**Providing Constructors for Your Classes**

A class contains constructors.

Constructors are invoked to create objects from the class.

Constructor declarations look like method declarations—except that they use the name of the class and **have no return type**.

For example, Bicycle has one constructor:

**constructor with argument:**

public Bicycle(int startCadence, int startSpeed, int startGear) {

gear = startGear;

cadence = startCadence;

speed = startSpeed;

}

When we create an object such as myBike, a constructor is called by the **new operator**:

Bicycle myBike = new Bicycle(30, 0, 8);

**new Bicycle(30, 0, 8)** creates space in memory for the object and initializes its fields.

Although Bicycle only has one constructor, it could have others, including a **no-argument constructor**:

public Bicycle() {

gear = 1;

cadence = 10;

speed = 0;

}

Bicycle yourBike = new Bicycle(); invokes the no-argument constructor to create a new Bicycle object called yourBike

the Java platform differentiates constructors on the basis of the number of arguments in the list and their types. Therefore **you cannot write two constructors that have the same number and type of arguments for the same class. ---------------------🡪** compile-time error.

You don't have to provide any constructors for your class in such a case:

* The compiler automatically provides a **no-argument constructor**, default constructor for any class without constructors.
* This default constructor will call the no-argument constructor of the **superclass**.
* In this situation, the compiler will **complain if the superclass doesn't have a no-argument constructor** so you must verify that it does.
* **If your class has no explicit superclass, then it has an implicit superclass of Object, which *does* have a no-argument constructor.**

# Defining Methods

Here is an example of a typical method declaration:

public double calculateAnswer(double wingSpan, int numberOfEngines,

double length, double grossTons) {

//do the calculation here

}

method declarations have six components, in order:

1. Modifiers—such as public, private, and others you will learn about later.
2. The return type—the data type of the value returned by the method, or void if the method does not return a value.
3. The method name—the rules for field names apply to method names as well, but the convention is a little different.
4. The parameter list in parenthesis—a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, (). If there are no parameters, you must use empty parentheses.
5. An exception list—to be discussed later.
6. The method body, enclosed between braces—the method's code, including the declaration of local variables, goes here.

**Typically, a method has a unique name *within its class*. However, a method might have the same name as other methods due to *method overloading***

## **Overloading Methods ----🡪**

This means that methods within a class can have the **same name** if they have **different parameter** lists

Java distinguish between methods by seeing ***method signatures***. So if method name are same but have different parameters then java can easily distinguish among them.

public class DataArtist {

...

public void **draw**(***String s***) {

...

}

public void **draw**(***int i***) {

...

}

public void **draw**(***double f***) {

...

}

public void **draw**(***int i, double f***) {

...

}

}

You cannot declare more than one method with the same name and the same number and type of arguments, because the compiler cannot tell them apart

The compiler does not consider return type when differentiating method, so you cannot declare two methods with the same signature even if they have a different return type.

## **Access Modifiers :**

 public modifier—the field is accessible from all classes.

 private modifier—the field is accessible only within its own class

In the spirit of **encapsulation**, it is common to **make fields private**. This means that they can only be *directly* accessed from the Bicycle class. We still need access to these values, however. This can be done *indirectly* by adding public methods that obtain the field values for us:

public class Bicycle {

**private int cadence; directly accessible to methods which are present**

**private int gear; inside the class Bicycle**

**private int speed;**

public Bicycle(int startCadence, int startSpeed, int startGear) {

gear = startGear;

cadence = startCadence;

speed = startSpeed;

}

public int getCadence() {

return cadence;

}

public void setCadence(int newValue) {

cadence = newValue; 🡨---------------

}

public int getGear() {

return gear;

}

public void setGear(int newValue) {

gear = newValue; 🡨---------------

}

public int getSpeed() {

return speed;

}

public void applyBrake(int decrement) {

speed -= decrement; 🡨---------------

}

public void speedUp(int increment) {

speed += increment; 🡨---------------

}

}

## **Types:**

* **primitive types** ---------------> int, float, boolean, etc.
* **reference types** ---------------> strings, arrays, or objects.

# Declaring Classes :

You've seen classes defined in the following way:

class *MyClass* {

// field, constructor, and

// method declarations

}

This is a *class declaration*

**NOTE:** constructors used for initializing new objects

class *MyClass extends MySuperClass implements YourInterface* {

// field, constructor, and

// method declarations

}

means that MyClass is a subclass of MySuperClass and that it implements the YourInterface interface

# Passing Information to a Method or a Constructor:

public double computePayment(

double **loanAmt**,

double **rate**,

double **futureValue**,

int **numPeriods**)

{

double interest = **rate** / 100.0;

double partial1 = Math.pow((1 + interest),

- **numPeriods**);

double denominator = (1 - partial1) / interest;

double answer = (-**loanAmt** / denominator)

- ((**futureValue** \* partial1) / denominator);

return answer;

}

# This method has four parameters: the loan amount, the interest rate, the future value and the number of periods.

# **Note:** *Parameters* refers to the list of variables in a method declaration. *Arguments* are the actual values that are passed in when the method is invoked. When you invoke a method, the arguments used must match the declaration's parameters in type and order.

# Here's an example of a method that accepts an array as an argument

public **Polygon** **polygonFrom**(**Point[] corners**) {

// method body goes here

}

Note: datatype of method ‘polygonFrom’ is class Polygon

# **Note:** If you want to pass a method into a method, then use a [lambda expression](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html) or a [method reference](https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html).

## Passing Primitive Data Type Arguments

Primitive arguments, such as an int or a double, are passed into methods *by value*. This means that any changes to the values of the parameters exist only within the scope of the method. When the method returns, the parameters are gone and any changes to them are lost. Here is an example:

public class PassPrimitiveByValue {

public static void main(String[] args) {

int x = 3;

// invoke passMethod() with

// x as argument

passMethod(x);

// print x to see if its

// value has changed

System.out.println("After invoking passMethod, x = " + x);

}

// change parameter in passMethod()

public static void passMethod(int p) {

p = 10; ---🡪 do not modify the value of ‘x’, r-> **pass by value case**

}

}

When you run this program, the output is:

After invoking passMethod, x = 3

## Passing Reference Data Type Arguments

Reference data type parameters, such as objects, are also passed into methods *by value*. This means that when the method returns, the passed-in reference still references the same object as before. *However*, the values of the object's fields *can* be changed in the method, if they have the proper access level.

For example, consider a method in an arbitrary class that moves Circle objects:

public void moveCircle(Circle circle, int deltaX, int deltaY) {

// code to move origin of circle to x+deltaX, y+deltaY

circle.setX(circle.getX() + deltaX);

circle.setY(circle.getY() + deltaY);

// code to assign a new reference to circle

circle = new Circle(0, 0);

}

Let the method be invoked with these arguments:

moveCircle(myCircle, 23, 56)

Inside the method, circle initially refers to myCircle. The method changes the x and y coordinates of the object that circle references (that is, myCircle) by 23 and 56, respectively. These changes will persist when the method returns. Then circle is assigned a reference to a new Circle object with x = y = 0. This reassignment has no permanence, however, because the reference was passed in by value and cannot change. Within the method, the object pointed to by circle has changed, but, when the method returns, myCircle still references the same Circle object as before the method was called.

# Creating Objects

As you know, a class provides the blueprint for objects; you create an object from a class.

**Point originOne** = new Point(23, 94);

**Rectangle rectOne** = new Rectangle(originOne, 100, 200);

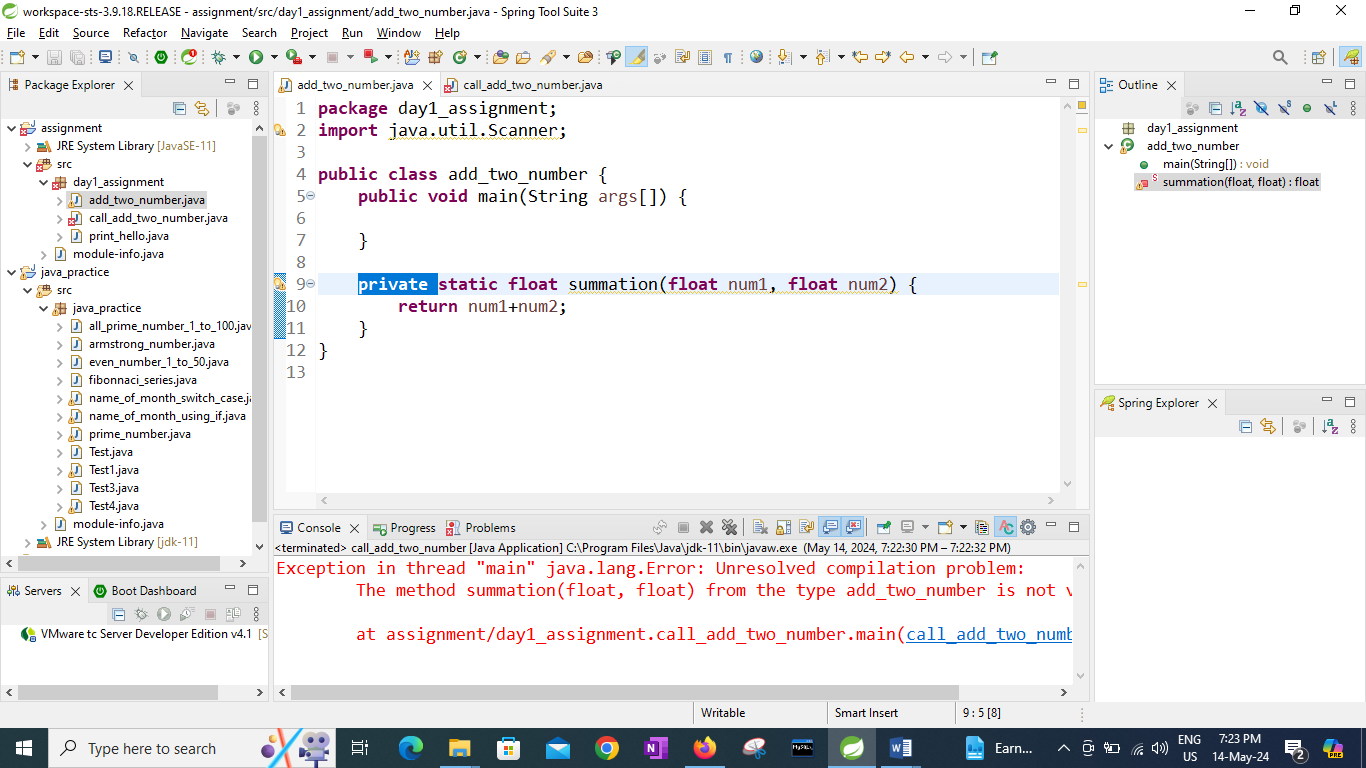
**Rectangle rectTwo** = new Rectangle(50, 100);

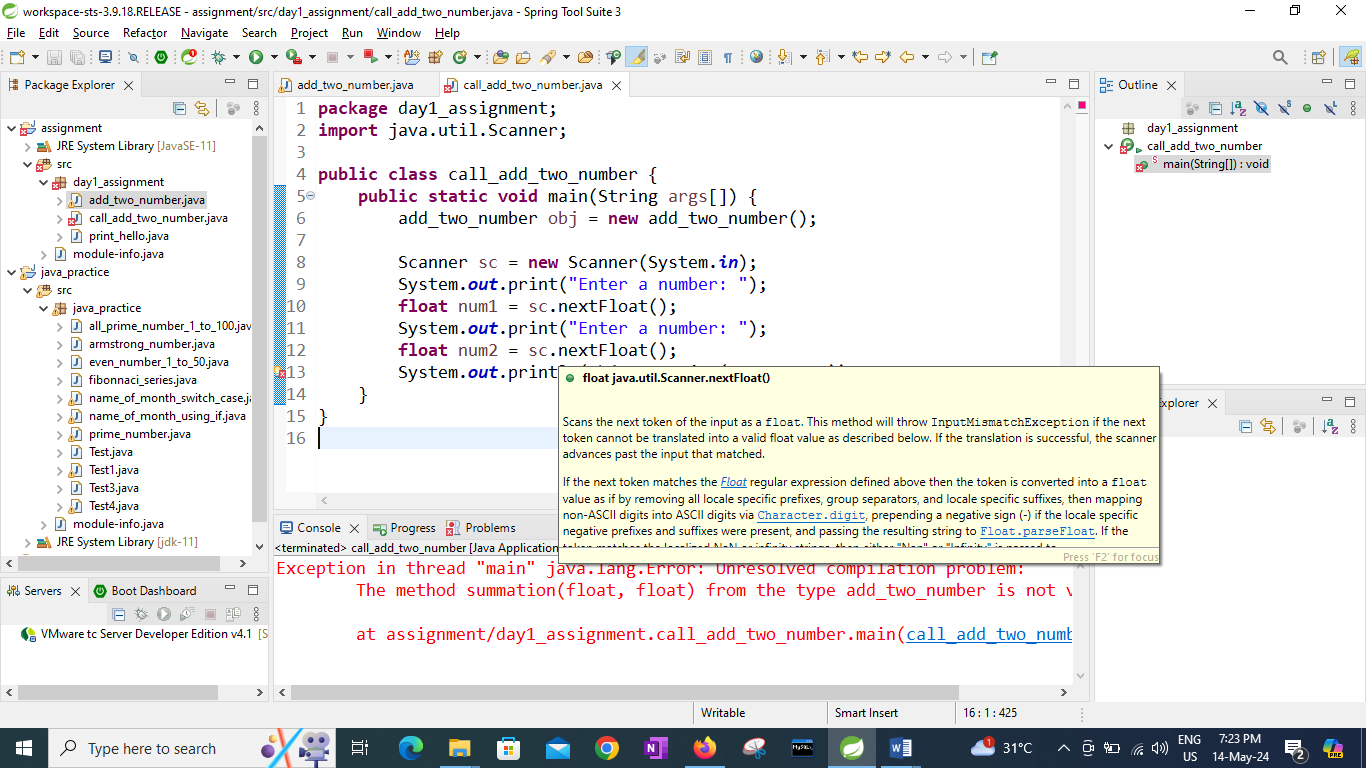
The first line creates an object of the [Point](https://docs.oracle.com/javase/tutorial/java/javaOO/examples/Point.java) class, and the second and third lines each create an object of the [Rectangle](https://docs.oracle.com/javase/tutorial/java/javaOO/examples/Rectangle.java) class.

Each of these statements has three parts (discussed in detail below):

1. **Declaration**: The code set in **bold** are all variable declarations that associate a variable name with an object type.
2. **Instantiation**: The **new** keyword is a Java operator that **creates the object.**
3. **Initialization**: The new operator is followed by a call to a constructor, which initializes the new object. -🡪 Point(23, 94); Rectangle(originOne, 100, 200); Rectangle(50, 100); constructor hai.

**# Scope of private – within a class**





The method summation(float, float) from the type add\_two\_number is not visible