

Lesson 3: Functions & Object-Oriented Programming (OOP)

- Object-Oriented Programming (OOP)** 3-1
- Object** 3-1
- Class** 3-1
 - Creating a Class 3-2
 - Providing Constructors for Your Classes 3-6
 - Class Inheritance 3-7
 - Abstract Class 3-11
 - Interface Class 3-15
 - Generic Class 3-21
 - Class Variables..... 3-23
 - Member Variables..... 3-23
- Kotlin Collections** 3-26
 - Hashmaps 3-26
 - ArrayList..... 3-30
 - listof and mutableListOf 3-35

Object-Oriented Programming (OOP)

If you have never used an object-oriented programming language before, you'll need to learn the basic concepts before you start coding. This lesson will introduce you to objects, classes, inheritance, interfaces, and packages.

Each discussion focuses on how these concepts relate to the real world, while simultaneously providing an introduction to the syntax of the Kotlin programming language. Learning the basics of OOP is necessary before starting any mobile application development.

Object

Objects are the key to understanding object-oriented technology. Look around you right now and you will find many examples of real-world objects: your car, your desk, your computer, your house...etc.

Real-world objects share two characteristics: state and behavior. Computers have a state (type, color, speed, capacity) and behavior (processing, playing media, browsing). A car also has a state (current gear, current pedal cadence, and current speed) and behavior (changing gear, changing pedal cadence, applying brakes). Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.

Class

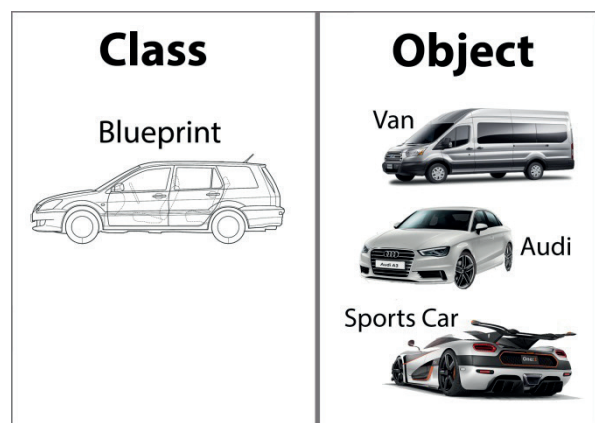
A class is a blueprint or prototype (template) from which objects are created. This section defines a class that models the state and behavior of a real-world object. It intentionally focuses on the basics, showing how even a simple class can cleanly model state and behavior. Classes are characterized by properties and functions.

For example, if the class is a car, it may have the following properties (attributes):

- Price
- Type
- Maximum speed
- Number of seats

And it may have the following functions:

- Number Of Seats
- Number Of Wheels
- get Year Built



Creating a Class

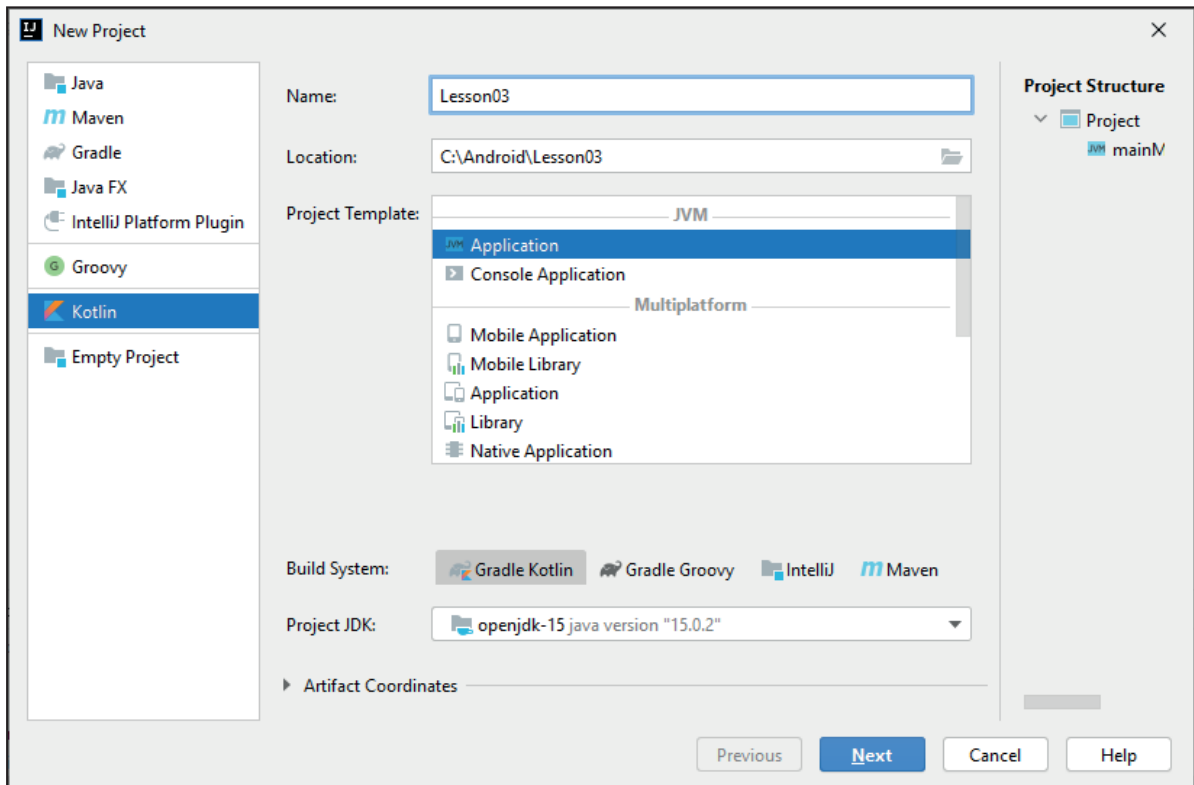
A class code includes the class name and a group of properties or attributes which will construct the class.

Example:

1- Open **IntelliJ IDEA**

2- Click **File** → **New** → **Project**

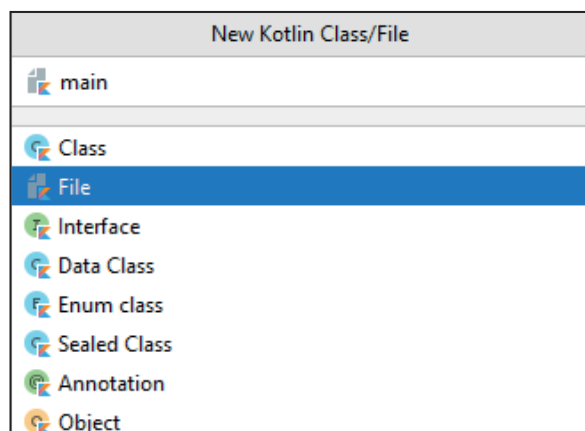
3- In the dialog box below, type **Lesson03** for the Kotlin project **Name**, select the project path, and the **Project JDK**. Click **Next**, then click **Finish**.



4- Select **New Window**

5- Right click the Kotlin folder (Lesson03 → src → main → kotlin) , select **New** → **Kotlin Class/File**

6- Type **main** for the file name. Press **Enter** (Return) as illustrated in the figure below.

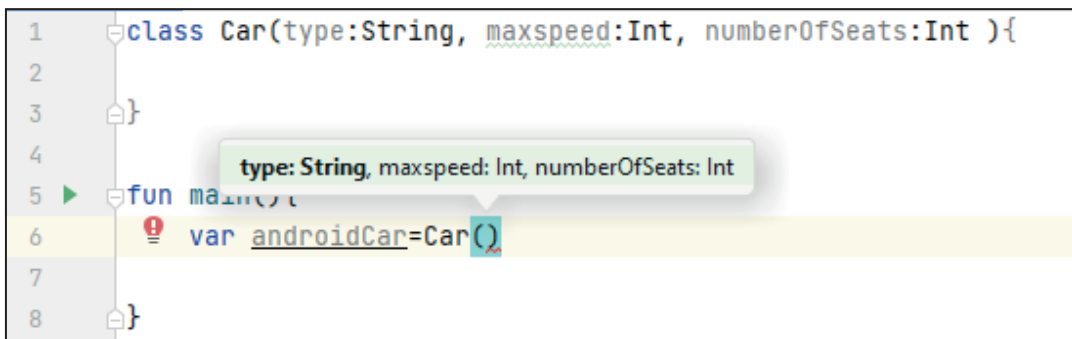


This example shows how you can create a class called **car** step by step:

Step 1: Declare an empty class called “**car**” and some properties (attributes), as illustrated in the following code:

```
class Car(type:String, maxspeed:Int, numberOfSeats:Int ){  
  
}  
fun main() {  
  
}
```

Step 2: When you are trying to create an object depending on this “**car**” class, IntelliJ will display the data type of each class properties as illustrated in the figure below:



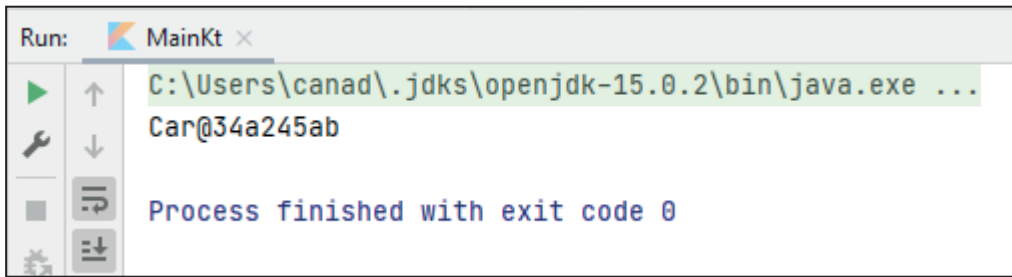
The code is:

```
class Car(type:String, maxspeed:Int, numberOfSeats:Int ){  
  
}  
  
fun main() {  
    var androidCar=Car("Toyota",200,4)  
  
}
```

If you add the **println** method to print the variable **androidCar**, the result will be as follows:

```
class Car(type:String, maxspeed:Int, numberOfSeats:Int ){  
  
}  
  
fun main() {  
    var androidCar=Car("Toyota",200,4)  
    println(androidCar)  
  
}
```

The result is below:



The output result **Car@34a245ab** consists of the class name with the memory address which includes the information about this object. This is not the right way to use the class property. The class properties in Kotlin must be initialized before use. The class properties or fields can be initialized in different ways to the default values when classes are loaded. For example, the initialized default value of an integer property is zero.

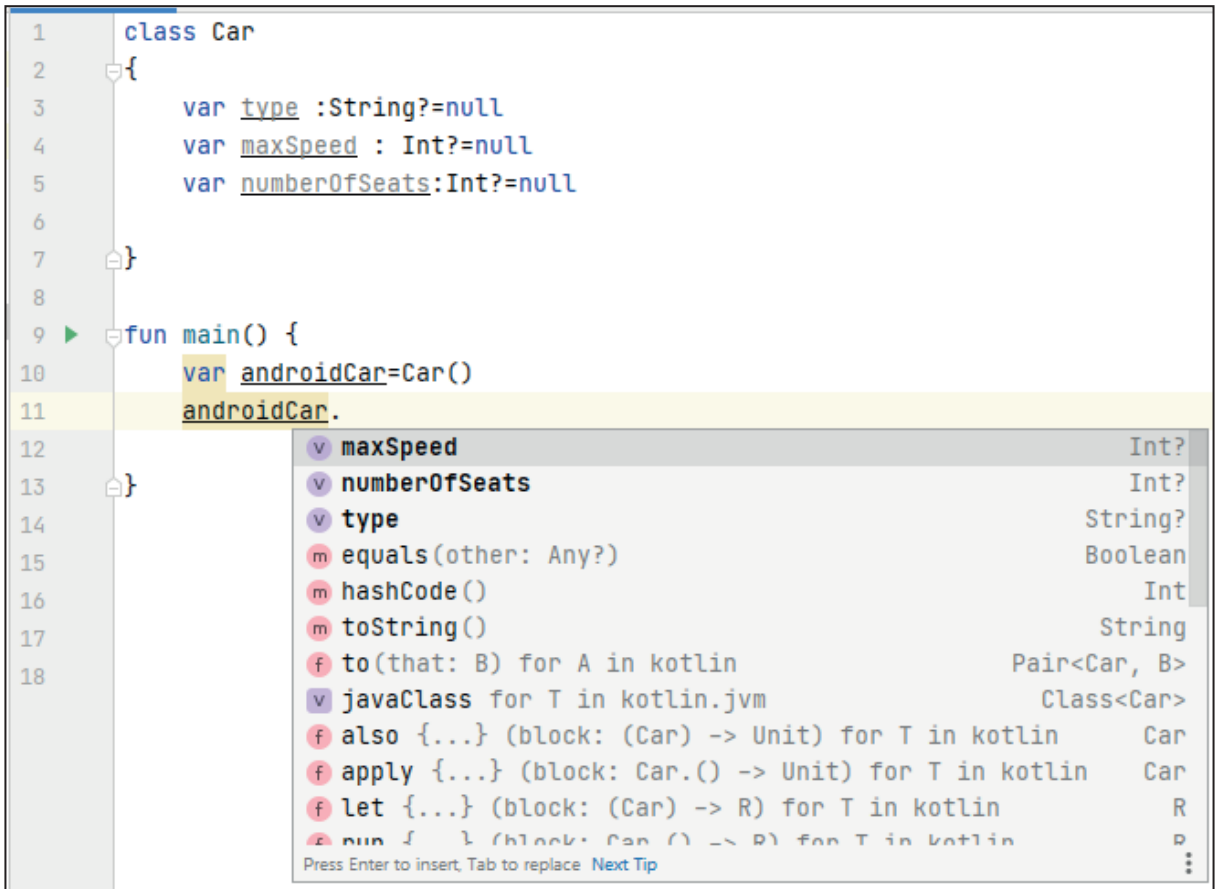
First, you should define the properties as variables with empty values. These variables will have values later when you use this class to define the object. The gray highlighted part of the below codes displays how you can configure the class properties as variables with empty values.

```
class Car
{
    var type :String?=null
    var maxSpeed : Int?=null
    var numberOfSeats:Int?=null
}

fun main() {
    var androidCar=Car()
}
```

Second, you should enter values for this class properties which will be used to create the object **"androidCar"**.

The following screenshot displays how you can configure the value of each class property:



The full code will be as follows:

```

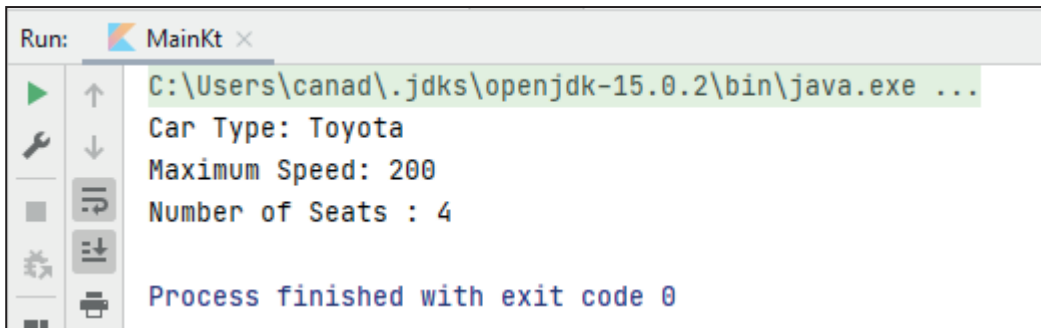
class Car
{
    var type :String?=null
    var maxSpeed : Int?=null
    var numberOfSeats:Int?=null
}

fun main() {
    var androidCar=Car()
    androidCar.type= "Toyota"
    androidCar.maxSpeed=200
    androidCar.numberOfSeats=4

    println("Car Type: ${androidCar.type}")
    println("Maximum Speed: ${androidCar.maxSpeed}")
    println("Number of Seats : ${androidCar.numberOfSeats}")
}

```

The run result is displayed below:



Providing Constructors for your Classes

The constructor is part of the class header; it goes after the class name as illustrated in the figure below:

```
class Student constructor(name: String) {  
    var name: String? = name  
    var college: String? = college  
    var age: Int? = age  
}
```

If the constructor does not have any annotations or visibility modifiers, the constructor keyword can be omitted as illustrated in the following code:

```
class Student(name: String) {  
    var name: String? = name  
    var college: String? = college  
    var age: Int? = age  
}
```

A **constructor** is a concise way to initialize class properties. It is a special member function that is called when an object is created.

Now, we are going to create another class using the **constructor** initializing technique. A class contains constructors that are invoked to create objects from the class blueprint. Constructor declarations are similar to method declarations except that they use the name of the class and have no return type.

Example:

You are going to create a **Student** class as you did in the previous example, but you will declare the class properties (name, college, and age) as illustrated in the code below:

```

/* Created by Android ATC */

class Student(name: String, college: String, age: Int) {
    var name: String? = name
    var college: String? = college
    var age: Int? = age
}

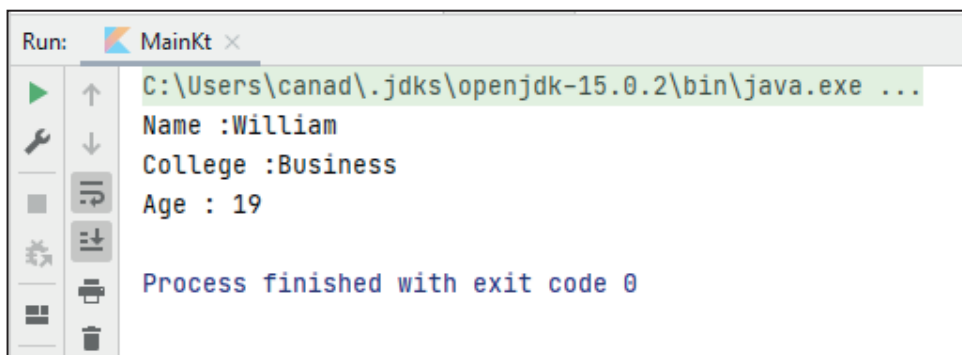
fun main () {

    var itStudent=Student("William","Business",19)
    println("Name :"+ itStudent.name)
    println("College :"+itStudent.college)
    println("Age : "+itStudent.age)

}

```

The run result of this code is displayed below:



Class Inheritance

The idea of inheritance is simple but powerful. When you want to create a new class and there is already a class that includes some of the codes you need, you can derive your new class from the existing class (called parent or super class). While doing this, you can reuse the properties and methods of the existing class without having to write them again.

For example, if you want to create a new class called **Teacher** with the same properties of the previous class **Student**, which is the primary class (parent or super class), you can add the following line to the previous code snippet.

```
class Teacher():Student ()
```

The following is the full code and the inheritance process. However, there are still some additions that must be done to avoid any errors that might occur.


```

/* Created by Android ATC */

class Student(name: String, college: String, age: Int) {
    var name: String? = name
    var college: String? = college
    var age: Int? = age
}

class Teacher ():Student ()

fun main () {

    var itStudent=Student("William","Business",19)
    println("Name :"+ itStudent.name)
    println("College :"+itStudent.college)
    println("Age : "+itStudent.age)

}

```

You will get the following error (underlined in red) because the **Student** class is closed by default.

```
class Teacher ():Student ()
```

To make the **Student** class open and allow any child class to make a copy of its properties, you need to move your mouse pointer over the Student class, then select: **Make 'Student' open** as illustrated in the figure below:

The screenshot shows an IDE with a Kotlin file. The code is as follows:

```

1  /* Created by Android ATC */
2
3  class Student(name: String, college: String, age: Int) {
4      var name: String? = name
5      var college: String? = college
6      var age: Int? = age
7  }
8
9  class Teacher ():Student ()
10
11 fun main () {
12
13     var itStudent=Student("William","Business",19)
14     println("Name :"+ itStudent.name)
15     println("College :"+itStudent.college)
16     println("Age : "+itStudent.age)
17
18 }
19

```

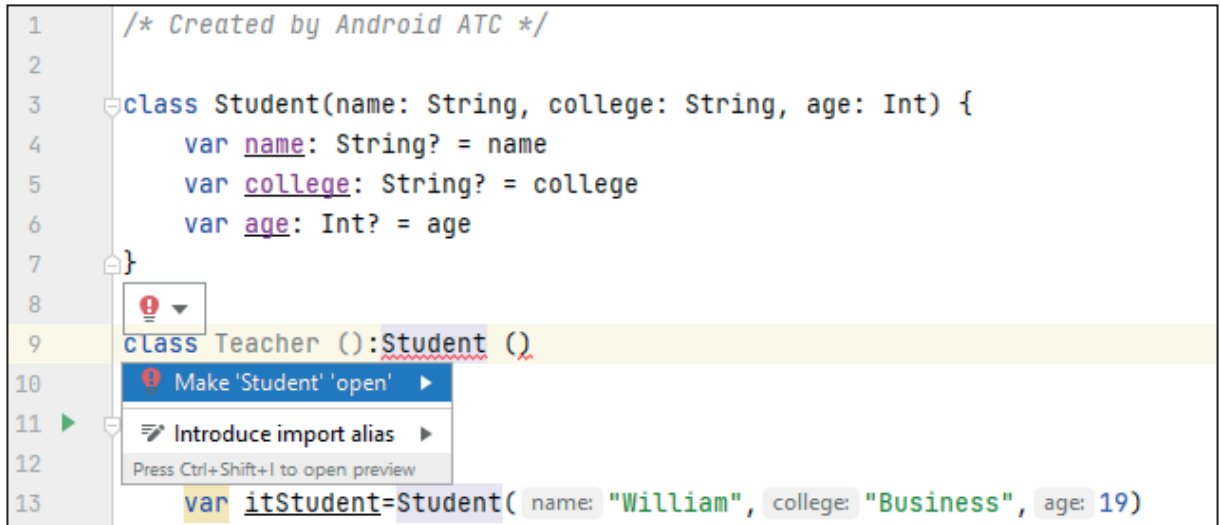
The line `class Teacher ():Student ()` has a red squiggly line under `Student ()`, indicating an error. A context menu is open over this line, displaying the message: "This type is final, so it cannot be inherited from". The menu offers the option "Make 'Student' 'open'" with the keyboard shortcut "Alt+Shift+Enter". Other options include "More actions..." and "Alt+Enter". Below the menu, the details for the `Student` class are shown, including its public constructor.

Or, you may click the Student class, then click the red pop-up warning icon (red lamp) and select : **Make 'Student' open**", as illustrated in the following figure.

```

1  /* Created by Android ATC */
2
3  class Student(name: String, college: String, age: Int) {
4      var name: String? = name
5      var college: String? = college
6      var age: Int? = age
7  }
8
9  class Teacher ():Student ()
10
11  var itStudent=Student( name: "William", college: "Business", age: 19)

```

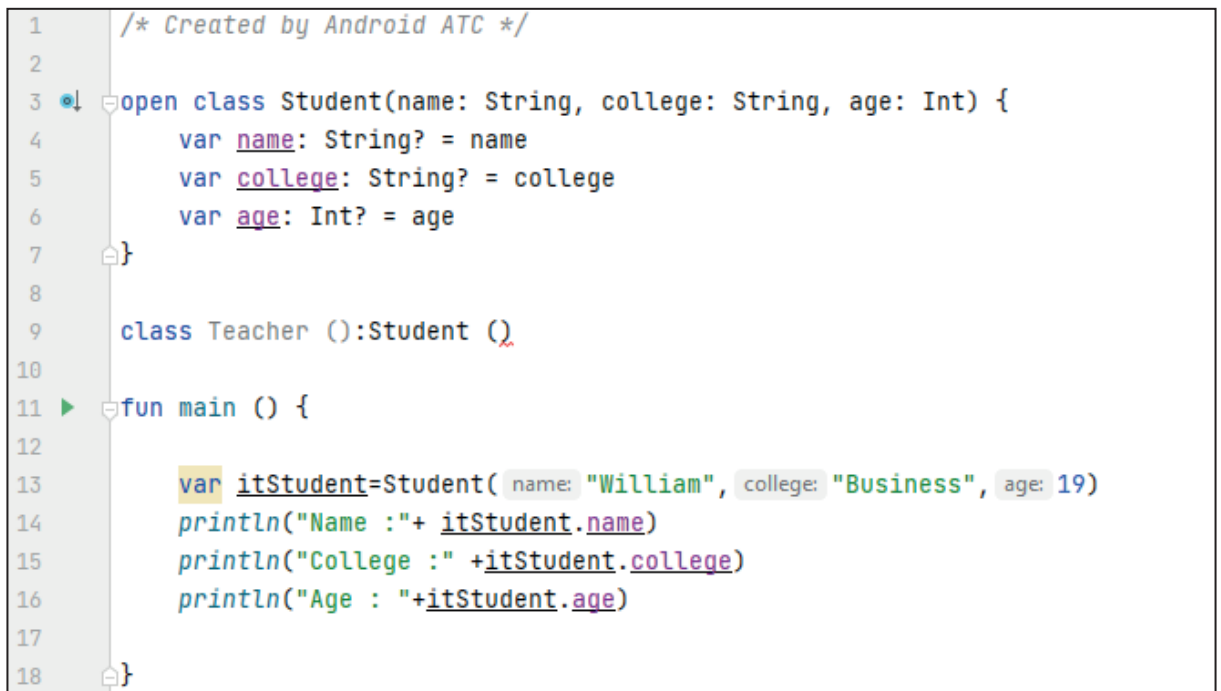


The “**open**” description (keyword) is added automatically to the **Student** class, as illustrated in the following figure to let other classes (sub class) inherit and reuse its properties and methods.

```

1  /* Created by Android ATC */
2
3  open class Student(name: String, college: String, age: Int) {
4      var name: String? = name
5      var college: String? = college
6      var age: Int? = age
7  }
8
9  class Teacher ():Student ()
10
11  fun main () {
12
13      var itStudent=Student( name: "William", college: "Business", age: 19)
14      println("Name :"+ itStudent.name)
15      println("College :"+itStudent.college)
16      println("Age : "+itStudent.age)
17
18  }

```

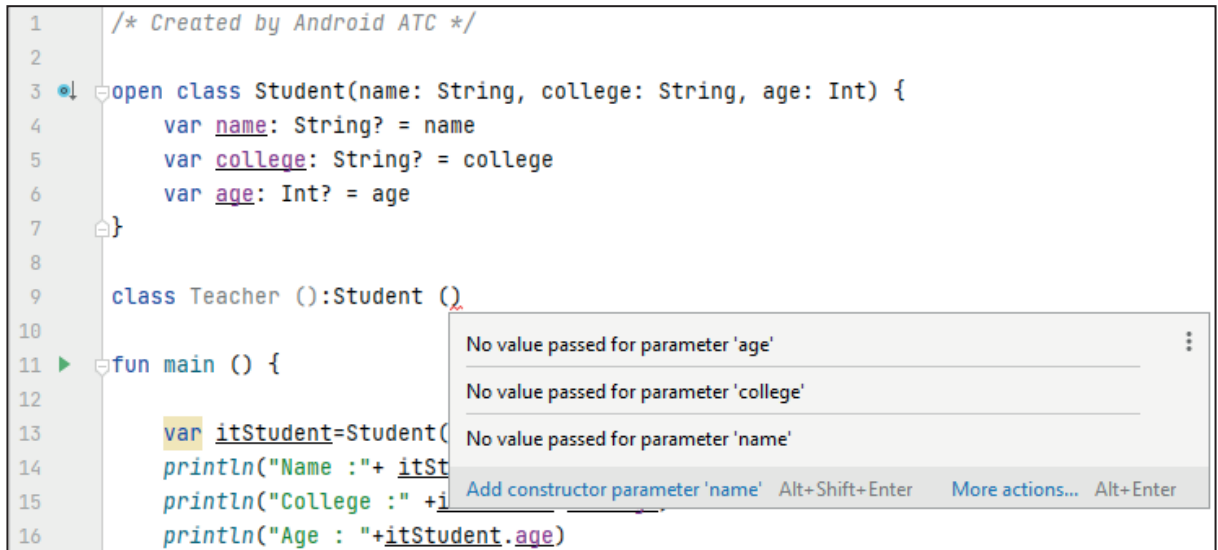


There is still a red line indicating that there is an error in configuring the relation between the class **Student** and the subclass **Teacher**, as illustrated in the following figure. **Student** class still needs to invoke its class constructor (properties):

```

1  /* Created by Android ATC */
2
3  open class Student(name: String, college: String, age: Int) {
4      var name: String? = name
5      var college: String? = college
6      var age: Int? = age
7  }
8
9  class Teacher ():Student ()
10
11 fun main () {
12
13     var itStudent=Student(
14     println("Name :"+ itSt
15     println("College :"+ i
16     println("Age : "+itStudent.age)

```



You should add the class parameters to your class inheritance configuration as follows:

```
class Teacher (name: String, college: String, age: Int):Student (name,college, age)
```

The full code should be as follows:

```

/* Created by Android ATC */

open class Student(name: String, college: String, age: Int) {

    var name: String? = name
        var college: String? = college
        var age: Int? = age
    }

class Teacher (name: String, college: String, age: Int):Student
(name,college,age)

fun main () {

    var itStudent=Student("William","Business",19)
    println("Name :"+ itStudent.name)
    println("College :"+ itStudent.college)
    println("Age : "+itStudent.age)

}

```

Now, the inheritance process is successfully completed.

You can now use the **Teacher** class which has the same or all properties of the **Student** class in your code. For example, you can simply replace the **Student** class with **Teacher** class in the **main()** function as illustrated in the following code:

```

/* Created by Android ATC */

open class Student(name: String, college: String, age: Int) {
    var name: String? = name
    var college: String? = college
    var age: Int? = age
}

class Teacher (name: String, college: String, age: Int):Student
(name,college,age)

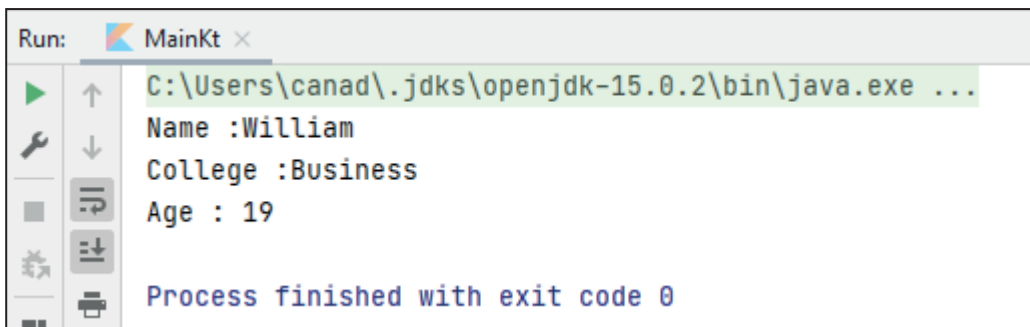
fun main () {

    var itStudent=Teacher("William","Business",19)
    println("Name :"+ itStudent.name)
    println("College :" +itStudent.college)
    println("Age : "+itStudent.age)

}

```

The run result of this code is explained below:



```

Run: MainKt x
C:\Users\canad\.jdk\openjdk-15.0.2\bin\java.exe ...
Name :William
College :Business
Age : 19
Process finished with exit code 0

```

Abstract Class

A class is called abstract when at least one of its members does not have an implementation. It is the responsibility of the derived / child class to provide the implementation of the abstract members of the parent class.

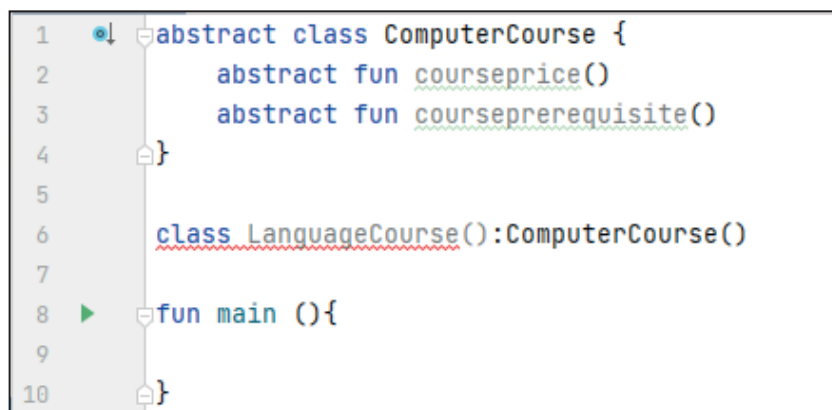
An abstract class needs to be qualified with the abstract keyword. An abstract class cannot be instantiated, and its only purpose is to be inherited. We do not need to annotate an abstract class or function with the open keyword.

Example:

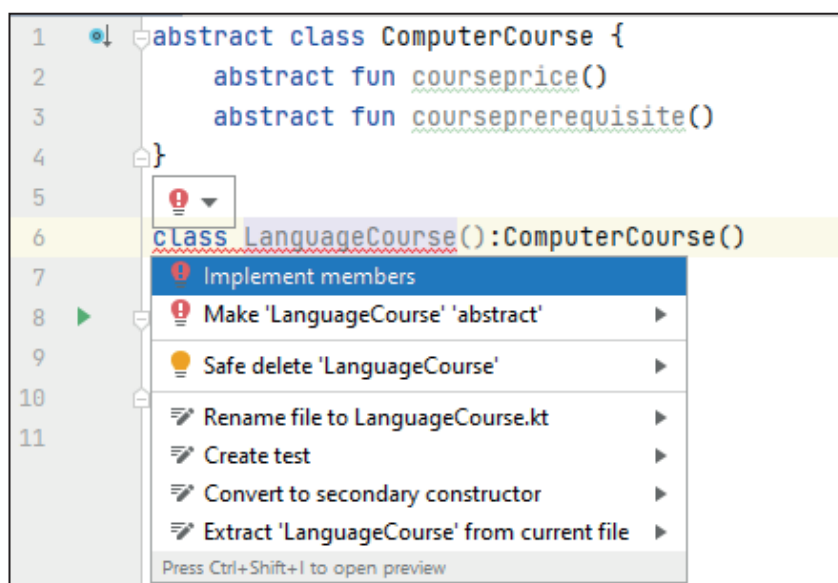
In the following code, the inheritance relation between the parent class (abstract class) **"ComputerCourse"** and the child class **"LanguageCourse"** is implemented without adding the **"open"** keyword at the beginning of the parent class. All implementations will be done within the child class.

```
abstract class ComputerCourse {  
  
}  
  
class LanguageCourse():ComputerCourse()  
  
    fun main () {  
  
}
```

The purpose of creating an abstract class is to have a main (parent) class with some functions (abstract functions) which can be automatically transmitted to be used easily inside the secondary (child function) as illustrated below:



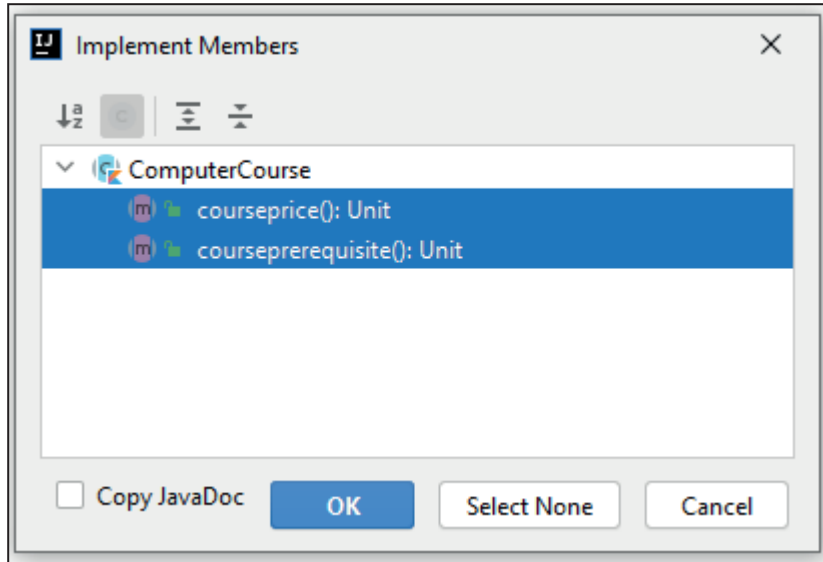
Click on the red underlined **LanguageCourse** class, click the red lamp icon as illustrated in the figure below, then select **Implement members**.



The “**Implement members**” option will import all the abstract functions (methods) in the abstract parent class to the child class.

Select the two methods `courseprice()` and `courseprerequisite()` as illustrated in the figure below.

When you click on “**Implement members**”, a dialog box appears asking you which abstract functions (methods) you want to implement inside the child class “**ComputerCourse**”. Select the two abstract functions and then click **OK**.



You will get the following:

```
1  abstract class ComputerCourse {
2      abstract fun courseprice()
3      abstract fun courseprerequisite()
4  }
5
6  class LanguageCourse():ComputerCourse() {
7      override fun courseprice() {
8          TODO(reason: "Not yet implemented")
9      }
10
11     override fun courseprerequisite() {
12         TODO(reason: "Not yet implemented")
13     }
14 }
15
16 fun main (){
17
18 }
```

Here is the full code after adding a simple method **"println()"** to the child classes:

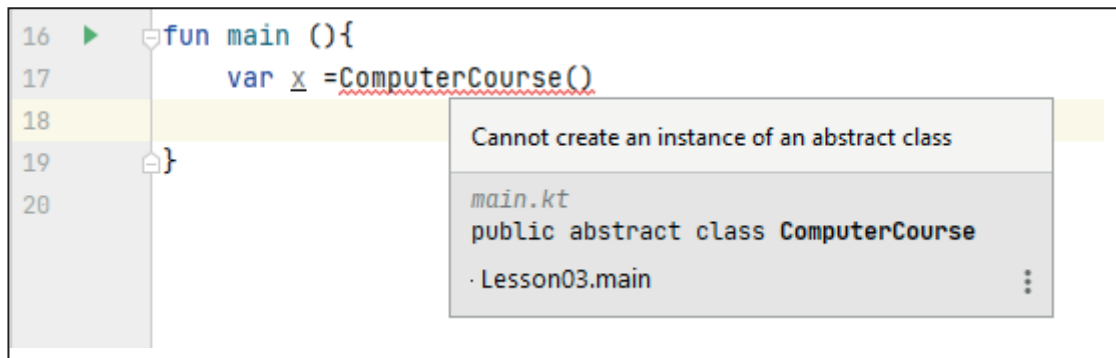
```
abstract class ComputerCourse {
    abstract fun courseprice()
    abstract fun courseprerequisite()
}

class LanguageCourse():ComputerCourse() {
    override fun courseprice() {
        println("Course Price")
    }

    override fun courseprerequisite() {
        println("Course Prerequisite")
    }
}

fun main () {
}
```

If you tried to initiate an object using the abstract class in the main function, you would get an error message displaying that this abstract function could not create an instance of an abstract class as illustrated below:



However, you can initiate an object or instance form the child class which inherited its properties from the abstract class by doing the following:

```
abstract class ComputerCourse {
    abstract fun courseprice()
    abstract fun courseprerequisite()
}
```

```

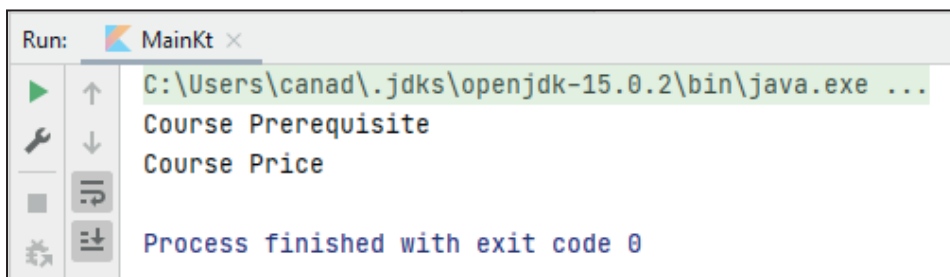
class LanguageCourse():ComputerCourse() {
    override fun courseprice() {
        println("Course Price")
    }

    override fun courseprerequisite() {
        println("Course Prerequisite")
    }
}

fun main () {
    var x =LanguageCourse()
    x.courseprerequisite()
    x.courseprice()
}

```

The run result of this code is:



Interface Class

An interface class is a class that consists of methods or functions without any implementations. You use an interface class when you need to use certain features of a class with another class. Here, the interface class becomes the parent class and the class which uses the features of the interface class is considered the child class.

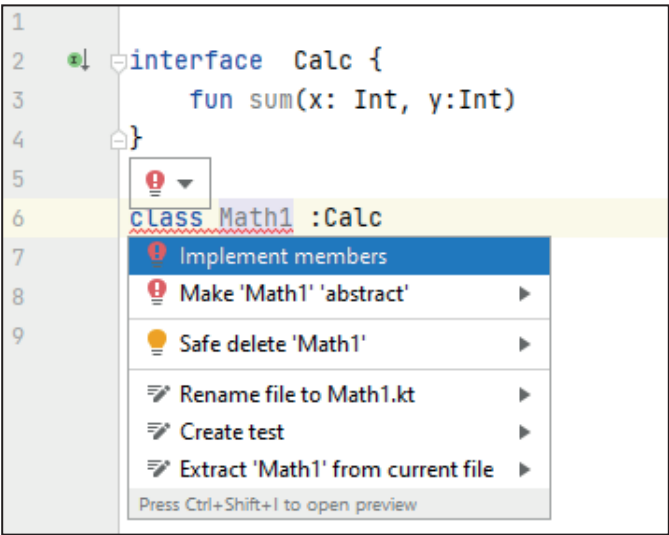
Example: In the following snapshot, the **Calc** interface class includes a method “**sum**” and **Math1** class (child) which will be configured to use the Calc class (parent) features (methods).

```

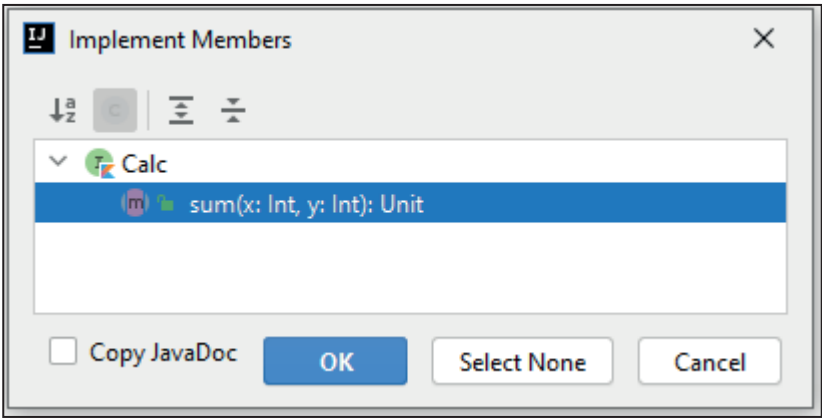
1
2  interface Calc {
3      fun sum(x: Int, y:Int)
4  }
5
6  class Math1 :Calc
7

```

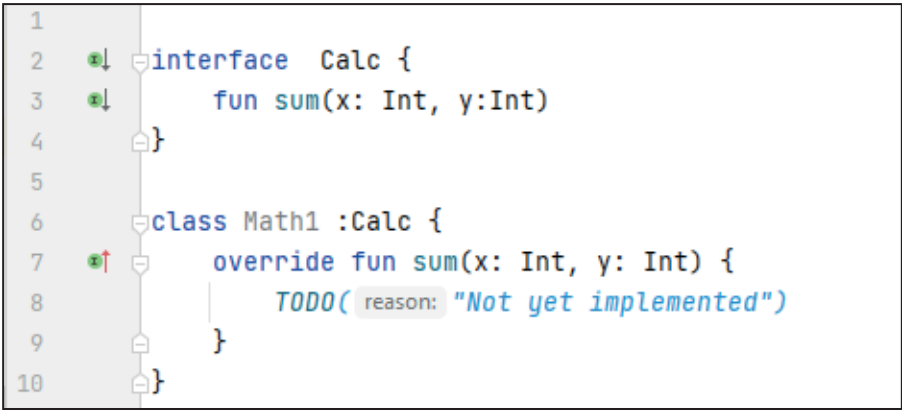

In the following figure, click the red bulb to import the methods of the interface class “Calc”.
Select the “**Implement members**” option :



As illustrated below, select the **sum** method, then click **OK**.



The implementation result will be as follows:



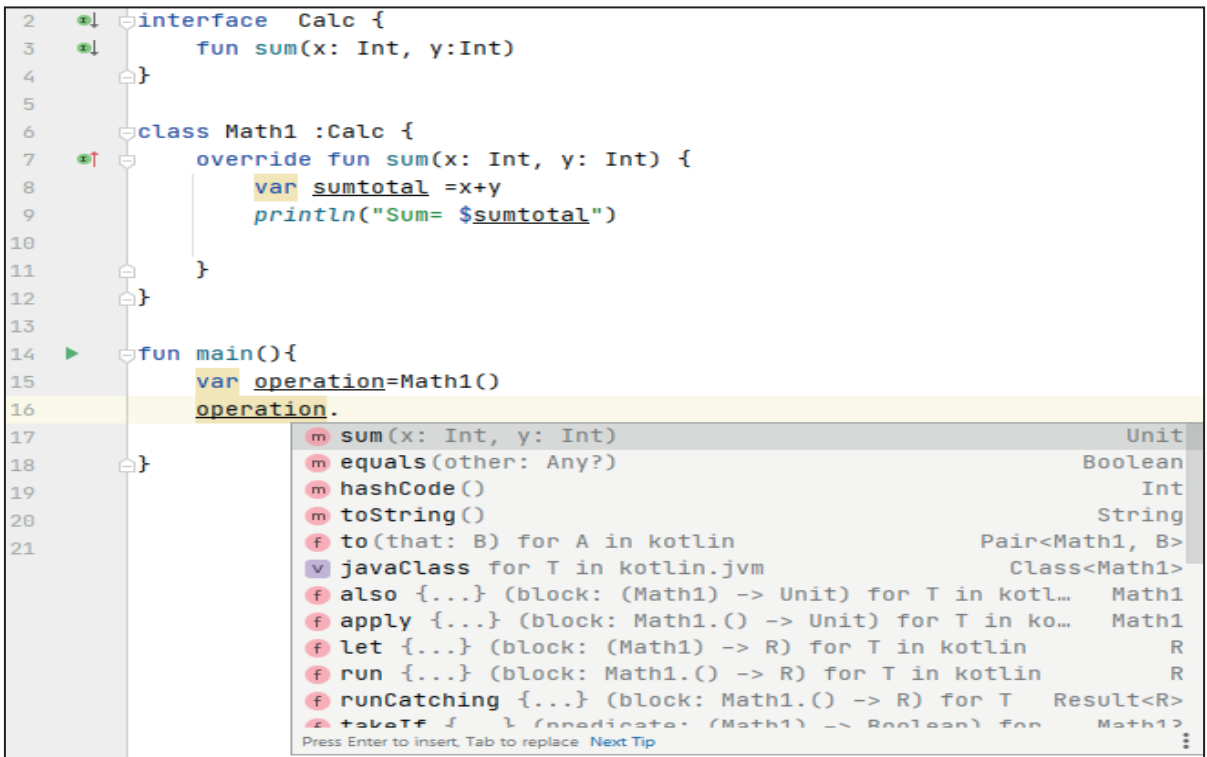
Add the **sumtotal** variable with print method to the **Math1** class body as illustrated in the code below:

```
interface Calc {
    fun sum(x: Int, y: Int)
}

class Math1 : Calc {
    override fun sum(x: Int, y: Int) {
        var sumtotal = x + y
        println("Sum= $sumtotal")
    }
}

fun main() {
}
```

You can use the methods which you have imported to Math1 class from the interface class with other methods as illustrated in the following figure:



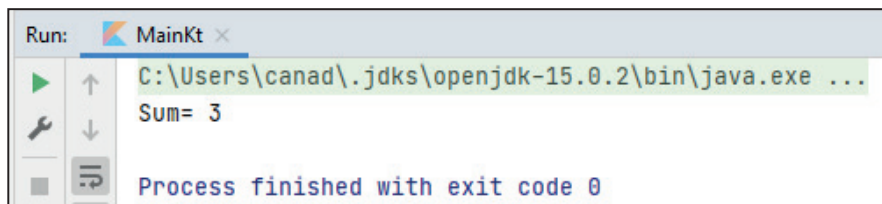
The code is as following:

```
interface Calc {
    fun sum(x: Int, y: Int)
}

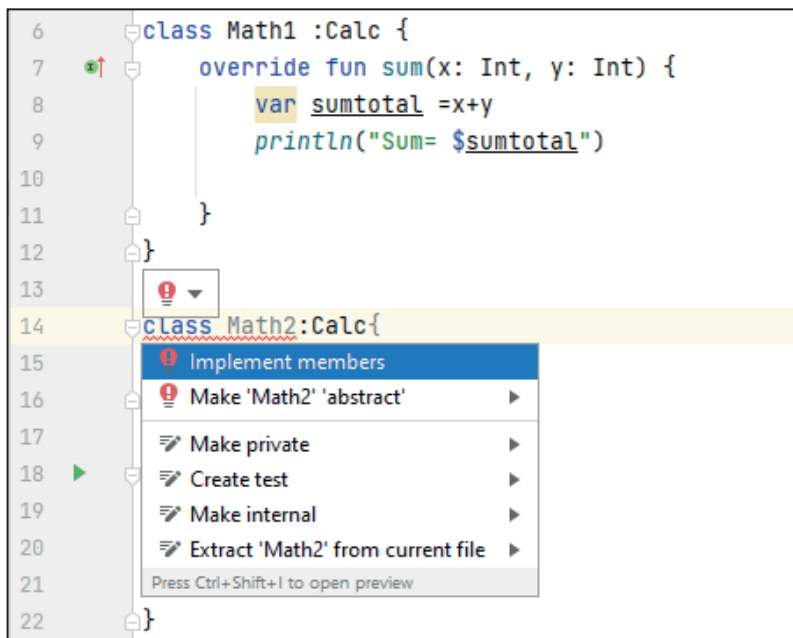
class Math1 : Calc {
    override fun sum(x: Int, y: Int) {
        var sumtotal = x + y
        println("Sum= $sumtotal")
    }
}

fun main() {
    var operation = Math1()
    operation.sum(1, 2)
}
```

The run result of this code is as follows:



If you configure another inheritance between the new subclass “**Math2**” and the interface class **Calc**, you will also be asked to add the implementation methods to this subclass, as demonstrated in the previous example. The following snapshot displays the new inheritance configuration:



The full code is:

```
interface Calc {
    fun sum(x: Int, y: Int)
}

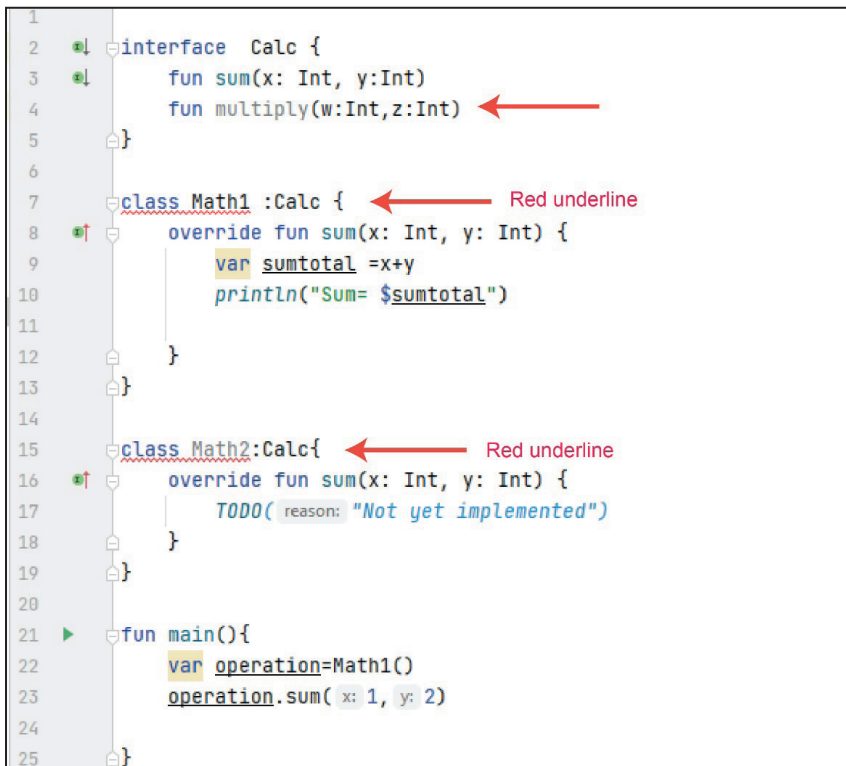
class Math1 : Calc {
    override fun sum(x: Int, y: Int) {
        var sumtotal = x + y
        println("Sum= $sumtotal")
    }
}

class Math2: Calc {
    override fun sum(x: Int, y: Int) {
        TODO("Not yet implemented")
    }
}

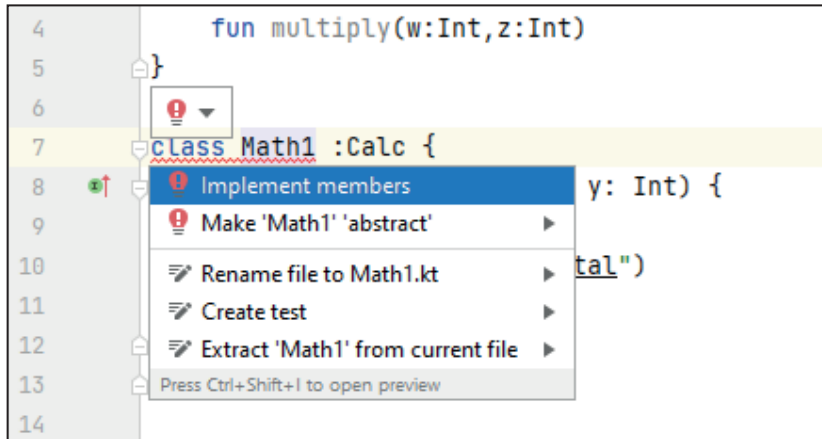
fun main() {
    var operation = Math1()
    operation.sum(1, 2)
}
```

If you add **fun multiply(w: Int, z: Int)** to the interface class, what happens to the subclasses **"Math1"** and **"Math2"**?

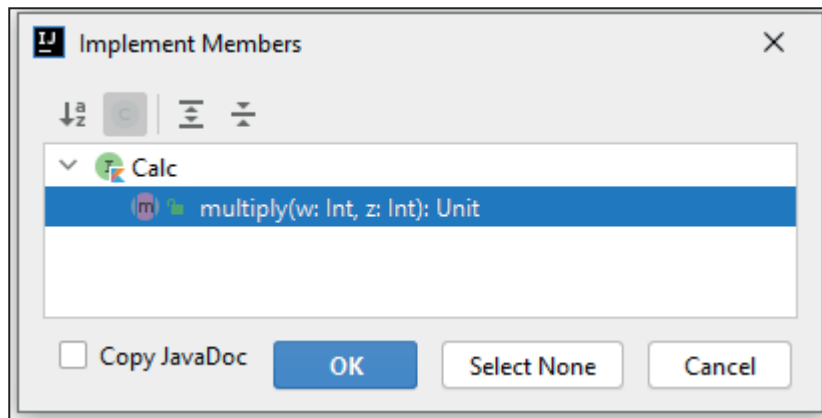
You will get an error message (code underlined in red) asking you to implement these new changes within the subclass as illustrated below:



If you click on Math1 class again, click the red bulb, then select **Implement members** as illustrated in the figure below:



Select **multiply** method, and click **OK** as illustrated in the figure below:



Repeat the same previous step to implement the multiply method for **"Math2"**.

The full code is:

```
interface Calc {
    fun sum(x: Int, y: Int)
    fun multiply(w: Int, z: Int)
}

class Math1 : Calc {
    override fun sum(x: Int, y: Int) {
        var sumtotal = x + y
        println("Sum= $sumtotal")
    }

    override fun multiply(w: Int, z: Int) {
        TODO("Not yet implemented")
    }
}
```

```

    }
}

class Math2:Calc{
    override fun sum(x: Int, y: Int) {
        TODO("Not yet implemented")
    }

    override fun multiply(w: Int, z: Int) {
        TODO("Not yet implemented")
    }
}

fun main(){
    var operation=Math1()
    operation.sum(1,2)
}

```

We conclude that the interface class works as a main class without any implementation and all subclasses whose implementation depends on the methods which belong to the interface class.

Generic Class

A generic class allows a single function to handle many types of different data as if there was a copy of the function defined for each data type. This reduces code duplication. As a result, you will be able to create various objects that have different types of data and content using the same generic class.

Example:

In the following code, the class **"Permission"** includes two parameters; each one of them is a **"String"** data type.

```
class Permission(userName:String, password:String)
```

If you want to make the data type of these parameters **"userName"** and **"password"** fixable so that they can accept any data type especially when creating objects, you can use the class **"Permission"** as a **generic** class. Generic types are defined in the **< >** block right after the class name.

The following code shows how you can write a generic code:

```
class Permission <T> {  
    var userName:T?=null  
    var password:T?=null  
    var iD:T?=null  
}
```

In the following example, we will use the generic class "**Permission**" to create object "**x**" using the String data type and the same class "**Permission**" to create object "**y**" using the Integer data type.

```
class Permission <T> {  
    var userName:T?=null  
    var password:T?=null  
    var iD:T?=null  
}  
  
fun main () {  
  
    val x =Permission<String> ()  
    x.userName="william@androidatc.com"  
    x.password= "Toronto@123"  
    println("User Name: ${x.userName}   & Password: ${x.password}")  
  
    val y =Permission<String> ()  
    y.userName="george@gmail.com"  
    y.password=123456.toString()  
    println("User Name: ${y.userName}   & Password: ${y.password}")  
  
    val z=Permission<Int>()  
    z.iD=2022  
    println("User ID: ${z.iD} ")  
}
```

The run output of this code is as follows:



```
Run: MainKt x  
C:\Users\canad\.jdk\openjdk-15.0.2\bin\java.exe ...  
User Name: william@androidatc.com & Password: Toronto@123  
User Name: george@gmail.com & Password: 123456  
User ID: 2022  
Process finished with exit code 0
```

Class Variables

When a number of objects are created from the same class blueprint, each object will have its own distinct instance variables. In the case of the previous “Student” class, the instance variables are “Name”, “Age” and “College”. Each object “Student” has its own values of these variables stored in different locations in the memory.

Sometimes, you want to have variables that are common to all objects. This is accomplished with the companion object. These are objects declared inside classes and they can be used in Kotlin to create and access static variables.

The following code snippet creates a static variable (i.e. class variable) in Kotlin:

```
class Game {
    companion object {
        val gamesPlayed = 10
    }
}
```

The variable **gamesPlayed** is a static variable that can be shared among all object instances of the class **Game**. Since this is a class variable, you can access it using the class name instead of using the class instance name.

The following line of code shows how to access a static variable in Kotlin:

```
class Game {
    companion object {
        val gamesPlayed = 10
    }
}

fun main() {
    println(Game.gamesPlayed)
}
```

Member Variables

In Kotlin, we have three types of variables: public, private, and protected variables.

1- Public variables

All variables are public by default (implicitly public) and are accessible from all classes.

2- Private variables

These variables are accessible only within their own class.

3- Protected variables

These variables are a version of public variables restricted only to subclasses. This means that only the current class and its subclasses will have access to the field or method of the protected class.

Example:

The following code includes a class called **"Airplane"**. This class has two variables: **"type"** and **"capacity"**. There is also another subclass (child class) called **"Bus"** which inherited the properties of the class **"Airplane"** as illustrated below:

```
open class Airplane() {
    var type: String? = null
    var capacity: Int? = null
}

class Bus():Airplane()

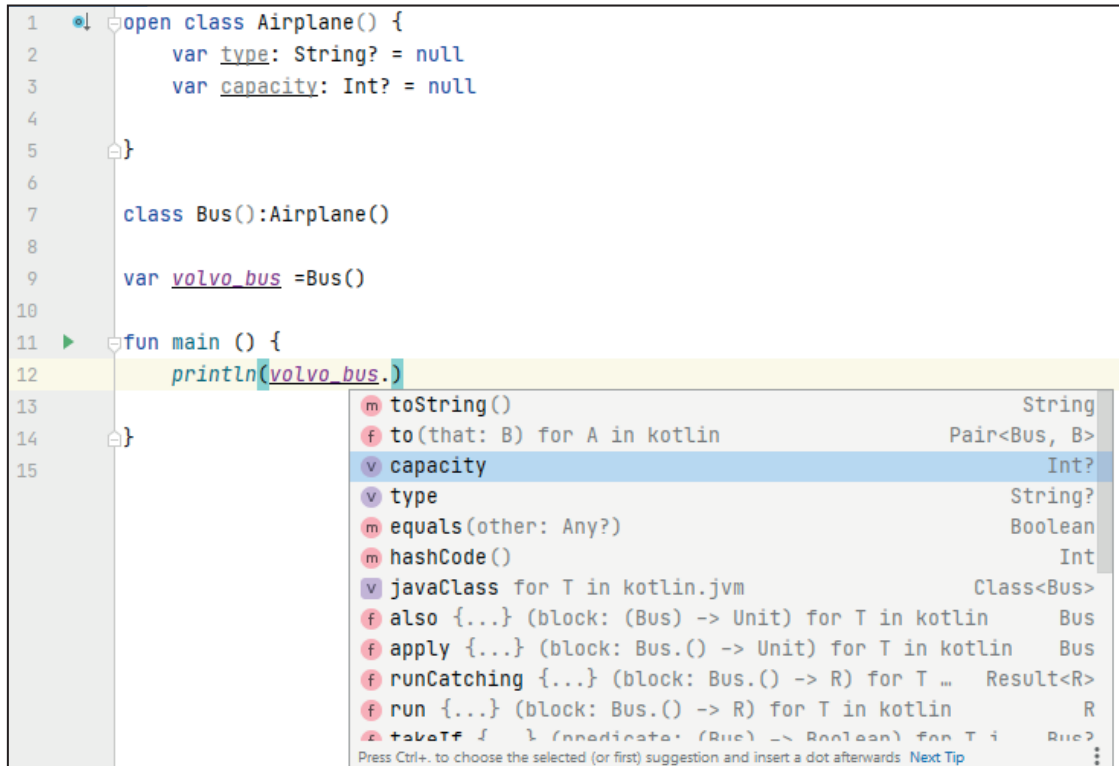
var volvo_bus =Bus()
fun main () {

}
```

The variables **"type"** and **"capacity"** in the previous code are public by default, and there is no need to add a modifier to declare them as public. But if you want to do this, then the top part of the code will be as follows:

```
open class Airplane() {
    var type: String? = null
    var capacity: Int? = null
}
```

Because these variables are public, we can use them in class **"Bus"**, as illustrated below:



However, if you only change the variables type of “**capacity**” variable to **protected**, this variable will be available only within its class or sub classes. The following screenshot displays that the variable “**capacity**” is not available in the main function.



But if you change the variables (type and capacity) type to **private** as illustrated in the code below, these variables will be available only within their class.

```

open class Airplane() {
    private var type: String? = null
    private var capacity: Int? = null
}

```

The figure below displays that the variables “**capacity**” and “**type**” are not available to use with another class.

```

1  open class Airplane() {
2      private var type: String? = null
3      private var capacity: Int? = null
4
5  }
6
7      class Bus():Airplane()
8
9      var volvo_bus =Bus()
10
11  fun main () {
12      println(volvo_bus)
13
14  }
15

```

IntelliJ IDEA autocomplete suggestions for `println(volvo_bus)`:

- m toString() String
- m equals(other: Any?) Boolean
- m hashCode() Int
- f to(that: B) for A in kotlin Pair<Bus, B>
- v javaClass for T in kotlin.jvm Class<Bus>
- f also {...} (block: (Bus) -> Unit) for T in kotlin Bus
- f apply {...} (block: Bus.() -> Unit) for T in kotlin Bus
- f runCatching {...} (block: Bus.() -> R) for T ... Result<R>
- f run {...} (block: Bus.() -> R) for T in kotlin R
- f takeIf {...} (predicate: (Bus) -> Boolean) for T i... Bus?
- f takeUnless {...} (predicate: (Bus) -> Boolean) for... Bus?
- f let {...} (block: (Bus) -> R) for T in kotlin R

Press Ctrl+. to choose the selected (or first) suggestion and insert a dot afterwards [Next Tip](#)

Kotlin Collections

The Kotlin collections framework is a set of classes and interfaces that implement commonly reusable collection data structures. Although it is referred to as a framework, it works as a library. The Kotlin collections provide both interfaces that define various collections and classes that implement them.

Following are some examples of Kotlin collections:

- Hashmaps
- ArrayList
- listof and mutableListOf

Hashmaps

This class uses a hash table to implement the Map interface. This saves the execution time of basic operations, such as `get ()` and `put ()` to remain constant even for large sets.

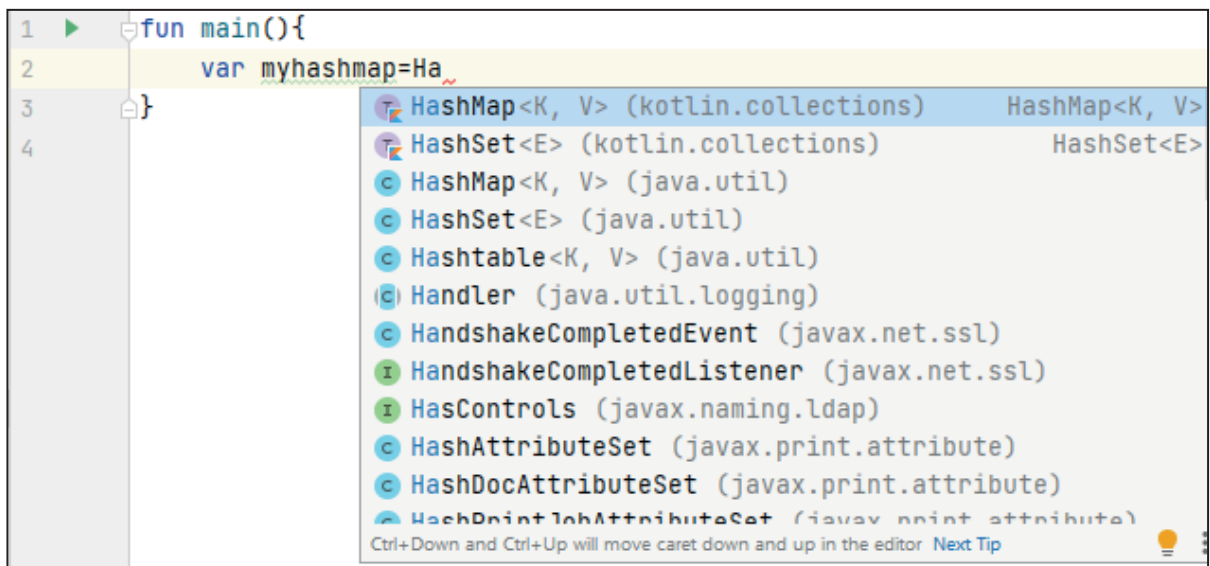
Each value can be represented by a key. For example, the following hash table includes keys (any word) and the values they represent.

Key	Value
Ok	Okay
Y	Yes
a	Android
Abc	Google

Example:

The following example explains how you can use a **HashMap** class to implement the previous hash table.

The following screenshot illustrates how to write the **HashMap** class in Kotlin:



In the following code, the data type of the keys and values in HashMap class are string .

```
fun main() {
    var myhashmap=HashMap<String, String>()
}
```

```
fun main() {
    var myhashmap=HashMap<String, String>()
}
```

↑ ↑
Key Value

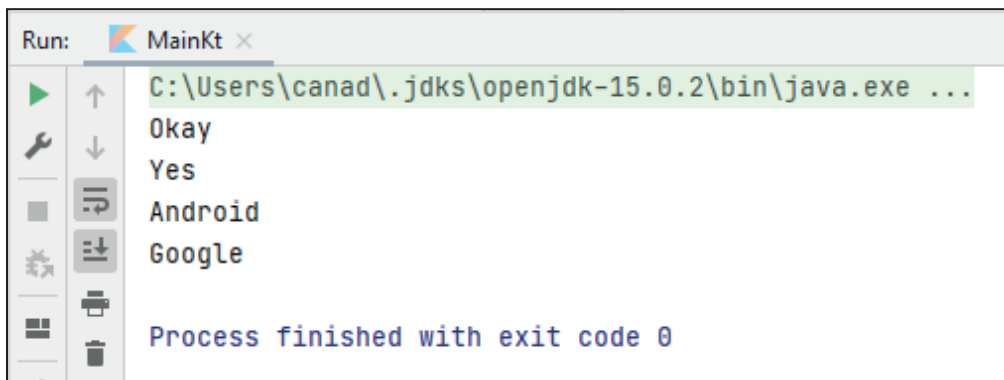
Now, to enter the keys and values of the previous hash table, you should use **put ()** method as illustrated in the following code:

```
fun main() {  
    var myhashmap=HashMap<String, String>()  
    myhashmap.put("Ok", "Okay")  
    myhashmap.put("Y", "Yes")  
    myhashmap.put("a", "Android")  
    myhashmap.put("Abc", "Google")  
}
```

The **get ()** method is used to get the hash table content as illustrated in the code below:

```
fun main() {  
    var myhashmap=HashMap<String, String>()  
    myhashmap.put("Ok", "Okay")  
    myhashmap.put("Y", "Yes")  
    myhashmap.put("a", "Android")  
    myhashmap.put("Abc", "Google")  
  
    println(myhashmap.get("Ok"))  
    println(myhashmap.get("Y"))  
    println(myhashmap.get("a"))  
    println(myhashmap.get("Abc"))  
}
```

The run result is:



There is another way to get all the hash table content by using **for** loop as illustrated in the code below:

```

fun main() {
    var myhashmap=HashMap<String, String>()
    myhashmap.put("Ok", "Okay")
    myhashmap.put("Y", "Yes")
    myhashmap.put("a", "Android")
    myhashmap.put("Abc", "Google")

    for (x in myhashmap.keys) {
        println(myhashmap.get(x))
    }
}

```

The run result will be the same as the previous one.

Note: You can use **"Any"** data type instead of **"String"**. If you want to use any data type, the code will be:

```
var myhashmap=HashMap<Any, Any>()
```

Also, you can enter the keys and values to the hash table directly using the following code:

```

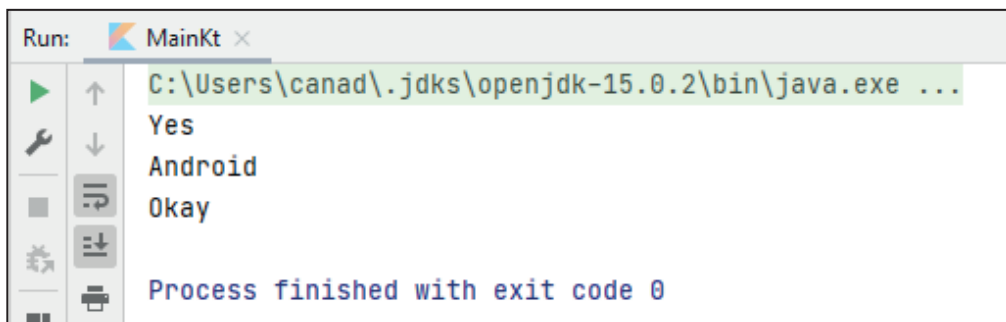
fun main () {

    var myhashmap =hashMapOf <String, String>("Ok" to "Okay","Y"
to "Yes","a" to "Android" )

    for (x in myhashmap.keys) {
        println(myhashmap.get(x))
    }
}

```

The run result is:



ArrayList

The **ArrayList** class implements all optional list operations and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. Array lists are created with an initial size. When this size extends, the collection is automatically enlarged. When objects are removed, the array may shrink in size.

Example:

The following example shows how to create an **ArrayList** class and how to extend or shrink its content using different methods.

The variable **myArrayList** represents an empty array; all its content is configured as integer values.

```
fun main () {  
  
    var myArrayList=ArrayList<Int>()  
}
```

You can add array values using “**add**” method at the specified position in the list as illustrated in the code below.

```
fun main () {  
  
    var myArrayList=ArrayList<Int>()  
    myArrayList.add(20)  
}
```

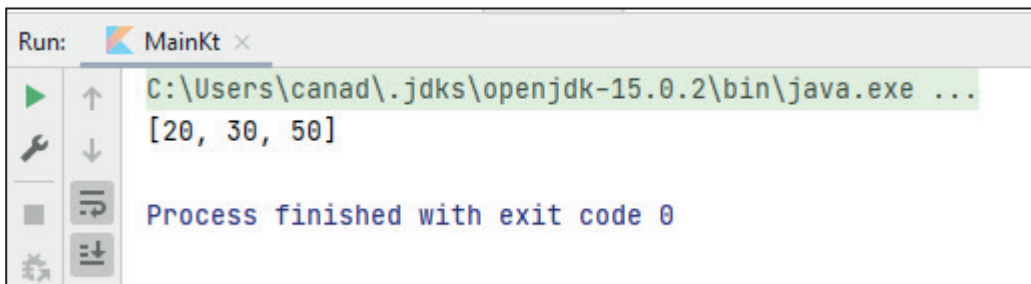
The **myArrayList.add(20)** command here assigned value 20 for the **index [0]** of the array. You can repeat this command many times to add different values to this array. This means that these methods or functions help us extend the size of this array. Here is the full code:

```
fun main () {  
  
    var myArrayList=ArrayList<Int>()  
    myArrayList.add(20)  
    myArrayList.add(30)  
    myArrayList.add(50)  
}
```

To print the content of this array, use the following **println** command:

```
fun main () {  
  
    var myArrayList=ArrayList<Int>()  
    myArrayList.add(20)  
    myArrayList.add(30)  
    myArrayList.add(50)  
  
    println(myArrayList)  
  
}
```

The run result of this code is:

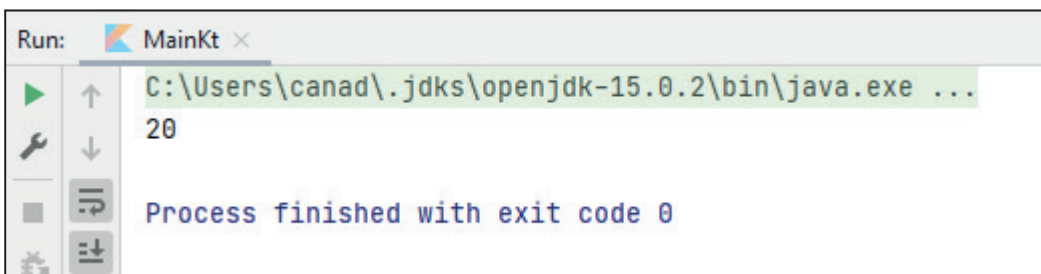


The screenshot shows the 'Run' window in Android Studio. The title bar says 'Run: MainKt x'. The command line shows 'C:\Users\canad\.jdk\openjdk-15.0.2\bin\java.exe ...'. The output shows '[20, 30, 50]'. Below the output, it says 'Process finished with exit code 0'.

If you want to print a specific array element, use the following **get** method to print the **index [0]** of this array as illustrated in the code below:

```
fun main () {  
  
    var myArrayList=ArrayList<Int>()  
    myArrayList.add(20)  
    myArrayList.add(30)  
    myArrayList.add(50)  
  
    println(myArrayList.get(0))  
  
}
```

The run output is:

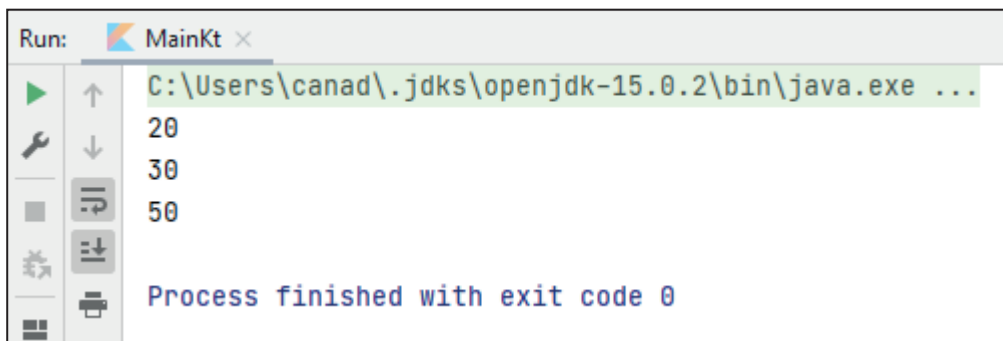


The screenshot shows the 'Run' window in Android Studio. The title bar says 'Run: MainKt x'. The command line shows 'C:\Users\canad\.jdk\openjdk-15.0.2\bin\java.exe ...'. The output shows '20'. Below the output, it says 'Process finished with exit code 0'.

Also, you can use **for** loop to print all the array elements:

```
fun main () {  
  
    var myArrayList=ArrayList<Int>()  
    myArrayList.add(20)  
    myArrayList.add(30)  
    myArrayList.add(50)  
  
    for (number in myArrayList) {  
        println(number)  
    }  
  
}
```

The run result is:



Or you can use the **for** loop in a different way as illustrated in the code below:

```
fun main () {  
  
    var myArrayList=ArrayList<Int>()  
    myArrayList.add(20)  
    myArrayList.add(30)  
    myArrayList.add(50)  
  
    for (index in 0..myArrayList.size-1) {  
        println(myArrayList[index])  
    }  
  
}
```

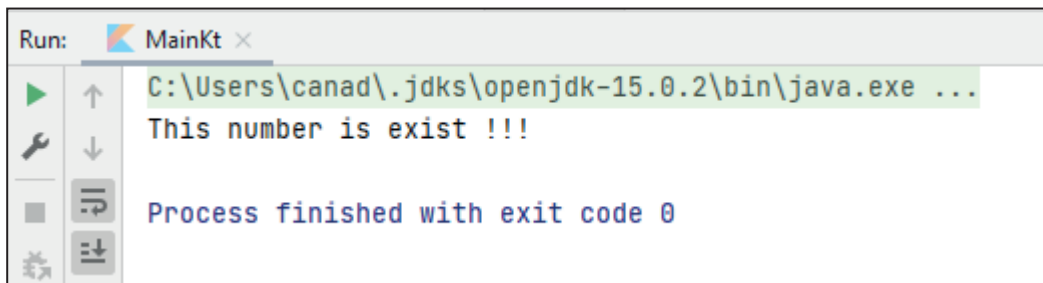
The run result is the same as previous run result.

Now, you can use the **Arraylist** class with the methods that help you extend the size of the array or make its size dynamic. This gives more flexibility in using arrays.

You can use the **contains** method with the **ArrayList** class to check if this array includes a specific element as illustrated in the code below:

```
fun main () {  
  
    var myArrayList=ArrayList<Int>()  
    myArrayList.add(20)  
    myArrayList.add(30)  
    myArrayList.add(50)  
  
    if (myArrayList.contains(30)) {  
        println("This number is exist !!!")  
    }  
}
```

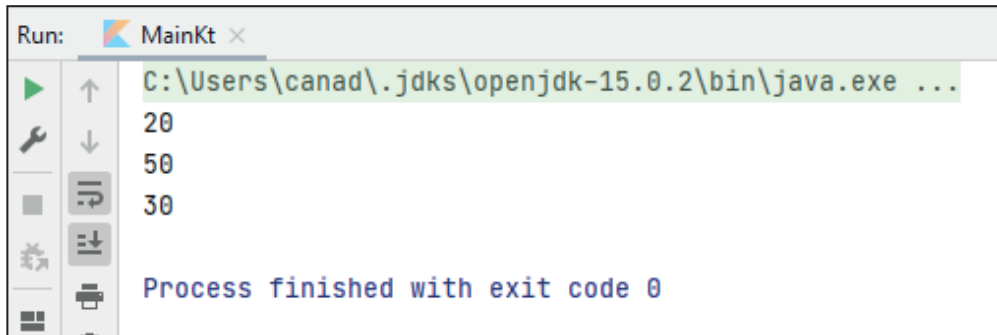
The run result is:



Or you can use the **remove** method with the **ArrayList** class to remove the content of all or some array elements as illustrated in the code below:

```
fun main () {  
  
    var myArrayList=ArrayList<Int>()  
    myArrayList.add(20)  
    myArrayList.add(30)  
    myArrayList.add(50)  
    myArrayList.add(30)  
  
    myArrayList.remove(30)  
  
    for (index in 0..myArrayList.size-1) {  
        println(myArrayList[index])  
    }  
}
```

Here, the **index [1]** and **index [3]** have elements equal to 30, and the remove method will remove the first element only which is equal to 30. The run result of this code is as follows:



```
Run: MainKt x
C:\Users\canad\.jdk\openjdk-15.0.2\bin\java.exe ...
20
50
30
Process finished with exit code 0
```

If you repeat the command `myArrayList.remove(30)` one more time, the second value will also be removed.

You can use the **set** method with the **ArrayList** class to add an element to your array as illustrated in the code below.

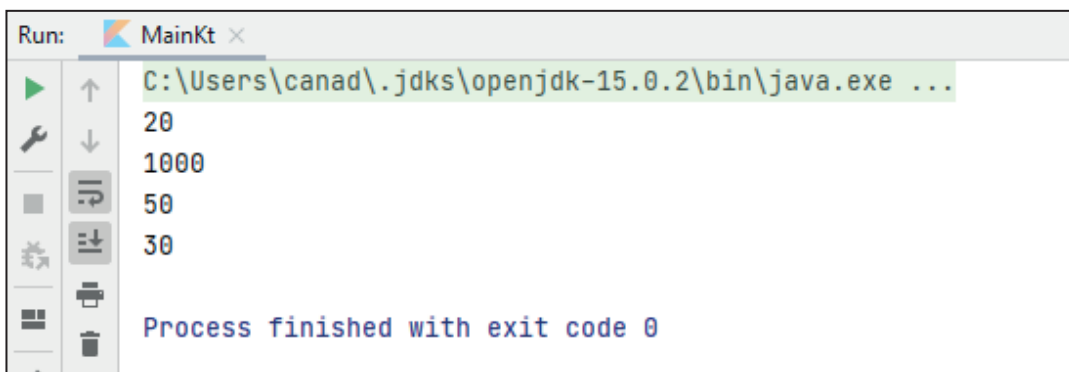
```
fun main () {

    var myArrayList=ArrayList<Int>()
    myArrayList.add(20)
    myArrayList.add(30)
    myArrayList.add(50)
    myArrayList.add(30)

    myArrayList.set(1,1000)

    for (index in 0..myArrayList.size-1) {
        println(myArrayList[index])
    }
}
```

Here, **set** method added the element value 1000 at index [1]. The run result of this code is as follows:



```
Run: MainKt x
C:\Users\canad\.jdk\openjdk-15.0.2\bin\java.exe ...
20
1000
50
30
Process finished with exit code 0
```

You can use the **Arrayof** class, as well, to do what is illustrated in the following code:

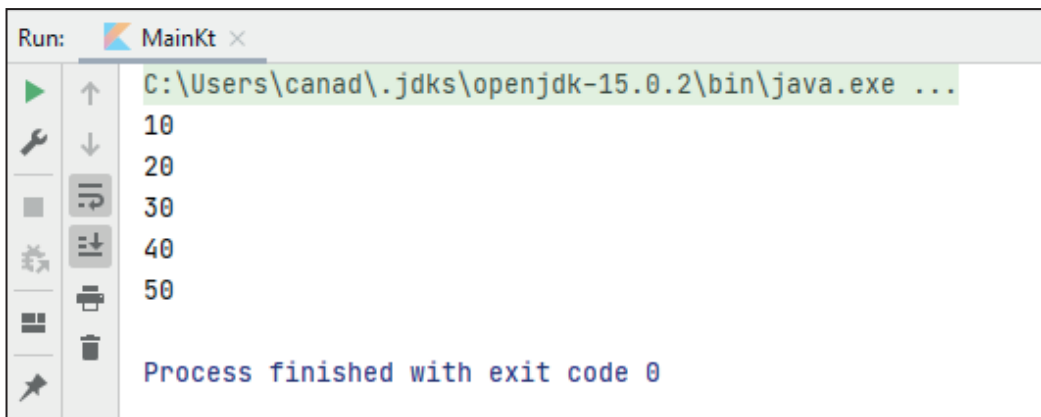
```
var x =arrayOf(10,20,30,40,50)

fun main () {

    for (index in x){
        println(index)
    }

}
```

The run result is as follows:



listof and mutableListOf

These two methods are used to store elements in a specified order and provide indexed access to them.

The difference between them is that the **mutableListOf** method supports adding and removing elements.

You can use the **listof** method to store elements as illustrated in the code below:

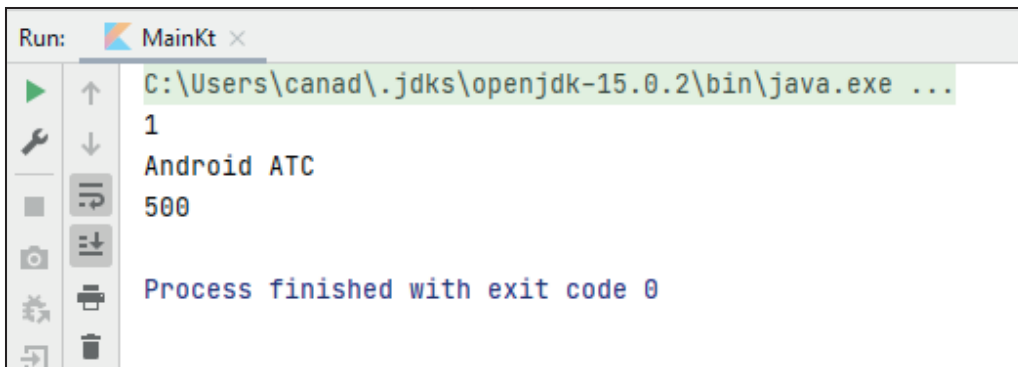
```
fun main () {

    var mylist= listOf <Any>(1,"Android ATC",500)

    for (index in 0..mylist.size-1) {
        println(mylist[index])
    }

}
```

The run result of this code is as follows:



By default, the values of the **listOf** elements are immutable; therefore, you cannot change them.

If you tried to update any value of list elements in this **listOf** method, you will get an error as illustrated in the following screenshot:



To make this list a mutable one, replace the **listof** method with **mutableListof** method, as illustrated in the following code:

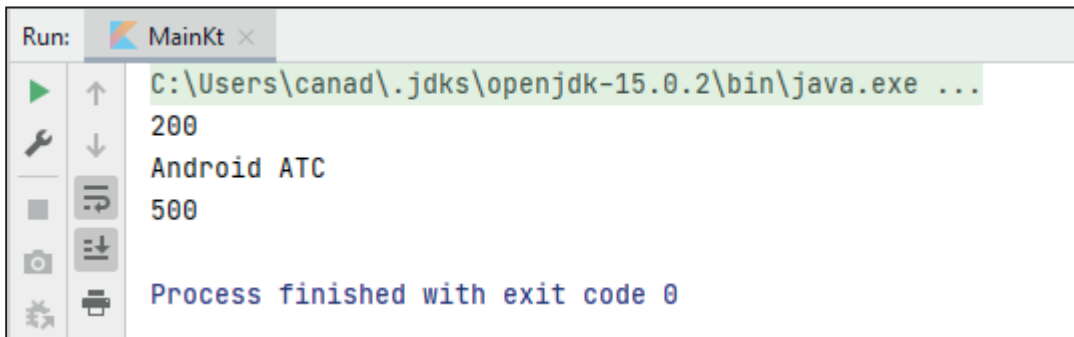
```
fun main () {

    var mylist= mutableListof <Any>(1,"Android ATC",500)

    mylist[0]=200

    for (index in 0..mylist.size-1) {
        println(mylist[index])
    }
}
```

The run result is:



```
Run: MainKt x
C:\Users\canad\.jdk\openjdk-15.0.2\bin\java.exe ...
200
Android ATC
500
Process finished with exit code 0
```