Introduction to Kotlin

Why should I consider yet another programming language?

Hello!

I am Andy Bowes

I am here because Kotlin has made coding fun & productive again.

Most good programmers do programming not because they expect to get paid or get adulation by the public, but because it is fun to program.

“ *Linus Torvalds*

What is a ‘Kotlin’?

- ‘New’ programming language
  - Developed since 2011, v1.0 released Feb 2016
- Developed by JetBrains
  - Makers of IntelliJ & Android Studio
- Runs on Java Virtual Machine (JVM)
- Open Source
- A ‘Better Java’
- Named after an island in the Baltic Sea

Why was I looking for a new language?

- Experience of a number of languages
- Mainly Java & Python
- Python:
  - Pros – Terse code, good support for functional coding
  - Cons – Dynamically typed, uncompiled.
- Java:
  - Pros – JVM provides portability, Statically Typed
  - Cons – Verbose, functional coding via interfaces
- Learning a new language also changes the way that you use your existing language

Characteristics of ideal language

- Runs on multiple platforms
  - Natively compiled or via JVM/CLR
- Statically typed & compiled
- True support for functional programming
- Terse & productive
- Wide community
  - Build tools (Gradle, Maven)
  - IDEs – IntelliJ, Eclipse
  - Availablity of frameworks (HTTP, JSON, ...)
    - Ability to reuse existing libraries

Characteristics of ideal language

- Potential compiled languages
  - Go – Google, statically typed with garbage collection
  - Rust – Mozilla, safe, concurrent systems language
- Decided JVM provided better platform
  - Portable to more target platforms
  - Better instrumentation & support
  - Less ‘lock-in’, easier to switch to alternative language.

What's wrong with plain old Java ?

- It's verbose.
  - Too much 'boilerplate' code.
  - Multiple overload methods/constructors.
- Runtime Errors
  - Null Pointers
  - Class Cast exceptions
- Functional paradigm is still an afterthought.
  - Java 8 over-promised & under-delivered
  - Lambda functions treated as instances of interfaces

```
Function<String,Integer> atoi = s -> Integer.valueOf(s);
```

JVM Languages considered before finding Kotlin

Scala

- The ‘go-to’ next generation JVM language
- Large development & user community
- Functional programming is almost mandatory
- Steep Learning Curve
  - Operational overloading makes code obscure
  - API Documentation reads like a academic exercise
- Conclusion
  - Enjoyed writing Scala but would hate to maintain it !!

JVM Languages considered before finding Kotlin

Closure

- Lisp-based JVM language
- Smaller user community
- Dynamically typed
- Tooling
  - Plugins for Eclipse, Gradle & Maven
- Conclusion
  - Dynamic typing & lack of refactoring support means I wouldn't want to support/develop a large project

JVM Languages considered before finding Kotlin

Frege

- Haskell-based JVM language
  - Strictly functional language
- Very small user community
- Statically typed
- Lack of tooling support
- Conclusion
  - Lack of maturity & 3rd Party libraries
  - Java interoperability is complex

10 reasons to consider Kotlin as your next language

1 100% Java Interoperability

- All code compiles to pure Java byte-code
- Kotlin classes can invoke methods in Java classes
- Java classes can invoke Kotlin functions
- Kotlin can use standard Java libraries
  - Many Kotlin specific libraries are available but can continue to use familiar Java libraries
- Allows incremental migration to Kotlin from Java
- Deploy mixed applications as a single artifact

2 Statically Typed Language

- All variables, parameters & return values have a statically defined type
- Many variables/return types are inferred
- Allows the IDE to suggest appropriate methods
- Fewer runtime errors
- Refactoring can be performed with more confidence

3 Simple Data Objects

- Define DTO's in a single line of code

```
data class User(val id:String, val name: String, val age: Int)
```

- DTO is immutable
  - Mutable properties use **var** rather than **val**
- Automatically generates boiler-plate code
  - Property accessors
  - equals()/hashCode()
  - toString()
  - copy() - allows properties to be modified
- Classes default to 'closed' (Java final)

Simple Data Objects

Java

```java
import java.util.Date;

public class Person {

    private final String id;
    private final String forename;
    private final String surname;
    private final Date dateOfBirth;

    public Person(String id, String forename, String
surname, Date dateOfBirth) {
        this.id = id;
        this.forename = forename;
        this.surname = surname;
        this.dateOfBirth = dateOfBirth;
    }

    public String getId() {
        return id;
    }

    public String getForename() {
        return forename;
    }

    public String getSurname() {
        return surname;
    }
```

Kotlin

```kotlin
import java.util.Date

data class Person(val id: String, val forename: String, val
surname: String, val dateOfBirth: Date)
```

Simple Data Objects

Java

```
...

    public Date getDateOfBirth() {
        return dateOfBirth;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return
false;

        Person person = (Person) o;

        if (!id.equals(person.id)) return false;
        if (!forename.equals(person.forename)) return false;
        if (!surname.equals(person.surname)) return false;
        return dateOfBirth.equals(person.dateOfBirth);

    }
...
```

Kotlin

```
Nothing to see here ...
```

Simple Data Objects

Java

```
...
    @Override
    public int hashCode() {
        int result = id.hashCode();
        result = 31 * result + forename.hashCode();
        result = 31 * result + surname.hashCode();
        result = 31 * result + dateOfBirth.hashCode();
        return result;
    }

    @Override
    public String toString() {
        return "Person{" +
                "id='" + id + '\'' +
                ", forename='" + forename + '\'' +
                ", surname='" + surname + '\'' +
                ", dateOfBirth=" + dateOfBirth +
                '}';
    }
}
```

Kotlin

Still nothing to see here ...

4 No more Null Pointer Exceptions *

- Need to explicitly state that variable/parameter allows nulls
  - **val** name: String – Cannot be assigned null
  - **val** name: String? – Can be set to a value or null
- Unsafe calls are prevented by the compiler
- Must check potentially null objects before use
- Or use Safe Calls
  - name?.length - returns the length or null
  - user?.department?.head?.name - can chain nullable calls
- Elvis Operator (thanks for nothing Groovy)

* Unless you really want them

Null Safety - Compile Time Checks

```
fun getCheckNulls(): Int{
    var a: String = "abc"
    a = null // compilation error, attempt to assign Null to NonNull variable

    // To allow nulls, we can declare a variable as nullable string, written String?:
    var b: String? = "abc"
    b = null // ok

    b = getName()  // Call a function that returns a Nullable String
    b.length // Compilation Error, invoking method on potentially Null Object
    if (b != null){
        b.length // Can now execute method on the variable after Null check
    }

    var i = b?.length // Returns length or Null as an Int?

    return b?.length ?: -1 // Returns either the length of the String or -1 if b is null
}
```

5 Smart Casting

- Type checks with the `is` or `!is` operator
- Compiler tracks `is` checks and performs automatic cast
- No need to create extra variables for cast results

Smart Casting Examples

```
// Pass generic object into the Function
fun demo(x: Any, y:Any?) {

    if (x is String) {
        print(x.length) // x is automatically cast to String
    }

    // The compiler is smart enough to know a cast to be safe if a negative check leads to a return:
    if (x !is String) return
    print(x.length) // x is automatically cast to String

    // Can be applied to each option in a when() statement
    when (y) {
        is Int -> print(y + 1)
        is String -> print(y.length + 1)
        is IntArray -> print(y.sum())
    }

    // Unsafe Cast - will fail if x is Null or Not a String
    var answer: String = y as String

    // Handle Null values & Strings
    var answer2: String? = y as String?

    // Safe Cast - Casts to String or null if Cast fails
    val answer3: String? = y as? String

}
```

6 Named Function Parameters with defaults

- Very similar to Python
- Also applies to Constructors
- Default Values
  - Reduces the need for overloaded methods
  - `fun read(b: Array<Byte>, off: Int = 0, len: Int = b.size())`
- Named Arguments
  - Improves readability
  - Pick relevant parameters at invocation
  - `read(bytes,len=1024)`

Overloaded Constructors

Java

```java
public class MyHashMap<K,V>{

   private static final double DEFAULT_FILLFACTOR = 0.75;
   private static final int DEFAULT_CAPACITY = 16;

   private int capacity;
   private final double fillFactor;

   public MyHashMap(){
       this(DEFAULT_CAPACITY);
   }

   public MyHashMap(int capacity) {
       this(capacity, DEFAULT_FILLFACTOR);
   }

   public MyHashMap(int capacity, double fillFactor) {
       this.fillFactor = fillFactor;
       this.capacity = capacity;
   }
}
```

Kotlin

```kotlin
class MyHashMapKt<K,T>(var capacity:Int = 16,
                       val fillFactory:Double = 0.75)
```

7 Extension Functions

- Add functions onto existing classes even those in stdlib
- No need to create a sub-classes to extend functionality
- For example:
  - ```
    fun MutableList<Int>.swap(index1: Int, index2: Int) {
      val tmp = this[index1] // 'this' corresponds to the list
      this[index1] = this[index2]
      this[index2] = tmp
    }
    ```
- This method is now available on all MutableList<Int>
    ```

Using Extension Methods

```
fun String.toCamelCase() : String {
    return this.split(' ')
                .map { it -> it.toLowerCase().capitalize() }
                .joinToString(separator = "")
}


fun main(args: Array<String>) {
    println("this is a test".toCamelCase())
    println("ANOTHer Test Case".toCamelCase())
}
```

8 Functional Programming

- Functions can be defined as variables & passed as parameters
- Kotlin can ‘inline’ some of these functions
- Java 8–style Streaming
- Handle collections as ‘Sequences’
  - Lazily evaluated Collections
  - Similar to Python Generators
- Supports tail recursion
  - Optimises recursion to a standard loop structure
  - Avoids stack overflow

Synchronised Locks

```
import java.io.File
import java.util.concurrent.locks.Lock
import java.util.concurrent.locks.ReentrantLock

inline fun <T>lock(lock:Lock, body: () -> T):T{
    lock.lock()
    try {
        return body()
    } finally {
        lock.unlock()
    }
}

class FileAccess(val file: File){
    private final val myLock: Lock
    init {
        this.myLock = ReentrantLock()
    }

    fun writeToFile(contents: ByteArray) {
        lock(myLock, {file.writeBytes(contents)})
    }
}
```

9

Inheritance & Composition

- Supports standard Java Inheritance
  - Extend single parent class
  - Implement multiple interfaces
- Abstract interfaces can define properties & functions
- Implementation by composition
  - Zero boilerplate coding
  - Compiler automatically generates forwarding functions

Interfaces with Properties

```
interface User {
    val nickname: String

    fun capitaliseName(): String {
        return this.nickname.toUpperCase()
    }
}

class PrivateUser(override val nickname: String) : User

class SubscribingUser(val email: String) : User {
    override val nickname: String
        get() = email.substringBefore('@')
}

class FacebookUser(val accountId: Int) : User {
    override val nickname = getFacebookName(accountId)
}
```

Implementation by Composition

```
interface Base {
    fun print()
}

class BaseImpl(val x: Int) : Base {
    override fun print() { println(x) }
}

interface Closeable{
    fun close()
}

class ClosableImpl():Closeable{
    override fun close(){ println("Closing item")}
}

/**
Delegate function calls to classes which implement the interfaces
**/
class Derived(b: Base, c: Closeable) : Base by b, Closeable by c

fun main(args: Array<String>) {
    val derived = Derived(BaseImpl(15), ClosableImpl())
    derived.print()
    derived.close()
}
```

10 Android Development

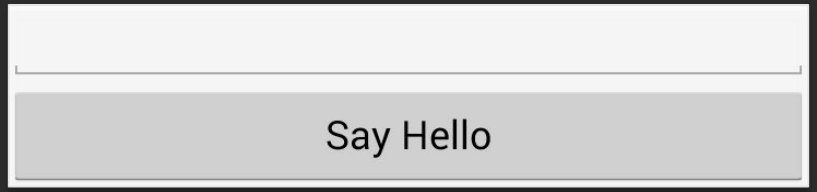
- Integrates with Android Studio
  - Both developed by JetBrains
- Programmatic layout development
  - Implemented by Anko library
  - Produces smaller code base than XML layouts
- Simplifies integration with SQLite database

Android Programming

Uses Anko DSL to create a simple layout with a field and a button.

Include lambda expression to add listener to button click.

```
verticalLayout {
    val name = editText()
    button("Say Hello") {
        onClick { toast("Hello, ${name.text}!") }
    }
}
```



Kotlin Summary - Part 1

Concise

Reduction of code verbosity

Increases clarity & maintainability.

Programs typically 30% smaller than Java.

Safe

Statically Typed

Null Checking

Safe Casting

Supported

Developed by JetBrains

Active Community

Multiple IDEs inc IntelliJ & Eclipse

Kotlin Summary - Part 2

Compatible

100% compatible with Java classes

Can reuse existing Java libraries

Reuse existing Java build processes

Portable

Runs anywhere that you can install a Java 1.6+ JVM

Deployed as standard Java application

Versatile

Multitude of environments:

Server Side

Rich Client - JavaFX

Android Apps

Even compiles to JavaScript

Thanks!

Any questions ?

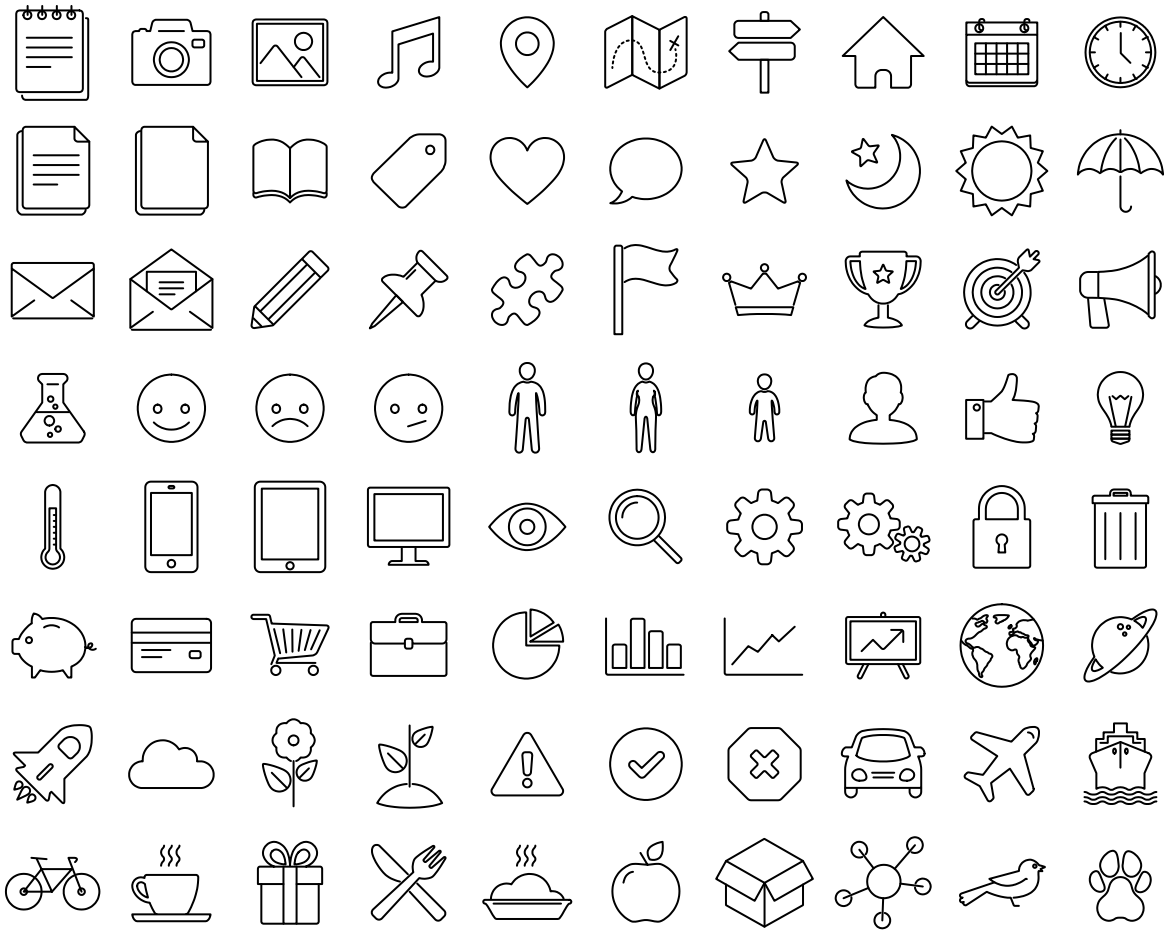
You can find me at

- @AndyJBowes
- andy.bowes@bjss.com
- http://andybowes.me.uk

Credits

Special thanks to all the people who made and released these awesome resources for free:

- Presentation template by SlidesCarnival



SlidesCarnival icons are editable shapes.

This means that you can:

- Resize them without losing quality.
- Change line color, width and style.

Isn't that nice? :)

Examples: