java just uses classes to organize these functions.

Non-static methods require a string as input, and instead of passing that string as a parameter, dot-notation is used: `s.toUpperCase()`. It's possible that there might be other methods named `toUpperCase` out there. Java knows which method to call because Java knows that `s` is a String, so Java looks in `String.java` for the method.

The `char` data type

Unlike in Python or Javascript, you cannot access the characters of the string using `[]` bracket notation; in Java, bracket notation is reserved only for arrays, and you must use the `charAt` method with strings. `charAt` does not return a String: instead, it returns something called a `char`.

Data types like `int`, `double`, and `boolean` in Java are called *primitive data types*, because variables of those types are built-in, and do not have methods available to act on them. `char` is also a primitive data type, representing a single character of text.

Although the `String` data type is built-in, the `String` data type is not a primitive: it is defined by a class, with methods and an internal representation defined by that class.

Objects organize data, some of which may be composed of primitive types. A string organizes the characters in the string, which are internally represented using `char` values.

To distinguish between `String` and `char` data, Java uses single quotes for character literals and double quotes for Strings:

```
package com.github.akarazhev.jacademy.jprog.basics;

public final class CharExample {

    public static void main(final String[] args) {
        final char character = 'Z';
        final String magicWord = "XYZZY";

        final char anotherChar = magicWord.charAt(0);
        System.out.println(character + anotherChar);
    }
}
```

The code above prints out 178, which might surprise you. The `+` operator concatenates strings, but characters are internally represented using numbers, with a code called *Unicode*. Since the unicode value of 'X' is 88, and the unicode value of 'Z' is 90, the `+` operator adds the two up and returns an `int`.

However, you can concatenate a character to a string easily with `+`, since the `+` operator converts both operands to strings if one of the operands is a string.

Strings are immutable in Java

We will see later that variables do not actually hold objects, but *references* to objects. A reference may be loosely thought of as the address of the data storing the object in memory.

One implication is that if you pass the reference to an object to a method, the method may change the object. This is different than primitive types. If I pass the value of a variable to the `sqrt` method, I can rest assured that the `sqrt` function has no way to change that variable: the method only has the variable's value.

Java protects against strings getting changed accidentally by making objects of the String class *immutable*. You can safely pass a reference to a string to a method without worrying that the String will be changed. Notice that the `toUpperCase()` method we saw above *does not change the string*: it just creates a new string that is uppercased, and returns the result.

Therefore, although there is a way to get a character from a String using the `charAt()` method, there is no way to change a character within a string: if you want a variable to refer to some other string, you need to create a new string. There's another class in Java, `StringBuffer`, that is useful if you need to efficiently manipulate character arrays. For example, you might use a `StringBuffer` to store the text data if you were writing a text editor.