Kotlin Features:

- Kotlin does not need ; to end statements
- Kotlin is null-safe
- Kotlin is 100% Java interoperable
- Kotlin has no primitives (but optimizes their object counterparts for the JVM, if possible)
- Kotlin classes have properties, not fields
- Kotlin offers data classes with auto-generated equals/hashCode methods and field accessors
- Kotlin only has runtime Exceptions, no checked Exceptions
- Kotlin has no new keyword.
- Creating objects is done just by calling the constructor like any other method.
- Kotlin supports (limited) operator overloading.
- Kotlin can not only be compiled to byte code for the JVM, but also into Java Script, enabling you to write both backend and frontend code in Kotlin
- Kotlin is fully compatible with Java 6, which is especially interesting in regards for support of (not so) old Android devices
- Kotlin is an officially supported language for Android development
- Kotlin's collections have built-in distinction between mutable and immutable collections.
- Kotlin supports Coroutines (experimental)

Kotlin Environment:

- To write kotlin code we required an IDE called IntelliJ IDEA, go through the following URL to download IntelliJ IDEA.
https://www.jetbrains.com/idea/
- To Install Intellij IDEA , JDK is pre requisite.
- IntelliJ is a product of Jet Brains.
- Android Studio will run on top of IntelliJ.

Hello World Program:

```
fun main(args: Array<String>) {
println("Hello, world!")
}
```

Prathmesh P Joshi

- **When targeting the JVM, the function will be compiled as a static method in a class with a name derived from the filename**
- **To run this program , right click on code window select Run 'FileNameKt'**

Declaring Variables:

- **In Kotlin, variable declarations look a bit different than Java's: val i : Int = 42**
- **They start with either val or var, making the declaration final ("value") or variable.**
- **The type is noted after the name, separated by a :**
- **Thanks to Kotlin's type inference the explicit type declaration can be omitted if there is an assignment with a type the compiler is able to unambiguously detect**

Java Kotlin

int i = 42; var i = 42 (or var i : Int = 42)
final int i = 42; val i = 42

Declaring Functions:

Basic Functions:

Functions are declared using the fun keyword, followed by a function name and any parameters. You can also specify the return type of a function, which defaults to Unit. The body of the function is enclosed in braces {}. If the return type is other than Unit, the body must issue a return statement for every terminating branch within the body.

```
fun sayMyName(name: String): String
{
 return "Your name is $name"
}
```

Prathmesh Pjoshi

Inline Functions:

Functions can be declared inline using the inline prefix, and in this case they act like macros in C - rather than being called, they are replaced by the function's body code at compile time. This can lead to performance benefits in some circumstances, mainly where lambdas are used as function parameters.

```
inline fun sayMyName(name: String) = "Your name is $name"
```

(inline keyword is optional)

Lambda Functions :

Lambda functions are anonymous functions which are usually created during a function call to act as a function parameter. They are declared by surrounding expressions with {braces} - if arguments are needed, these are put before an arrow ->.

```
{   name: String ->
    "Your name is $name" //This is returned     }
```

Conditional Statements

When-statement argument matching:

When given an argument, the when-statement matches the argument against the branches in sequence. The matching is done using the == operator which performs null checks and compares the operands using the equals function. The first matching one will be executed.

```
when (x) {
            "English" -> print("How are you?")
       "German" -> print("Wie geht es dir?")
       else -> print("I don't know that language yet :(")
}
```

The when statement also knows some more advanced matching options:

```
val names = listOf("John", "Sarah", "Tim", "Maggie")
 when (x) {
                in names -> print("I know that name!")
                !in 1..10 -> print("Argument was not in the range from 1 to 10")
```

Pratmesh Pjoshi

```
        is String -> print(x.length) // Due to smart casting, you can use
String-functions here
    }
```

When-statement as expression:

Like if, when can also be used as an expression:

```
val greeting = when (x) {
 "English" -> "How are you?"
"German" -> "Wie geht es dir?"
else -> "I don't know that language yet :("
}
print(greeting)
```

To be used as an expression, the when-statement must be exhaustive, i.e. either have an else branch or cover all possibilities with the branches in another way.

Loops in Kotlin:

You can loop over any iterable by using the standard for-loop:

```
 val list = listOf("Hello", "World", "!")
for(str in list) {
 print(str)
}
```

Lots of things in Kotlin are iterable, like number ranges:

```
 for(i in 0..9) {
print(i)
 }
```

If you need an index while iterating:

```
for((index, element) in iterable.withIndex()) {
print("$element at index $index")
}
```

There is also a functional approach to iterating included in the standard library, without apparent language constructs, using the forEach function:

```
 iterable.forEach {
print(it.toString())
}
```

Prathmesh Ploshi

While Loops

While and do-while loops work like they do in other languages:

```
while(condition) {
 doSomething()
 }
do {
doSomething()
 } while (condition)
```

In the do-while loop, the condition block has access to values and variables declared in the loop body.

Ranges in Kotlin :

Range expressions are formed with rangeTo functions that have the operator form .. which is complemented by in and !in. Range is defined for any comparable type, but for integral primitive types it has an optimized implementation

Integral Type Ranges :

Integral type ranges (IntRange , LongRange , CharRange) have an extra feature: they can be iterated over. The compiler takes care of converting this analogously to Java's indexed for-loop, without extra overhead

```
for (i in 1..4)
print(i) // prints "1234"
for (i in 4..1) print(i) // prints nothing
```

downTo(): downTo( ) function if you want to iterate over numbers in reverse order? It's simple. You can use the downTo() function defined in the standard library

```
for (i in 4 downTo 1)
print(i) // prints "4321"
```

step(): function Is it possible to iterate over numbers with arbitrary step, not equal to 1? Sure, the step() function will help you

```
 for (i in 1..4 step 2)
print(i) // prints "13"
for (i in 4 downTo 1 step 2)
print(i) // prints "42"
```

until : function To create a range which does not include its end element, you can use the until function:

Prathmesh P Joshi

```
for (i in 1 until 10) {  // i in [1, 10), 10 is excluded
println(i)
}
```

Constructor in Kotlin:

Primary Constructor :

A class in Kotlin can have a primary constructor and one or more secondary constructors. The primary constructor is part of the class header: it goes after the class name (and optional type parameters).

```
class Person constructor(firstName: String) { }
```

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

```
class Person(firstName: String) {   }
```

Secondary Constructor:

The class can also declare secondary constructors, which are prefixed with constructor:

```
class Person {
    constructor(parent: Person) {
        parent.children.add(this)
    }
}
```

If the class has a primary constructor, each secondary constructor needs to delegate to the primary constructor, either directly or indirectly through another secondary constructor(s). Delegation to another constructor of the same class is done using the this keyword:

```
class Person(val name: String) {
    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }
}
```

Prathmesh P Joshi

Creating instances/Object of classes:

To create an instance of a class, we call the constructor as if it were a regular function:
val invoice = Invoice()
val customer = Customer("Joe Smith")

Note that Kotlin does not have a new keyword.
Classes and Inheritance
Classes in Kotlin are declared using the keyword class:
class Invoice { }
The class declaration consists of the class name, the class header (specifying its type parameters, the primary constructor etc.) and the class body, surrounded by curly braces. Both the header and the body are optional; if the class has no body, curly braces can be omitted.
class Empty

Inheritance

All classes in Kotlin have a common superclass Any, that is the default superclass for a class with no supertypes declared:
class Example // Implicitly inherits from Any
Note: Any is not java.lang.Object; in particular, it does not have any members other than equals(),hashCode() and toString(). Please consult the Java interoperability section for more details.
To declare an explicit supertype, we place the type after a colon in the class

header:

open class Base(p: Int)
class Derived(p: Int) : Base(p)
The open annotation on a class is the opposite of Java's final: it allows others to inherit from this class. By default, all classes in Kotlin are final, which corresponds to Effective Java, 3rd Edition, Item 19: Design and document for inheritance or else prohibit it.
If the derived class has a primary constructor, the base class can (and must) be initialized right there, using the parameters of the primary constructor.

Prathmesh P Joshi

If the class has no primary constructor, then each secondary constructor has to initialize the base type using the super keyword, or to delegate to another constructor which does that. Note that in this case different secondary constructors can call different constructors of the base type:

```
class MyView : View {
        constructor(ctx: Context) : super(ctx)
        constructor(ctx: Context, attrs: AttributeSet) : super(ctx, attrs)
}
```

Interfaces

Interfaces in Kotlin are very similar to Java 8. They can contain declarations of abstract methods, as well as method implementations. What makes them different from abstract classes is that interfaces cannot store state. They can have properties but these need to be abstract or to provide accessor implementations.

An interface is defined using the keyword interface

```
interface MyInterface {
   fun bar()
   fun foo() {
     // optional body
   }
}
```

Implementing Interfaces

A class or object can implement one or more interfaces

```
class Child : MyInterface {
   override fun bar() {
      // body
   }
}
```

Prathmesh P Joshi