Scientific Programming Using Object-Oriented Languages
Module 2: Objects and Classes

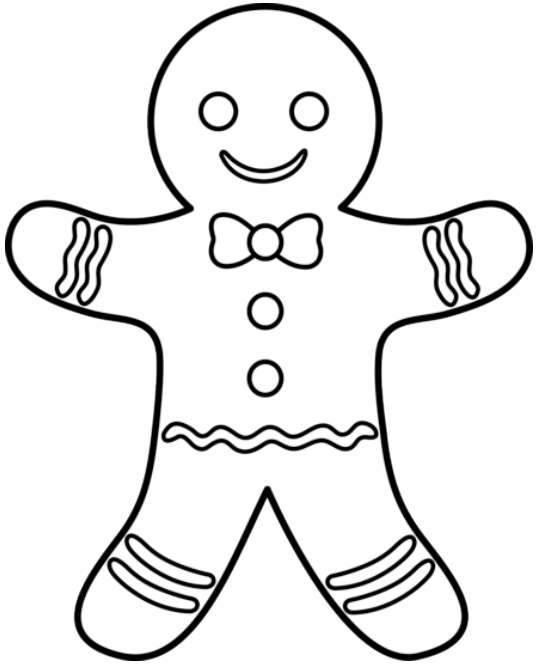
Aims of Module 2:

- Be able to define classes and use objects.
- Understand and apply the concept of encapsulation.
- Know when and how to use **`static`** methods and variables.
- Know when and how to use **`final`** variables.
- Know about and be able to use the `String` class and the numeric data-type wrapper classes.
- Know how to find out about other classes in the Java API (docs.oracle.com/javase/8/docs/api/).

Why Are Objects Used?

- Represent more complicated data-types than single numbers
- Represent objects from the real world
- Modularise code:
  - easier to understand
  - easier to extend
- Encapsulation:
  - objects can only be manipulated through controlled methods
  - protects user from unintended consequences of directly changing variables
- Polymorphism/inheritance
  - but that's another module!

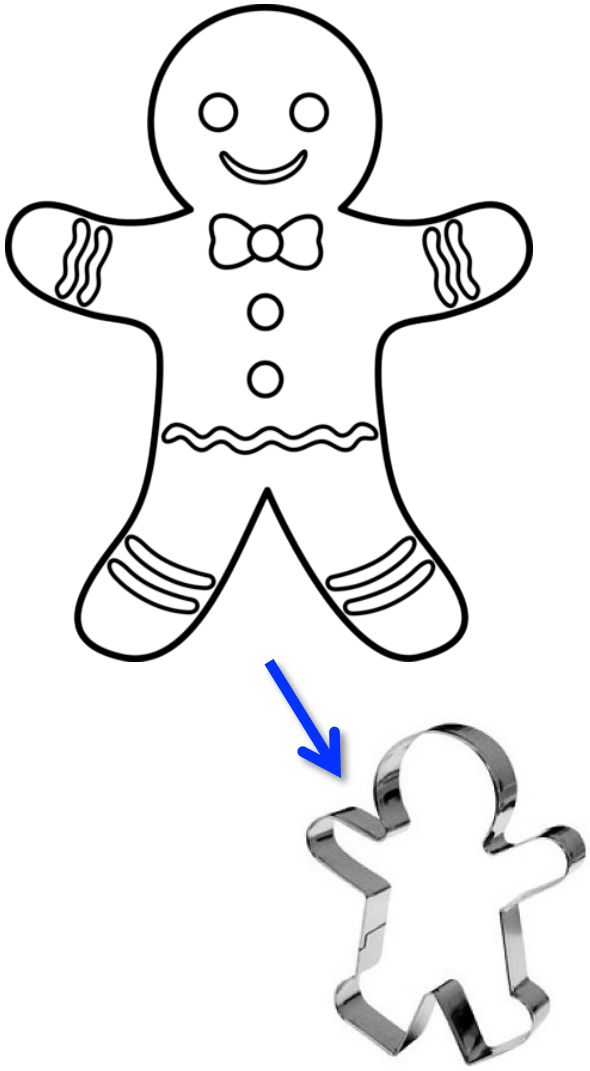
Classes & Objects: The Class



A *class* is a data type that can represent more complicated entities than simply single real or integer numbers, characters or boolean variables.

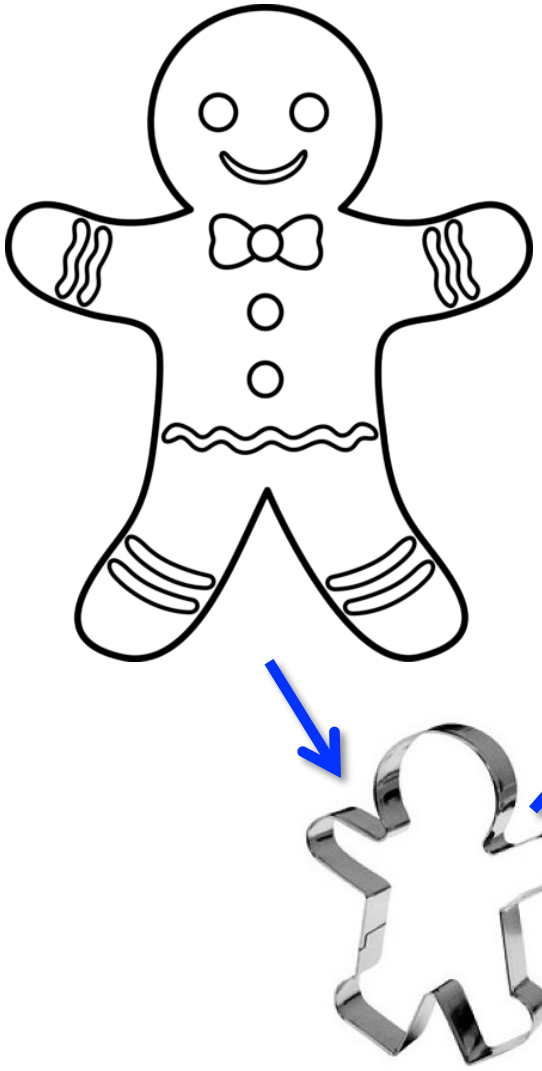
You create objects using the “blueprint” defined by the class.

Classes & Objects: The Constructor



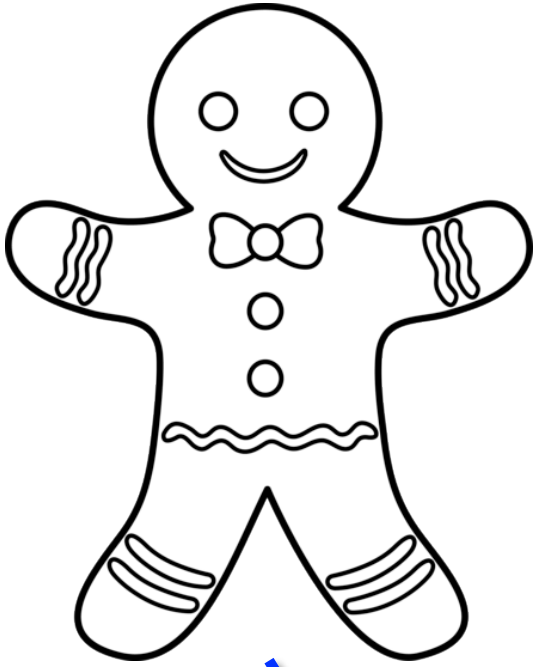
A *constructor* is used to set up an object when we tell Java to create a new object using the `new` command. The constructor is defined within the class and populates the structure of the object when you first *instantiate* it.

Classes & Objects: Instantiation



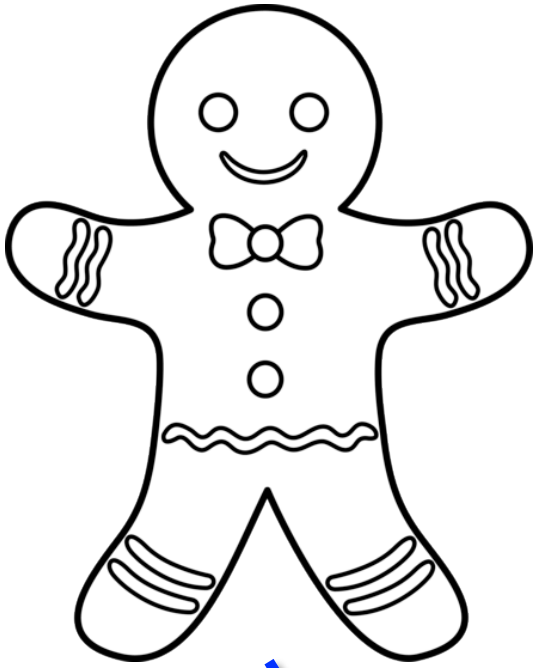
Creating an object using a constructor from a given class is called *instantiation*. This creates a new *instance* of that class in the form of the new object.

Classes & Objects: A New Object



Once you instantiate a new object using the constructor, you now have an object whose "layout" follows the "blueprint" defined by the class. The new object contains all the *methods* defined by that class as well as the default member variables.

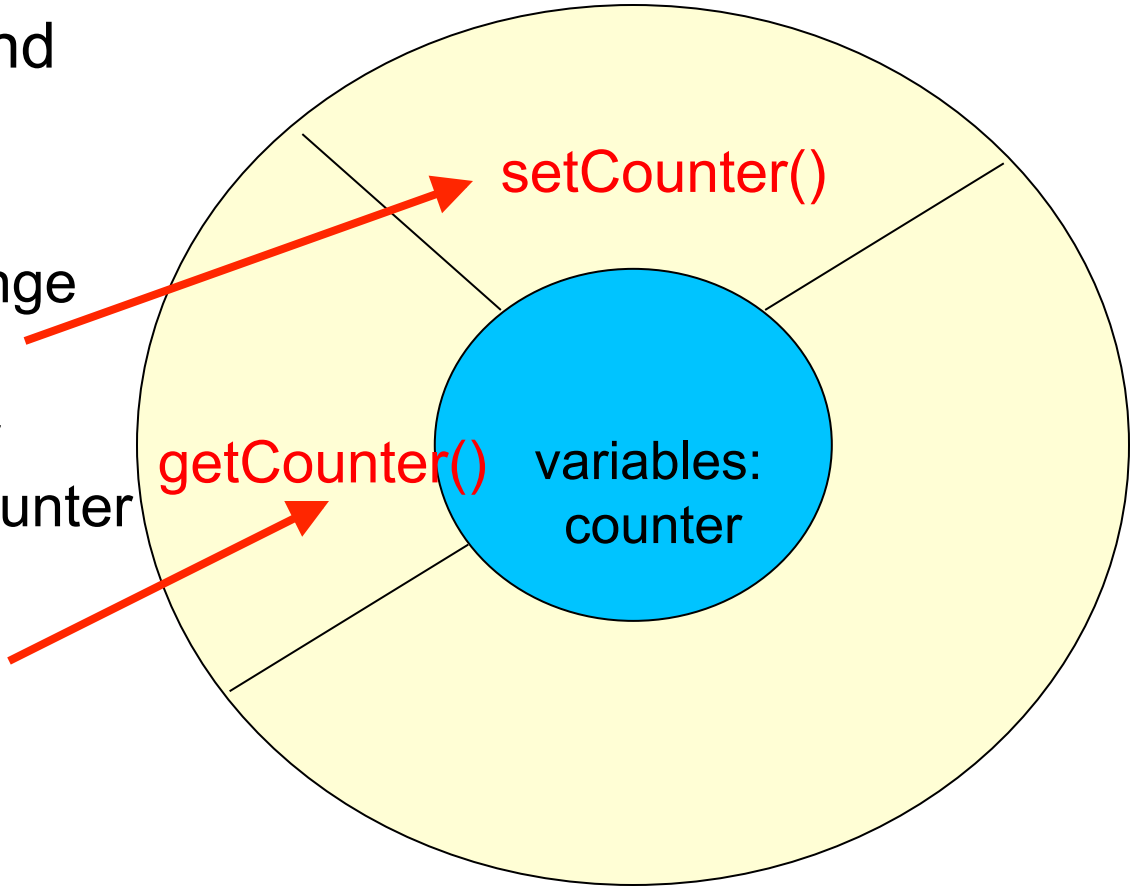
Classes & Objects: Methods



You can now manipulate the data contained within the object using the *methods* associated with that class. These “functions” allow you to modify the state of the object – such as passing data back and forth – or make specific calculations defined by the class.

Encapsulation

- Communicate with an object by sending and receiving *messages*
  - send message to setCounter to change counter to 10
  - ask getCounter for current value of counter



A Simple Class

```
public class SimpleCounter {

   int counter;                  // member variable
   public SimpleCounter() {} // constructor (ignore for now)

   int getCounter() {return counter;} // method to set value
   void setCounter(int val) {counter = val;} // retrieve value

}
```

Using Our Class

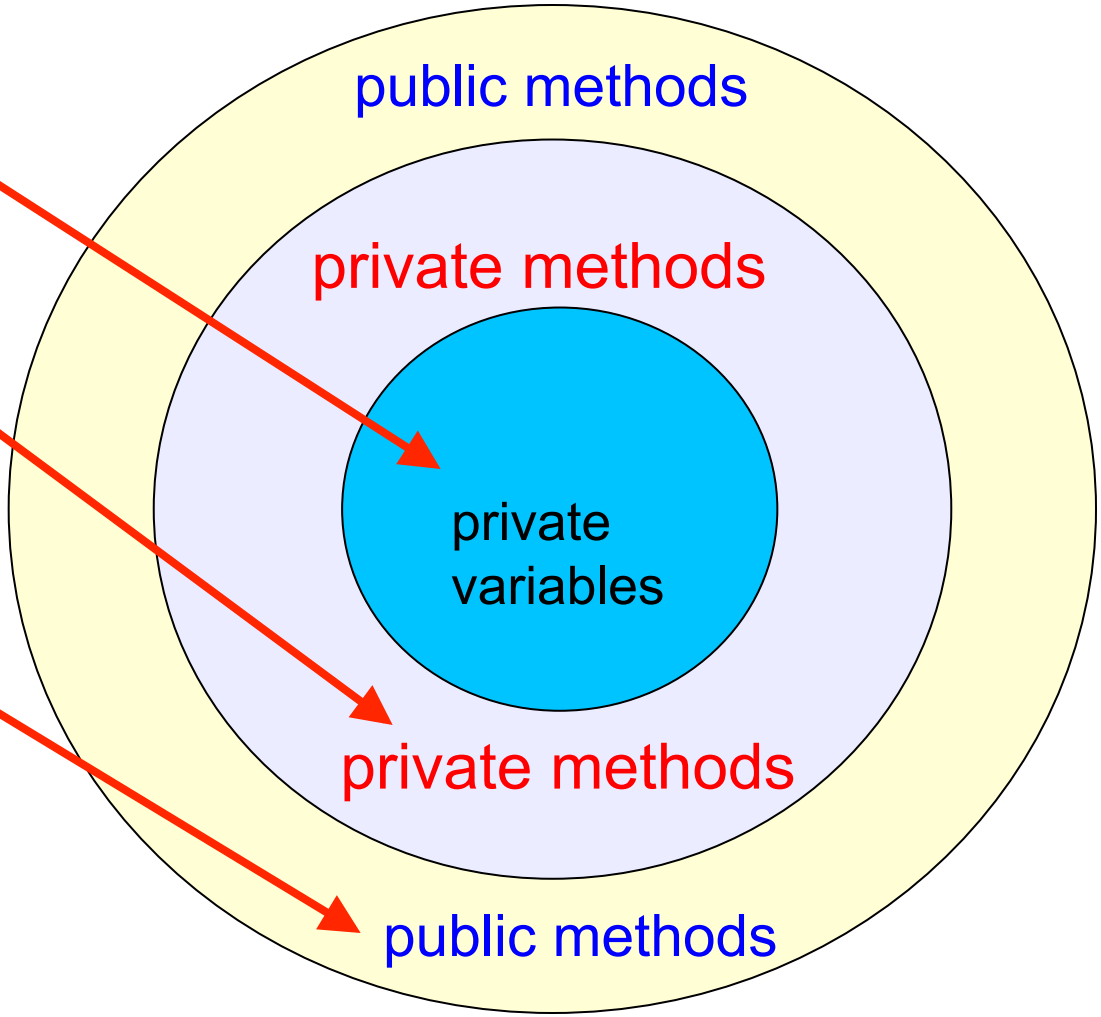
```
public class TestSimpleCounter {

    public static void main(String[] args) {

        SimpleCounter count1 = new SimpleCounter();
        count1.setCounter(10);
        System.out.println("value of counter is
                            "+count1.getCounter());

    }

}
```

Constructors

```
public SimpleCounter() {}
public SimpleCounter(int val) { counter = val; }


SimpleCounter count1 = new SimpleCounter();  // set to 0
SimpleCounter count1 = new SimpleCounter(3); // set to 3
```

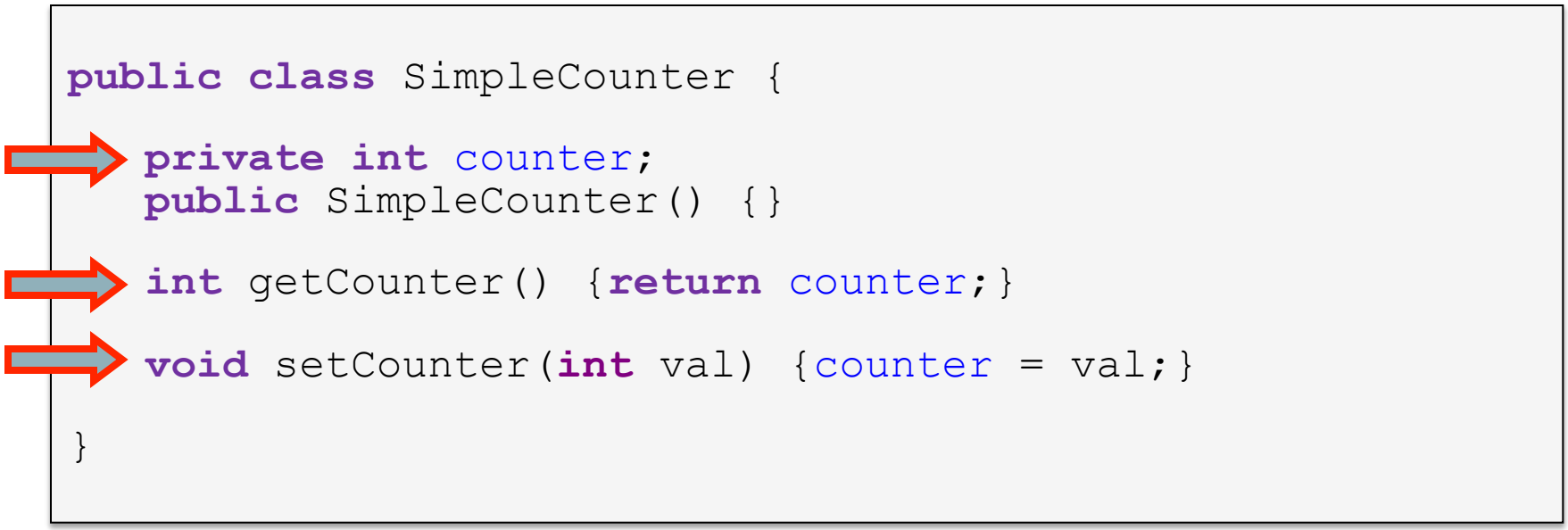
Encapsulation

- Prevent direct access to variables by making them private
- Can also have private methods, only usable from within the class
- Other classes/objects can only access this one through its public methods



Access Control

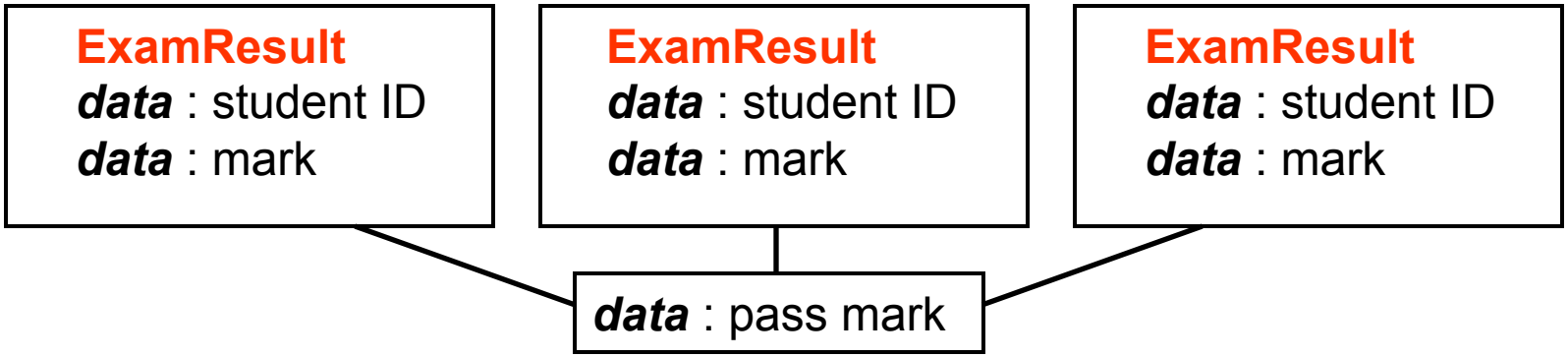
- Why not count1.counter = 10 ?
- How to stop people from doing this?



```
public class SimpleCounter {

   private int counter;
   public SimpleCounter() {}

   int getCounter() {return counter;}

   void setCounter(int val) {counter = val;}

}
```

Static Variables

- In general, each instance of an object of a particular type holds its own set of data that can differ from one instance to another.
- But there are occasions when you want the universe to contain a single data instance which is "***shared***" or "***viewed***" by all object instances.
- For example in a piece of teaching software we might have:



Static Variables

- This is achieved by declaring such variables to be `static`
- Another example:

```
public class SimpleCounter {

  private static int max = 1000; // same val for every SimpleCounter

  private int counter;
  public SimpleCounter() {}

  int getCounter() {return counter;}

  void setCounter(int val) {
    if (val<=max ) counter = val; // only change if new value < max
  }

}
```

can be referred to using "`class_name.variable_name`", e.g. "`SimpleCounter.max`"

Static Methods

- `static` methods do not depend on any non-static variables.
- In other words, the method does not depend on the unique state of a particular object.
- A static method can be used without even bothering to create an object at all by invoking "`class_name.method_name(args)`"
- For example:

```
double cos_90 = Math.cos(Math.toRadians(90));
```

The results of these math functions depend only on the arguments passed and not on the state of the object

"this"

- The keyword "**this**" is used to refer to the current object instance.

```
public static SimpleCounter add(SimpleCounter x, SimpleCounter y) {
  int sum = x.counter + y.counter;
  return new SimpleCounter(sum);
}
```

```
public SimpleCounter add(SimpleCounter x) {
  int sum = x.counter + this.counter;
  return new SimpleCounter(sum);
}
```

```
SimpleCounter sumA = sc1.add(sc2) // non-static
SimpleCounter sumB = SimpleCounter.add(sc1,sc2)
```

Rewrite to avoid code duplication.
HINT !!!! Exercises …

```
public SimpleCounter add(SimpleCounter x) {
  return new add(this,x);
}
```

call static version

Final Variables

- Variables can be declared final to prevent them being modified (either accidentally or misguidedly) during program execution.
- For example :

```
public static final double PI;
```

should never change …

which can be accessed by:

```
Math.PI;
```

- Physics simulations contain lots of constants : fundamental and specific.

Converting Objects to Strings

- By defining a `toString` method with the following signature, it becomes straightforward to print the state of an object to the screen. For example, for our `SimpleCounter` class:

SimpleCounter class definition

```
public String toString() {
    return "counter = "+counter+ ", max = "+max;
}
```

This method is invoked when the object is converted to a string:

Take care to provide an informative yet concise string version of your object.

Calling Code

```
SimpleCounter c = new SimpleCounter();
System.out.println("state of c:"+c);
```

- See Module 2 notes for more details of the `String` class.

Java API: docs.oracle.com/javase/8/docs/api/