- **Object-Oriented Programming**

Object-oriented programming (OOP) enables you to develop large-scale software and Graphical User Interfaces (GUIs) effectively. It is essentially a technology for developing reusable software. Having learned Java programming language in the previous courses, you are able to solve many computer solvable problems using selections, loops, methods, and arrays. However, these Java features are not sufficient for developing GUIs and large-scale software systems.

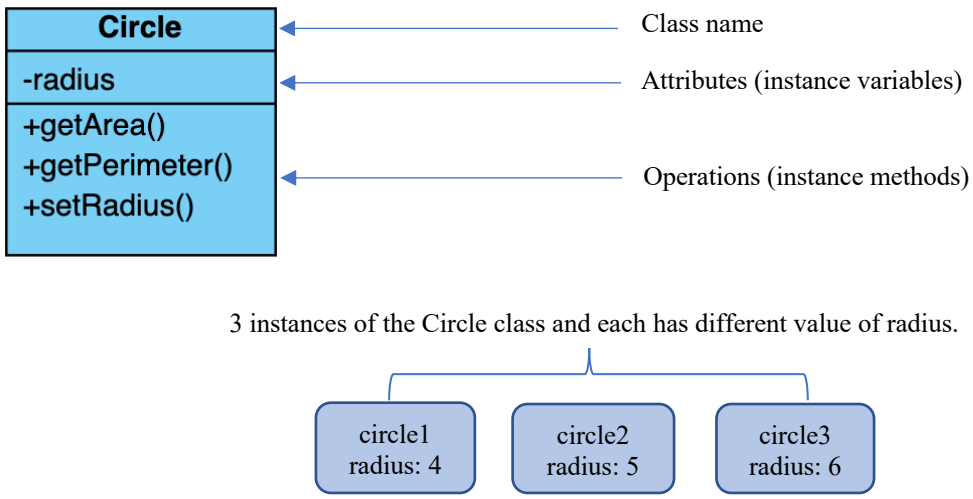
- **Class and objects**

A **class** defines the properties and behaviors for objects. OOP involves programming using objects. An object is typically used to represent an entity in the real world that can be distinctly identified, either tangible or intangible. For example, a student, a desk, a circle, a building, a loan, or an event can all be viewed as objects. An object has a unique identity, state, and behavior, depending on how you model it based on software requirements. For example, processing student tuitions may need to keep track of the credit hours students currently enrolled. The "credit hours" should be included as one of the attributes of a student.

The **state** of an object (also known as its properties or attributes) is represented by data fields with their current values. A circle object, for example, has a data field radius, which is the property that characterizes a circle. A rectangle object, for example, has the data fields width and height, which are the properties that characterize a rectangle.

The **behavior** of an object (also known as its operations) is defined by methods. To invoke a method on an object is to ask the object to perform an action, which is to manipulate the data fields. For example, you may define getter methods named **getArea()** and **getPerimeter()** for circle objects. A circle object may invoke **getArea()** to return its area and **getPerimeter()** to return its perimeter. You may also define the **setRadius(radius)** method. A circle object can invoke this method to change its radius.

Objects of the same type are defined using a common class. **A class is a template**, blueprint, or contract that defines what an object's data fields and methods will be. An object is an instance of a class. You can create many instances of a class and every instance has its own "state". Creating an instance is referred to as **instantiation**. The terms object and instance are often interchangeable. The relationship between classes and objects is analogous to that between an apple-pie recipe and apple pies: You can make as many apple pies as you want from a single recipe.



- **Class and Object Implementation with Java**

Java is a pure object-oriented programming language. In a Java class, instance variables define the attributes of the objects and methods define the operations that can be performed to manipulate the values of the instance variables. A Java class also provides a special type of methods, known as **constructors**, which can be invoked to create a new object. A constructor can perform any actions. However, constructors are mainly designed to perform initializing actions, such as initializing the instance variables of the new object.

It is a good software development practice to always define the instance variables as "private" to better protect the data. In this case, the data are hidden and are only "visible" within the class, meaning the data can be directly accessed within the class without the instantiation of an object. For example, the Circle class below is a template for creating circle objects with different radius values. Note that, if the Circle class doesn't have a "main" method, it cannot be run by itself. However, a "testbed main" can be created as a driver for the purpose of unit testing the Circle class. In other cases, the Circle class can be used by **a client class**, which may include a main method as the starting point of program execution.

```
// Circle class is a template for circle objects with different radius values.
public class Circle {
    private double radius; //data are private; no direct access from outside the class

    /** Default constructor; create a circle object with a default value */
    public Circle() {
        radius = 1.0; // set radius to the default value
    }

    /** Parameterized Constructor; create a circle object with a specified radius */
    public Circle(double radius) {
        this.radius = radius;
    }

    /** A getter method that returns the current value of the radius */
    public double getRadius() {
        return radius;
    }

    /** A setter method that sets the radius of a circle object to a given value. */
    public void setRadius(double radius) {
        this.radius = radius;
    }

    /** Compute and return the area of the circle object. */
    public double getArea() {
        return radius * radius * Math.PI;
    }

    /** Compute and return the perimeter of the circle object */
    public double getPerimeter() {
        return 2 * radius * Math.PI;
    }
    //This is called a testbed main, which is used as a driver to exercise the code in this class
    public static void main(String[] args) {
        Circle circle1 = new Circle(4.0);
        System.out.println("The area of circle1 with radius " + circle1.radius + " is "
              + circle1.getArea());
        Circle circle2 = new Circle(5.0);
        System.out.println("The area of circle2 with radius " + circle2.radius + " is "
              + circle2.getArea());
        Circle circle3 = new Circle(6.0);
        System.out.println("The area of circle3 with radius " + circle3.radius + " is "
              + circle3.getArea());
    }
}
```

Program output for test running the Circle class.

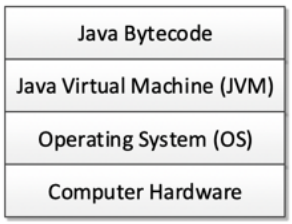
```
The area of circle1 with radius 4.0 is 50.26548245743669
The area of circle2 with radius 5.0 is 78.53981633974483
The area of circle3 with radius 6.0 is 113.09733552923255
```

When you run a Java program, the Java runtime system invokes the main method in the main class. You can put two classes into one file, but only one class in the file can be the "public" class. **The public class must have the same name as the file name**. For example, the Circle class above must be stored as **Circle.java** since the Circle class is "public".

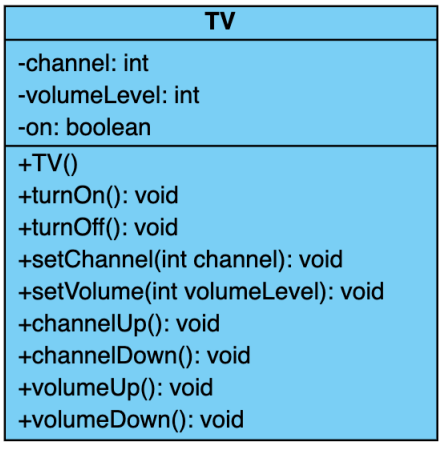
Each class in the source code (.java file) is compiled into a **.class** file. When you compile Circle.java, a Circle.class file is generated. Note that, Java uses a combination of a compiler and an interpreter. Java programs are first compiled into bytecode, which is interpreted and run by JVM (Java Virtual Machine. Java bytecode is portable and can be run on any platform running JVM.

C:\> javac myprogram.java

C:\> java myprogram



As another example, consider television sets. Each TV is an object with states (current channel, current volume level, and power on or off) and behaviors (change channels, adjust volume, and turn on/off). You can use a class to model TV sets. Depending on the needs of the software you are developing, you can also define a different set of attributes and associated operations for a TV object.



- **Constructors**

A constructor is invoked to create (instantiate) an object using the "new" operator. Constructors are a special kind of methods. They have three peculiarities:

1. A constructor must have the same name as the class itself.
2. Constructors do not have a return type—not even void.
3. Constructors are invoked using the "new" operator when an object is created. Constructors play the role of initializing objects.

A constructor has exactly the same name as its defining class. Like regular methods, constructors can be overloaded (i.e., multiple constructors can have the same name but different signatures), making it easy to construct objects with different initial data values. There are 3 kinds of constructors.

1. Default constructor – no-parameter constructors.
2. Parameterized constructor – a various numbers of parameters are defined.
3. Copy constructor – to clone an object; a single parameter with the class type is defined.

Since data are "private" and cannot be accessed directly from outside of the class, constructors are used to "construct" objects. To construct an object from a class, invoke a constructor of the class using the "new" operator as follows: **new ClassName(arguments);** for example, **new Circle()** creates an object of the Circle class using the first constructor (default constructor) defined in the Circle class, and **new Circle(25.0)** creates an object using the second constructor (parameterized constructor) defined in the Circle class. A class normally provides a default constructor (e.g., Circle()). A class may be defined without any constructors. In this case, a public default constructor with an empty body is implicitly defined. Note that, this constructor is provided automatically ONLY if not a single constructor has been explicitly defined in the class. For example, even if a class defines a parameterized constructor without defining a default constructor, Java will NOT generate a default constructor for the class.

- **Accessing Objects**

An object's data and methods can be accessed through the dot (.) operator via the object's reference variable. Newly created objects are allocated in the memory. They can be **accessed via reference variables**. Objects are accessed via the object's reference variables, which contain references to the objects. A class is essentially a programmer-defined type. A class is a reference type, which means that a variable of the class type can reference an instance of the class. You can write a single statement that combines the declaration of an object reference variable, the creation of an object, and the assigning of an object reference to the variable with the following syntax:

```
ClassName objectVariable = new ClassName();
```

An object reference variable that appears to hold an object actually contains a reference to that object. Strictly speaking, an object reference variable and an object are different, but most of the time the distinction can be ignored. Therefore, it is fine, for simplicity, to say that **myCircle** is a Circle object rather than use the long-winded description that **myCircle** is a variable that contains a reference to a Circle object.

In OOP terminology, an object's members refer to its instance variables (data) and methods. After an object is created, its data can be accessed and its methods can be invoked using the dot operator (.), also known as the **object members access operator**. For example, in the Circle class, the data field **radius** is referred to as an **instance variable** because it is dependent on a specific instance. For the same reason,

the method **getArea()** is referred to as an **instance method** because you can invoke it only on a specific instance. The object on which an instance method is invoked is called a **calling object**.

Recall that you use **Math.methodName(arguments)** (e.g., Math.pow(3, 2.5)) to invoke a method in the Math class. Can you invoke **getArea()** using **Circle.getArea()?** The answer is NO. All the methods in the Math class are **static methods**, which are defined using the "static" keyword. However, **getArea()** is an instance method, and thus non-static. It must be invoked from an object using objectVariable**.**methodName(arguments) (e.g., myCircle.getArea()).

Usually you create an object and assign it to a variable, then later you can use the variable to reference the object. Occasionally, an object does not need to be referenced later. In this case, you can create an object without explicitly assigning it to a variable using the syntax:

```
System.out.println("Area is " + new Circle(5.0).getArea());
```

The former statement creates a Circle object. The latter creates a Circle object and invokes its getArea method to return its area. An object created in this way is known as an **anonymous object**. The instance variables can be of reference types. For example, the following Student class contains a variable **name** of the String type. String is a predefined Java class.

```
public class Student {
    private String  name;           // name has the default value null
    private int     age;            // age has the default value 0
    private boolean isScienceMajor; // isScienceMajor has default value false
    private char    gender;         // gender has default value '\u0000'
}
```

If an instance variable of a reference type does not reference any object, the instance variable holds a special Java value, **null**, which is a literal just like **true** and **false**. While **true** and **false** are boolean literals, **null** is a literal for a reference type. The default value of an instance varialbe is **null** for a reference type, **0** for a numeric type, **false** for a boolean type, and **\u0000** for a char type. However, Java assigns no default value to a local variable defined within a method. The following code displays the default values of name, age, isScienceMajor, and gender of a Student object. However, not assigning values to local variables and trying to print the content of the variables will cause compile errors.

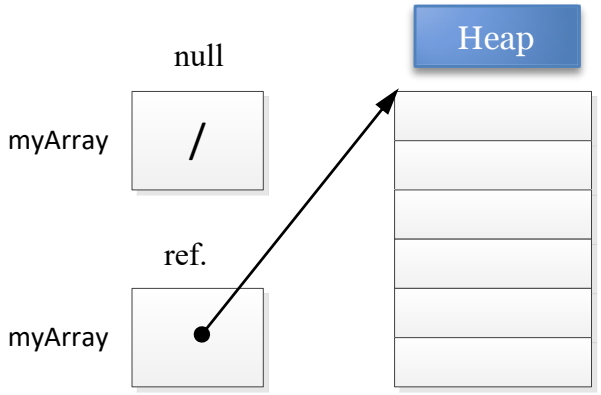
```
 1
 2 public class Student {
 3     private String name;             // name has the default value null
 4     private int age;                 // age has the default value 0
 5     private boolean isScienceMajor;  // isScienceMajor has default value false
 6     private char gender;             // gender has default value '\u0000'
 7
 8     public static void main(String[] args) {
 9         Student student = new Student();
10         System.out.println(student.name);
11         System.out.println(student.age);
12         System.out.println(student.isScienceMajor);
13         System.out.println(student.gender);
14     }
15 }
16
17
18
```

```
Problems  Javadoc  Declaration  Console
<terminated> Student [Java Application] /Library/Java/JavaVirtualMachines/jdk-14.0.1.jdk/Contents/Home/bin/java  (Jan 26, 2021, 4:09:39 PM – 4:09:41 PM)
null
0
false
```

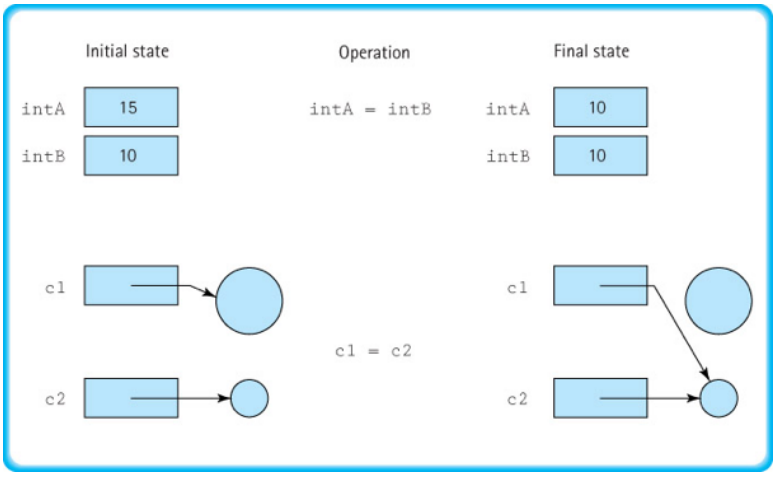
```
 7
 8    public static void main(String[] args) {
 9        int x;
10        String y;
11        System.out.println(x);
12        System.out.println(y);
13    }
14 }
15
16
```

NullPointerException is a common runtime error. It occurs when you invoke a method on a reference variable with a **null** value. Make sure you assign an object reference to the variable before invoking the method through the reference variable.

Every variable contains a reference, which represents a memory location that holds a value. When you declare a variable, you are telling the compiler what type of value the variable can hold. For a variable declared with a primitive type, the value is of the primitive type. For a variable declared with a reference type, the value is a reference to a memory location where the object is stored.



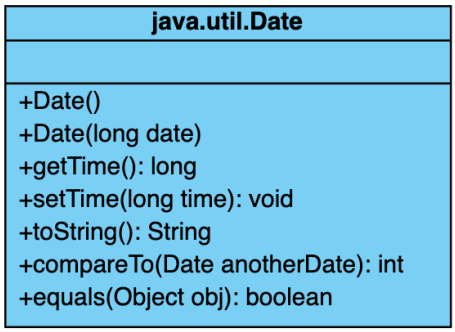
When you assign one variable to another, the other variable is set to the same value. For a variable of a primitive type, the real value of one variable is assigned to the other variable. For a variable of a reference type, the reference of one variable is assigned to the other variable.



For example, in the above figure, after the assignment statement **c1 = c2**, **c1** points to the same object referenced by **c2**. The object previously referenced by **c1** is no longer accessible and therefore is now known as **garbage**. Garbage occupies memory space, so the Java runtime system detects garbage and automatically reclaims the space it occupies. This process is called **garbage collection**.

- **Java library classes**

One of the benefits of using Java language is the well-established Java library classes. There are many existing Java classes that can be used to solve problems without writing new classes from scratch. For example, the **Date class** is commonly used by many software developers.

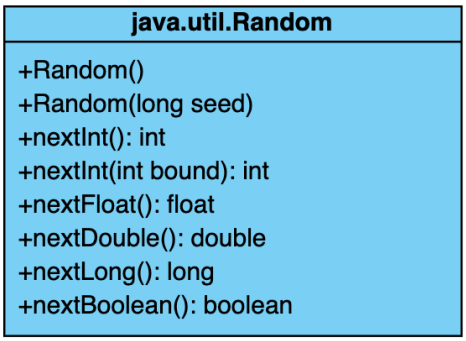


You can use the default constructor in the **Date** class to create an instance for the current date and time, the **getTime()** method to return the elapsed time in milliseconds since January 1, 1970, GMT, and the **toString()** method to return the date and time as a string. For example, the following code generates the output below.

```
java.util.Date date = new java.util.Date();
System.out.println("The elapsed time since Jan 1, 1970 is " + date.getTime() + " milliseconds");
System.out.println(date.toString());
```

```
The elapsed time since Jan 1, 1970 is 1596383892252 milliseconds
Sun Aug 02 11:58:12 EDT 2020
```

As another example, the **Random class** is commonly used by software developers to generate random numbers for different purposes.



When you create a Random object, you have to specify a seed or use the default seed. A seed is a number used to initialize a random number generator. The default constructor creates a Random object using the current elapsed time as its seed. If two Random objects have the same seed, they will generate identical

sequences of numbers. The ability to generate the same sequence of random values is useful in software testing and many other applications. In **software testing**, oftentimes you need to reproduce the test cases from a fixed sequence of random numbers.

You can generate random numbers using the **java.security.SecureRandom** class rather than the Random class. The random numbers generated from the Random are deterministic and they can be predicated by hackers. The random numbers generated from the SecureRandom class are nondeterministic and are more secure.

For a list of Java API library classes (version 17), please visit:
https://docs.oracle.com/en/java/javase/17/docs/api/index.html

- **The Object Class**

Class **Object** is the root of the Java class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class, especially the **equals()** and **toString()** method. We know that "**overloading**" is to define multiple methods with the same name, however with different signatures. "**Overriding**" is to change the method behaviors through the subclasses, however, keep the method signatures defined in the superclass unchanged.

Modifier and Type	Method
protected Object	clone()
boolean	equals(Object obj)
protected void	finalize()
Class<?>	getClass()
int	hashCode()
void	notify()
void	notifyAll()
String	toString()
void	wait()
void	wait(long timeoutMillis)
void	wait(long timeoutMillis, int nanos)

Since a Java class defines an ADT (abstract data type), which is NOT a primitive data type, you cannot compare 2 objects with the logical operator "==". We need to always include the **equals()** method in an entity class, such as a Stduent class, in order to compare the contents of the 2 Student objects and determine if they are "equal". If you change the signature of the equals() method, then you are "overloading", not "overriding". Below is a sample code for overriding the equals() method.

```
@Override //the tag to avoid the change of signature
public boolean equals(Object obj) {
    if (obj instanceof Student) {
        Student student = (Student) obj; //casting
        return student.name.equals(this.name);
    }
    return false;
}
```

Similarly, we always override the **toString()** method in the Object class to provide a textual representation of the object as each ADT has different data (instance variables.) If you don't override the toString() method, the toString() method of the Object class will be performed. As a result, it returns a string consisting of the name of the class of which the object is an instance, with the at-sign character `@', and the unsigned hexadecimal representation of the hash code of the object. In other words, this returns a string equal to the value of: getClass().getName() + '@' + Integer.toHexString(hashCode()). For example, the output for the following code looks something like Loan@15037e5. This message is not very helpful or informative. Usually you should override the toString() method so that it returns a descriptive string representation of the object.

- **Static variables, constants, and methods**

A static variable is shared by all objects of the class. A static method cannot access instance members (i.e., instance variables and methods) of the class.

The **radius** in the circle class is known as an instance variable. An instance variable is tied to a specific instance of the class; it is not shared among objects of the same class. If you want all the instances of a class to share data, use static variables, also known as **class variables**. Static variables store values for the variables in a common memory space. Because of this common location, if one object changes the value of a static variable, all objects of the same class are affected. Java supports static methods as well as static variables. Static methods can be called without creating an instance of the class. To declare a static variable or define a static method, add the modifier **static** to the variable or method declarations.

Constants in a class are shared by all objects of the class. Thus, constants should be declared as **final static.** For example, the constant PI in the Math class is defined as follows:

```
final static double PI = 3.14159;
```

The main method is static as well. Static variables and methods can be accessed without creating objects. Use **ClassName.methodName(arguments)** to invoke a static method and **ClassName.staticVariable** to access a static variable. This improves readability because this makes static methods and data easy to spot.

An instance method can invoke an instance or static method and access an instance or static variable. A static method can invoke a static method and access a static variable. However, a static method of a class cannot invoke an instance method or access an instance variable without creating an object, since instance methods and instance variable must be associated with a specific object. The relationship between static and instance members is summarized in the following table.

Static/or non-static	Invoke instance methods	Access instance variables	Invoke static methods	Access static variables
Instance methods	√	√	√	√
Static methods	X	X	√	√

How do you decide whether a variable or a method should be instance or static? A variable or a method that is dependent on a specific instance of the class should be an instance variable or method. A variable or a method that is not dependent on a specific instance of the class should be a static variable or method. For example, every circle has its own radius, so the radius is dependent on a specific circle object. Therefore, radius is an instance variable of the Circle class. Since the getArea() method is dependent on a circle object's radius value, it is also an instance method. None of the methods in the Math class, such as random, pow, sin, and cos, is dependent on a specific instance. Therefore, these methods are static methods. The main method of a class is static and can be invoked directly from a class. It is a common design error to define an instance method that should have been defined as static. For example, the method **factorial(int n)** should be defined as static, because it is independent of any specific instance.

- **Visibility Modifiers**

Visibility modifiers can be used to specify the visibility of a class and its members. You can use the "public" modifier for classes, methods, and instance variables to denote that they can be accessed from any other classes. If no visibility modifier is used, then by default the classes, methods, and instance variables are directly accessible by any class in the same package. This is known as package-private or package-access.

Packages are used to organize classes. To do so, you need to add the following line as the first non-comment and nonblank statement in the program.

```
package packageName;
```

If a class is defined without the package statement, it is said to be placed in the **default package**. Java recommends that you place classes into packages rather than using a default package.

In addition to the **public** and default visibility modifiers, Java provides the **private** and **protected** modifiers for class members. The **private** modifier makes methods and instance variables directly accessible only from within its own class. If a class is not defined as public, it can be accessed only within the same package. Using the modifiers **public** and **private** on local variables would cause a compile error.

Modifier	directly accessible within the class	directly accessible within the package	directly accessible within subclasses in the same package or a different package	directly accessible everywhere
public	√	√	√	√
protected	√	√	√	X
package	√	√	X	X
private	√	X	X	X

In most cases, constructors should be public. However, if you want to prohibit the client class from creating an instance of a class, define the constructor as private. In this case, private constructors are hidden from the external client classes and can only be invoked from within the class. For example, there is no reason to create an instance of the Java Math class, because it contains only static variables and static methods. To prevent the user from creating the objects of the Math class, the default constructor of the **java.lang.Math** is defined as follows.

```
private Math() {
}
```

- **Data Encapsulation**

Making instance variables private protects the data and makes the class easier to maintain.

1. Data may be tampered with if the data is made public where everyone has the direct access. This means the update of data is not well-controlled and it is difficult to trace the changes. For example, **numberOfObjects** is to count the number of objects created, but it may be mistakenly set to an arbitrary value (e.g., Circle.numberOfObjects = 10). The class becomes difficult to maintain and vulnerable to bugs.
2. Suppose that you want to modify the Circle class to ensure that the radius is nonnegative after other programs have already used the class. You have to change not only the Circle class but also the programs that use it because the client classes may have modified the radius directly.

To prevent direct modifications from other classes on the data, you should always declare the instance variables as private using the "private" modifier. This is known as **data encapsulation**.

A private instance variable cannot be accessed by an object from outside the class. However, a client class outside the class often needs to retrieve and modify the data contained in the private instance variable. To make a private data accessible, provide a "getter" method to return its value. To enable a private data to be updated, provide a "setter" method to set a new value. A getter method is also referred to as an **accessor** and a setter method to a **mutator**.

- **Passing Objects to Methods**

Passing an object to a method is to pass the reference of the object. You can pass objects to methods. Like passing an array, passing an object is actually **passing the reference of the object**. For example, The following code passes the **circle** object as an argument to the **printCircle()** method:

```
public void printCircle(int times, Circle c) { }

public static void main(String[] args) {
    int n = 5;
    Circle circle = new Circle(4.0);
    circle.printCircle(n, circle);
    …
}
```

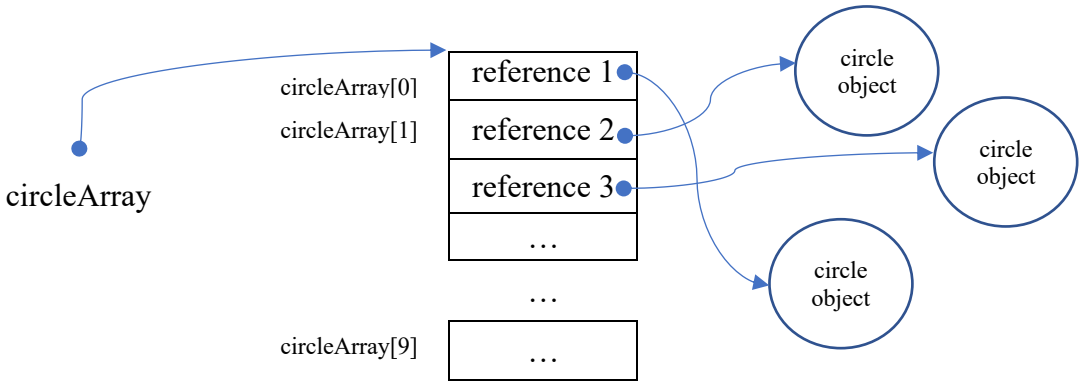
Pass by reference

Pass by value

Pass-by-value refers to the situation where the method call is passing the value of an argument of a primitive data type. In the above example, the value of *n* (i.e., 5) is passed to the parameter **times**. In the **printCircle()** method, if the content of the variable **times** is changed, the value of *n* in the main method remains unchanged. When passing an argument of a reference type, the reference of the object is passed. In this case, *c* contains the same reference to the **circle** object. Therefore, changing the data values of the **circle** object through *c* in the **printCircle()** method has the same effect as doing so outside the method through the variable "circle". Pass-by-reference can be best described semantically as **pass-by-sharing**; that is, the object referenced in the method is the same as the object being passed.

- **Array of Objects**

An array of objects is actually an array of references. Thus, invoking circleArray[1].getArea() involves two levels of referencing, as shown below. `Circle[] circleArray = new Circle[10];` declares an array of Circle objects with a size of 10. The **circleArray** contains a reference to the beginning address of a consecutive memory block that is allocated for storing 10 object references of the Circle class. Each array element stores a reference to an instance of Circle class. For example, **circleArray[1]** references the second element of the array, which is a reference to a Circle object. Similarly, **circleArray[1].getArea()** invokes the method of the second Circle object in the array. Note that, an array occupies a block of consecutive memory addresses; however, the memory addresses that are storing the actual circle objects are not necessarily consecutive. When an array of objects is created using the **new** operator, each element in the array is a reference variable with a default value of **null**.



- **Immutable Objects and Classes**

Normally, you create an object and allow its contents to be changed later. However, occasionally it is desirable to create an object whose contents cannot be changed once the object has been created. We call such an object as **immutable object** and its class as **immutable class**. The String class, for example, is immutable. If you deleted the setter method in the Circle class, the class would be immutable because radius is private and cannot be changed without a setter method.

If a class is immutable, then all its instance variables must be declared as private, and the class cannot contain any public setter methods for the instance variables. Note that, a class with all the instance variables declared as private and contains no mutators is not necessarily immutable. For example, the **Employee** class below contains only private data and has no setter methods, but it is NOT an immutable class. The variable **hired** is a reference type and it is returned in the **getDateHired()** method. The variable **hired** contains a reference to a Date object. Through this reference, the content for hired can be changed by an external object. Therefore, to make the Employee class an immutable class, one can convert the Date object to a string or create a clone object, and then return the string or new object.

```
public class Employee {
    private int id;
    private String name;
    private java.util.Date hired;

    public Employee (int id, String name) {
        this.id = id;
         this.name = name;
         hired = new java.util.Date();
    }

    public int getId() {
        return id;
    }

    public String getName() {
         return name;
    }

    //this method returns the reference to the calling method, thus is mutable.
    public java.util.Date getDateHired() {
         return hired;
    }
}
```

For a class to be immutable, it must meet the following requirements:

1. All instance variables must be declared as private.
2. Cannot contains any mutator methods for changing the data contained in the instance variables.
3. No accessor methods can return an instance variable of reference type that contains the reference to a mutable object.

- **Scope of Variables**

Instance and static variables defined in a class are referred to as the **class variables**. Variables defined inside any methods are referred to as **local variables**. The scope of class variables is the entire class, regardless of where the variables are declared. Class variables and methods can appear in any order in the class. The exception is when an instance variable is initialized based on a reference to another instance variable. In such cases, the other instance variable must be declared first. For example, the getArea() method in the Circle class below can be declared before radius; however, the integer i must be declared before the integer j, as the value of j depends on integer i.

```
public class Circle {
    public double getArea() {
        return radius * radius * Math.PI;
    }
    private double radius = 1.0;
}
public class Foo {
    private int i = 1;
    private int j = i + 1;
}
```

You can declare a class variable only once, but you can use the same variable name in a method many times within different non-nesting blocks. If a local variable has the same name as a class variable, the local variable takes precedence and the class variable with the same name is hidden. For example, in the

code segment below, *x* is defined both as an instance variable and as a local variable in the method. However, to avoid confusion and possible bugs, DO NOT use the same names for class variables and local variables, EXCEPT for method parameters.

```
public class Foo {
    private int x = 0; //an instance variable
    private int y = 0;

    public Foo() {
    }

    public void myMethod() {
        int x = 1; //a local variable
        System.out.println("x = " + x); //reference to the local variable x
        System.out.println("y = " + y);
    }
}
```

- **The Keyword this**

The keyword **this** contains the reference to the object itself. It can also be used inside a constructor to invoke another constructor of the same class. The **this** keyword is the name of a reference that an object can use to refer to itself. You can use the **this** keyword to reference the object's instance members. For example, the **this** reference is omitted for brevity in the following code. However, the **this** reference is needed to reference the data hidden by a method or constructor parameter, or to invoke an overloaded constructor.

```
public double getArea() {
    return radius * radius * Math.PI;
    // the above statement is equivalent to
    // return this.radius * this.radius * Math.PI;
    }
```

It is a good practice to use the name of an instance variable as the parameter name in a setter method or a constructor to make the code easy to read and to avoid creating unnecessary names. In this case, you need to use the **this** keyword to reference the instance variable in the setter method. For example, the setRadius() method below use the same variable names for the instance variable and the local variable defined as a parameter. It would be wrong if the statement is written as `radius = radius;`

```
public void setRadius(double radius) {
    this.radius = radius;
}
```

local variable

instance variable

The **this** keyword can be used to invoke another constructor of the same class. For example, you can rewrite the default constructor of the **Circle** class as follows. The default contractor invokes the

parameterized constructor to initialize the radius. Note that, Java requires that the **this(arg-list)** statement appear first in the constructor before any other executable statements.

```
public Circle(double radius) {
    this.radius = radius;
}

public Circle() {
    this(1.0); // call the above constructor
}
```

If a class has multiple constructors, it is better to implement them using **this(arg-list)** as much as possible. In general, a constructor with no or fewer arguments can invoke a constructor with more arguments using this(arg-list). This syntax often simplifies coding and makes the class easier to read and to maintain.

- **Wrapper Classes**

A primitive-type value is not an object, but it can be wrapped in an object using a wrapper class in the Java API. Owing to performance considerations, primitive data type values are not objects in Java. Because of the overhead of processing objects, the language's performance would be adversely affected if primitive data type values were treated as objects. However, many Java methods require the use of objects as arguments. Java offers a convenient way to incorporate, or wrap, a primitive data type value into an object (e.g., wrapping an int into an Integer object, wrapping a double into a Double object, and wrapping a char into a Character object). By using a wrapper class, you can process primitive data type values as objects. Java provides Boolean, Character, Double, Float, Byte, Short, Integer, and Long wrapper classes in the **java.lang** package for primitive data types. The Boolean class wraps a Boolean value true or false. This section uses Integer and Double as examples to introduce the numeric wrapper classes. Most wrapper class names for a primitive type are the same as the primitive data type name with the first letter capitalized. The exceptions are Integer for int and Character for char. The instances of all **wrapper classes are immutable**; this means that, once the objects are created, their internal values cannot be changed. Numeric wrapper classes are very similar to each other. Each contains the methods doubleValue(), floatValue(), intValue(), longValue(), shortValue(), and byteValue(). These methods "convert" objects into primitive-type values. The key features of Integer and Double are shown below.

  - **java.lang.Double**

Fields

Modifier and Type	Field	Description
static int	BYTES	The number of bytes used to represent a double value.
static int	MAX_EXPONENT	Maximum exponent a finite double variable may have.
static double	MAX_VALUE	A constant holding the largest positive finite value of type double, $(2-2^{-52})\cdot 2^{1023}$.
static int	MIN_EXPONENT	Minimum exponent a normalized double variable may have.
static double	MIN_NORMAL	A constant holding the smallest positive normal value of type double, 2^{-1022}.
static double	MIN_VALUE	A constant holding the smallest positive nonzero value of type double, 2^{-1074}.
static double	NaN	A constant holding a Not-a-Number (NaN) value of type double.
static double	NEGATIVE_INFINITY	A constant holding the negative infinity of type double.
static double	POSITIVE_INFINITY	A constant holding the positive infinity of type double.
static int	SIZE	The number of bits used to represent a double value.
static Class<Double>	TYPE	The Class instance representing the primitive type double.

Modifier and Type	Method	Description
byte	byteValue()	Returns the value of this Double as a byte after a narrowing primitive conversion.
int	compareTo(Double anotherDouble)	Compares two Double objects numerically.
Optional<Double>	describeConstable()	Returns an Optional containing the nominal descriptor for this instance, which is the instance itself.
double	doubleValue()	Returns the double value of this Double object.
boolean	equals(Object obj)	Compares this object against the specified object.
float	floatValue()	Returns the value of this Double as a float after a narrowing primitive conversion.
int	hashCode()	Returns a hash code for this Double object.
int	intValue()	Returns the value of this Double as an int after a narrowing primitive conversion.
boolean	isInfinite()	Returns true if this Double value is infinitely large in magnitude, false otherwise.
boolean	isNaN()	Returns true if this Double value is a Not-a-Number (NaN), false otherwise.
long	longValue()	Returns the value of this Double as a long after a narrowing primitive conversion.
Double	resolveConstantDesc(MethodHandles.Lookup lookup)	Resolves this instance as a ConstantDesc, the result of which is the instance itself.
short	shortValue()	Returns the value of this Double as a short after a narrowing primitive conversion.
String	toString()	Returns a string representation of this Double object.

- **java.lang.Integer**

Fields

Modifier and Type	Field	Description
static int	BYTES	The number of bytes used to represent an int value in two's complement binary form.
static int	MAX_VALUE	A constant holding the maximum value an int can have, $2^{31}-1$.
static int	MIN_VALUE	A constant holding the minimum value an int can have, -2^{31}.
static int	SIZE	The number of bits used to represent an int value in two's complement binary form.
static Class<Integer>	TYPE	The Class instance representing the primitive type int.

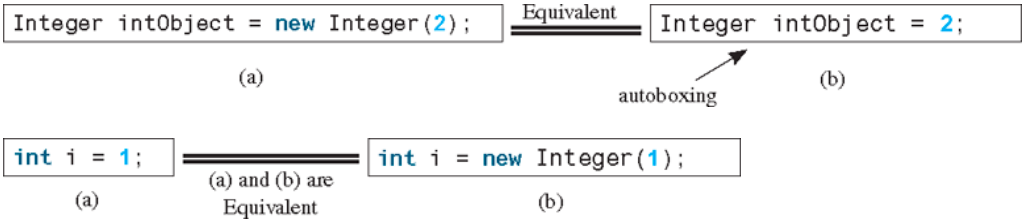
Modifier and Type	Method	Description
byte	byteValue()	Returns the value of this Integer as a byte after a narrowing primitive conversion.
int	compareTo(Integer anotherInteger)	Compares two Integer objects numerically.
Optional<Integer>	describeConstable()	Returns an Optional containing the nominal descriptor for this instance, which is the instance itself.
double	doubleValue()	Returns the value of this Integer as a double after a widening primitive conversion.
boolean	equals(Object obj)	Compares this object to the specified object.
float	floatValue()	Returns the value of this Integer as a float after a widening primitive conversion.
int	hashCode()	Returns a hash code for this Integer.
int	intValue()	Returns the value of this Integer as an int.
long	longValue()	Returns the value of this Integer as a long after a widening primitive conversion.
Integer	resolveConstantDesc(MethodHandles.Lookup lookup)	Resolves this instance as a ConstantDesc, the result of which is the instance itself.
short	shortValue()	Returns the value of this Integer as a short after a narrowing primitive conversion.
String	toString()	Returns a String object representing this Integer's value.

Each numeric wrapper class has the constants MAX_VALUE and MIN_VALUE. MAX_VALUE represents the maximum value of the corresponding primitive data type. For Byte, Short, Integer, and Long, MIN_VALUE represents the minimum byte, short, int, and long values. Float and Double, MIN_VALUE represents the minimum positive float and double values. The numeric wrapper classes contain the **compareTo** method for comparing two numbers and returns **1**, **0**, or **–1**, if this number is greater than, equal to, or less than the other number.

The numeric wrapper classes have a useful static method, **valueOf(String s)**. This method creates a new object initialized to the value represented by the specified string. Each numeric wrapper class has two

overloaded parsing methods to parse a numeric string into an appropriate numeric value based on 10 (decimal) or any specified radix (e.g., 2 for binary, 8 for octal, and 16 for hexadecimal).

A primitive-type value can be automatically converted to an object using a wrapper class, and vice versa, depending on the context. Converting a primitive value to a wrapper object is called **boxing.** The reverse conversion is called **unboxing**. Java allows primitive types and wrapper classes to be converted automatically. The compiler will automatically box a primitive value that appears in a context requiring an object and unbox an object that appears in a context requiring a primitive value. This is called **autoboxing** and **autounboxing.**



There are BigInteger and BigDecimal classes can be used to represent integers or decimal numbers of any size and precision. If you need to compute with very large integers or high-precision floating-point values, you can use the BigInteger and BigDecimal classes in the **java.math** package. **Both are immutable**. The largest integer of the long type is Long.MAX_VALUE (i.e., 9223372036854775807). An instance of BigInteger can represent an integer of any size. You can use new BigInteger(String) and new BigDecimal(String) to create an instance of BigInteger and BigDecimal, use the add, subtract, multiply, divide, and remainder methods to perform arithmetic operations, and use the compareTo method to compare two big numbers.

- **String class**

A String object is immutable; its contents cannot be changed once the string is created.

```
String newString = new String(stringLiteral);
```

The argument **stringLiteral** is a sequence of characters enclosed in double quotes. The following statement creates a **String** object **message** for the string literal **"Welcome to Java"**:

```
String message = new String("Welcome to Java");
```

Java treats a string literal as a **String** object. Thus, the following statement is valid:

```
String message = "Welcome to Java";
```

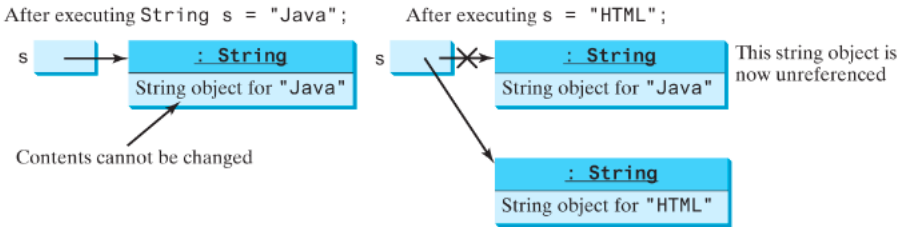
You can also create a string from an array of characters. For example, the following statements create the string **"Good Day"**:

```
char[] charArray = {'G', 'o', 'o', 'd', ' ', 'D', 'a', 'y'};
String message = new String(charArray);
```

A String variable holds a reference to a String object that stores a string value. Strictly speaking, the terms String variable, String object, and string value are different, but most of the time the distinctions between them can be ignored. For simplicity, the term string will often be used to refer to String variable, String object, and string value. **A String object is immutable**; its contents cannot be changed. Does the following code change the contents of the string?

```
String s = "Java";
s = "HTML";
```

The answer is NO. The first statement creates a String object with the content "Java" and assigns its reference to s. The second statement creates a new String object with the content "HTML" and assigns its reference to s. The first String object still exists after the assignment, but it can no longer be accessed, because variable s now points to the new object, as shown below.



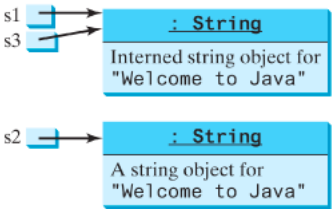
Because strings are immutable and are ubiquitous in programming, the JVM uses a unique instance for string literals with the same character sequence in order to improve efficiency and save memory. Such an instance is called an **interned string**. For example, the following statements:

```
String s1 = "Welcome to Java";

String s2 = new String("Welcome to Java");

String s3 = "Welcome to Java";

System.out.println("s1 == s2 is " + (s1 == s2));
System.out.println("s1 == s3 is " + (s1 == s3));
```




```
s1 == s2 is false
s1 == s3 is true
```

In the preceding statements, s1 and s3 refer to the same interned string—"Welcome to Java"—so s1 == s3 is true. However, s1 == s2 is false, because s1 and s2 are two different string objects, even though they have the same contents. Strings are not arrays, but a string can be converted into an array and vice versa. To convert a string into an array of characters, use the **toCharArray** method. Another way of converting a number into a string is to use the overloaded static **valueOf** method. This method can also be used to convert a character or an array of characters into a string.

static String	valueOf(boolean b)	Returns the string representation of the boolean argument.
static String	valueOf(char c)	Returns the string representation of the char argument.
static String	valueOf(char[] data)	Returns the string representation of the char array argument.
static String	valueOf(char[] data, int offset, int count)	Returns the string representation of a specific subarray of the char array argument.
static String	valueOf(double d)	Returns the string representation of the double argument.
static String	valueOf(float f)	Returns the string representation of the float argument.
static String	valueOf(int i)	Returns the string representation of the int argument.
static String	valueOf(long l)	Returns the string representation of the long argument.
static String	valueOf(Object obj)	Returns the string representation of the Object argument.

The String class contains the static format method to return a formatted string. This method is similar to the printf method except that the format method returns a formatted string, whereas the **printf** method displays a formatted string.

```
String s = String.format("%7.2f%6d%-4s", 45.556, 14, "AB");
System.out.println(s);
```

console output: `--45.56----14AB--`

The StringBuilder and StringBuffer classes are similar to the String class except that the String class is immutable. In general, the StringBuilder and StringBuffer classes can be used wherever a string is used. StringBuilder and StringBuffer are more flexible than String. You can add, insert, or append new contents into StringBuilder and StringBuffer objects, whereas the value of a String object is fixed once the string is created. The StringBuilder class is similar to StringBuffer except that the methods for modifying the buffer in StringBuffer are synchronized, which means that only one task is allowed to execute the methods. Use StringBuffer if the class might be accessed by multiple tasks concurrently, because synchronization is needed in this case to prevent corruptions to StringBuffer. Using StringBuilder is more efficient if it is accessed by just a single task, because no synchronization is needed in this case. The constructors and methods in StringBuffer and StringBuilder are almost the same. You can replace StringBuilder in all occurrences in this section by StringBuffer. The program can compile and run without any other changes. For more information, visit https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/StringBuilder.html

The **StringTokenizer class** allows an application to break a string into tokens. The tokenization method is much simpler than the one used by the StreamTokenizer class. The StringTokenizer methods do not distinguish among identifiers, numbers, and quoted strings, nor do they recognize and skip comments. The set of delimiters (the characters that separate tokens) may be specified either at creation time or on a per-token basis.

A StringTokenizer object internally maintains a current position within the string to be tokenized. A token is returned by taking a substring of the string that was used to create the StringTokenizer object. The following is one example of the use of the tokenizer. The code:

```
StringTokenizer st = new StringTokenizer("this is a test");
while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

prints the following output:

```
this
is
a
test
```

StringTokenizer is a legacy class that is retained for compatibility reasons although its use is discouraged in new code. It is recommended that anyone seeking this functionality use the split method of String or the java.util.regex package instead. The following example illustrates how the String.split method can be used to break up a string into its basic tokens and generate the same output as the above example. **Note:** `"\\s"` delimiter is a single space, where `"\\s+"` delimiter is one or more spaces.

```
String[] result = "this is a test".split("\\s");
for (int x = 0; x < result.length; x++)
    System.out.println(result[x]);
```

- **enum class**

An enum class is a special data type that enables for a variable to be a set of **predefined constants**. The variable must be equal to one of the values that have been predefined for it. Common examples include compass directions (values of NORTH, SOUTH, EAST, and WEST) and the days of the week. Because they are constants, the names of an enum type's data fields are in **uppercase letters**.

You should use enum types any time you need to represent a fixed set of constants. That includes natural enum types such as the planets in our solar system and data sets where you know all possible values at compile time—for example, the choices on a menu, command line flags, and so on.

In the Java programming language, you define an enum class by using the enum keyword. For example, you would specify a days-of-the-week enum class as below and use the constants for coding.

```
//define an enum class
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY;
}
switch (example.day) { //use the constants in a switch case statement
    case MONDAY:
        System.out.println("Mondays are bad.");
        break;
    case FRIDAY:
        System.out.println("Fridays are better.");
        break;
    case SATURDAY:
    case SUNDAY:
        System.out.println("Weekends are best.");
        break;
    default:
        System.out.println("Midweek days are so-so.");
        break;
}
```

Java enum types are much more powerful than their counterparts in other languages. The enum class body can include methods and other data fields. Java automatically adds some special methods when it creates an enum. For example, the static **values()** method returns an array containing all of the values of the constants defined in the enum class, in the order they are declared. This method is commonly used in combination with the for-each construct to iterate over the values of an enum type. For example, the code below will display: SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY,

```
for (Day day : Day.values()) {
    System.out.print(day.toString() + ", ");
}
```

An enum class can also define constants with additional properties. For example, Planet is an enum type that represents the planets in the solar system. They are defined with constant mass and radius properties. Each planet is declared with values for the mass and radius parameters. These values are passed to the constructor when the constant is created. **Java requires that the constants be defined first**, prior to any fields or methods. Also, when there are fields and methods, the list of enum constants must end with a semicolon. The constructor for an enum type must be package-private or private access. It

automatically creates the constants that are defined at the beginning of the enum body. **You cannot invoke an enum constructor yourself.**

```
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS   (4.869e+24, 6.0518e6),
    EARTH   (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27,   7.1492e7),
    SATURN  (5.688e+26, 6.0268e7),
    URANUS  (8.686e+25, 2.5559e7),
    NEPTUNE (1.024e+26, 2.4746e7); //end with semicolon

   private final double mass;   // in kilograms
    private final double radius; // in meters

    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }

    double surfaceGravity() { }
    double surfaceWeight(double otherMass) { }
    ...
}
```

Enum class Javadoc: https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Enum.html

- **Scanner class**

Scanner class is a simple **text scanner** which can parse primitive types and strings using regular expressions. A Scanner breaks its input into tokens using a delimiter pattern, which by default matches whitespace. The resulting tokens may then be converted into values of different types using the various next methods. The default whitespace delimiter used by a scanner is as recognized by **Character.isWhitespace(),** such `as \n, \t, \s, \r,` etc. A scanning operation may block waiting for input. The next() and hasNext() methods and their companion methods (such as nextInt() and hasNextInt()) first skip any input that matches the delimiter pattern, and then attempt to return the next token. Both hasNext() and next() methods may block waiting for further input. Whether a hasNext() method blocks has no connection to whether or not its associated next() method will block. The tokens() method may also block waiting for input.

For example, this code allows a user to read a number from **System.in**, which stands for the standard input, such as the IDE's console.

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
```

The scanner can also use delimiters other than whitespace. This example reads several items in from a string:

```
String input = "1 fish 2 fish red fish blue fish";
Scanner s = new Scanner(input).useDelimiter("\\s*fish\\s*");
```

```
System.out.println(s.nextInt());
System.out.println(s.nextInt());
System.out.println(s.next());
System.out.println(s.next());
s.close();
```

prints the following output:

```
1
2
red
blue
```

Scanner class Javadoc:
https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Scanner.html