Lesson 3:

Basic Tip Calculator

Instructor: Ahmet Geymen

About this lesson

- Lesson 3:
  - Kotlin fundamentals
    - Classes and objects
    - Functions
    - Lambda expressions
- Interacting with UI and state

Get started

Kotlin fundamentals

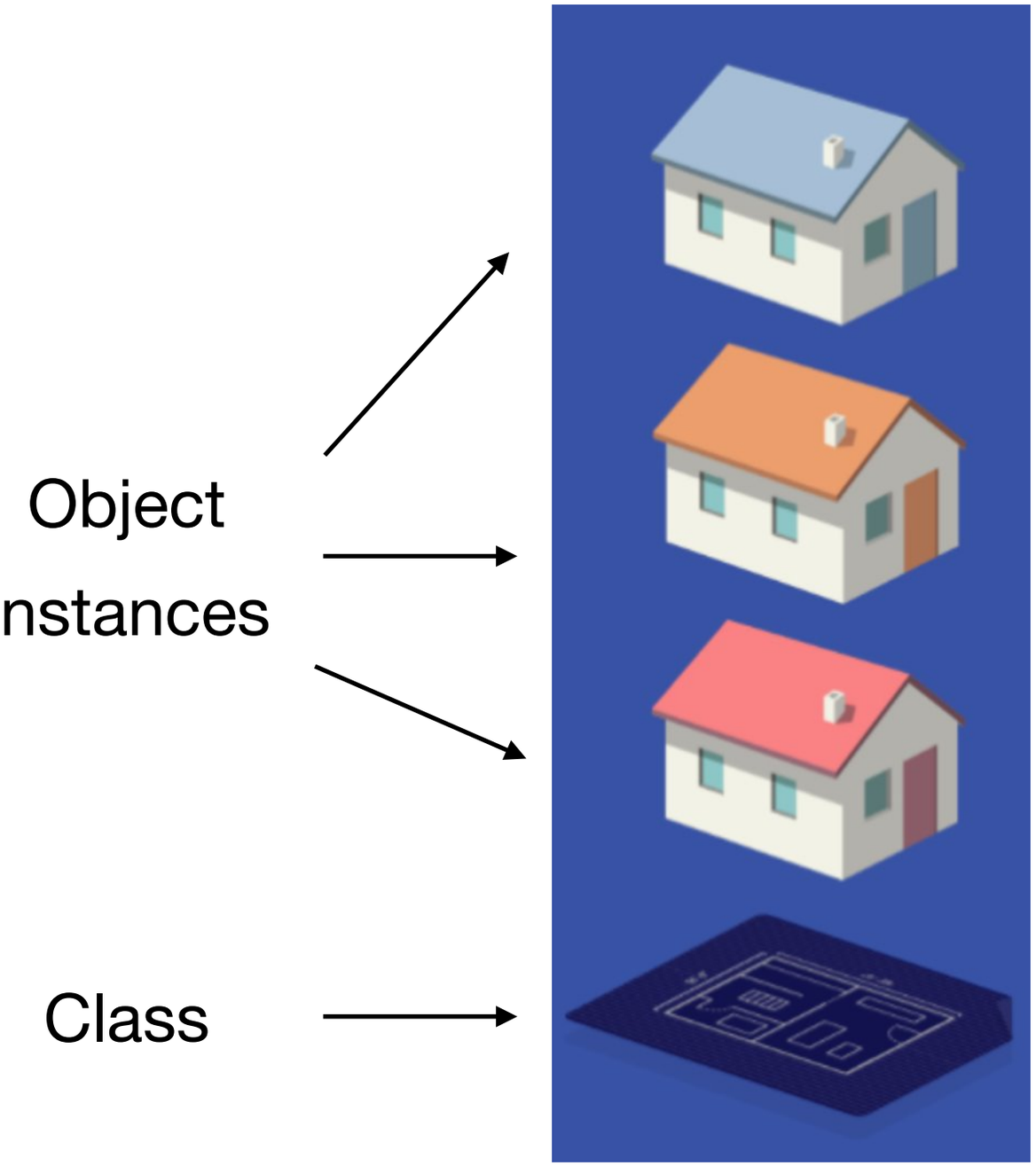
Classes

Class

- Classes are blueprints for objects
- Classes define methods that operate on their object instances

Class

- Classes are blueprints for objects
- Classes define methods that operate on their object instances



Class vs object instance

House Class

Data

- House color (String)
- Number of windows (Int)
- Is for sale (Boolean)

Behavior

- `updateColor()`
- `putOnSale()`

Define and use a class

- Class Definition

```
class name {
    body
}
```

- Create New Object Instance

```
val name = ClassName()
```

Define and use a class

- Class Definition

```
class House {
  val color: String = “white"
  val numberOfWindows: Int = 2
  val isForSale: Boolean = false

  fun updateColor(newColor: String){...}
  ...
}
```

- Create New Object Instance

```
val myHouse = House()
println(myHouse)
```

Constructors

When a constructor is defined in the class header, it can contain:

- No parameters
  ```
  class A
  ```
- Parameters
  - Not marked with `var` or `val` → copy exists only within scope of the constructor
    ```
    class B(x: Int)
    ```
  - Marked `var` or `val` → copy exists in all instances of the class
    ```
    class C(val y: Int)
    ```

Constructors

- `class A`

```
val aa = A()
```

- `class B(x: Int)`

```
val bb = B(12)
println(bb.x)
```

```
=> compiler error unresolved reference
```

- `class C(val y:Int)`

```
val cc = C(42)
println(cc.y)
```

```
=> 42
```

Default parameters

Class instances can have default values.

- Use default values to reduce the number of constructors needed
- Default parameters can be mixed with required parameters
- More concise (don’t need to have multiple constructor versions)

```
class Box(val length: Int, val width:Int = 20, val height:Int = 40)

val box1 = Box(100, 20, 40)
val box2 = Box(length = 100)
val box3 = Box(length = 100, width = 20, height = 40)
```

Primary constructor

```
class Person constructor(firstName: String) { /*...*/ }
```

```
class name constructor( parameters ){
    body
}
```

If the primary constructor does not have any annotations or visibility modifiers, the constructor keyword can be omitted

Primary constructor

```
class Person constructor(firstName: String) { /*...*/ }
```

```
class name constructor( parameters ){
    body
}
```

```
class Person(firstName: String) { /*...*/ }
```

If the primary constructor does not have any annotations or visibility modifiers, the constructor keyword can be omitted

Initializer block

- Any required initialization code is run in a special `init` block
- Multiple `init` blocks are allowed
- `init` blocks become the body of the primary constructor

Initializer block example

Use the init keyword:

```
class Square(val side: Int) {
    init {
        println(side * 2)
    }
}

val s = Square(10)
=> 20
```

Multiple constructors

- Use the `constructor` keyword to define secondary constructors
- Secondary constructors must either call:
  - The primary constructor using `this` keyword
  - Or another secondary constructor that calls the primary constructor
- Secondary constructor body is not required

Multiple constructors example

```
class Circle(val radius:Double) {
    constructor(name:String) : this(1.0)

    constructor(diameter:Int) : this(diameter / 2.0) {
        println("in diameter constructor")
    }

    init {
        println("Area: ${Math.PI * radius * radius}")
    }
}

val c = Circle(3)
```

Class properties

- Define properties in a class using `val` or `var`
- Access these properties using dot (.) notation with property name
- Set these properties using dot (.) notation with property name (only if declared a `var`)

Class properties example

```
class Person(var name: String)

fun main() {
    val person = Person("Alex")
    println(person.name)   ←————   Access with .<property name>
    person.name = "Joey"   ←————   Set with .<property name>
    println(person.name)
}
```

Custom getters and setters

If you don't want the default `get/set` behavior:

- Override `get()` for a property
- Override `set()` for a property (if defined as a `var`)

```
var propertyName: DataType = initialValue
        get() = ...
        set(value) {
            ...
          }
```

Custom getter example

```
class Person(val firstName: String, val lastName: String) {
    val fullName: String
        get() {
            return "$firstName $lastName"
        }
}


val person = Person("John", "Doe")
println(person.fullName)
=> John Doe
```

Custom setter example

```
var fullName: String = ""
    get() = "$firstName $lastName"
    set(value) {
        val components = value.split(" ")
        firstName = components[0]
        lastName = components[1]
        field = value
    }


person.fullName = "Jane Smith"
```

Member functions

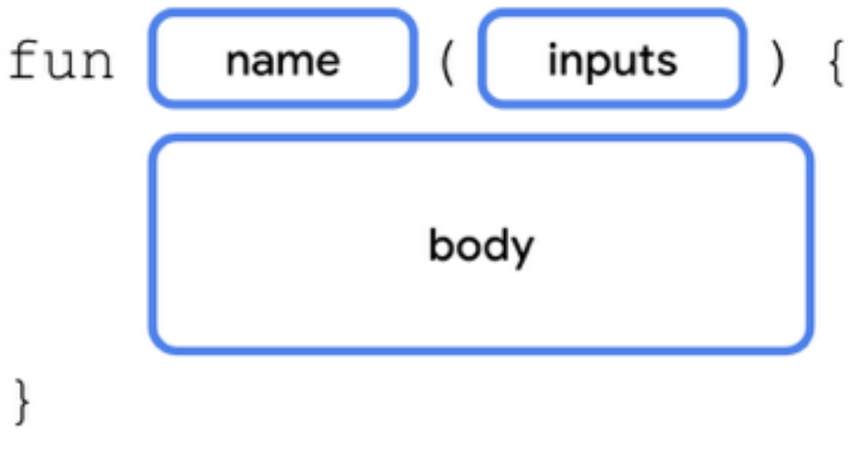
- Classes can also contain functions
- Functions:
  - Declare with `fun` keyword
  - Can have default or required parameters
  - Specify return type (if not Unit)

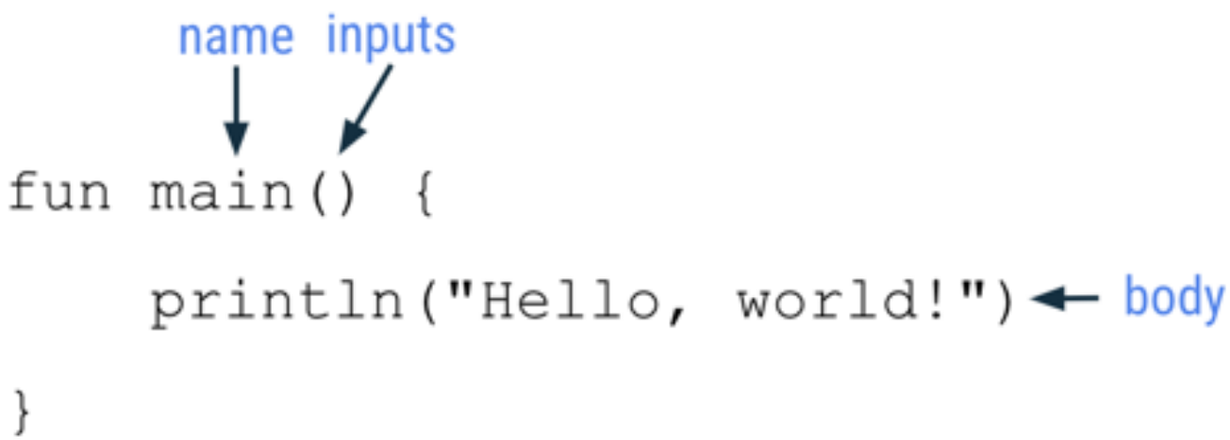
Functions

About functions

- A block of code that performs a specific task
- Breaks a large program into smaller modular chunks
- Declared using the `fun` keyword
- Can take arguments with either named or default values

Parts of a function





Unit returning functions

- If a function does not return any useful value, its return type is `Unit`.

```
fun printHello(name: String?): Unit {
    println("Hi there!")
}
```

Unit returning functions

- The `Unit` return type declaration is optional.

```
fun printHello(name: String?): Unit {
    println("Hi there!")
}
```

is equivalent to:

```
fun printHello(name: String?) {
    println("Hi there!")
}
```

Function arguments

Functions may have:

- Default parameters
- Required parameters
- Named arguments

Default parameters

- Default values provide a fallback if no parameter value is passed.

```
fun drive(speed: String = "fast") {
    println("driving $speed")
}
```

Use "=" after the type
to define default values

```
drive() ⇒ driving fast
drive("slowly") ⇒ driving slowly
drive(speed = "turtle-like") ⇒ driving turtle-like
```

Required parameters

- If no default is specified for a parameter, the corresponding argument is required.

```
fun tempToday(day: String, temp: Int) {
    println("Today is $day and it's $temp degrees.")
}


// day and temp are required parameters
```

Default versus required parameters

- Functions can have a mix of default and required parameters.

```
fun reformat(
    str: String,
    normalizeCase: Boolean = true,
    upperCaseFirstLetter: Boolean = true,
    wordSeparator: Char = ' '
) { /*...*/ }


reformat("This is a long String!")
```

Named arguments

- To improve readability, use named arguments for required arguments.

```
fun reformat(
    str: String,
    normalizeCase: Boolean = true,
    upperCaseFirstLetter: Boolean = true,
    wordSeparator: Char = ' '
) { /*...*/ }


reformat(
   str = "This is a short String!",
   normalizeCase: Boolean = true,
   upperCaseFirstLetter = false
   )
```

Single-expression functions

- Single-expression functions make your code more concise and readable.

```
fun double(x: Int): Int {
    x * 2
}

fun double(x: Int): Int = x * 2

fun double(x: Int) = x * 2
```

Lambda Expressions

Kotlin functions are first-class

- Kotlin functions can be stored in variables and data structures
- They can be passed as arguments to, and returned from, other higher-order functions
- You can use higher-order functions to create new "built-in" functions

Lambda expressions

- A lambda is an expression that makes a function that has no name.
- A lambda expression is always surrounded by curly braces.
- The body goes after the ->
- If the inferred return type of the lambda is not Unit, the last (or possibly single) expression inside the lambda body is treated as the return value.

```
val sum: (Int, Int) -> Int = { x: Int, y: Int -> x + y }
```

Function type Lambda expression Function body

Higher-order functions

- Higher-order functions take functions as parameters, or return a function.

```
fun encodeMsg(msg: String, encode: (String) -> String): String {
    return encode(msg)
}
```

The body of the code calls the function that was passed as the second argument, and passes the first argument along to it.

Higher-order functions

- To call this function, pass it a string and a function.

```
val enc1: (String) -> String = { input -> input.toUpperCase() }

println(encodeMsg("abc", enc1))
```

Using a function type separates its implementation from its usage.

Passing a function reference

- Use the `::` operator to pass a named function as an argument to another function.

```
fun enc2(input: String): String = input.reversed()

encodeMessage("abc", ::enc2)
```

The `::` operator lets Kotlin know that you are passing the function reference as an argument, and not trying to call the function.

Last parameter call syntax

- Kotlin prefers that any parameter that takes a function is the last parameter.

```
encodeMessage("acronym", { input -> input.toUpperCase() })
```

- You can pass a lambda as a function parameter without putting it inside the parentheses.

```
encodeMsg("acronym") { input -> input.toUpperCase() }
```

Last parameter call syntax

- Many Kotlin built-in functions are defined using last parameter call syntax.

```
inline fun repeat(times: Int, action: (Int) -> Unit)
```

```
repeat(3) {
    println("Hello")
}
```

Workshop