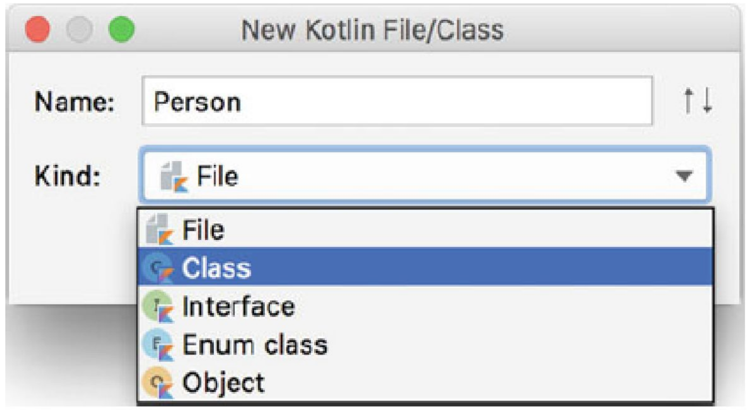
Objects and Classes in Kotlin

Lab Session 3

Just like many other modern advanced programming
languages, Kotlin is object-oriented

To create a class say Person
right click com.example.<your package> ->New->Kotlin File/Class,

then
type in Person in the prompt dialog.
By default, it will choose File for Kind, and option class is to be chosen



Name: Person

Clicking “OK” will finish the creation of the new class and will generate the
following code.

```
class Person {
}
```

This class is empty for now,

Kotlin also uses class keyword to declare a class which is the same as Java.

Now let’s add fields and functions to this class.

```
class Person {
    var name = ""
    var age = 0
    fun eat() {
        println(name + " is eating. He is " + age + " years old.")
    }
}
```

We use var. to declare name and age because we need to assign their values after we create the instance and if we use val, we cannot reassign the value after initiation.
We also define a simple eat() function which will print a string.

After definition, let’s use the following code to instantiate this class:

```
val p = Person()
```

Kotlin doesn’t need to use new keyword to instantiate the class

and the reason is because when you call the constructor function of a class, the only possible reason is
you want to instantiate the class

Here without keyword new, it still clearly shows your intention.

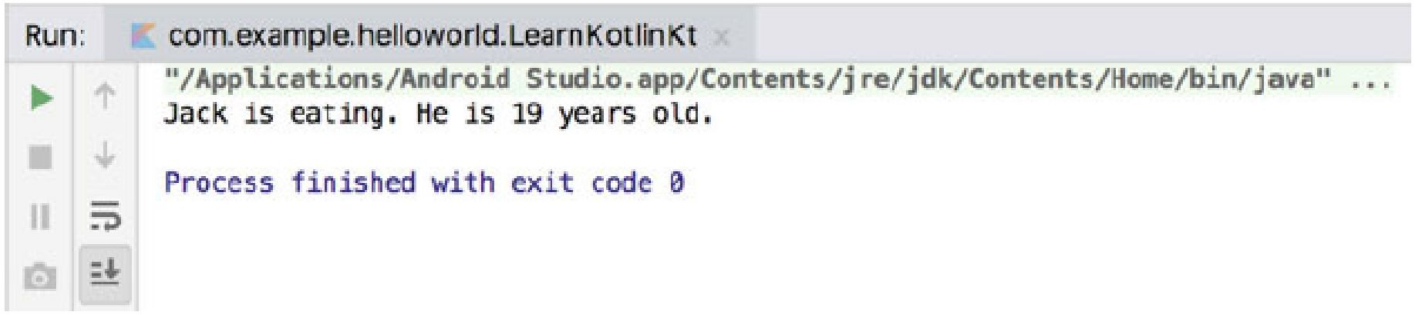
Kotlin follows the minimalism principle and eliminates redundant structure like new keyword and semi-colons at the end of the statements.

Let’s assign this value to variable p. Now p is an object and an instance of Person class.

Now let’s start to operate on p inside the main() function.

```
fun main() {
    val p = Person()
    p.name = "Jack"
    p.age = 19
    p.eat()
    }
```

the result is shown



```
"/Applications/Android Studio.app/Contents/jre/jdk/Contents/Home/bin/java" ...
Jack is eating. He is 19 years old.

Process finished with exit code 0
```

Inheritance and Constructor Function

Right click the com.example.yourproject>New->Kotlin File/Class and

type in Student in the dialog
then choose Class for Kind.

Click “OK” to finish the creation, and add student number and grade in Student class as shown below:

```
class Student {
var sno = ""
var grade = 0
}
```

We design Student class to inherit Person class and then Student will get the fields and
functions in the Person for free and only needs to define its own fields and functions.

First, we need to make Person inheritable

In Kotlin any non-abstract class is not inheritable by default which is equivalent as adding final keyword in Java.

if a class is inheritable by default, then it would be difficult to avoid the risk of random inheritance implementation.

In Effective Java, it is highlighted that if a class is not intentionally designed to be inherited, then we need to proactively add final keyword to prevent inheriting from this class.

Apparently Kotlin designers followed this principle and made it by default that all non-abstract classes are not inheritable

We just need to add the open keyword before the Person class to make it inheritable

```
open class Person {
...
}
```

With the open keyword, we’re telling the Kotlin compiler that Person is designed to be inheritable.

The second thing we need to do is to let Student class inherit Person class.

In Java the keyword is extends and in Kotlin it is a colon

```
class Student: Person() {
var sno = ""
var grade = 0
}
```

there are parentheses after the Person which do not exist in Java.

this has something to do with
More advanced concepts like primary constructor, secondary constructor

Any OOP language have the concept of constructor, so does Kotlin.

However, Kotlin has two kinds of constructors:
primary constructor and the secondary constructor.

The primary constructor is the constructor you use the most and each class will have a parameterless constructor, and of course can also explicitly define the parameters there.

The primary constructor has no function body and is defined right after the class name

```
class Student(val sno: String, val grade: Int) Person()
{ }
```

when instantiating the Student class, the caller has to pass in the parameters required by the constructor

```
val student = Student("a123", 5)
```

The code above creates an instance of the Student class and also assigns a value of a123 to student number and 5 to grade.

Notice that the parameters are all defined as val because the values are passed in instantiation and no need to reassign the value.

if we want to add some code in the primary constructor, what can I do?
Kotlin actually provides an init block in which we can add the logic as shown below:

```
class Student(val sno: String, val grade: Int) : Person() {
init {
       println("sno is " + sno)
       println("grade is " + grade)
      }
}
```

The constructor of sub class has to call the constructor of the parent class and
Kotlin follows this rule.

We defined a primary constructor and based on the rule above, we need to call the Person's constructor.

However, if there is no function body in the primary constructor. So how can we call the parent's constructor?

answer might be to call in the init block. This may work but is not ideal since under most circumstances, we don't need the init block.

Kotlin adopted a simple yet not straightforward approach to resolve the issue: **parentheses**.

The sub class's primary constructor will call the parent's constructors
by using parentheses. And now you should understand the code below.

```
class Student(val sno: String, val grade: Int) : Person() {}
```

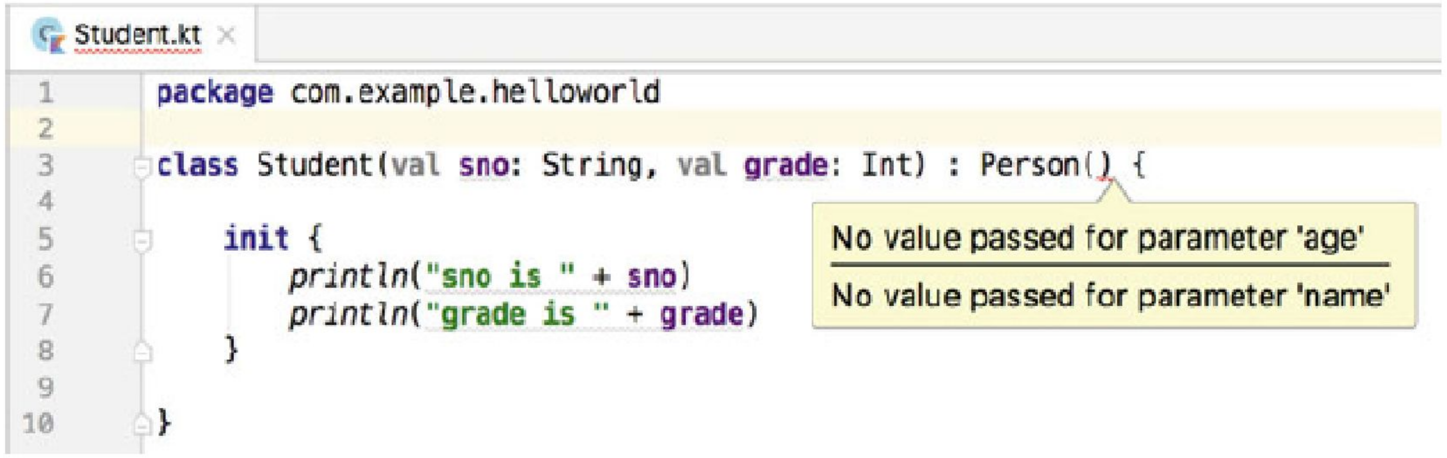
Here the empty parentheses after the Person class means that the Student class's primary constructor will call the parameter less constructor of the Person class.

The parentheses here cannot be omitted even if there is no param at all.

Modify the Person class as shown below:

```
open class Person(val name: String, val age: Int) {
...
}
```

Then there will be errors in the Student class



Student.kt

```
package com.example.helloworld

class Student(val sno: String, val grade: Int) : Person() {

    init {
        println("sno is " + sno)
        println("grade is " + grade)
    }

}
```

No value passed for parameter 'age'

No value passed for parameter 'name'

there is no parameter less constructor in the Person class anymore
Which causes the error.

To resolve the error, we need to pass in the name and age

and we need to add name and age in the primary constructor of Student class.

Then pass these two arguments to the constructor of person class as shown below:

```
class Student(val sno: String, val grade: Int, name: String, age: Int) :
Person(name, age) {
...
}
```

Notice that we cannot declare name and age to be val as the var.
and val keyword will automatically make them the fields of the class which will conflict with the name and age fields in the parent class.

Thus, we don’t need to add any keyword before the name and age parameter and the scope of these two parameters will be within the primary constructor.

We can use the code below to create an instance of the Student class:

```
val student = Student("a123", 5, "Jack", 19)
```

In real world, you might never need to use the secondary constructor since Kotlin provides a way to set the default value to the parameters

which should be able to replace the secondary constructor completely

Any class can only have one primary constructor but can have multiple secondary constructors.

We can use secondary constructor to instantiate a class which is no different from the primary constructor, except that it will have a function body.

In Kotlin, when the class has primary constructor and secondary constructor,

All the secondary constructor(s) must call the primary constructor directly or indirectly.

```
class Student(val sno: String, val grade: Int, name: String, age: Int):
Person(name, age) {
constructor(name: String, age: Int) : this("", 0, name, age) {
  }
constructor() : this("", 0) {
  }
}
```

```
class Student(val sno: String, val grade: Int, name: String, age: Int):
Person(name, age) {
constructor(name: String, age: Int) : this("", 0, name, age) {
  }
constructor() : this("", 0) {
  }
}
```

The secondary constructor is declared with the keyword constructor
and here we define two secondary constructors:

the first secondary constructor takes name and age parameter, and then it calls primary constructor by using this keyword and initializes sno and grade;

the second secondary constructor does not have any parameter and it uses this keyword to call the first secondary constructor we just defined and initializes name and age.

The second secondary constructor indirectly calls the primary constructor, and thus, it is still legit.

Now we have 3 ways to instantiate Student class.

They are parameterless constructor,
constructor with 2 parameters,
and constructor with 4 parameters

as shown below:

```
val student1 = Student()
val student2 = Student("Jack", 19)
val student3 = Student("a123", 5, "Jack", 19)
```

Consider a very special case:
there is only secondary constructor but no primary constructor.

This is very rare but legit case in Kotlin.

When a class does not explicitly define primary constructor but instead defines secondary constructor, then
it is still legit, and we can use the following code to demonstrate:

```
class Student : Person {
constructor(name: String, age: Int) : super(name, age) {
  }
}
```

the differences here.
First, Student class does not explicitly define the primary constructor,
and since it defines secondary constructor, there is no primary constructor inside the Student class.

Since there is no primary constructor, there is no need to add the parentheses when inheriting the Person class. This should clarify when to add parentheses and when not to do so.

Besides this, since there is no primary constructor, the secondary constructor can only call the parent constructor directly

and in the code above, this gets replaced by super which is similar to Java and easy to understand.

Interface

interface in Kotlin is almost identical as in Java.

Interface is an important concept in polymorphism.

Java only supports single inheritance which means that any class can only inherit one super class while being able to implement any number of interfaces.

Kotlin is exactly like Java in this.

We can define a series of abstract functions and then the concrete classes will implement these abstract functions

First, create a Study interface and then define a few study-related functions.

Right click com.example.helloworld package- > New- > Kotlin File/Class

and then type in “Study “, select “ Interface “ for Kind option.

Then add a few functions related to study in the Study interface,

notice that
The function body or the implementation of the function is not required as shown below:

```
interface Study {
fun readBooks()
fun doHomework()
}
```

Then get the Student class to implement the interfaces of Study.

```
class Student(name: String, age: Int): Person(name, age), Study {
override fun readBooks() {
  println(name + " is reading.")
  }
override fun doHomework() {
  println(name + " is doing homework.")
  }
}
```

In Java,
extends keyword is used for inheritance
and implements keyword is used for implementing interface.

In Kotlin both inheritance and implements are done away by using colon and between them are commas.

Here Student class inherits Person class and implements Study interface.

The Study interface declares two abstract functions

readBooks() and
doHomework(),

thus, Student class has to implement these two functions.

Kotlin uses override keyword to override the function in parent class or to implement the abstract function in the interface.

Here we simply print a string in the console.

```
fun main() {
val student = Student("Jack", 19)
doStudy(student)
}
fun doStudy(study: Study) {
study.readBooks()
study.doHomework()
}
```

First, create an instance of Student class. We can directly call the readBooks() and doHomework() functions of the instance,

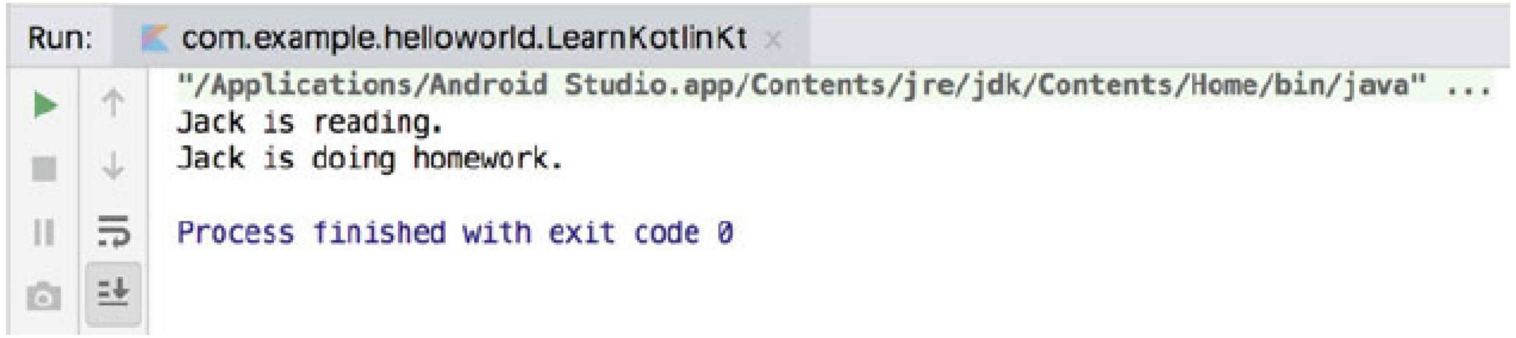
Instead, an instance of Student is passed to the doStudy() function.

This function will take an argument of type Study and since the Student class implemented Study interface,

the Student instance can be passed to doStudy() function without any issue.

Then the code calls the readBooks() and doHomework() implementations of the Study interface.

This is called interface-oriented programming or polymorphism



```
"/Applications/Android Studio.app/Contents/jre/jdk/Contents/Home/bin/java" ...
Jack is reading.
Jack is doing homework.

Process finished with exit code 0
```

In order to make the interface more flexible, Kotlin added another extra functionality

which is to allow default implementation for the function declared in the interface.

Java started to support this since JDK1.8.

So overall, Kotlin and Java are pretty much the same in interface design.

Let’s change the Study interface as follows:

```
interface Study {
fun readBooks()
fun doHomework() {
  println("do homework default implementation.")
  }
}
```

We added the function body to doHomework() which will print a string in the console.

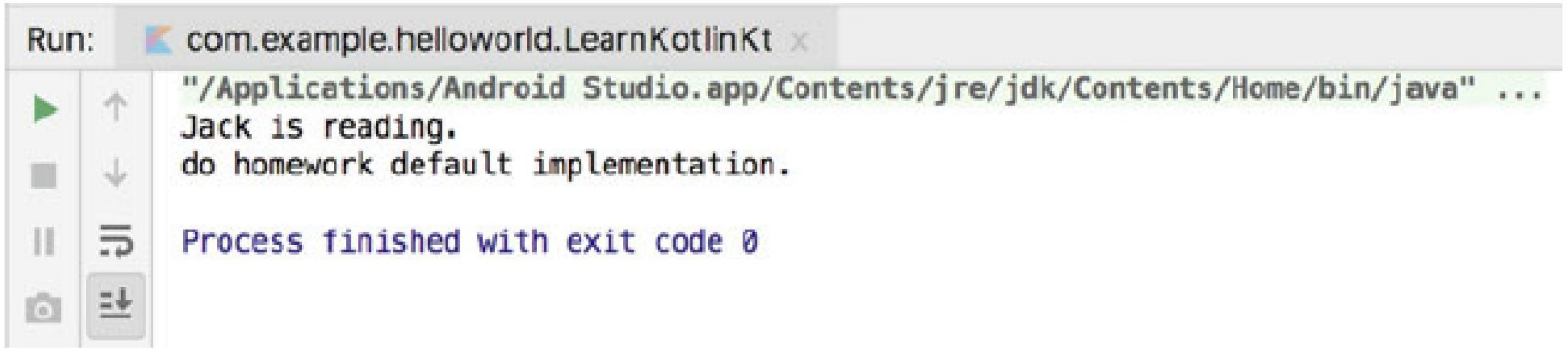
If a function in the interface has function body, then the function body is its default implementation.

Now when a class tries to implement the Study interface,
doHomework() won’t need to be implemented by the class and if not implemented, the default implementation will
used.

Get back to the Student class.

If we delete doHomework(), there will be no error
but deleting readBooks() will cause error.

And let’s delete the doHomework() function and rerun the main() function



```
"/Applications/Android Studio.app/Contents/jre/jdk/Contents/Home/bin/java" ...
Jack is reading.
do homework default implementation.

Process finished with exit code 0
```

scopes

In Java, we have four values of scope for functions which are *public, private, protected, and default* (no keyword).

There are 4 types of scope in Kotlin too which are *public, private, protected, and internal.*

We can add the keyword before the fun keyword to apply the scope.

The scope of private in these two languages is the same which means that the function is only accessible inside the class.

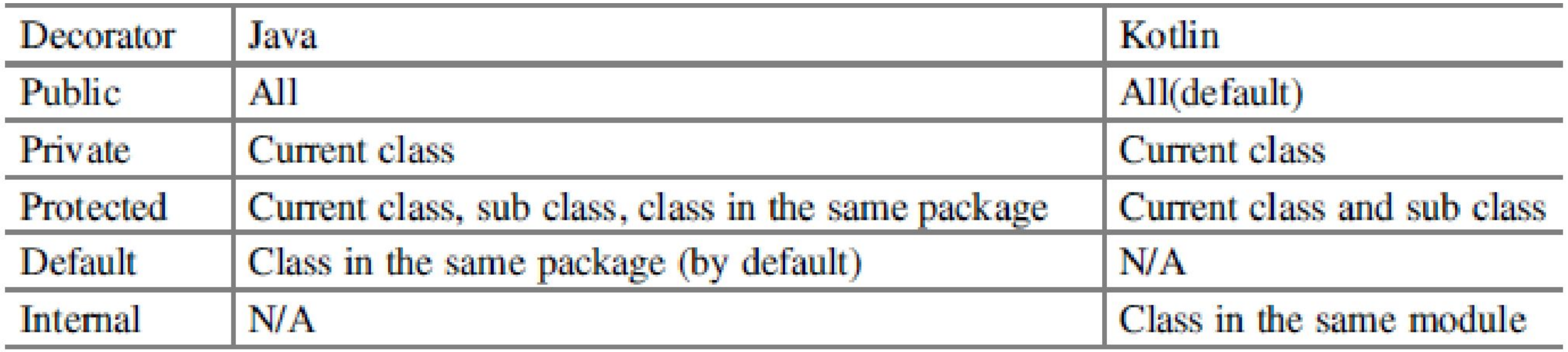
Public is mostly the same which means accessible to all classes.
However, in Kotlin public is the default scope and in Java package private is the default scope.

We didn’t add any scope keywords for our functions before and by default they are public.

For Java, protected means accessible to the current class, sub class, and the classes in the same package. However, for Kotlin it is only accessible to the current class and sub class.

Kotlin eliminates the package private scope (default in Java), and instead introduced another accessibility
concept which is accessible by classes in the same module and the keyword for it is internal.

if we implement a module for other developers to use, and if certain functions can only be called within the module, then we can declare functions to be internal.



Decorator	Java	Kotlin
Public	All	All(default)
Private	Current class	Current class
Protected	Current class, sub class, class in the same package	Current class and sub class
Default	Class in the same package (by default)	N/A
Internal	N/A	Class in the same module

Table . compares more visually the differences between function visibility modifiers in Java and Kotlin.

Data Class and Singleton

In more formal frameworks, data classes play a very important role.

They provide the data models for programming logics and map the data in the database or on the server to memory.

For data classes, usually it is necessary to override the equals(), hashCode(), toString() functions.

The equals() function is used to determine if two data class instances are equal.

The hashCode() function needs overriding otherwise, HashMap, HashSet, and other hash-related classes won’t work.

The toString() function can be used to provide more useful information about the class,

for example, if we try to print a class, by default it will print the memory address of the class which is not useful in most cases.

Data classes, i.e. **Java classes whose sole purpose is to hold data and make it accessible via getters and setters**,

They are among the largest collection points of boilerplate code in many software projects

Data class is a simple class which is used **to hold data/state and contains standard functionality**.

A data keyword is used to declare a class as a data class.

Declaring a data class must contain at least one primary constructor with property argument (val or var).

In order to create a data class, we need to fulfill the following requirements:

1. Contain primary constructor with at least one parameter.

2. Parameters of primary constructor marked as val or var.

3. Data class cannot be abstract, inner, open or sealed.

4. Before 1.1,data class may only implement interface.

The Kotlin compiler automatically generates the following functionality for them:
A correct, complete, and readable toString() method.
Value equality-based equals()
and hashCode() methods.

Utility copy() and componentN() methods.

Now let’s create a simple cellphone class which has brand field and price field.

If we use Java to implement this data class, it should be something like the following code

```
public class Cellphone {
   String brand;
   double price;
   public Cellphone(String brand, double price) {
   this.brand = brand;
   this.price = price;
}
@Override
public boolean equals(Object obj) {
   if (obj instanceof Cellphone) {
   Cellphone other = (Cellphone) obj;
   return other.brand.equals(brand) && other.price ==
price;
 }
   return false;
}
@Override
public int hashCode() {
  return brand.hashCode() + (int) price;
  }
@Override
public String toString() {
    return "Cellphone(brand=" + brand + ", price=" + price +
")";
  }
}
```
It already looks cluttered even without any real business logic and just serves as a container for data.

Kotlin will provide a much simpler way to implement the same functionality.

Right click com.example.<project name>. package ->New ->Kotlin File/Class,

type in “Cellphone” in the dialog and select “Class” for the Kind option.

Then type in the following code in the class:

```
data class Cellphone(val brand: String, val price: Double)
```

The magic happens because of the **data** keyword. When the class is declared with the data keyword, you explicitly show

your intent to make it a data class

and Kotlin will automatically generate
the equals(),
hashCode(),
toString(),
and other frequently used functions and can reduce the workload of development drastically.

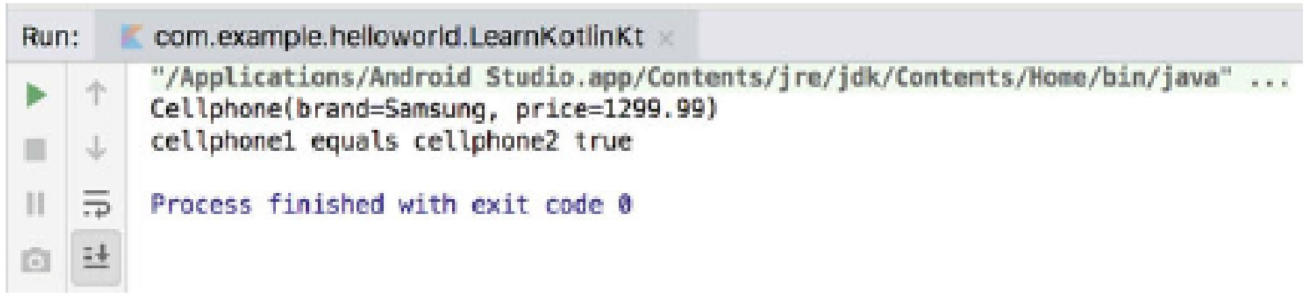
when there is no code inside the class, the braces at the end can be omitted.

We do some test around this data class and type code as follows:

```
fun main() {
   val cellphone1 = Cellphone("Samsung", 1299.99)
   val cellphone2 = Cellphone("Samsung", 1299.99)
   println(cellphone1)
   println("cellphone1 equals cellphone2 " + (cellphone1 ==cellphone2))
}
```

Here we create two Cellphone instances and print the first instance,

then, compare
if these two instances are equal. The result is shown



Run: com.example.helloworld.LearnKotlinKt

```
"/Applications/Android Studio.app/Contents/jre/jdk/Contents/Home/bin/java" ...
Cellphone(brand=Samsung, price=1299.99)
cellphone1 equals cellphone2 true

Process finished with exit code 0
```

Apparently, Cellphone data class works as expected.

However, if there is no data keyword in front of the Cellphone class, the result will be quite different

There is another special functionality Kotlin provides—*singleton class.*

Singleton Pattern, which is one of the most widely used design patterns that can prevent creating duplicated instances.

For instance, if we only want to create a single instance globally, then we should use Singleton Pattern.

Static Method in Java With Examples

The static keyword is used to construct methods that will exist regardless of whether or not any instances of t
Any method that uses the static keyword is referred to as a static method.

Features of static method:

- A static method in Java is a method that is part of a class rather than an instance of that class.
- Every instance of a class has access to the method.
- Static methods have access to class variables (static variables) without using the class's object (instance).
- Only static data may be accessed by a static method. It is unable to access data that is not static (instance varia
- In both static and non-static methods, static methods can be accessed directly.

Syntax to declare the static method:

Access_modifier static void methodName()
{ // Method body. }

The name of the class can be used to invoke or access static methods.

Syntax to call a static method:

className.methodName();

```java
// Java program to demonstrate that
// The static method does not have
// access to the instance variable

import java.io.*;

public class GFG {
   // static variable
   static int a = 40;

   // instance variable
   int b = 50;

   void simpleDisplay()
   {
      System.out.println(a);
      System.out.println(b);
   }

   // Declaration of a static method.
   static void staticDisplay()
   {
     System.out.println(a);
   }

// main method
   public static void main(String[] args)
   {
      GFG obj = new GFG();
      obj.simpleDisplay();

      // Calling static method.
      staticDisplay();
   }
}
```

There are many ways to implement a singleton and here is a common way to do it in Java:

```
public class Singleton {
  private static Singleton instance;
  private Singleton() {}
 public synchronized static Singleton getInstance() {
  if (instance == null) {
     instance = new Singleton();
  }
  return instance;
}
public void singletonTest() {
  System.out.println("singletonTest is called.");
  }
}
```

It’s very straightforward.

In order to prevent other classes to create instances of the Singleton class, we make the scope of the constructor private and
then provide a static function getInstance() to get the instance of the Singleton.

Then in the getInstance() method,
if there is no instance of the class,
  then we create a new instance and return it;
otherwise,
  we just return the existing one.

This is how Singleton Pattern works

If we want to call the method inside the singleton,
for example the singletonTest() method,
then we can write code below:

```
Singleton singleton = Singleton.getInstance();
singleton.singletonTest()
```

Kotlin provides a better way to implement singleton

As the design of data class, Kotlin will also hide the logic for some frequently used functions and provides an elegant way to implement the class.

It is extremely easy to create a singleton class in Kotlin,

We just need to replace the class keyword to object keyword.

Right click com.example.<project name>. package ->New ->Kotlin File/Class,
type in“Singleton” in the dialog
and select “Object” for Kind option,
click “OK” to finish creation

and the following code will be generated

```
object Singleton {
}
```

Now, Singleton is a singleton and we can write functions inside this class,
For instance, a singletonTest() function:

```
object Singleton {
  fun singletonTest() {
    println("singletonTest is called.")
  }
}
```

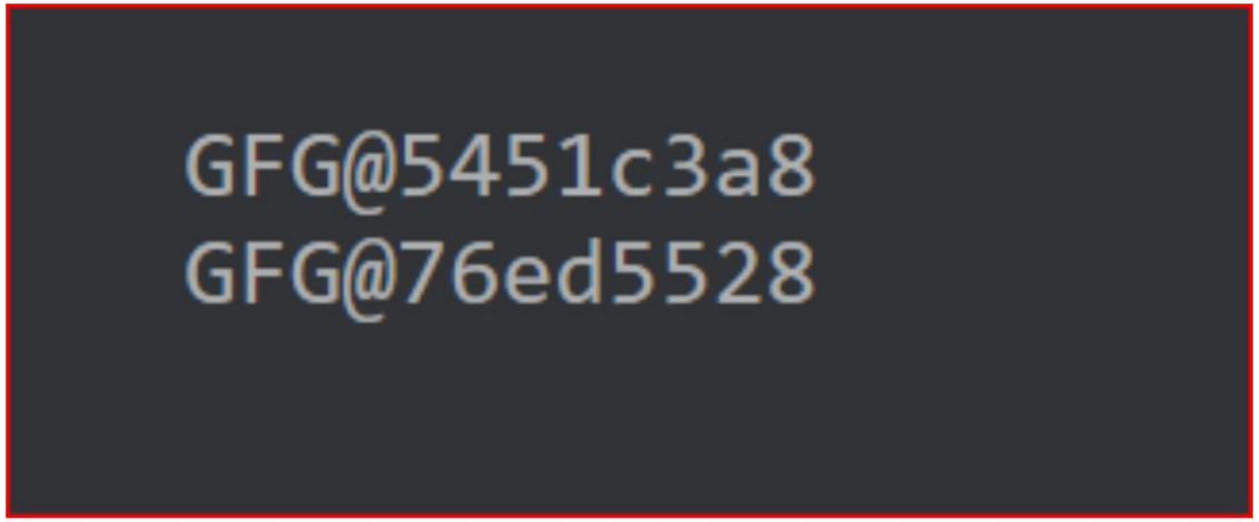
Normally we create the two different objects of the same class for application purpose
but creation of two different objects also requires the allocation of two different memories for the objects.

So it is sometimes better to make a single object and use it again and again.

```
// Kotlin program
fun main(args: Array<String>)
{
  val obj1 = GFG()
  val obj2 = GFG()
  println(obj1.toString())
  println(obj2.toString())
}

class GFG
{

}
```



```
GFG@5451c3a8
GFG@76ed5528
```

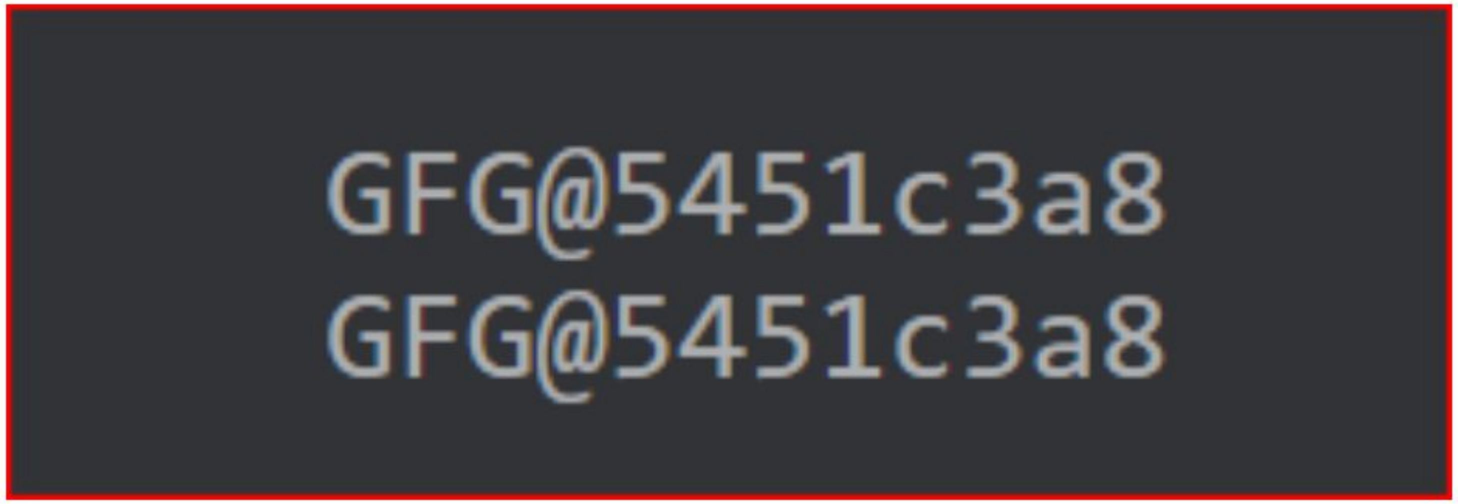
It has been observed that both the objects have a different address, thus it is a wastage of memory.

In the following program, same thing has been accomplished , but the singleton class is used .

```
// Kotlin program
fun main(args: Array<String>)
{
  println(GFG.toString())
  println(GFG.toString())
}

// GFG is the singleton class here
object  GFG
{

}
```



```
GFG@5451c3a8
GFG@5451c3a8
```

So when we use an object instead of a class, Kotlin actually uses the Singelton and allocates the single memory.

In Java, a singleton class is made making a class named **Singleton**.
But in Kotlin, we need to use the **object** keyword.

The object class can have functions, properties, and the init method.

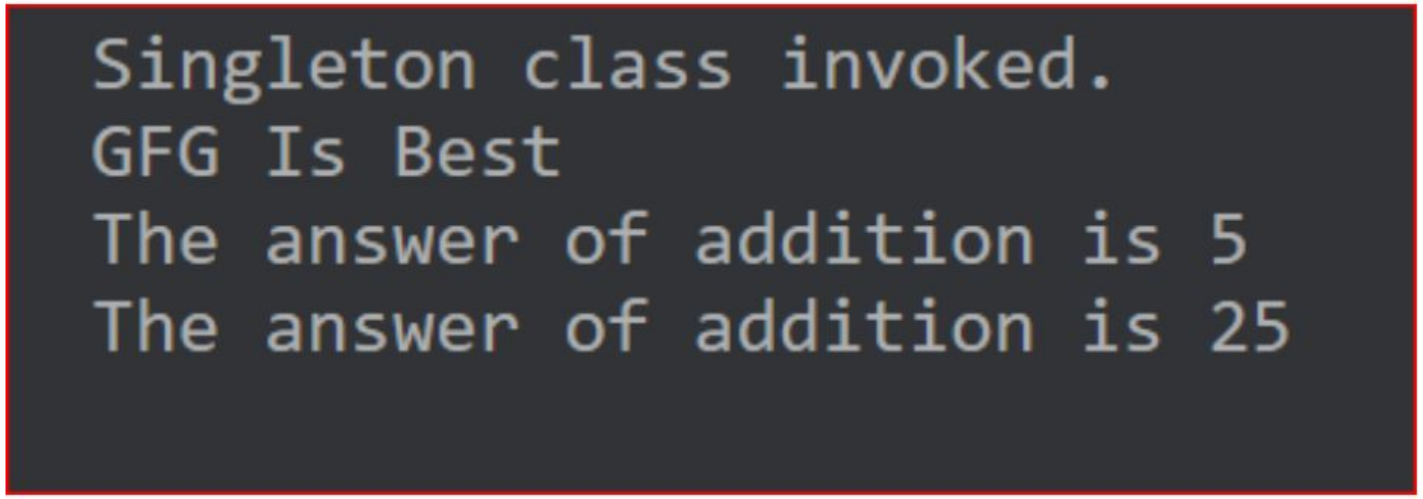
The constructor method is not allowed in an object so we can use the init method if some initialization is required and the object can be defined inside a class.

We call the method and member variables present in the singleton class using the class name, as we did in the **companion objects**.

```
// Kotlin program
fun main(args: Array<String>)
{
  println(GFG.name)
  println("The answer of addition is ${GFG.add(3,2)}")
  println("The answer of addition is ${GFG.add(10,15)}")
}

object  GFG
{
   init
  {
    println("Singleton class invoked.")
  }

  var name = "GFG Is Best"
  fun add(num1:Int,num2:Int):Int
  {
    return num1.plus(num2)
  }
}
```



```
Singleton class invoked.
GFG Is Best
The answer of addition is 5
The answer of addition is 25
```

We don’t need to create private constructor, nor static method like getInstance().

All we need to do is to declare the class with object keyword and then we get a singleton class.

To use the method in the singleton is just like using static method in Java:

```
Singleton.singletonTest()
```

It looks like we’re calling a static method,

but under the hood, Kotlin created an instance of the Singleton class and can guarantee there will be only one Singleton instance globally.

Properties of Singleton Class

The following are the properties of a typical singleton class:

- **Only one instance**: The singleton class has only one instance and this is done by providing an instance of the class, within the class.

- **Globally accessible**: The instance of the singleton class should be globally accessible so that each class can use it.

- **Constructor not allowed**: We can use the init method to initialize our member variables.

Suppose you have an app, and users can Login to it after undergoing user authentication,
so if at the same time two user with same name and password tries to Login to the account,
the app should not permit as due to concern of security issues.

So singleton objects help us here to create only one instance of the object,
i.e user here so that multiple logins can’t be possible.

- *Consider the case when you are working with the repository in MVVM architecture,*
  So, in that case, you should create only 1 instance of the repository, as repositories are not going to change, and creating different instances would result in space increment and time wastage.

Creation and Iteration of Collection

Usually when we talk about collections, we are talking about List and Set,

and
key-value data structures like Map.

List, Set, and Map are interfaces in Java.

The ArrayList class and LinkedList class are the most used classes that implement List.

HashSet is the most used class that implements Set.

HashMap is the most used class that implements Map.

Using Java, you might instantiate an instance of ArrayList and then add the names in the ArrayList one by one.

Of course, we can do the same thing in Kotlin as the code below:

```
val list = ArrayList<String>()
list.add("Apple")
list.add("Banana")
list.add("Orange")
list.add("Pear")
list.add("Grape")
```

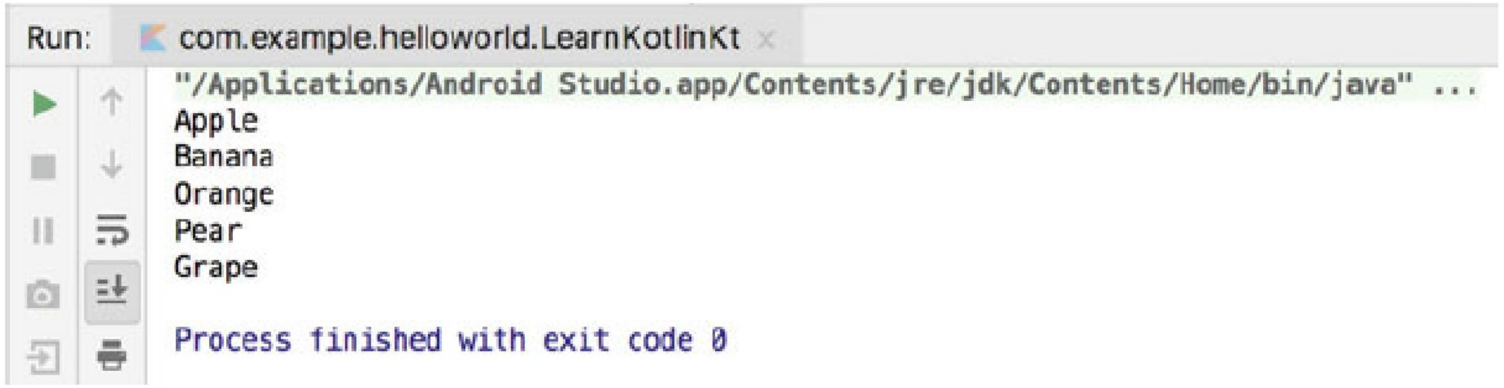
Kotlin is a concise language, and it provides a listOf() method to simplify the initialization of lists as shown below:

```
val list = listOf("Apple", "Banana", "Orange", "Pear",
"Grape")
```

for-in can be used to not only iterate the interval but also iterate the collections.

Now let’s try it out to iterate the fruit names list, and add the code below to main() function:

```
fun main() {
 val list = listOf("Apple", "Banana", "Orange", "Pear",
"Grape")
 for (fruit in list) {
   println(fruit)
  }
}
```



Run: com.example.helloworld.LearnKotlinKt

```
"/Applications/Android Studio.app/Contents/jre/jdk/Contents/Home/bin/java" ...
Apple
Banana
Orange
Pear
Grape

Process finished with exit code 0
```

Notice that listOf() function will create an immutable collection

which means that
this list is read-only and we cannot add, update, or remove items in the list.

The reason behind this is the same as the design of val keyword and class is not inheritable by default.

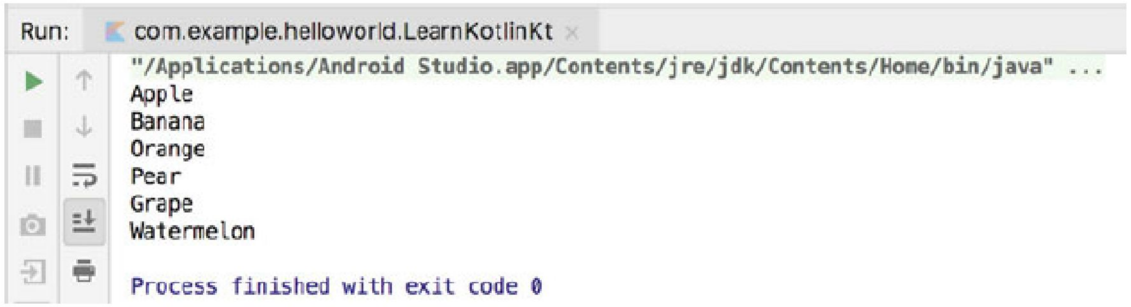
What if we need to create a mutable collection?

Then we can just use mutableListOf() function, as shown below:

```
fun main() {
  val list = mutableListOf("Apple", "Banana", "Orange", "Pear","Grape")
  list.add("Watermelon")
  for (fruit in list) {
    println(fruit)
  }
}
```

Here we use mutableListOf() to create a mutable list and then add a new fruit name in this list.

Finally, we use for-in to iterate the list. Rerun the app.



```
"/Applications/Android Studio.app/Contents/jre/jdk/Contents/Home/bin/java" ...
Apple
Banana
Orange
Pear
Grape
Watermelon

Process finished with exit code 0
```

Set collection is used almost the same way as List.

We can use setOf() and mutableSetOf() for corresponding functionality

```
val set = setOf("Apple", "Banana", "Orange", "Pear",
"Grape")
for (fruit in set) {
println(fruit)
}
```

It is worth noting that under the hood, Set uses hash to store the data.

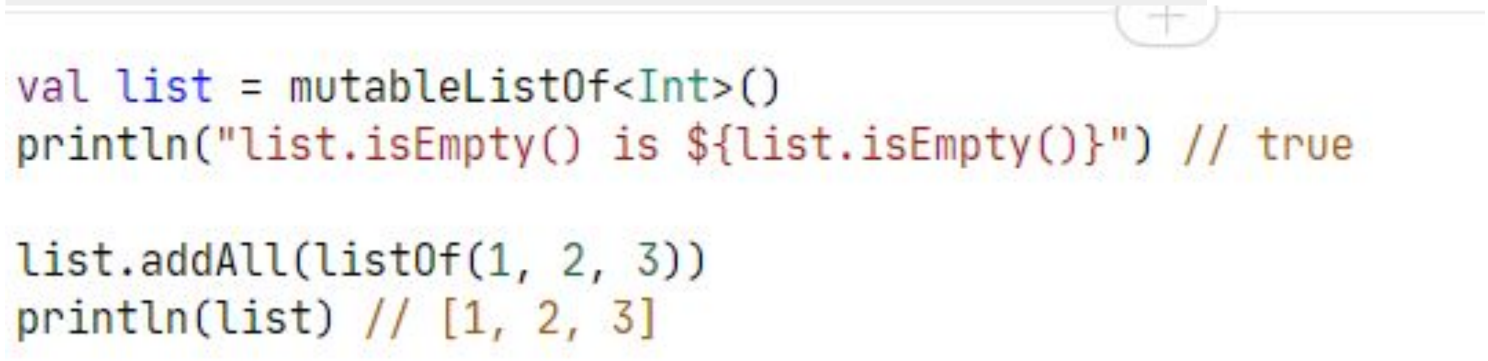
Thus, the elements in the set don’t have order which is the biggest difference from List

Kotlin mutableListOf()

MutableList class is used to create mutable lists in which the elements can be added or removed.
The method mutableListOf() returns an instance of MutableList Interface and takes the array of a particular type or mixed
 (depends on the type of MutableList instance) elements or it can be null also.

interface MutableList<E> : List<E>, MutableCollection<E>
(source)
mutableListOf
fun <T> mutableListOf(): MutableList<T>
)
Returns an empty new MutableList.

fun <T> mutableListOf(vararg **elements**: T): MutableList<T>
Returns a new MutableList with the given elements..



```
val list = mutableListOf<Int>()
println("list.isEmpty() is ${list.isEmpty()}") // true

list.addAll(listOf(1, 2, 3))
println(list) // [1, 2, 3]
```

API

API stands for Application Programming Interface. It's a software interface that allows different computer programs to communicate with each other. APIs are used in many applications, including mobile apps, web apps, and cloud services.

API architecture is usually explained in terms of client and server.
The application sending the request is called the client, and the application sending the response is called the server.
So in a weather example, the bureau's weather database is the server, and the mobile app is the client.

REST APIs

These are the most popular and flexible APIs found on the web today.
The client sends requests to the server as data.
The server uses this client input to start internal functions and returns output data back to the client.

REST stands for Representational State Transfer.
REST defines a set of functions like GET, PUT, DELETE, etc. that clients can use to access server data.
Clients and servers exchange data using HTTP.

The main feature of REST API is statelessness.
Statelessness means that servers do not save client data between requests.
Client requests to the server are similar to URLs you type in your browser to visit a website.
The response from the server is plain data, without the typical graphical rendering of a web page.

The benefits of REST APIs

REST APIs offer four main benefits:

1. Integration
APIs are used to integrate new applications with existing software systems. This increases development speed because each functionality doesn't have to be written from scratch. You can use APIs to leverage existing code.

2. Innovation
Entire industries can change with the arrival of a new app. Businesses need to respond quickly and support the rapid deployment of innovative services. They can do this by making changes at the API level without having to re-write the whole code.

3. Expansion
APIs present a unique opportunity for businesses to meet their clients' needs across different platforms. For example, maps API allows map information integration via websites, Android,iOS, etc. Any business can give similar access to their internal databases by using free or paid APIs.

4. Ease of maintenance
The API acts as a gateway between two systems. Each system is obliged to make internal changes so that the API is not impacted. This way, any future code changes by one party do not impact the other party.

What are the different types of APIs?

APIs are classified both according to their architecture and scope of use.

Private APIs
These are internal to an enterprise and only used for connecting systems and data within the business.

Public APIs
These are open to the public and may be used by anyone. There may or not be some authorization and cost associated with these types of APIs.

Partner APIs
These are only accessible by authorized external developers to aid business-to-business partnerships.

Composite APIs
These combine two or more different APIs to address complex system requirements or behaviors.

Functional API

A **functional API** is a way to create models that are more flexible and customizable than sequential APIs.
It's used in machine learning and deep learning.

Benefits of a functional API

- Build models with non-linear topology, such as residual connections or dense networks.
- Create multi-input and multi-output models.
- Create shared layers.
- Promote modular design and efficient parameter sharing.
- Create more compact and expressive models.

Functional APIs of Collections

few examples to understand the syntax of functional APIs which has the same syntax as Lambda expression

```
val list = listOf("Apple", "Banana", "Orange", "Pear", "Grape","Watermelon")
 var maxLengthFruit = ""
 for (fruit in list) {
 if (fruit.length > maxLengthFruit.length) {
    maxLengthFruit = fruit
  }
 }
 println("max length fruit is " + maxLengthFruit)
```

How do we find the fruit that has the longest name?

This can be made more concise by using the functional APIs as below:

```
val list = listOf("Apple", "Banana", "Orange", "Pear", "Grape","Watermelon")
val maxLengthFruit = list.maxBy { it.length }
println("max length fruit is " + maxLengthFruit)
```

We just need one line of code to accomplish the same
goal.

So, what is Lambda?
In plain words, Lambda is a few lines of code that can be used as an argument.

This is very useful because we used to only pass data as argument,

but Lambda allows the user to pass a few lines of code as argument.

but how many lines of code can be considered as a few lines?

Kotlin actually doesn’t have limitation on this,
but it is recommended to not write very long code in Lambda expression for the sake of
readability

The syntax of Lambda expression looks like this:

{param name 1: param type, param name 2: param type -> function body}.

we have ->between the params list and function.

We can write any lines of code in the function though it is recommended to keep the code short
and the result of the last line of code will be the return value.

However, maxBy is just a normal function
that takes a Lambda expression as param and when we iterate the collection,

Every value will be passed as argument to the Lambda expression.

Now let’s use functional API with the complete form as shown below:

```
val list = listOf("Apple", "Banana", "Orange", "Pear", "Grape","Watermelon")
val lambda = { fruit: String -> fruit.length }
val maxLengthFruit = list.maxBy(lambda)
```

maxBy function takes a Lambda expression as argument,
and this Lambda expression is using the syntax we just mentioned

What are the four types of functional interfaces in Java?

FUNCTIONAL INTERFACES	DESCRIPTION
Consumer	It accepts a single input argument and returns no result.
Predicate	It accepts a single argument and returns a boolean result.
Function	It accepts a single argument and returns a result.
Supplier	It does not take any arguments but provides a result.

Java Functional API

We can call Java methods in Kotlin with functional APIs with certain limitations.

That is, if we call a Java method in Kotlin and this method takes only one functional interface as argument, then we can use functional API.

Java functional interface is an interface that contains only one abstract method.

If there are multiple abstract methods in an interface, then it is not a functional interface.

The Runnable interface is a widely used functional interface in Java which has only one abstract method

—run():

```
public interface Runnable {
 void run();
}
```

what are the Java methods that can take Runnable as param?

There are many such use cases but
Runnable interface usually is getting used in thread

The constructor in Thread class takes a Runnable param

The following Java code is used to create and run this thread:

```
new Thread(new Runnable() {
@Override
public void run() {
  System.out.println("Thread is running");
  }
 }).start();
```

Anonymous class syntax is applied here.

An instance of anonymous class is created - that implements Runnable interface and pass it to the constructor of the Thread class.

Then use start() method in Thread class to run this thread instance.

The Kotlin version is :

```
Thread(object : Runnable {
 override fun run() {
  println("Thread is running")
  }
}).start()
```

Kotlin eliminates new keyword.

Thus, it uses object to create an instance.

Now we apply the rules - mentioned above to try to simplify it as shown below:

```
Thread(Runnable {
  println("Thread is running")
}).start()
```

Now it is more concise and not confusing at all.

This is because Runnable class only has one abstract method and
even it is not necessary to explicitly override run() method,

Kotlin knows that the Lambda expression after Runnable is the implementation of run() method.

Also, if we only have one functional interface as param then we can omit the interface name too as shown below:

```
Thread({
  println("Thread is running")
}).start()
```

Like the functional APIs in Kotlin,

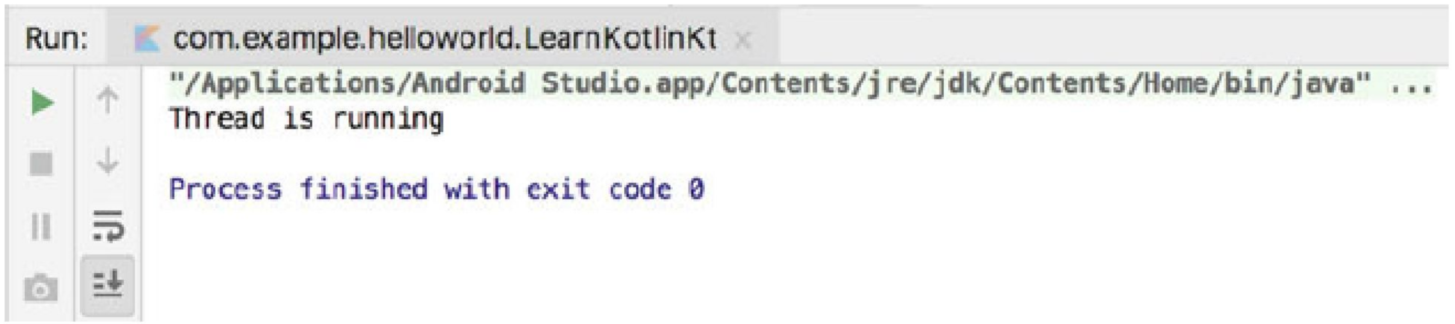
when the Lambda expression is the last param of a method,
the Lambda expression can be moved to the outside of the parentheses.

And if the Lambda expression is the only param of the method, then parentheses can be omitted altogether.

The final version of simplified code is as below:

```
Thread {
   println("Thread is running")
}.start()
```

Copy and paste the code above into the main() method and rerun the code.



```
"/Applications/Android Studio.app/Contents/jre/jdk/Contents/Home/bin/java" ...
Thread is running

Process finished with exit code 0
```

In Kotlin, we can use functional API to simplify the code to the following:

```
button.setOnClickListener {
}
```

all the Java functional API usages here are all about calling Java method in Kotlin
and the functional interface must be defined in Java.

This is because Kotlin has higher-order functions to implement even more powerful functional
APIs. thus, there is no need to use functional interface as Java does.

String Interpolation

Kotlin supports this feature since the very beginning.

In Kotlin we don’t need to concatenate the strings using very clumsy ways as in Java,

instead, we can put expression in string
and even building very complex string can be very simple task

The syntax of string interpolation in Kotlin is shown as code below:

```
"hello, ${obj.name}. nice to meet you!"
```

Kotlin allows to use ${} expression and when this string is getting used,
the expression will be evaluated and will be replaced by the evaluated result.

Also, if there is only one variable, then we can omit the braces like shown below:

```
"hello, $name. nice to meet you!"
```

```
val brand = "Samsung"
val price = 1299.99
println("Cellphone(brand=" + brand + ", price=" + price + ")")
```

We can see that there are four + operators which is not easy to write, and readability is also very poor.
However, with string interpolation, it will be much easier, as shown below:

```
val brand = "Samsung"
val price = 1299.99
println("Cellphone(brand=$brand, price=$price)")
```

Function Default Arguments

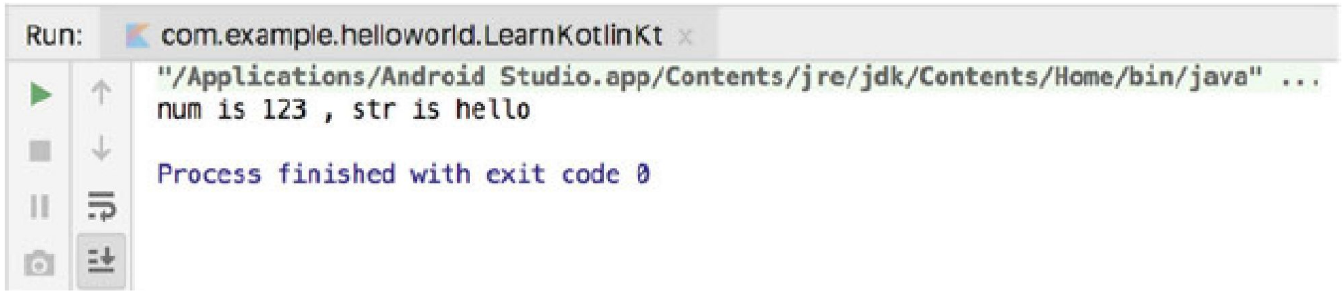
we can assign a default value to any argument
and when this function gets called, the caller doesn’t need to pass value to this argument

If there is no value passed in - then the default value of the argument will be used.

```
fun printParams(num: Int, str: String = "hello") {
  println("num is $num , str is $str")
}
```

When we call printParams(), passing value to the second param is optional and if no argument gets passed in, default value will be used.
Now let’s test it by modifying the code as shown below:

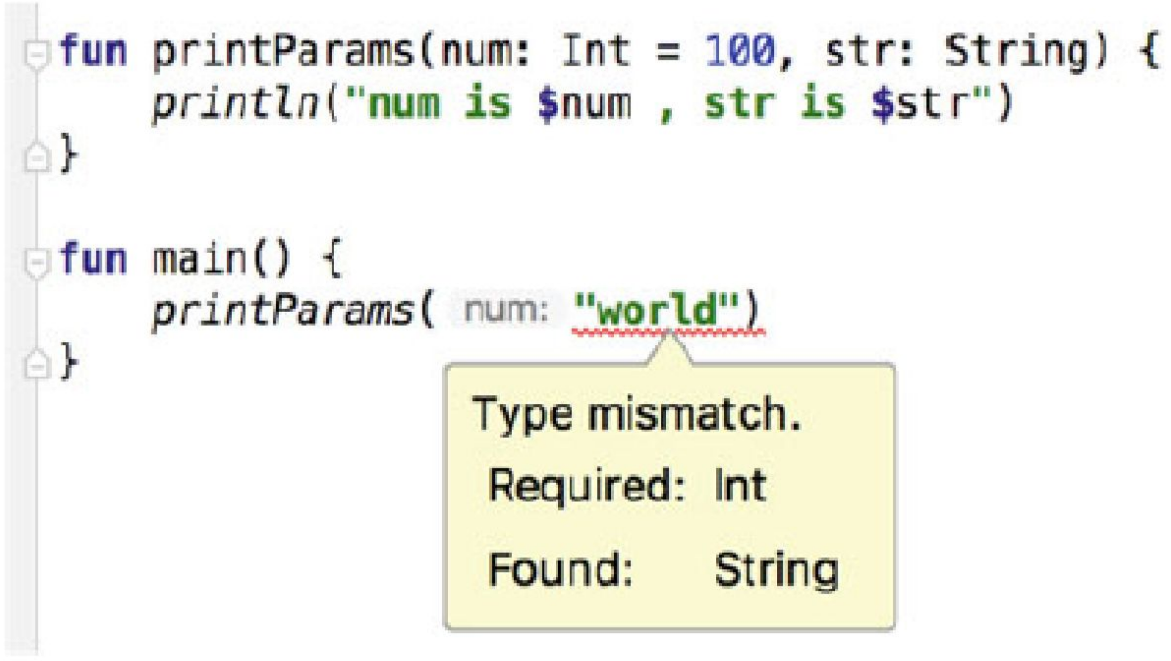
```
fun main() {
  printParams(123)
}
```



```
"/Applications/Android Studio.app/Contents/jre/jdk/Contents/Home/bin/java" ...
num is 123 , str is hello

Process finished with exit code 0
```

```
fun printParams(num: Int = 100, str: String) {
  println("num is $num , str is $str")
}
```



```
fun printParams(num: Int = 100, str: String) {
    println("num is $num , str is $str")
}

fun main() {
    printParams( num: "world")
}
```

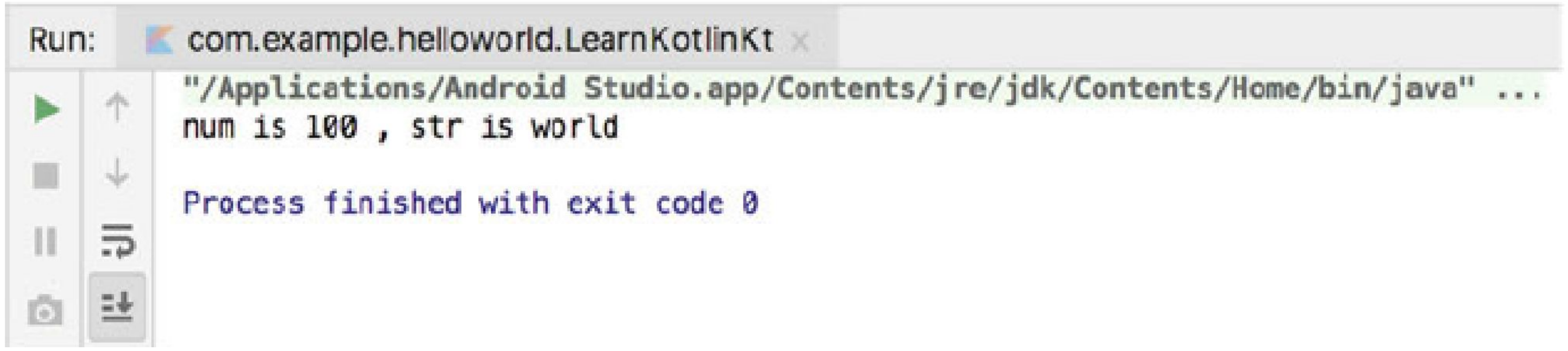
```
Type mismatch.
 Required:  Int
 Found:     String
```

Kotlin provides named argument to solve this problem which doesn't even need to follow the order of params.

For instance, to use printParams() we can write code like below:

```
printParams(str = "world", num = 123)

fun printParams(num: Int = 100, str: String) {
println("num is $num , str is $str")
}
fun main() {
printParams(str = "world")
}
```



Run: com.example.helloworld.LearnKotlinKt

```
"/Applications/Android Studio.app/Contents/jre/jdk/Contents/Home/bin/java" ...
num is 100 , str is world

Process finished with exit code 0
```

So how can function default arguments replace secondary constructor?

the code using secondary constructor is as follows:

```
class Student(val sno: String, val grade: Int, name: String, age: Int) :
Person(name, age) {
  constructor(name: String, age: Int) : this("", 0, name, age) {
  }
   constructor() : this("", 0) {
  }
}
```

The code above has one primary constructor and two secondary constructors.

The secondary constructors are used to instantiate the Student class with less params.
The parameterless constructor will call the constructor with two params and assign values to these two params.
The constructor with two params will call the primary constructor with four params
and assign values to the missing two params.

In Kotlin - just write one primary constructor
and use function default arguments to do the same thing.
The code is shown as below:

```
class Student(val sno: String = "", val grade: Int = 0, name: String ="", age: Int = 0) :
  Person(name, age) {
}
```

After assigning default values to each parameter, we can use any combination of params
to instantiate Student class which includes the cases of the two secondary constructors.

Thank
You