

Kotlin

Arthur Hoskey, Ph.D.
Farmingdale State College
Computer Systems Department

- Kotlin Basics

Today's Lecture

Kotlin

- Full programming language.
- Kotlin runs on a Java Virtual Machine (JVM).
- Kotlin can be used to develop Android programs.

- Getting started link:

<https://kotlinlang.org/docs/tutorials/getting-started.html>

Kotlin

IntelliJ IDE

- IntelliJ can be used to develop in Kotlin.
- Good to learn the basics of the language (instead of trying to write Android programs).
- IntelliJ Community Edition is free to use.

New Kotlin Project in IntelliJ

- Go to File|New|Project in Menu.
- New Project Dialog
 - Choose Kotlin in the left pane
 - Choose Kotlin/JVM in the right pane

IntelliJ IDE

Program Entry Point

- All Kotlin programs start from main.

```
fun main()  ← fun stands for function
{
    println("Hello World!") ← println is used to print
                             data to the console
}
```

Program Entry Point

Running Kotlin Program in IntelliJ

- Create a new Console Application project.
- The project should come with a main function already there.
- If the Run|Run menu item is grayed out, right click main in the code to bring up a context menu. Choose the Run menu option (there will be a green triangle next to Run). Doing this will cause IntelliJ to generate a class to hold the main method behind the scenes. You can open the edit configuration to see the name of the class it created. The Run|Run menu item will no longer be grayed out.

```
fun main()  
{  
    println("Hello World!")  
}
```

Running Kotlin Program in IntelliJ

Comments

- Similar to C++ and Java
- Use `//` for a single line comment
`// This is a Kotlin single line comment`
- Use `/* */` combination for multiline comments
`/*`
**This is
a multiline
comment**
`*/`

Comments

Datatypes

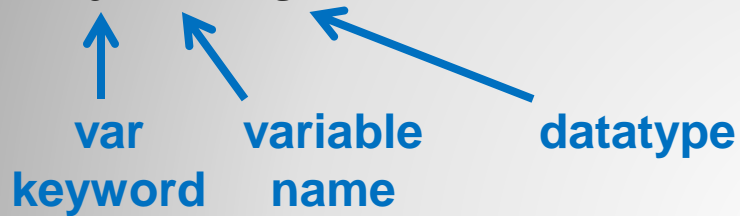
- Int – Used for integers
- Double – Used for floating-point numbers
- Boolean – Used for true/false values
- String – Used for string data

Basic Datatypes

Variables

- Int – Used for integers

var x:Int



- Double variable

var y:Double

- Boolean variable

var z:Boolean

- String variable

var z:String

Variables

val - Read Only Local Variables

- val – Used to make a read only local variable.

val x = 10

val keyword variable name Value to put in the variable (will cause the data type to be Int)

~~x = 20~~ Once a val is assigned to it cannot be changed (x already has the value 10)

val – Read Only Local Variables

Type Inference

- You can leave the datatype out of a declaration if you initialize the variable with a value.
- The datatype will be inferred by the value the variable is being set to.
- In the following line variable a has a String datatype because it is being assigned a string literal:

var a = "abc"

var
keyword

a
variable
name

"abc"
Double quotes is for a string
literal

Type Inference

if (statement)

- If can be used as a normal statement.
- Here is an example:

```
var sales = 150  
var goodSales : Boolean
```

```
if (sales > 100)  
{  
    goodSales = true  
}  
else  
{  
    goodSales = false  
}
```

The if statement is
similar to other
languages

If (statement)


if (expression)

- If can be used as an expression (evaluates to a value).
- Here is an example:

```
var sales = 150
```

```
var goodSales : Boolean
```

Set the variable equal to the result of the if expression



```
goodSales = if (sales > 100)
```

```
{
```

```
  true
```

```
}
```

```
else
```

```
{
```

```
  false
```

```
}
```

The last statement in the block that runs is what the if evaluates to. The goodSales variable gets the value true if sales > 100 and false otherwise.



If (expression)

when (statement)

- when can be used as a normal statement.
- Here is an example:

Similar to switch
in Java and C++

```
var grade = "A"
```

```
var description : String
```

```
when (grade) {
```

```
  "A" -> {
```

```
    description = "Excellent"
```

```
  }
```

```
  "B" -> description = "Good"
```

```
  "C" -> description = "Average"
```

```
  else -> {
```

```
    description = "Needs improvement"
```

```
  }
```

```
}
```

Can use a block { } for
the body of the branch


Can use a single statement
as the body of the branch

If no branches match, then
the else branch is executed

When (statement)

when (expression)

- when can be used as an expression (evaluates to a value).
- Here is an example:

var grade = "A"  **Set the variable equal to the result of the when expression**

```
var description = when (grade) {  
    "A" -> "Excellent"  
    "B" -> "Good"  
    "C" -> "Average"  
    else -> "Needs improvement"  
}
```

When (expression)

for Statement

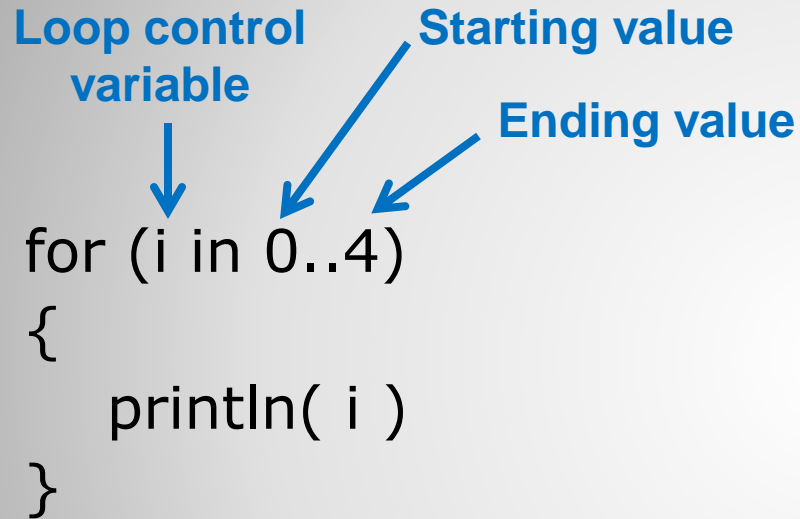
- The following for loop prints the numbers 0 to 4.

Loop control
variable

Starting value

Ending value

```
for (i in 0..4)
{
    println( i )
}
```



For

while Statement

- The following for loop prints the numbers 0 to 4.

Loop control
variable



```
var i = 0
```

Test condition
for loop



```
while (i < 5) {  
    println( i )  
    i++  
}
```

While

Function

- Function with no parameters and no return value that declares a local variable.

Use fun keyword to
define a function


Function name

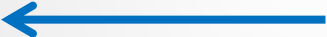
```
fun myFunction()  
{  
    var s = "abc"  
}
```

Function

Calling a Function

- Here is code to define and call a function.

```
fun main(args: Array<String>) {  
    myFunction()  Call myFunction from main  
}
```

```
fun myFunction()  Define  
myFunction  
{  
    println("myFunction ran")  
    var s = "abc"  
}
```

Calling a Function

Function - Parameters

- Function with one parameter and no return value.

Parameter name is x

Parameter type is Int

```
fun myFunction2(x : Int)  
{  
  
}
```

Put a colon between
the parameter name
and type

- Function with two parameters and no return value

```
fun myFunction3(x: Int, s : String)  
{  
  
}
```

Function - Parameters

Function – Return Value

- Function with no parameters and an Int type return value.

Put a colon after
parenthesis

Function return type

```
fun myFunction4() : Int
```

```
{
```

```
    var num = 100
```

```
    return num
```

```
}
```

Use return statement
to return a value

```
fun main(args: Array<String>) {
```

```
    var retVal = myFunction4()
```

```
}
```

Call function and put return
value in the retVal variable

Function – Return Value


Function Reference in Variable

- Function reference – Stores the address of a function.
- The test function below is located at address 2800.
- The fv variable below is a function reference. It is storing the starting address of a function (the test function in this example).

```
var x = 555
var y = 777
var fv = ::test

fun test() {
    println("test ran")
}
```

Get address of
function test



Memory Location	Data at Location
1000	var x, 555
1004	var y, 777
1008	var fv, 2800
Other locations...	
Other locations...	
2800	fun test { println("testran") }

Function Reference in Variable

Function Reference in Variable

- Function references can be stored in variables.
- `::` operation – Creates a reference.

`::` gets a reference to the test function. The `::` operator will return the address of test. That value is then assigned to the fv variable.

`var fv = ::test` ← Put a reference for the test function in the fv variable. Do not use `()` because the test method is not being executed.

`fv()` ← fv has address 2800 in it. Call whatever function is at address 2800 (test function in this example).

```
fun test() {  
    println("test ran")  
}
```

Memory Location	Data at Location
1008	var fv, 2800
Other locations...	
Other locations...	
2800	fun test { println("test ran") }

Function Reference in Variable

Function With Parameters Reference in Variable

- Variables can store references to functions that have parameters.

:: creates reference of testWithParms function

Put a reference for test function in the fv variable (do not use () here)

```
var fv = ::testWithParms  
fv("abc", 100)
```

Call test function using the variable and pass in arguments for the parameters


```
fun testWithParms(s: String, num: Int) {  
    println(s)  
    println(num)  
}
```

Function with Parameters Reference in Variable


Lambda Expression

- A lambda expression is an anonymous function.
- The function body is in memory, but it has no name (it's anonymous).
- The address of this function is only referred to in the variable. No other part of the program knows of it.

```
val anonFunc = {  { is the start of the anonymous function body  
    println("Anonymous method ran")
```

```
}  } is the end of the  
    anonymous  
    function body
```

```
anonFunc()
```

 Calls the function at location
9600 (9600 is the address
stored in anonFunc)

Memory Location	Data at Location
7000	val anonFunc, 9600
Other locations...	
Other locations...	
9600	{ println("Anonymous method ran") }

Lambda Expression

Review - Declaring and Initializing Variables

- Use val or var.
- This should be followed by a variable name.
- Then a colon comes next.
- This is followed by a data type.
- An assignment operator comes after that (=).
- The data to put in the variable is last.
- For example:

Variable Name Data Type Data to put in variable Name

val x: Int = 555

The diagram illustrates the structure of a variable declaration. Three blue labels are positioned above the code 'val x: Int = 555'. 'Variable Name' has an arrow pointing to 'x'. 'Data Type' has an arrow pointing to 'Int'. 'Data to put in variable Name' has an arrow pointing to '555'.

Review – Declaring and Initializing Variables

Review - Declaring and Initializing Variables

- Here are other examples:

Variable Name Data Type Data to put in variable
↓ ↓ ↓
`val s: String = "I love Android"`

Variable Name Data Type Data to put in variable
↓ ↓ ↓
`val e: Employee = Employee()`

← `Employee()` creates a new instance of `Employee` and returns the address of that instance. The address is then stored in `e`.

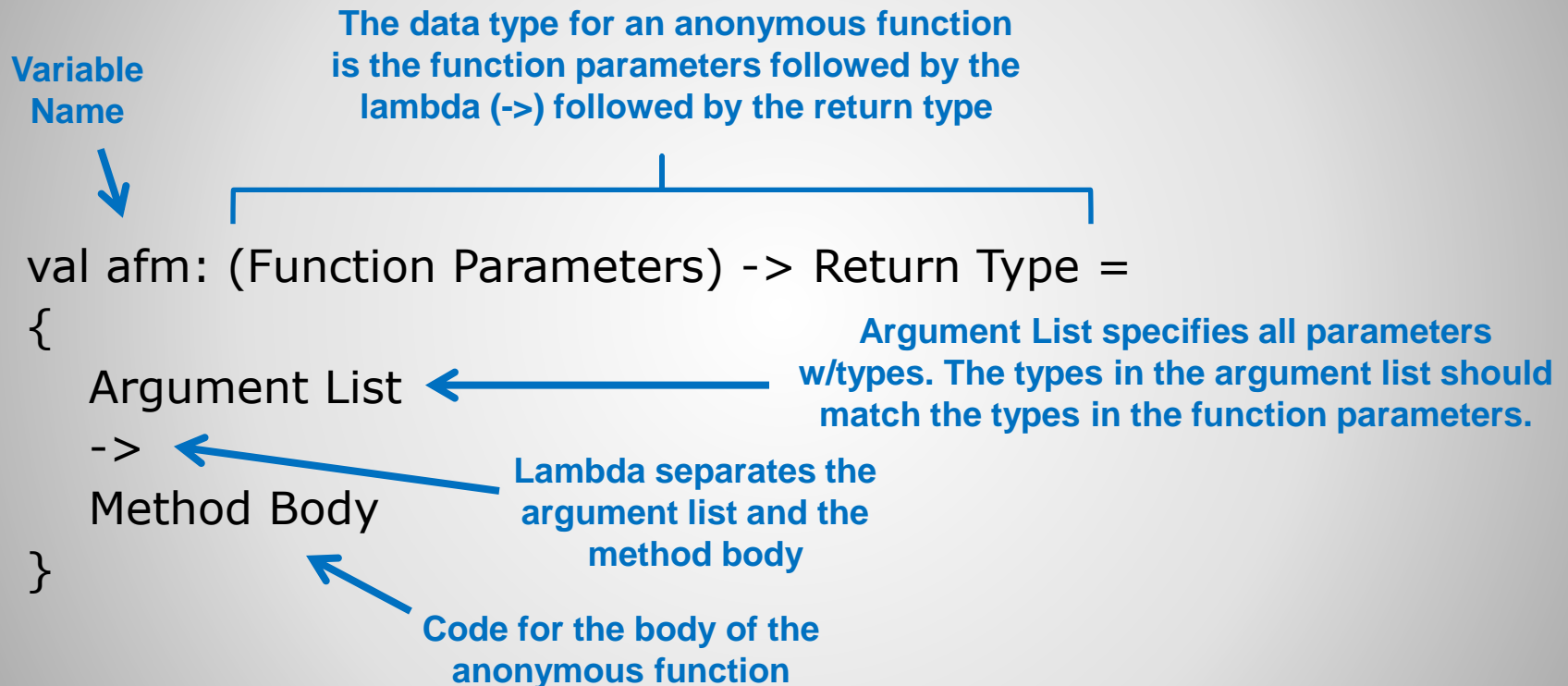
Variable Name Data Type Data to put in variable
↓ ↓ ↓
`val z: Double = (10.0*20.0)/2.0`

← The expression is evaluated first then the result is put in the variable

Review – Declaring and Initializing Variables

Anonymous Function Format

- General format for an anonymous function with parameters:



Anonymous Function Format

Anonymous Function with One Parameter

- Anonymous functions can take parameters.
- Need to specify the parameters before the function body.
- Unit means no return type (like void in other languages).

Specify parameter and return types
of function
(one string parm and no return type)

Give parameter a name to
use in the function body

```
val afm: (String) -> Unit = { m:String ->  
    println("Message: ")  
    println(m)  
}
```

Body of method
appears after the
arrow (->)

Call function using variable

```
afm("Hello")
```

Anonymous Function with One Parameter

Anonymous Function with Two Parameters and Return Value

- Need to specify parameters and return type before the function body.

Takes String and Int
parms. Returns an Int.

Give parameters names to
use in the function body

```
val afm: (String, Int) -> Int = { s:String, i:Int ->  
  println(s)  
  println(i)  
  i+5  
}
```

Return value. The last expression in the
body is the value that gets returned .

```
val result = afm("abc", 100)  
println(result)
```

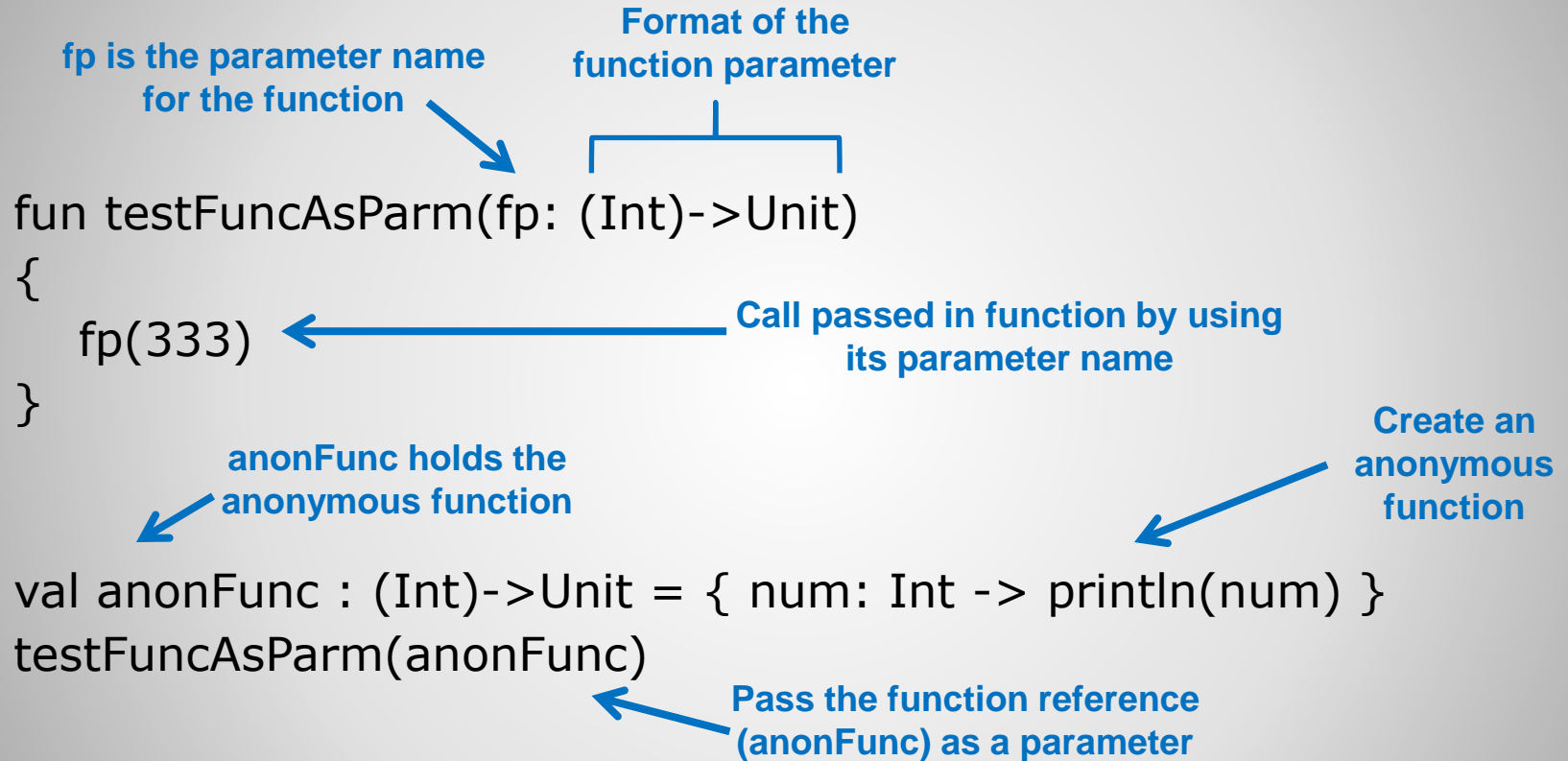
Call function using variable

Prints 105

Anonymous Function with Two Parameters and Return Value

Pass a Function Reference as a Parameter

- Function references can be passed as parameters to functions.



Pass a Function Reference as a Parameter

Pass Function Reference Parameter as a Trailing Lambda

- Kotlin has special syntax for passing in a function reference that is the last parameter in a function header.
- You can add a lambda expression right after the function header instead of passing it inside the () parenthesis.

```
fun testFuncAsParm(fp: (Int)->Unit) {  
    fp(333)  
}
```

← Function definition that has a function reference as the last parameter (the only parameter in this case)

```
testFuncAsParm( { num: Int -> println(num) } )
```

← Normal function call passing in a lambda (the lambda is inside the parenthesis)

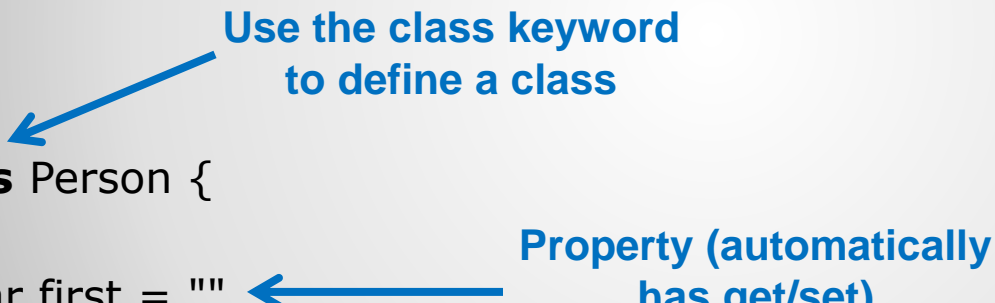
```
testFuncAsParm() { num: Int ->  
    println(num)  
}
```

← Function call with trailing lambda. The { } block after the function header is a trailing lambda. This block is actually being passed as a parameter.

Pass Function Reference Parameter as a Trailing Lambda

Class Definition

- Each class should be in its own file.
- The name of the file should be the name of the class followed by a .kt extension
- The class below should be in a class named Person.kt
- A property is a field+get/set (the field is the variable).
- Default visibility is public.



```
class Person {  
    var first = ""  
}
```

Class Definition

Create and Use a Class Instance

- A property is a field+get/set (the field is the variable).
- Default visibility is public.

```
class Person {  
    var first = ""  
}
```

← **Class definition**

```
fun myFunction()  
{  
    var p = Person()
```

← **Create an instance (no new keyword)**

```
    p.first = "Arthur"  
    println( p.first )  
}
```

← **Calls set on the first property**

← **Calls get on the first property**

Create and Use a Class Instance

Class - Access Modifiers

- **public** – Accessible from anywhere.
- **private** – Accessible from within the class.
- **protected** – Same as private except derived classes also have access.
- public is the default access modifier in Kotlin.

Class - Access Modifiers

Class - lateinit

- Use lateinit when a member property will be initialized later on.
- In the code below, first will be initialized when myFunction is called.

```
class Person {  
    private lateinit var first : String  
  
    fun myFunction()  
    {  
        first = "Rose"  
    }  
}
```

lateinit - Initialization
does not take place at
the declaration

Initialize the member
when myFunction runs

Class - lateinit

Class – Primary Constructor

- Part of the class header.
- The primary constructor does not have a normal function body.
- Use the constructor keyword to define it.
- Primary constructor parameters become member variables of the class.
- Note: The constructor keyword is optional if you are not using annotations or visibility modifiers.

This primary constructor
is defined to take one
string parameter

```
class Person constructor(var first: String)
```

```
{
```



first will be a member variable of
the class since it is in the
primary constructor

```
}
```

```
fun main(args: Array<String>) {
```

```
    var p = Person("Ali")
```

```
    println("p.first = " + p.first)
```

```
}
```

Call primary constructor passing
in the string "Ali". This will set
the first member variable of
Person to "Ali".

Class – Primary Constructor


Class – init Block

- You can add more initialization code to a class using an init block.
- The init block runs after the constructor.

Primary constructor
goes in the header



```
class Person constructor(var first: String)
{
    init {
        // More initialization code can go here...
    }
}
```



init block (code for the
primary constructor)

Class – init Block

Class with Two Constructors

- You can have more than one constructor in a class.

```
class Person constructor(var first: String)
{
    lateinit var last : String

    constructor(first: String, last: String) : this(first)
    {
        println("Other constructor ran")
        this.last = last
    }

    init {
        println("init block ran")
    }
}
```

Primary constructor is in the header

Other constructor function definition

The other construction must call the primary constructor. `this(first)` calls the primary constructor and initializes first. The primary constructor will run before the secondary constructor runs.

Initialize last

Class with Two Constructors

data Class Definition

- data class – Used to just store data values.
- If you need to write member methods with specific functionality then use a normal class instead.
- The primary constructor should have at least one member variable. Most of the time you will put all member variables in the primary constructor for a data class.

data class Person (
 var first: String,
 var last: String
)

Use the data keyword in
the class header

Two member variables in this data
class. They are both declared in
the primary constructor.

data Class Definition

Interface - Definition

- Use the interface keyword.
- Interfaces cannot store a "state" (no member variables).
- Functions can have an implementation.

Use interface keyword



```
interface MyInterface
{
    fun myFunction1()
    fun myFunction2() {
        // Code goes here...
    }
}
```

myFunction1 is abstract (no function body) and must be overridden in any classes that implement this interface

myFunction2 has a default implementation. Classes that implement this interface are NOT required to override this function (they can override if they want to).

Interface - Definition

Interface - Implementation

- Use : to have a class implement an interface.

MyClass implements
MyInterface



```
class MyClass : MyInterface {  
    override fun myFunction1() {  
        // Code goes here...  
    }  
}
```

myFunction1 **MUST** be overridden
because it is abstract in the
interface definition



```
    override fun myFunction2() {  
        // Code goes here...  
    }  
}
```

myFunction2 is overridden but it does not
have to be since the interface has a default
implementation (default implementation of
myFunction2 shown on previous slide)



Interface - Implementation

Array of Int

- Array of int.

```
//Declare and set values
```

```
var x = IntArray(3)
```

```
x[0] = 10
```

```
x[1] = 20
```

```
x[2] = 30
```

```
x.forEach(System.out::println)
```

```
// Declare with data
```

```
var y : Array<Int> = arrayOf(1, 2, 3, 4, 5)
```

```
y.forEach(System.out::println)
```

Array of Int

Array of String

- Array of String.

// Declare with data

```
var a : Array<String> = arrayOf("Mon", "Tues", "Wed")  
a.forEach(System.out::println)
```

Array of String

List of Int

- List of Int.

This list cannot be added to after initialization (need to define a mutable list to add other items after initialization)

```
// Declare with data
```

```
var numList : List<Int> = listOf(100, 200, 300)
```

```
numList.forEach(System.out::println)
```

You do not have to explicitly state the interface type. The type List<int> will be inferred.

```
var numList2 = listOf(400, 500, 600)
```

```
numList2.forEach(System.out::println)
```

Note: The following does not add to the list, it creates a whole new list instance with the original values and 400 in it (very inefficient):

```
numList += 400
```

List of Int

MutableList of Int

- A MutableList of Int allows you to add data items after initialization.

MutableList allows items to be added after initialization

Initializes with 100, 200, 300

```
// Declare with data  
var numList : MutableList<Int> = mutableListOf(100, 200, 300)  
var otherNumList : MutableList<Int> = mutableListOf()
```

```
numList.forEach(System.out::println)
```

Creates an empty list instead

```
numList.add(888)
```

Add 888 to the list

```
numList.forEach(System.out::println)
```

Note: Can infer the type instead using the following:

```
var numList = mutableListOf(100, 200, 300)
```

Mutable List of Int

MutableList of String

- A MutableList of Int allows you to add data items after initialization.

MutableList allows items to be added after initialization

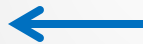


```
// Declare with data
```

```
var stringList = mutableListOf("Mon", "Tues", "Wed")
```

```
stringList.forEach(System.out::println)
```

```
stringList.add("Thurs")
```



Add Thurs to the list

```
stringList.forEach(System.out::println)
```

Mutable List of String

Coroutine

- An instance of a suspendable computation.
 - Similar to a thread in that it runs code currently with the other code in the program.
 - A coroutine is not associated with a particular thread (the same coroutine can be run on different threads).
 - A coroutine can suspend its execution on one thread and complete its execution on another thread.
-
- Taken from the following link:
<https://kotlinlang.org/docs/coroutines-basics.html>

Coroutine

Coroutine Example

- An instance of a suspendable computation.

```
fun normalMethod() {  
    runBlocking { // this: CoroutineScope  
        launch { // launch a new coroutine and continue  
            delay(1000L) // Non-blocking delay for 1 second  
            Log.d("MY_DEBUG", "2") // print after delay  
        }  
        Log.d("MY_DEBUG", "1") // Runs concurrently with launch block  
    }  
    Log.d("MY_DEBUG", "3") // Runs after runBlock finishes  
}
```

runBlocking. Connects coroutine code with non-coroutine code. The thread that executes a runBlock will block or stop until all code in that block completes.

launch. Coroutine builder that launches a new coroutine. Code inside launch runs concurrently with non-coroutine code.

Coroutine code (inside runblocking)

Code outside coroutine (outside runblocking)

Output
1
2
3

Coroutine Example

- End of Slides

End of Slides