Lecture - Classes and Methods

Overview

Classes are the most important language feature and the basis for object-oriented programming (OOP). They are central to Java and you have already come across some: remember *String* and *Scanner?* They are classes that come with Java's standard library. A class has methods (actions) as well as data.

In this unit, you will learn about:

- Class definitions
  - Class structure
  - Variables
  - Methods
- Access modifiers (e.g., public vs private)
- Accessor and mutator methods
- Overloading
- Constructors

Q1

```
int a = 5, b = 10;
System.out.println(a++ + --b);
```

What's the output?

5 f 9 : 14

Q2:

```
System.out.println(7%2);
```

What's the output?

1

Q3:

```
int a = 1, b = 0;
if(a++ > 0 || ++b >0)
System.out.println(b);
```

What's the output?

0

→ This one is important.
Because the first part a++ is already true, it won't process the second expression. Therefore, b is still 0.

Q4

Which of the following is a valid if-else statement in Java?

```
if (a = b) {
...
}
```

✓
```
if (a == b) {
...
} else if (a > b) {
...
} else {
...
}
```

```
if a == b then {
...
} else {
...
}
```

29

Q6:

```
for (int i = 0; i < 5; i++) {
System.out.print(i + " ");
}
```

What's the output?

0 1 2 3 4

Q6: Which statement is used to terminate a loop in Java?

continue

break

Q7:

A class defines the data items and methods/actions to take on the data items.

True

False

Which keyword is used to create objects?

Q8.

new

old

Q9.

the dot operator is used to access

- data items
- methods
- both

16

Q10.

Every class comes with two methods

- toString
- equals

class is a blueprint of object
object is actual in a memory

Terminology

Programming in Java consists of defining a number of classes as every program is a class. Java software, therefore, consists of a collection of classes. All programmer-defined types are classes. The class is at the core of Java. It is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object. Any concept you wish to implement in a Java program must be written within a class.

class contain data and methods.

A class is a **type** and you can declare variables of your own **class type** (*e.g.*, you can define a `class Car` and then declare a variable `Car myCar`). A value of a class type is called an **object** or an **instance** of the class. The process of creating an object of a class is called **instantiation.**

> i If A is a class, then the statements "the type of B is A", "B is an instance of A", and "B is an object of the class A" all mean the same thing.

An object has both **data** and **actions** (i.e., **methods**) associated with it. Both the data items and the methods are also called **members** of the object. Data items are also referred to as **fields**. Methods

class are non primitive data type

define what objects of your class are capable of. Each object can have different data, but all objects of a class have the same *types* of data and all objects of a class have the same methods (*e.g.*, if either `myCar` `oryourCar` has a method `drive()` then both do). You can think of methods and the list of variable names/types as belonging to the class, and the data stored in the variables as belonging to the instance of the class.

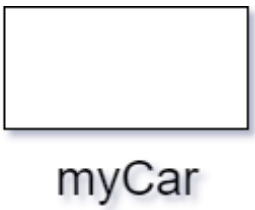
```
public class Car {
    // data fields or instance variables
    private String manufacturer; //private is a modifier that makes sure that this instance variabl
    private String model;
    private int yearBuilt;
    private int horsepower;

    // a method
    public void drive() {
        //...
    }

    // another method
    public void writeOutput() {
        System.out.printf("%s (%s, %d) with %d hp\n", manufacturer, model, yearBuilt, horsepower);
    }

    // program entry
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.manufacturer = "BMW";
        myCar.model = "X7";
        myCar.yearBuilt = 2020;
        myCar.horsepower = 523;

        myCar.writeOutput();
    }
}
```

Each object of this class has four pieces of data:

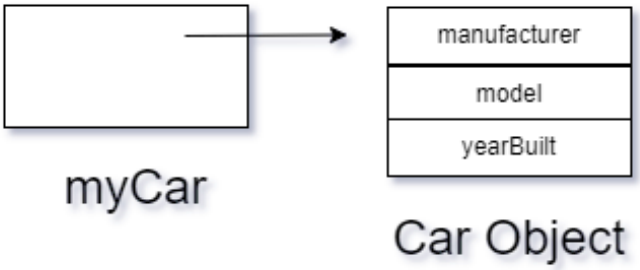
1. a string for the manufacturer
2. a string for the car's model
3. an integer for the year it was built, and
4. another integer for its horsepower (hp).

A class type value or object can have multiple pieces of data as well as actions. This is different from primitive type values, which are single pieces of data (*e.g.*, int, short, char). All objects of a class have the same pieces of data (*i.e.*, manufacturer, model, yearBuilt, and horsepower) and methods (*i.e.*, writeOuput). For a given object, each piece of data can hold a different value. The effect of the statements above in the code is depicted in the image below:

Car myCar;



myCar = new Car();



Class and Method Definitions

Classes are usually defined in their own file. So, if you want to create a new class called *Car* you will want to write the class definition into a file called *Car.java* (for now, make sure that your program and all the classes it uses are in the same directory or folder). A class definition specifies the data items (instance variables) and methods (actions) that all of its objects will have. Instance variable declarations and method definitions can be placed in any order within the class definition.

The *new* Operator

An object of a class is named or declared by a variable of the class type.

```
ClassName classVariable; //declaration
```

The **new operator** must then be used to create the object and associate it with its variable name:

```
classVariable = new ClassName(); //instantiation
```

Declaration and instantiation can be combined as follows:

```
ClassName classVariable = new ClassName();
```

The above statement creates a variable called `classVariable` of the class `ClassName`. To create an object, you must use the `new` operator to create a "new" object.

Advanced: Doing this will allocate memory for the object. That memory is automatically released when Java determines that the object is finished with. In some languages such as C/C++ the programmer has to delete the object explicitly. This is more efficient, but often causes bugs that are hard to track down.

Instance Variables

Instance variables are defined within the class and can take any valid *type* (including that of another class or even the same class!).

```
public String instanceVariable1;
public int instanceVariable2;
```

The modifier *public* is used to make the instance variable accessible from outside the class. We will learn about modifiers in a later unit.

In order to refer to an instance variable within an object, preface it with its object name as follows:

```
classVariable.instanceVariable1;
```

Instance variables can be both set and read as in the prior example:

```
Car myCar = new Car();
myCar.manufacturer = "BMW";
System.out.println("Manufacturer: " + myCar.manufacturer);
```

Methods

As mentioned at the beginning , classes usually consist of two things: instance variables and methods. Method definitions are divided into two parts: a **heading** and a **method body.**

```
public void myMethod() { //heading
    //body
}
```

Methods are invoked using the name of the calling object and the method name as follows:

```
classVariable.myMethod();
```

Invoking a method is equivalent to executing the method body. There are two kinds of methods:

1. methods that compute and return a value and
2. methods that perform an action

A method that returns a value must specify the type of its return value in its heading:

```
public typeReturned methodName(parameter_List) {}
```

Methods that only perform an option without returning a valid are called **void methods**. A void method uses the keyword *void* in its heading to show that it does not return a value:

```
public void methodName(parameter_List) {}
```

As we have already seen, a program in Java is just a class that has a **main** method. When you run a Java program, the run-time system invokes that method main, which is a void method as indicated by its heading:

```
public static void main(String[] args) {}
```

The method body consists of a list of declarations and statements enclosed in a pair of braces. The body of a method that returns a value must also contain one or more **return statements**. A return statement specifies the value returned and ends the method invocation:

```
return Expression;
```

Expression can be any expression that evaluates to something of the type returned listed in the method heading. A void method does not need to contain a return statement unless there is a situation that requires the method to end before all its code is executed. In this case, since it does not return a value, a return statement is used without an expression:

```
return;
```

Note that `return` is one of the few Java keywords that is followed by a value that is not in brackets. Another is `case`.

Exercise: Can you think of any others?

Method Invocation

An invocation of a method that returns a value can be used as an expression anywhere that a value of the `typeReturned` can be used:

```
typeReturned tRVariable;
tRVariable = objectName.methodName();
```

An invocation of a void method, on the other hand, is simply a statement:

```
objectName.methodName();
```

An invocation of a method that returns a value of type *boolean* returns either *true* or *false*. It is, therefore, common practice to use an invocation of such a method to control statements and loops where a Boolean expression is expected, i.e., in *if-else* statements or *while* loops.

A method that returns a value can also perform an action. If you want the action performed, but do not need the returned value, you can invoke the method as if it were a void method, and the returned value will simply be discarded.

Here is an example of a class definition that has both type of methods:

```
public class Date {
    public int day;
    public int month;
    public int year;

    // a void method
    public void writeOutput() {
        System.out.println(day + "/" + month + "/" + year);
    }

    // a method that returns a value
    public int getYear() {
        return year;
    }

    public static void main(String[] args) {
        // object declaration & creation
        Date date1 = new Date();

        date1.day = 13; //data initialization/update
        date1.month = 8;
        date1.year = 2021;
        System.out.println("date1:");
        date1.writeOutput(); //void method invocation  involve method
```

```
        int year = date1.getYear(); //method invocation
        System.out.printf("Year: %d\n", year);
    }
}
```

Variables, Parameters, and Arguments

A variable declared within a method definition is called a **local variable**. In the example above, *date1* is a local variable that only exists within the context of the main method. Another local variable in this example is the *args* variable, which is also called a **method parameter**. All method parameters are local variables.

> ⚠ Unlike other programming languages, Java does not have **global variables**.

A **block** is another name for a compound statement or a method body; that is, a set of Java statements enclosed in braces, {}. A variable declared within a block is local to that block. When the block ends, all variables declared within the block disappear.

> ⚠ In Java, you cannot have two variables with the same name inside a single block definition (e.g., inside a sub-block and outside a sub-block) .

Exercise: Fix the following code. Can you fix it in more than one way?

```
class Main {
    static public void main (String[] args) {
        int i = 0;
        System.out.println(i);

        if (i < 1) {
            double i = 10d;
            System.out.println(i);
        }
    }
}
```

global variable (instance variable): you define within the class outside any istance variable

local variable: define within the method.

block variable: a variable that is define between say the braces.

error : You can not have a variable with same name in the method.

You can declare one or more variables within the initialization portion of a *for* statement.

```
int sum = 0;
for(int i=1; i<=100; i++) {
    sum = sum+1;
}
```

The variable *i* is local to the for loop, and cannot be used outside of the loop. If you need to use such a variable outside of a loop, then you must declare it outside the loop.

Parameters

Most methods we have seen so far have had no parameters (except `main(String[] args)`) as indicated by an empty set of parentheses in the method heading. Some methods, however, need to

receive additional data via a list of **parameters** in order to perform their work. These parameters are also called **formal parameters**.

A parameter list provides a description of the data required by a method. It indicates the number and types of data pieces needed, the order in which they must be given, and the local name for these pieces as used in the method:

```
public double myMethod(int param1, int param2, double param3) {}
```

When a method is invoked, the appropriate values must be passed to the method in the form of **arguments**. Arguments are also called **actual parameters**. The number and order of the arguments must exactly match that of the parameter list. The type of each argument must be compatible with the type of the corresponding parameter.

```
int a=1,b=2,c=3;
double result = myMethod(a,b,c);
```

If argument and parameter types do not match exactly, Java will attempt to make an automatic type conversion. In the example above, the int value of argument c would be cast to a double. A primitive argument can automatically be typecast from any of the following types to any of the types that appear to its right:

```
byte > short > int > long > float > double > char
```

The value of each argument (not the variable name) is plugged into the corresponding method parameter. This method of plugging in arguments for formal parameters is known as the **pass-by-value mechanism**. When a method is invoked, the value of its argument is computed/evaluated, and the corresponding parameter (i.e., the local variable) is initialized to this value. Even if the value of a formal parameter is changed within a method (i.e., it is used as a local variable) the value of the argument cannot be changed.

If the parameter is a variable of object type (class or array), then it can *seem* as if the value can be changed. This will be explained in more detail in a later lecture. For now just think of primitive types as being passed by value, and object types as being "it's complicated".

In the following example, we use a method with parameters to set the instance variables:

```
public class Date {
    public int day;
    public int month;
    public int year;

    // a void method
    public void writeOutput() {
        System.out.println(day + "/" + month + "/" + year);
    }
```

```java
    // a method that returns a value
    public int getYear() {
        return year;
    }

    // a method with parameters
    public void setDate(int newDay, int newMonth, int newYear) {
        day = newDay;
        month = newMonth;
        year = newYear;
    }

    // a method with return type
    public String monthString(int monthNumber)
    {
        switch(monthNumber) {
            case 1:
                return "January";
            case 2:
                return "February";
            case 3:
                return "March";
            case 4:
                return "April";
            case 5:
                return "May";
            case 6:
                return "June";
            case 7:
                return "July";
            case 8:
                return "August";
            case 9:
                return "September";
            case 10:
                return "October";
            case 11:
                return "November";
            case 12:
                return "December";
            default:
                return "ERROR: no such month";
        }
    }

    public static void main(String[] args) {
        // object declaration & creation
        Date date1 = new Date();

        date1.setDate(13, 8, 2021); //method invocation with parameters
        System.out.println("date1:");
        date1.writeOutput(); //void method invocation

        int year = date1.getYear(); //method invocation
```

```
        System.out.printf("Year: %d, Month: %s\n", year, date1.monthString(date1.month));
    }
}
```

The terms **parameter** and **argument** are often used wrongfully interchangeably. You may need to infer their meaning from context.

Bill Example

This is an example program that uses two classes:

- *Bill.java*: this class holds all data and methods necessary to bill a client for a service
- *BillingDialog.java*: this class holds the program execution code and manages the user dialogue

Have a look at how formal parameters are used as local variables and study the program flow between the classes.

Not every class should have main method.

```
public class Date {

    private int day;
    private int month;
    private int year;

    public int getDay(){
        return day;
    }

    public void setDay(int dayOfMonth){
        if(dayOfMonth >0 && dayOfMonth <32){
            day = dayOfMonth
        }
    }
}
```

(I can access the value through this method)

← getter method

Use these methods outside the classes.

← setter

(If I want to set the value, through this method)

```
public class Date{

    private int day;
    private int month;
    private int year;

    public int getDay(){
        return day;
    }

    public void setDay(int dayOfMonth){
        if(dayOfMonth > 0 && dayOfMonth < 32){
            day = dayOfMonth;
```

```java
import java.util.Scanner;

class Bill
{
    public static double RATE = 150.00; //Dollars per quarter hour

    private int hours;
    private int minutes;
    private double fee;

    public void inputTimeWorked()
    {
        Scanner keyboard = new Scanner(System.in);

public static void main(String[] args)
{
    System.out.println("Welcome to the law office of Better Call Saul!");
    Bill.RATE = 200;
    Bill yourBill = new Bill();

    yourBill.inputTimeWorked();

    yourBill.updateFee();
    yourBill.outputBill();

    System.out.println("It has been a pleasure to serve you.");
}
```

this line will compile, even this line haven't invoke

This is because here

```java
import java.util.Scanner;

class Bill
{
    public static double RATE = 150.00; //Dollars per quarter h

    private int hours;
    private int minutes;
    private double fee;
    private Date startDate;

    public Date getStartDate(){
        return startDate;
    }

    public void setStartDate(int day, int month, int year){
        startDate = new Date();
        startDate.setDay(day);
        startDate.setMonth(month);
        startDate.setYear(year)
    }
```

```
c class BillingDialog

ublic static void main(String[] args)

    System.out.println("Welcome to the law office of Better Call S

    Bill yourBill = new Bill();
    yourBill.setStartDate(1,4,2020);

    yourBill.inputTimeWorked();

    yourBill.updateFee();
    yourBill.outputBill();
    yourBill.computeFee(38,20);

    System.out.println("It has been a pleasure to serve you.");
```

```
public class Date {
    private int day;
    private int month;
    private int year;

    // accessor methods
    public int getDay() {
        return day;
    }

    public int getMonth() {
        return month;
    }

    public int getYear() {
        return year;
    }

    public void setDay(int day) {
        if((day <= 0) || (day > 31)) {
            System.out.println("Fatal Error");
            System.exit(0);
        } else {
            this.day = day;
        }
    }

    public void setMonth(int monthNumber) {
        if((monthNumber <= 0) || (monthNumber > 12)) {
            System.out.println("Fatal Error");
            System.exit(0);
        } else {
            month = monthNumber;
        }
    }
    public void setYear(int year) {
        if((year < 1000) || (year > 9999)) {
            System.out.println("Fatal Error");
            System.exit(0);
        } else {
            this.year = year;
        }
    }
```

The this Parameter (Keyword)

Take a look at the *writeOuput()* method in the following example:

```
public class Car {

    private String manufacturer;
    private String model;

    public void writeOutput() {
        System.out.printf("%s / %s\n", manufacturer, model);
    }

    // program entry
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.manufacturer = "BMW";
        myCar.model = "X7";

        myCar.writeOutput();
    }
}
```

```
myCar.writeOutput();
```

is equivalent to

```
System.out.printf("%s / %s\n", myCar.manufacturer, myCar.model);
```

This is because, although the definition of *writeOuput* reads

```
public void writeOutput() {
    System.out.printf("%s / %s\n", manufacturer, model);
}
```

it really means:

```
public void writeOutput() {
    System.out.printf("%s / %s\n", <the calling object>.manufacturer, <the calling object>.model);
}
```

All instance variables are understood to have <the calling object> in front of them. If an explicit name for the calling object is needed, the keyword **this** can be used. The following is, therefore, a valid Java method definition that is equivalent to the original *writeOutput()* method.

```
public void writeOutput() {
    System.out.printf("%s / %s\n", this.manufacturer, this.model);
```

```
}
```

The *this* object reference must be used if a parameter or other local variable with the same name is used in the method. Otherwise, all instances of the variable name will be interpreted as local.

```
int someVariable = this.someVariable;
//      |_local            |_instance
```

The *setDate()* method in our previous example could, therefore, be rewritten as follows:

```
public class Date {
    public int day;
    public int month;
    public int year;

    public void setDate(int day, int month, int year) {
        this.day = day; //note how <this> is used to distinguish instance from local variable
        this.month = month;
        this.year = year;
    }

    public void writeOutput() { //here we don't need <this> as the context is unambigious
        System.out.println(day + "/" + month + "/" + year);
    }

    public static void main(String[] args) {
        Date date1 = new Date();
        date1.setDate(13, 8, 2021);
        date1.writeOutput();
    }
}
```

The *this* parameter is a kind of hidden parameter. Even though it does not appear on the parameter list of a method, it is still a parameter. When a method is invoked, the calling object is automatically plugged in for *this*.

equals() and toString()

Java expects certain methods, such as **equals** and **toString**, to be implemented by all, or almost all, classes.

The equals() Method

The purpose of the *equals()* method, a boolean-valued method, is to compare two objects of the class to see if they satisfy the notion of "being equal".

You can't just use == to compare objects as it often refers to references in memory!

We have already encountered the *String.equals()* method:

```
String userInput = Scanner.nextLine();
if(userInput.equals("exit")){
    System.exit(0);
}
```

For your classes, you should always implement an *equals()* method to be able to compare instances to each other. How that comparison is made is up to your implementation. In the following example we assume dates are equal if their *day*, *month*, and *year* variables are the same:

```
public class Date {
    public int day;
    public int month;
    public int year;

    public void setDate(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
    }

    public boolean equals(Date anotherDate) { //equals expects an argument of the same type as itse
        return (day==anotherDate.day && month==anotherDate.month && year==anotherDate.year);
    }

    public static void main(String[] args) {
        Date date1 = new Date();
        date1.setDate(13, 8, 2021);

        Date date2 = new Date();
        date2.setDate(26, 8, 2021);

        Date date3 = new Date();
```

```
        date3.setDate(13, 8, 2021);

        System.out.println("date1 and date2 are equal: " + date1.equals(date2));
        System.out.println("date1 and date3 are equal: " + date1.equals(date3));
    }
}
```

If you don't provide an `equals()` method, a default will generated -- but the default often doesn't do what you would expect, and so it is good to provide your own.

The toString() Method

The purpose of the `toString()` method is to return a String value that represents the data in the object.

```
public class Date {
    public int day;
    public int month;
    public int year;

    public void setDate(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
    }

    public String toString() { //the toString() method does not take parameters and returns a Strin
        return (day + "/" + month + "/" + year);
    }

    public static void main(String[] args) {
        Date date1 = new Date();
        date1.setDate(13, 8, 2021);
        System.out.println("date1: " + date1.toString());
    }
}
```

In the example above, check what happens if you replace line 19 with the following statement:

```
System.out.println(date1);
```

What happens here is that Java knows that the `println()` function takes a `String` as an argument. As a result, it tries to convert the *date1* object into a `String`, which it does by calling the `toString()` method.

Access Modifiers

Java's access modifiers are public, private, and protected. Java also defines a default access level. protected applies only when inheritance is involved which we will cover when we talk about inheritance in Week 6. We will only cover public and private access modifiers here.

Modifiers: *public* and *private*

Java uses **modifiers** to restrict access to certain variables and methods. They contribute to the language's ability to hide information and are a crucial component of encapsulation.

The modifier **public** means that there are no restrictions on where an instance variable or method can be used. The modifier **private**, on the other hand, means that an instance variable or method cannot be accessed by name from outside of the class.

It is considered good programming practice to make all instance variables `private` by default. Most methods are `public` and thus provide controlled access to the object. Usually, methods are `private` only if used as **helper methods** for other methods in the class.

Accessors and Mutators

Accessor and Mutator Methods

Despite instance variables should be made `private` *(*unless you have a good reason not to), we may still need ways to access and modify them. That's where **accessor and mutator** methods come in.

Accessor methods allow the programmer to obtain the value of an object's instance variables. The data can be accessed but not changed. The name of an accessor method typically starts with the word *get.*

```
public class Car {

    private String manufacturer;
    private String model;

    // accessor methods
    public String getManufacturer() {
        return manufacturer;
    }

    public String getModel() {
        return model;
    }
}
```

An accessor method may also perform some computation to return the result, rather than only returning instance variable values; this flexibility is an important part of encapsulation.

Mutator methods allow the programmer to change the value of an object's instance variables in a controlled manner. It allows you as the creator of the class to test and filter incoming data and the data manipulation attempts. The name of a mutator method typically starts with the word *set*.

Mutator methods, often also called *setter* methods, allow you to test and enforce certain conditions of the state of the data. The **precondition** of a method states what is assumed to be true when the method is called. The **postcondition** of a method states what will be true after the method is executed, as long as the precondition holds. It is a good practice to always think in terms of preconditions and postconditions when designing a method, and when writing the method comment.

```
public class Date {
    private int day;
    private int month;
    private int year;

    // accessor methods
```

```
    public int getDay() {
        return day;
    }

    public int getMonth() {
        return month;
    }

    public int getYear() {
        return year;
    }

    // mutator method with test of incoming data
    public void setDay(int day) {
        if((day <= 0) || (day > 31)) {
            System.out.println("Fatal Error");
            System.exit(0);
        } else {
            this.day = day;
        }
    }

    public void setMonth(int monthNumber) {
        if((monthNumber <= 0) || (monthNumber > 12)) {
            System.out.println("Fatal Error");
            System.exit(0);
        } else {
            month = monthNumber;
        }
    }

    public void setYear(int year) {
        if((year < 1000) || (year > 9999)) {
            System.out.println("Fatal Error");
            System.exit(0);
        } else {
            this.year = year;
        }
    }

    public boolean equals(Date otherDate) {
        if((otherDate.day == day) //within the definition of Date, you can directly access private
            && (otherDate.month == month)
            && (otherDate.year == year)) {
                return true;
            } else {
                return false;
            }
    }
}
```

Within the definition of a class, private members of any object of the class can be accessed, not just

private members of the calling object. That is why in the example above, the *equals* function can directly access the *day* as well as the *otherDate.day* variable.

Overloading

Overloading is when two or more methods in the same class have the same method name. To be valid, any two definitions of the method name must have different **signatures**. A signature consists of the name of a method together with its parameter list, for example:

```
// method takes one parameter of type Date
public void setDate(Date date) {}

// method takes 3 parameters
public void setDate(int day, int month, int year) {}

// method takes 3 parameters of different types
public void setDate(int day, String month, int year) {}
```

Differing signatures must have different numbers and/or types of parameters.

If Java cannot find a method signature that exactly matches a method invocation, it will try to use automatic type conversion. Be aware that the interaction of overloading and automatic type conversion can have unintended results. In some cases of overloading, because of automatic type conversion, a single method invocation can be resolved in multiple ways. Ambiguous method invocations will produce an error in Java.

The signature of a method only includes the method name and its parameter types. The signature does not include the type returned. Java does not permit methods with the same name and different return types in the same class.

Exercise: What could go wrong if there were two methods with the same parameter list but a different return type?

Although many programming languages, such as C++, allow you to overload operators (+, -, etc.), Java does not permit this. You may only use a method name and ordinary method syntax to carry out the operations you desire.

overload

different parameter

⇒ define in different data type.

```
public class Date {
    private int day;
    private int month;
    private int year;

    // accessor methods
    public int getDay() {
        return day;
    }

    public int getMonth() {
```

```
    return month;
}

public int getYear() {
    return year;
}

public void setDate(Date date) {
    day = date.day;
    month = date.month;
    year = date.year; //notice how no condition test is applied as we rely on the precondition
}

// method takes 3 parameters
public void setDate(int day, int month, int year) {
    setDay(day);
    setMonth(month);
    setYear(year);
}

// mutator methods with test of incoming data
public void setDay(int day) {
    if((day <= 0) || (day > 31)) {
        System.out.println("Fatal Error");
        System.exit(0);
    } else {
        this.day = day;
    }
}

public void setMonth(int monthNumber) {
    if((monthNumber <= 0) || (monthNumber > 12)) {
        System.out.println("Fatal Error");
        System.exit(0);
    } else {
        month = monthNumber;
    }
}

public void setYear(int year) {
    if((year < 1000) || (year > 9999)) {
        System.out.println("Fatal Error");
        System.exit(0);
    } else {
        this.year = year;
    }
}

public boolean equals(Date otherDate) {
    if((otherDate.day == day)
        && (otherDate.month == month)
        && (otherDate.year == year)) {
            return true;
        } else {
```

```
                return false;
            }
    }

    public static void main(String[] args) {
        Date date1 = new Date();
        date1.setDay(1);
        date1.setMonth(2);
        date1.setYear(2021);

        Date date2 = new Date();
        date2.setDate(date1);

        System.out.println(date1.equals(date2));
    }
}
```

Constructors

A **constructor** is a special kind of method that is designed to initialize the instance variables for an object.

```
public ClassName(anyParameters) {}
```

A constructor must have the same name as the class itself, it has no return type (not even *void),* and is typically overloaded. A constructor is called when an object of the class is created using *new*:

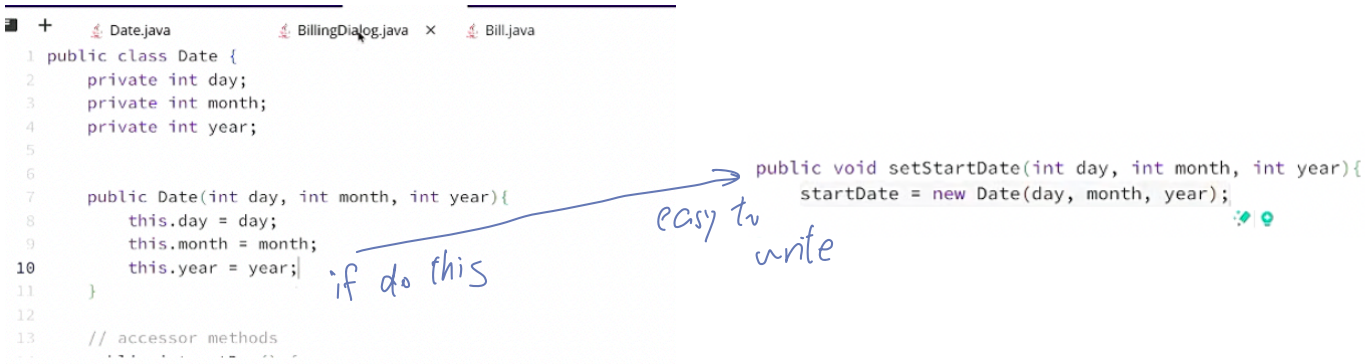
```
ClassName objectName = new ClassName(anyArgs);
```

The name of the constructor and its parenthesized list of arguments (if any) must follow the new operator. This is the only valid way to invoke a constructor: a constructor cannot be invoked like an ordinary method. If a constructor is invoked again (using *new*), the first object is discarded and an entirely new object is created. If you need to change the values of instance variables of the object, use mutator methods instead.

```
public class Date {
    private int day;
    private int month;
    private int year;

    // constructor → constructor with parameters
    public Date(int day, int month, int year) {
        setDay(day);
        setMonth(month);
        setYear(year);
    }

    // copy constructor
    public Date(Date date) {
        day = date.day;
        month = date.month;
        year = date.year;
    }

    // accessor methods
    public int getDay() {
        return day;
    }

    public int getMonth() {
        return month;
    }

    public int getYear() {
```

```
        return year;
    }

    // mutator methods with test of incoming data
    public void setDay(int day) {
        if((day <= 0) || (day > 31)) {
            System.out.println("Fatal Error");
            System.exit(0);
        } else {
            this.day = day;
        }
    }

    public void setMonth(int monthNumber) {
        if((monthNumber <= 0) || (monthNumber > 12)) {
            System.out.println("Fatal Error");
            System.exit(0);
        } else {
            month = monthNumber;
        }
    }

    public void setYear(int year) {
        if((year < 1000) || (year > 9999)) {
            System.out.println("Fatal Error");
            System.exit(0);
        } else {
            this.year = year;
        }
    }

    public boolean equals(Date otherDate) {
        if((otherDate.day == day)
            && (otherDate.month == month)
            && (otherDate.year == year)) {
                return true;
            } else {
                return false;
            }
    }

    public static void main(String[] args) {
        Date date1 = new Date(1, 2, 2021);
        Date date2 = new Date(date1);
        date2.setMonth(5);

        System.out.println(date1.equals(date2));
    }
}
```

The first action taken by a constructor is to create an object with instance variables. It is, therefore, legal to invoke another method within the definition of a constructor, since it has the newly created object as its calling object. For example, mutator methods can be used to set the values of the


Date.java
BillingDialog.java
Bill.java
public class Date {
private int day;
private int month;
private int year;
public Date(int day, int month, int year){
this.day = day;
this.month = month;
this.year = year;
}
// accessor methods
public void setStartDate(int day, int month, int year){
startDate = new Date(day, month, year);
if do this
easy to write

instance variables inside the constructor. It is even possible for one constructor to invoke another.

A constructor that takes an object of the same class as a parameter to create a copy of it is often referred to as a **copy constructor**.

Like any ordinary method, every constructor has a *this* parameter. The *this* parameter can be used explicitly but is more often understood to be there than written down. The first action taken by a constructor is to automatically create an object with instance variables. Then within the definition of a constructor, the *this* parameter refers to the object created by the constructor.

If you do not include any constructors in your class, Java will automatically create a **default** or **no-argument constructor** that takes no arguments, initializes variables to default values (see below), but allows the object to be created. If you include one or more constructors in your class, Java will not provide this default constructor. If you include constructors in your class, it is good practice to always provide your own no-argument constructor.

It is also good practice to use constructors to initialize instance variables. If you do not explicitly initialize instance variables, Java automatically initializes them as follows:

- *boolean* types are initialized to *false*
- Other primitives are initialized to the *zero* of their type
- Class types and array types are initialized to *null*

Local variables are not automatically initialized. It is a compile-time error to leave them uninitialized.

Implement a Person

Write a Java class called *Person* that has three instance variables:

- a name (String)
- an age (int)
- a gender (String)

Additionally, implement a constructor that initializes these instance variables when called as follows:

```
Person person = new Person("Joe", 25, "male");
```

Make sure that a person's age is never < 0. The default value for *age* should be set to 0, the name to "anonymous", and the gender to "unknown" if any invalid arguments are provided.

For each instance variable, make sure to provide the following accessor methods:

```
public String getName() {...}
public int getAge() {...}
public String getGender() {...}
```

Don't forget the *setter* methods:

```
public void setName(String name) {...}
public void setAge(int age) {...}
public void setGender(String gender) {...}
```

Add a *toString()* method that returns the value of the instance variables as follows:

```
Joe (25 years old), gender: male
```

Also, add an *equals()* method that takes another *Person* object as a parameter and returns *true* if all instance variable values are equal.

```
public boolean equals(Person anotherPerson) {}
```

The *Program.java* file contains the program entry (*i.e.,* the main method). You should not modify it. Instead, you need to modify your *Person.java* class in order to make the program run correctly.

Extension: Make gender case-insensitive, so that "Female" matches "female". You may want to use `String.toLowerCase(String s)`.

Extension: Write a `partial_match()` method that returns `true` if all of the non-default fields match. For example, `new Person ("", 25, "male").partial_match(new Person ("John", 25,`

`""))` should return `true`. Write this so that the literal text of the default values (like "anonymous") only appear once in the code, by using `static final` variables.

```
public Person(){}

public Person(String name, int age, String gender ){
    this.name = name;
    if(age >= 0 ){
        this.age = age;
    }else{
        this.age = 0;
    }
    this.gender = gender;

}


public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public String getGender() {
    return gender;
}

public void setGender(String gender) {
    this.gender = gender;
}

public int getAge() {
    return age;
}

public void setAge(int age) {
    if(age >= 0 ){
        this.age = age;
    }
}
```

```
public Person(String name, int age, String gender ){
    this.name = name.equals("") ? "anonymous": name;
    this.age = age > 0 ? age : 0;
    if (gender.equals("male") || gender.equals("female")){
        this.gender =gender ;
    }else{
        this.gender = "unknown";
    }

}
```

```
public String toString() {
    return "Person{" +
            "name='" + name + '\'' +
            ", gender='" + gender + '\'' +
            ", age=" + age +
            '}';
}
```

Relevant Reading Resources

Additional Reading Resources

- WALTER, S. Absolute Java, Global Edition. [Harlow]: Pearson, 2016. (Chapter 4)
- SCHILDT, H. Java: The Complete Reference, 12th Edition: McGraw-Hill, 2022 (Chapter 6 and 7)
- Classes and Objects (accessible on 14-02-2024) Oracle's Java Documentation. Available at: https://docs.oracle.com/javase/tutorial/java/javaOO/index.html