Note #2 – Object Oriented Programming with Java

Dr. Lily Chang

1. Object-Oriented Programming

Having learned Java language in previous courses, you are able to solve many computer-solvable problems using selections, loops, methods, and arrays. However, these Java features are insufficient for developing large-scale software systems with graphical user interfaces (GUIs.) Object-oriented programming (OOP) is one of the software methodologies that enables you to develop large-scale software and GUIs effectively. Important OO features include,

- ✓ Data encapsulation. A Java class encapsulates the data (private) and operations (public methods) and provides indirect access to the private data, which will better protect the data.
- ✓ Information hiding. Decouple the dependency between the software specification (Application Programming Interface - APIs) and implementation (coding). As a result, a change of implementation will not impact how the clients use/call the APIs.
- ✓ Readability. Mapping class objects to real-world entities would make it easier to comprehend the logic of the code. For example, a student object encapsulates the information about a student entity, which can directly map to a student entity in the real world.
- ✓ Extendibility. Java inheritance can reuse existing code and extend new features without the impact on existing code.
- ✓ Reusability. Java inheritance can generalize code for reuse to remove redundancy.
- ✓ Maintainability. Java inheritance supports polymorphism, a.k.a. runtime binding. This supports the changes of existing functionalities without the impact on existing code.

Any change of existing code could break the software system and required software testing at all levels. The OO features are intended to minimize the impact/cost of future software changes.

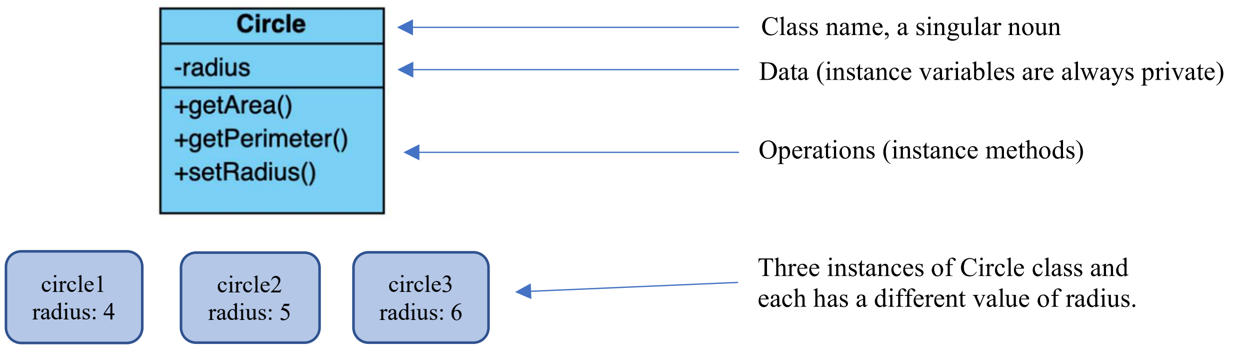
2. Class and object

A **Class** defines the data and operations for a group of similar 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. An object encapsulates data and operations, 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 data of a student object, and “calculate tuition” could be considered as an operation (method) to calculate the tuition.

The **data** encapsulated in an object is also known as “states”, “properties”, or “attributes” in object-oriented design. They are always defined as private data fields when defining a Java class. For example, a circle object may have a data field “radius”, which is a property that characterizes a circle. A rectangle object may have the data fields “width and “height”, which are the properties that characterize a rectangle. With these properties, a circle object and a rectangle object can be drawn on the computer screen, or the areas of the objects can be calculated.

The **operation** encapsulated in an object is also known as “behaviors” in object-oriented design. They are defined as public methods when defining a Java class to provide indirect manipulation of the private data of an object. To invoke a method on an object is to perform an operation on the private data, which typically involves getting or changing the current values (states) of the data fields. For example, you may define the getter methods **getArea()** and **getPerimeter()**. A circle object may invoke **getArea()** to return the its area and **getPerimeter()** to return its perimeter. You may also define a setter method **setRadius (radius)** that changes the value of the radius.

A Class is an **Abstract Data Type (ADT)**, which is a template, blueprint, or contract that defines what data fields and methods will be encapsulated in an object. An object is an instance of a class. You can create many instances of a class and every instance has its own "state" when running the Java programs. Creating an instance is referred to as **instantiation**, which involves memory allocation in in the dynamic memory area at runtime. 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 wanted from the same recipe.


Circle
-radius
+getArea()
+getPerimeter()
+setRadius()
Class name, a singular noun
Data (instance variables are always private)
Operations (instance methods)
circle1 radius: 4
circle2 radius: 5
circle3 radius: 6
Three instances of Circle class and each has a different value of radius.

3. Class and Object in Java

Java is a pure object-oriented programming language. In a Java class, instance variables define the data 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 values of the instance variables. It is a good 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 Java class, meaning the data can only be directly accessed within the class without creating 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 does not have a "main" method, it cannot run by itself. However, a "testbed main" can be created as a driver for the purpose of unit testing the Circle class. A "testbed main" is simply the main method defined within the Java class with the method signature: **`public static void main (String args[]).`** Every Java class can have a main() method to unit test the public methods defined within the class. The public methods can then be provided as the "API" to be used by other Java classes (clients.)

A software system developed in Java may contain many Java classes. When you want to run the software, you must invoke the main() method defined in the main Java class of the software. You can put two Java classes into a single Java source file, but only one class in the source file can be the "public" class. In addition, **the public class must have the same name as the name of the source file**. For example, the Circle class below must be stored as **Circle.java** since the Circle class is "public".

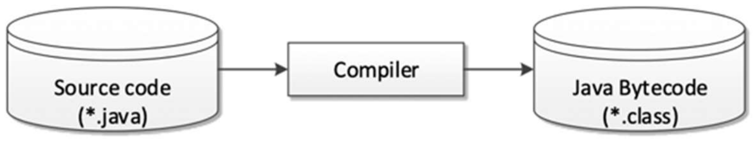
```java
// This class is a template for circle objects with different radius values.
public class Circle {
    private double radius; //data are private; no direct access from outside the class

    /** Default constructor (no-parameters); 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 given radius */
    public Circle(double radius) {
        this.radius = radius;
    }

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

    /** A setter method that sets the radius of the object to a given radius. */
    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;
    }
    //Testbed main, a driver to exercise and test the public methods defined within 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.

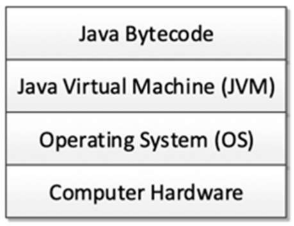
```
The area of circle1 with radius 4.0 is 50.26548245743669
The area of circle2 with radius 5.0 is 78.53981633974483
The area of circle3 with radius 6.0 is 113.09733552923255
```

Each Java class in the source file (*.java) is compiled into a **.class** file. For example, when you compile Circle.java, a Circle.class file is generated. Note that Java uses a combination of compiler and interpreter. Java programs are first compiled into bytecode, which is portable and runnable under any operating system with the Java Virtual Machine (JVM) installed. When you run the bytecode, JVM runs the bytecode with an interpreter.

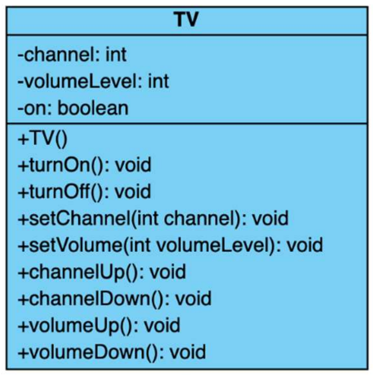
```
C:\> javac  myprogram.java
```



```
C:\> java  myprogram
```



As another example, if a software system is required to model the behaviors of a TV, we can create a TV class containing the necessary states and operations. Data may include an integer channel number, the volume level, and power on or off. The operations then involve changing the states of the TV, including changing the channel number, adjusting the volume, and switching the power on or off. Depending on the requirements of the software system, a different set of data fields and their associated operations are possible. For example, if TVs are a product of a retailer, then we might need to define a set of data fields such as, product number, product name, model number, serial number, quantity in stock, unit price, etc.



4. Constructors

Since data are "private" and cannot be accessed directly from outside of the class, constructors are used to "construct" objects. A constructor is invoked to create (instantiate) an object using the "new" keyword in Java. Constructors are a special kind of method. They have three peculiarities:

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

Like the regular methods, constructors can be overloaded, i.e., multiple constructors having the same name but different numbers/types of parameters. Overloading constructors makes it easy to create objects with different initial data values and types. There are 3 types of constructors.

(a) Default constructor – also known as no-parameter constructors.
(b) Parameterized constructor – various numbers/types of parameters are defined.
(c) Copy constructor – to clone an object (deep copying) with a single parameter in the class type.

Invoking a constructor to create an object, simply use the "new" keyword. For example,

```
Circle circle = new Circle();
```

creates an object of the Circle class. Since there is no argument provided, the no-parameter constructor (default constructor) will be invoked. On the other hand, if `new Circle(25.0)` is being used, then the other matching constructor with the same number of parameters and types will be invoked.

A good practice is to always define a default constructor. Because a Java class may be defined without any constructors, in this case, a public default constructor with an empty body will be implicitly defined by Java runtime. Note that, Java generated default constructor is provided automatically ONLY if not a single constructor has been explicitly defined in the class. Therefore, if a Java class defines a parameterized constructor without defining a default constructor, Java will NOT generate a default constructor for the class.

5. Accessing Object Members

An object's data and methods can be accessed through the dot (.) operator via the object's reference variable. Memory spaces are allocated for newly created objects. They can be **accessed via reference variables**, which contain the memory addresses of the objects. A Java class is essentially a programmer-defined data type, and a class is also called a reference type. You can write a single statement that combines the declaration of a reference variable, the creation of an object, and the assignment of the memory address to the reference variable. For example, the statement below declares a reference variable "student" in the Student type and creates a new object with the default constructor of the Student class. The "**new**" keyword means memory allocation for the new object and the memory address is assigned to the reference variable "student".

```
Student student = new Student();
```

Strictly speaking, a reference variable and an object are different, but the distinction can often be ignored. Therefore, it is fine, for simplicity, to say that the **student** is a Student object rather than use the long-winded description that the **student** is a variable that contains a reference to a Student 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. Instance methods must be invoked from an object using the dot operator, for example, **myCircle.getArea()**.

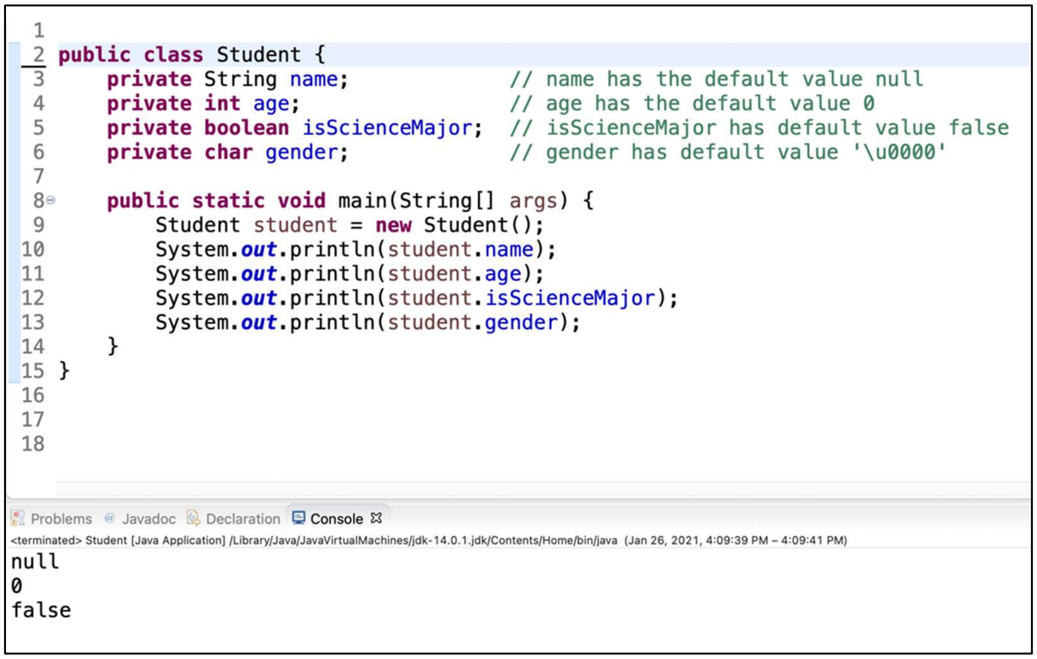
Usually, you create an object and assign it to a variable; then later, you can use the variable to reference the object. Occasionally, an object does not need to be referenced later. In this case, you can create an object without explicitly assigning it to a variable using the syntax. For example, the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

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

Instance variables can be of reference types. For example, the Student class below contains a variable **name** of the String type. String is a predefined Java class.

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

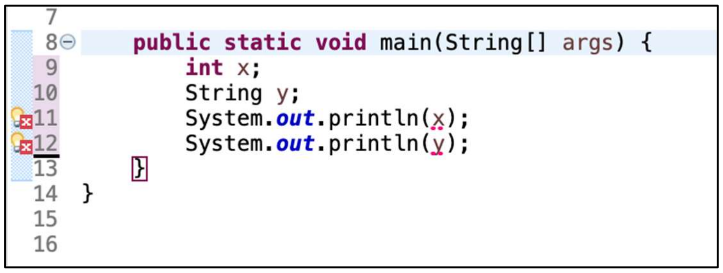
If an instance variable of a reference type does not reference any object, the instance variable holds a special Java value, **null**, which is a literal, just like **true** and **false**. While **true** and **false** are boolean literals, **null** is a literal for a reference type. The default value of an instance variable is **null** for a reference type, **0** for a numeric type, **false** for a boolean type, and **\u0000** for a char type. Note that Java assigns no default value to a local variable defined within a method.



```java

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'

    public static void main(String[] args) {
        Student student = new Student();
        System.out.println(student.name);
        System.out.println(student.age);
        System.out.println(student.isScienceMajor);
        System.out.println(student.gender);
    }
}
```

```
<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
```

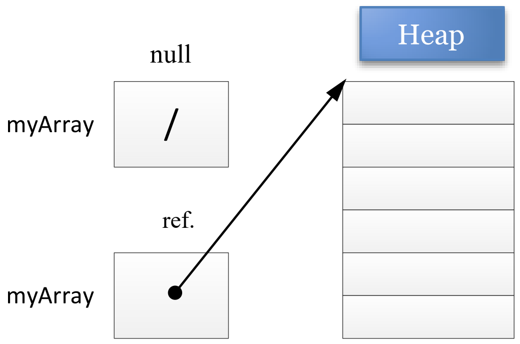
The code above displays the default values of name, age, isScienceMajor, and gender of a Student object. However, not assigning values to local variables and trying to print the content of the variables will cause compile errors as shown below.



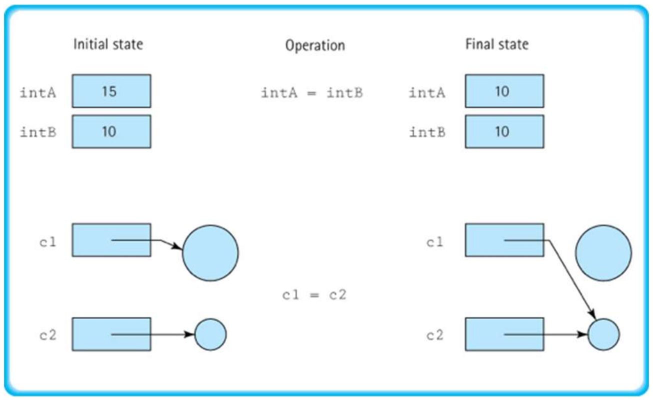
```
    public static void main(String[] args) {
        int x;
        String y;
        System.out.println(x);
        System.out.println(y);
    }
}
```

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 declared is a reference, which is a memory address containing a value in declared type. 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.



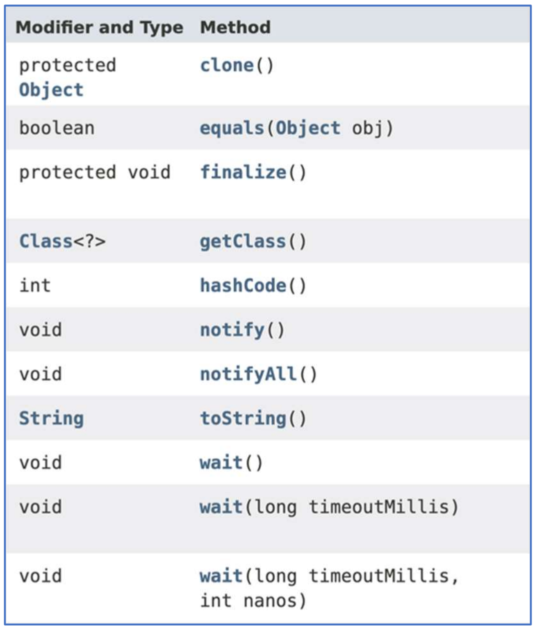
For example, in the left figure, after the assignment statement **c1 = c2; c1** points to the same object referenced by **c2**. The object previously referenced by **c1** is no longer accessible and, therefore, is now known as **garbage**. Garbage occupies memory space, so the Java runtime detects garbage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

6. 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. This greatly reduces the time for software development. When you have the Java Runtime (JRE) installed, your computer has the JVM and Standard Java Library packages ready for you to use. Under the existing Java library class hierarchy, you can either include the Java classes or extend the existing Java classes when you are developing Java applications. For example, Java `java.util` and `java.lang` packages contain many Java classes that are commonly used by Java developers, such as `Scanner class` for reading text input, `ArrayList class` for a collection of objects, `String class` for processing character data, etc. See the list of Java packages here. These Java packages (modules) are also called Application Programming Interfaces (APIs).

Additional Java library classes may be archived as JAR files and are open-source for Java developers to use for free. A JAR ("Java archive") file is a package file format typically used to aggregate many Java class files and associated metadata and resources (text, images, etc.) into one file for distribution. If you develop a set of Java source files and you want to share them with others, you can create a JAR file to include all the Java sources you would like to distribute. JavaFX is a good example. JavaFX contains many Java packages extending from the existing Java standard library. They are compressed into executable JAR files for distribution. Java developers can download the JARs and include them as part of the

7. The Object Class



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)

The Java class **`Object`** is the root of the Java class inheritance hierarchy. That is, the Object class is automatically the superclass of all Java classes, including the classes in the standard APIs and the Java classes you create. As a result, all objects, including arrays, inherit the methods defined in this class. For example, the **`equals()`** and **`toString()`** methods are two of the methods all Java classes inherit and are commonly used.

We mentioned **overloading** earlier, which is to define multiple methods with the same name, but with different method signatures. On the other hand, **overriding** is to define multiple methods with the same method signature but different implementations for the method bodies. Overriding always happens in the subclasses, which "overrides" the code defined in the superclass when software change is necessary. Let's use the **`equals()`** method as an example to demonstrate how overriding works.

A Java class defines an ADT (abstract data type), which encapsulates data and operation. It is NOT a primitive data type, so you cannot determine if two objects are equal with the logical operator "==".

Therefore, we need to always use the **`equals()`** method to determine if two objects are equal. The default implementation of the **`equals()`** method in the Object class is to use the logical operator `"=="` to compare the objects. As a result, the memory addresses of two objects are being compared. This may result in an undesirable behavior. Therefore, whenever you define a new Java class, you would always want to "override" the **`equals()`** method and write the code to compare the objects properly. Note that if you change the signature of the **`equals()`** method, then you are overloading, NOT "overriding".

Below is an example of the typical way of overriding the **`equals()`** method. Let's assume that there is a Student class defined to include an instance variable "name" declared in String type. The **`equals()`** method below is defined in the Student class. First, the method signature must remain unchanged as defined in the Object class. To avoid accidental change of the method signature, we can add the `@Override` tag to ensure that accidental change will be caught at compile time. Next, we need to ensure that we are comparing the two objects in the same type. Since the parameter type is Object, and every object is an instance of Object class. It is possible an object not in Student type is passed to the method. For example, `student.equals(employee)` will perform the **`equals()`** method in Student class, however, use an employee object as the argument. Finally, we down cast the Object type to Student type so the proper **`equals()`** method can be performed to determine the equality.

```
@Override //so the accidental change of signature will be caught at compile time
public boolean equals(Object obj) { //same signature defined in Object class
    if (obj instanceof Student) {//ensuring obj is a Student object
        Student student = (Student) obj; //down casting to Student object
        return student.name.equals(this.name); //the equals() in String class
    }
    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. The **`toString()`** method in the Object class 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, an object of the Loan class performs the **`toString()`** method will return **`Loan@15037e5`**. This is not very helpful or informative. Therefore, you should always override the **`toString()`** method so that it returns a descriptive string representation of the object.

8. 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. Add the modifier static to the variable or method declarations to declare a static variable or define a static method.

A constant is a numeric value that always remains unchanged during program execution. Thus, constants should be declared as **final static or static final;** 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. For example, `Math.PI` or `Math.pow(3, 2)`. This improves readability because this makes static methods and data easy to spot.

An instance method of a class can invoke an instance method or static method and access an instance variable or static variable within the same class. A static method can invoke a static method and access a static variable; however, it cannot invoke an instance method or access an instance variable without creating an object because instance methods and instance variables 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.

9. 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 classes 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 non-blank 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. A good practice is to use all lower-case letters for a package name. For example, Java packages like `java.util` and `java.lang`, etc.

In addition to the **public** and default visibility modifiers, Java provides **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 public and private modifiers on local variables would cause a compile error.

Modifier	directly accessible within the class	directly accessible within the package	directly accessible by subclasses in the same package or in different packages	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 private as shown below.

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

10. Data Encapsulation with the private modifier

Making instance variables private better protects the data and enhance the maintainability. 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, a static variable `numberOfObjects` is to count the number of objects created, but it may be mistakenly set to an arbitrary value and shared by all objects, such as `Circle.numberOfObjects = 10`. The class becomes difficult to maintain and vulnerable to bugs. As another example, 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 must change not only the Circle class but also the programs that use it because the client classes may have modified the radius directly. This creates "coupling," meaning that any software change creates a ripple effect where more code needs to be modified. To prevent direct modifications from other classes on the data and enhance the maintainability, you should always declare the instance variables as private using the "private" modifier.

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 private data accessible, provide a "getter" method to return its value. To enable 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**.

11. Passing Objects to Methods

Method signatures may contain parameters in reference types (class types.) You can pass objects to methods. Like passing an array, passing an object is passing **the reference of the object**. For example, The following code passes the **circle** object as an argument to the **printCircle()** method:

```java
public void printCircle(int times, Circle c) {
    times = 10; //times is a local variable doesn't change the value of n
    c.setRadius(6.0); //c contains the same reference of circle
      ...
}

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

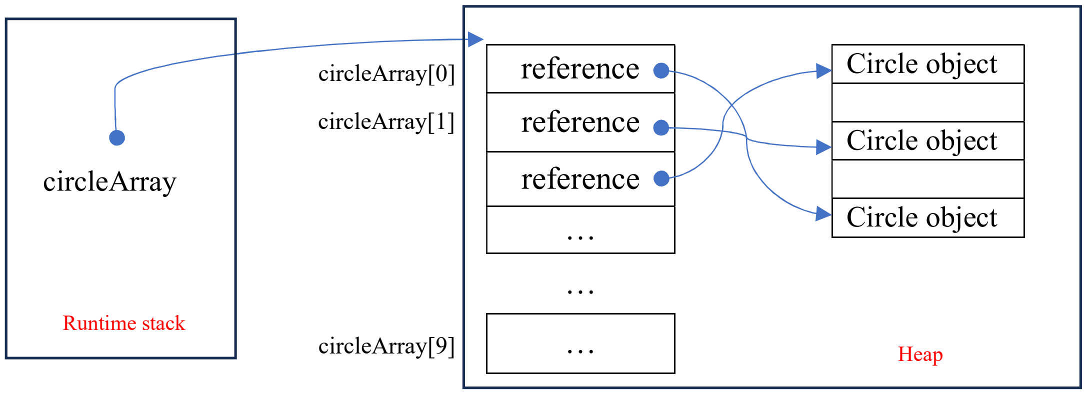
Pass-by-value refers to the situation where the method call is passing the value of an argument of a primitive data type. In the above example, the value of *n* (i.e., 5) is passed to the parameter **times**. In the **printCircle()** method, if the content of the variable **times** is changed, the value of *n* in the main method remains unchanged, as **times** is a local variable to **printCircle()** method. 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.

12. Array of Objects

An array of objects is an array of references. Thus, invoking `circleArray[1].getArea()` involves two levels of referencing. For example,

```
Circle[] circleArray = new Circle[10];
```

The above statement declares an array of Circle objects with a capacity of 10, the array indexes are running from 0 to 9. The variable **circleArray** contains the reference to the beginning address of a consecutive memory block allocated to store the 10 object references. Each array element stores a reference to an instance of the Circle class. For example, **circleArray[1]** references the second element of the array, where a reference to a Circle object is stored. 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 used to store the circle objects are not necessarily consecutive. When an array of objects is created using the **new** operator, each element in the array contains the default value **null**.



13. 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 the reference 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 the cloned 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:

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

14. Scope of Variables

Instance and static variables defined in a class are referred to as **class variables**. Variables defined inside any method 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 the 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 as an instance variable and 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.

```java
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 the local variable x
        System.out.println("y = " + y);
    }
}
```

15. 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.

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

```java
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.

16. 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. Refer to the Javadoc for the Double class and Integer class by following the links below.

java.lang.Double

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Double.html

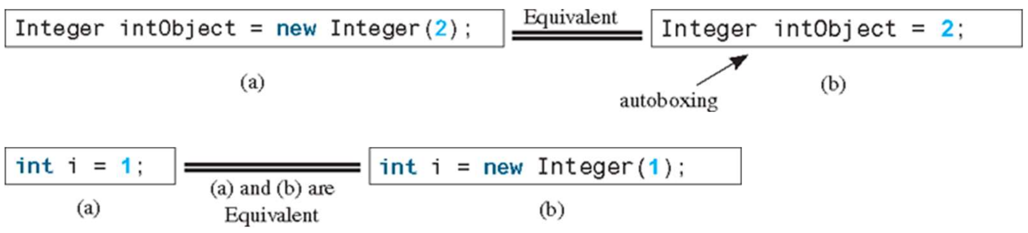
java.lang.Integer

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Integer.html

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 **auto unboxing.**



There are BigInteger and BigDecimal classes that 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 **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.

17. 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";
```

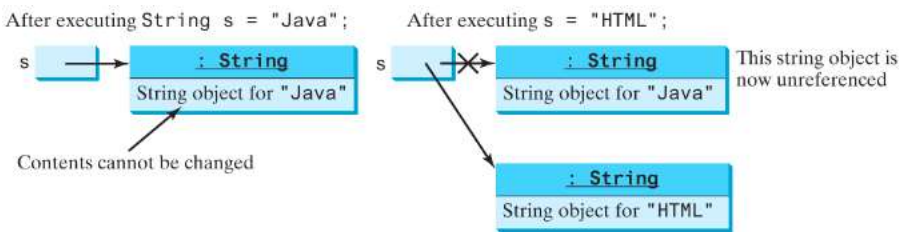
You can also create a string from an array of characters. For example, the following statements create the string **"Good Day"**:

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

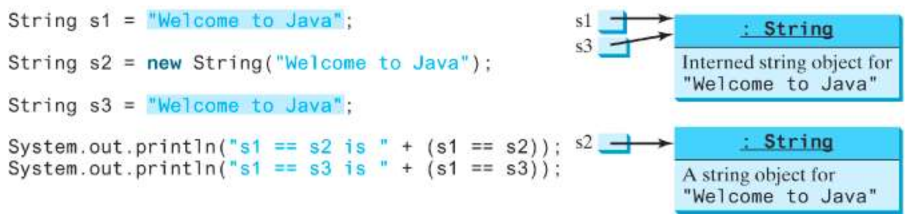
A String variable holds a reference to a String object that stores a string value. Strictly speaking, the terms String variable, String object, and string value are different, but most of the time, the distinctions between them can be ignored. For simplicity, the term string will often be used to refer to a 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.


After executing String s = "Java";
s
: String
String object for "Java"
Contents cannot be changed
After executing s = "HTML";
s
: String
String object for "Java"
This string object is now unreferenced
: String
String object for "HTML"

Because strings are immutable and 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:

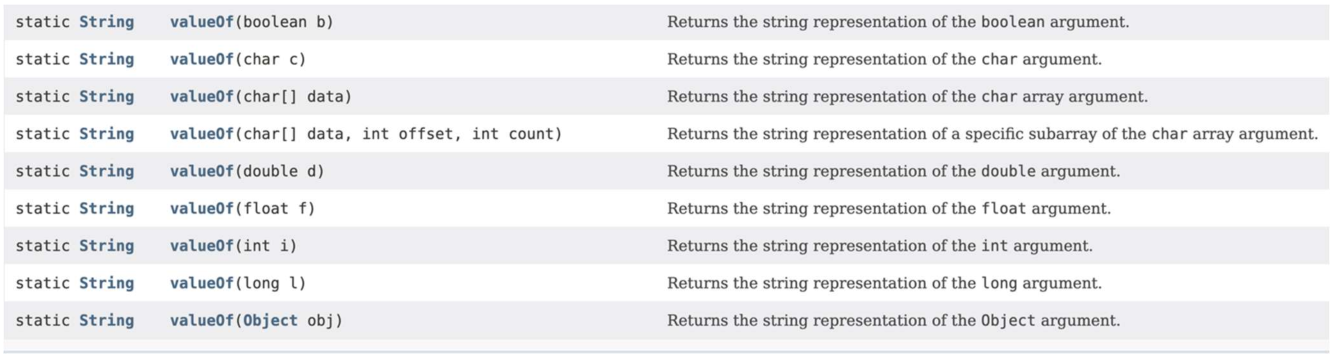


```
String s1 = "Welcome to Java";

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
s3
: String
Interned string object for "Welcome to Java"
s2
: String
A string object for "Welcome to Java"

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

In the preceding statements, s1 and s3 refer to the same interned string—"Welcome to Java"— therefore, `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]);
```

18. Enum class

An Enum class is a special data type that can be used to define 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. Java Enum types are much more powerful than their counterparts in other languages. The Enum class body can include methods and other data fields. A Enum class inherits the methods defined in **java.lang.Enum**, which is the base class for all the Enum classes defined. **Note that**, all the methods defined in the base class are final, which means that you cannot override the methods. The only exception is the `toString()` method. For example, you CANNOT override the `compareTo()` method.

All the constants of an Enum class can be obtained by calling the implicit public static T[] values() method of that class. That is, the **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.

As an example, a days-of-a-week Enum class is defined as shown below.

```
//define an enum class for the days of a week
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY;
}

//a switch case statement to use the constant defined in Day class
switch (event.day) { //event object's day in Day type.
    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;
}
```

By using the `values()` method in the for-each statement, the code below will display: SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY,

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

`Calendar class` is one of the Java library classes that deal with dates and times. It defines commonly used calendar dates and times as constant fields. You can also utilize the constants defined in the `Calendar class` without defining additional Enum classes.

An Enum class can also define constants with additional properties. For example, the `Planet class` below is an Enum type that represents the planets in the solar system. They are defined with two additional properties – mass and radius. Each planet is declared with a name and the constant values for the mass and radius parameters. Each pair of the constant values is passed to the constructor when an instance of the `Plant class` is created.

Java requires that the constants be defined first, prior to defining any data fields or methods. Also, when there are data 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. The private constructor automatically creates the instances of this class that are listed at the beginning of the Enum class body. The constructor is invoked by Java runtime. You CANNOT write code to invoke an Enum constructor outside of the Enum class.

```
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6), //constants with specified values
    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; must be final
   private final double radius; // in meters; must be final

    Planet(double mass, double radius) { //private; called by JVM
        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

19. 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 block has no connection to whether or not its associated next() method will block. The tokens() method may also block waiting for input.

For example, the code below 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. The example below reads several items 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

20. Calendar class

The Calendar class is an abstract class that provides methods for converting between a specific instant in time and a set of calendar fields such as YEAR, MONTH, DAY_OF_MONTH, HOUR, and so on, and for manipulating the calendar fields.

- To get an instance of current date and time: `Calendar today = Calendar.getInstance();`
- Set the instance to a specific date: `today.set(year, month, day);`
- Note that, `Calendar.AUGUST` is 7.
- To get the calendar fields:

  ```
  today.get(Calendar.YEAR); //return an integer representing the year
  today.get(Calendar.DAY_OF_WEEK); //return the day of the week
  ```

- For other data fields and methods, see the API document here: https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Calendar.html