kotlin: NewHope

In a Java 6 Wasteland

Statically typed programming language targeting the JVM and JavaScript

100% interoperable with Java™

concise

/kənˈsīs/
adjective

giving a lot of information clearly and in a few words;
brief but comprehensive.

expressive

/ik'spresiv/
adjective

effectively conveying thought or feeling.

safe

/sāf/
adjective

protected from or not exposed to danger or risk; not likely to be harmed or lost.

versatile

/ˈvərsədl/
adjective

able to adapt or be adapted to many different functions or activities.

interoperable

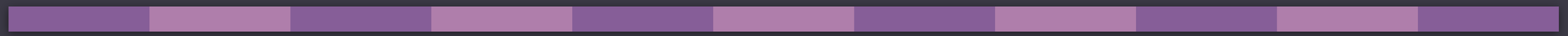
/ˌin(t)ərˈäp(ə)rəb(ə)l/
adjective

(of computer systems or software) able to exchange and make use of information.

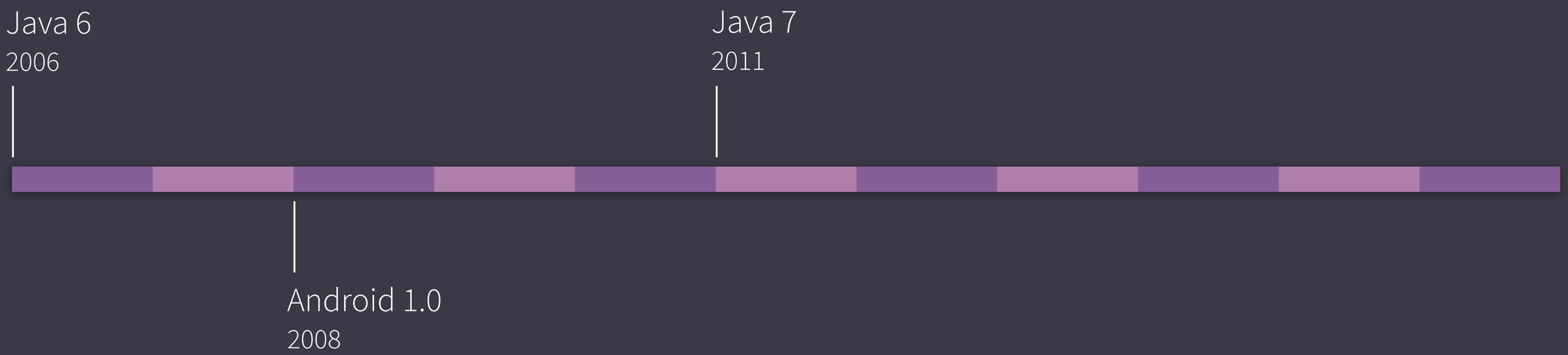
Why not wait for Java 8?

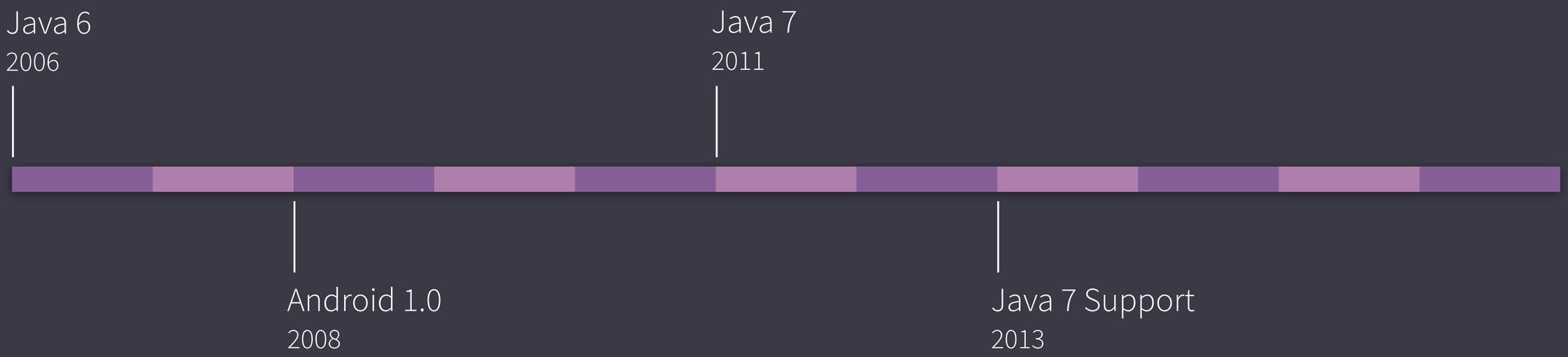
Java 6

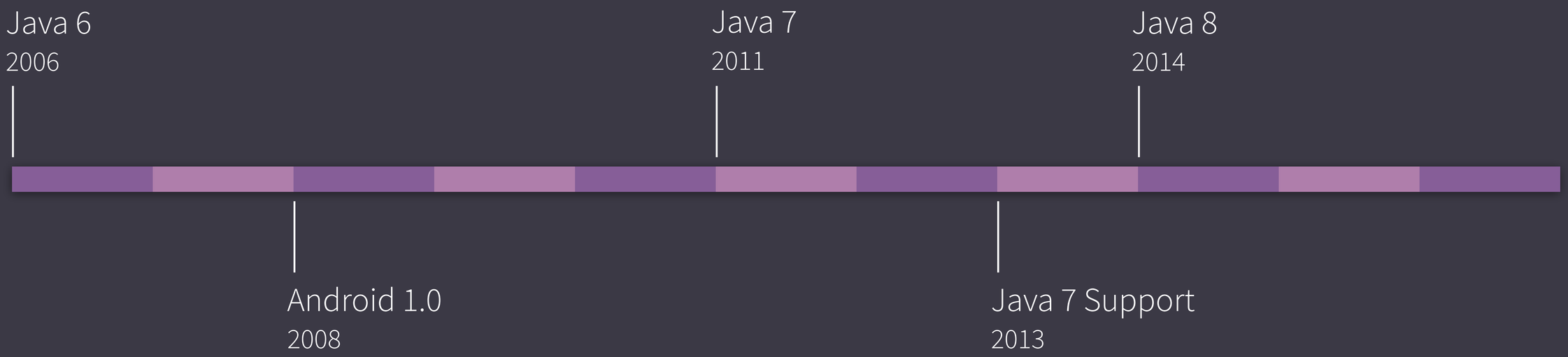
2006

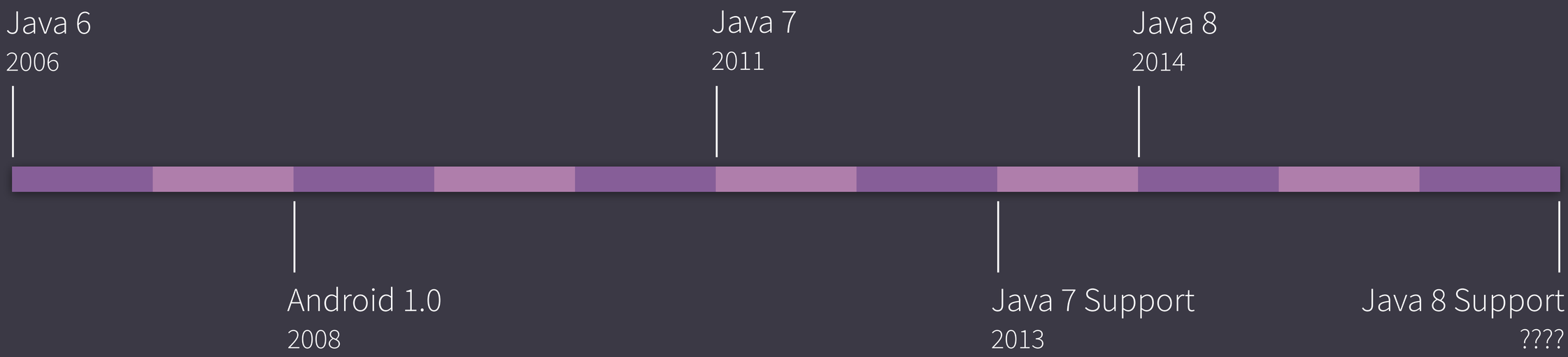


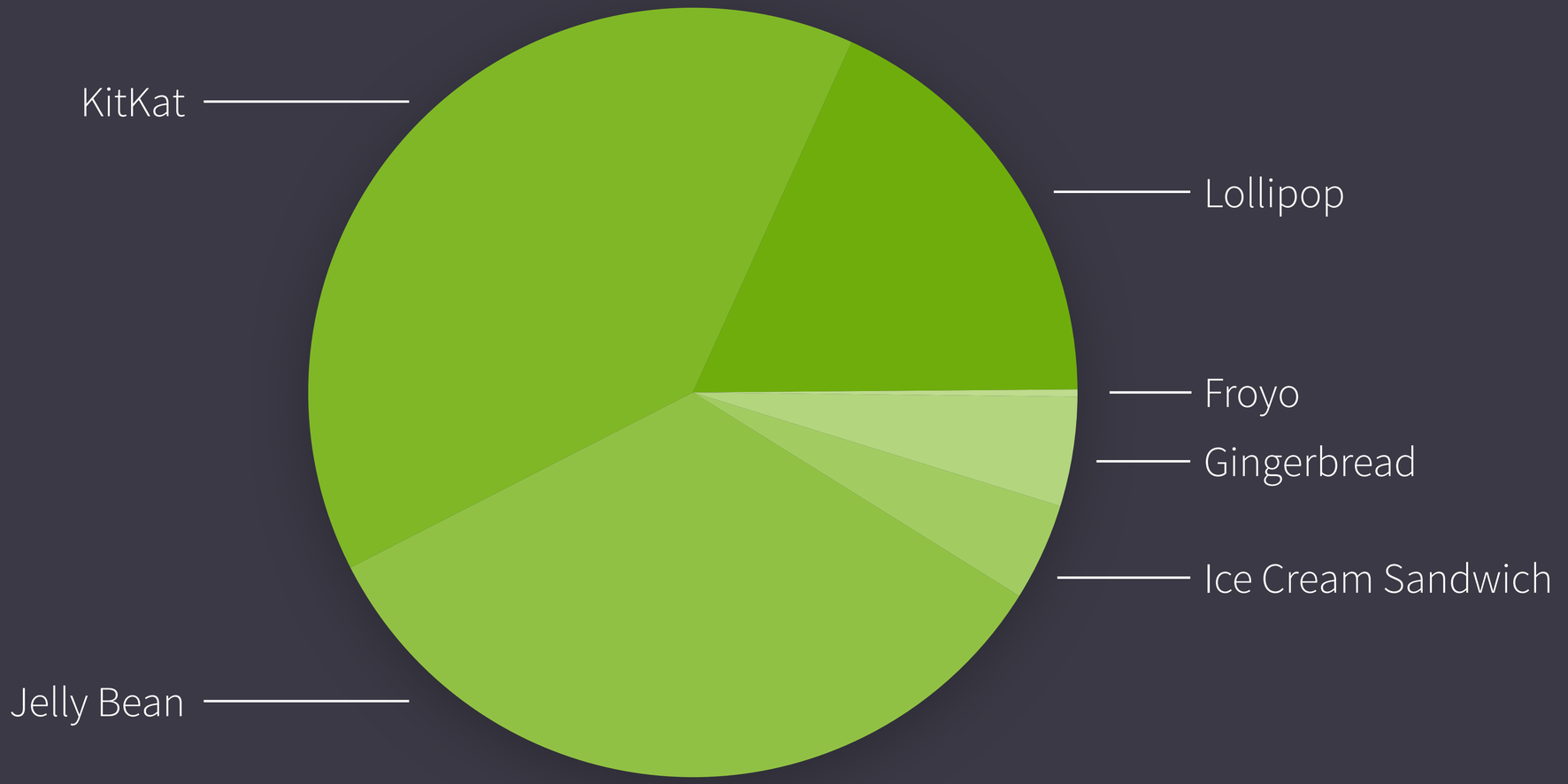

Java 6
2006
Android 1.0
2008

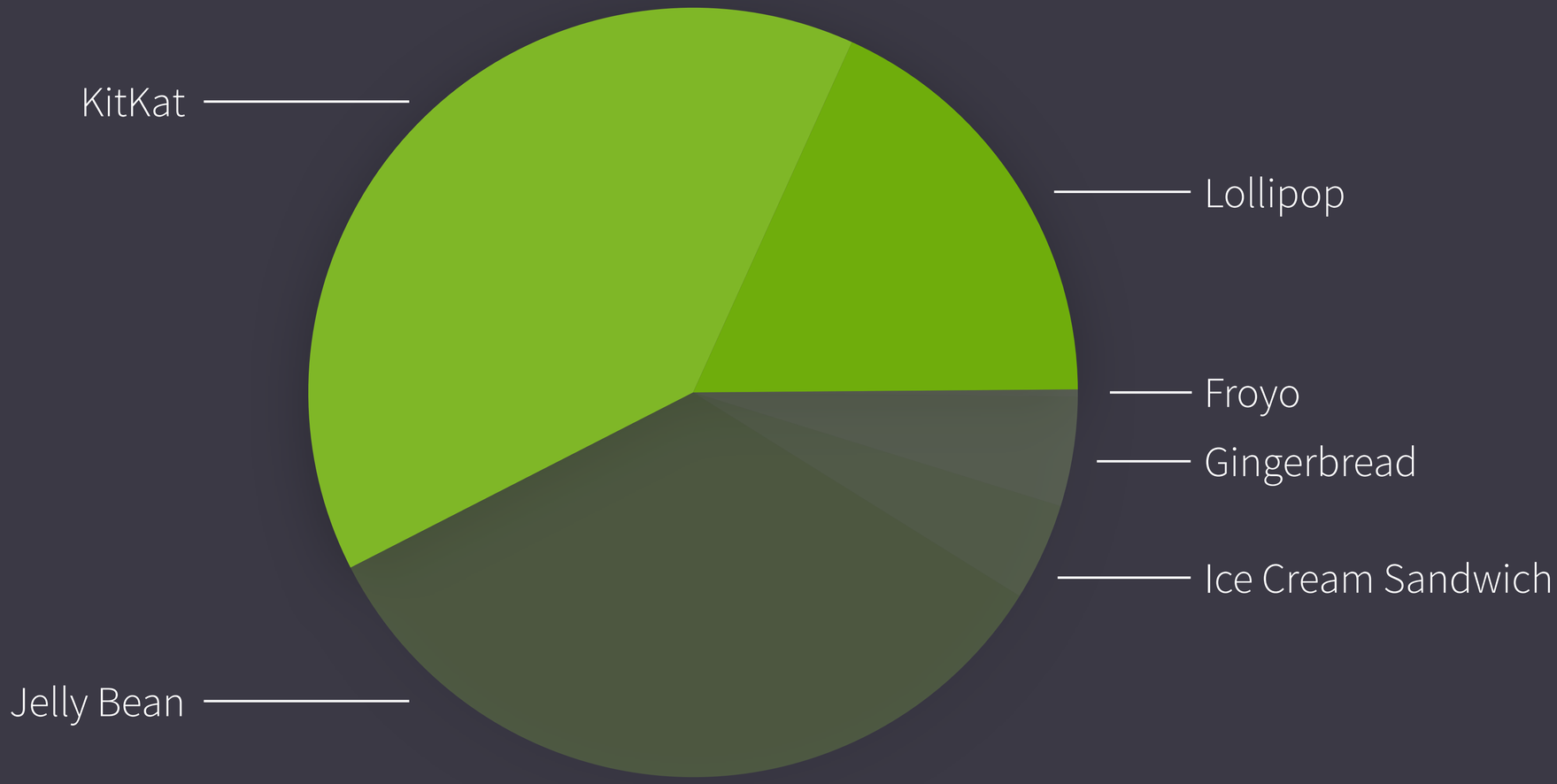

Java 6
2006
Java 7
2011
Android 1.0
2008


Java 6
2006
Java 7
2011
Android 1.0
2008
Java 7 Support
2013


Java 6
2006
Java 7
2011
Java 8
2014
Android 1.0
2008
Java 7 Support
2013


Java 6
2006
Java 7
2011
Java 8
2014
Android 1.0
2008
Java 7 Support
2013
Java 8 Support
????


KitKat
Lollipop
Froyo
Gingerbread
Ice Cream Sandwich
Jelly Bean


KitKat
Lollipop
Froyo
Gingerbread
Ice Cream Sandwich
Jelly Bean

Problems with Java

Null references

“I call it my billion-dollar mistake… [which] has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.”

— Tony Hoare

Raw types

```
List numbers = getNumberList();
int sum = 0;

for (Object num : numbers) {
    sum += (Integer) num; // Unchecked cast
}
```

Covariant arrays

```
String[] strings = { "hello" };
Object[] objects = strings;

objects[0] = 1; // java.lang.ArrayStoreException
```

SAM types

```
interface Func1<T1, R> {
    R call(T1 t1);
}

interface Func2<T1, T2, R> {
    R call(T1 t1, T2 t2);
}

interface Func3<T1, T2, T3, R> {
    R call(T1 t1, T2 t2, T3 t3);
}

// ...
```

Wildcards

“I am completely and totally humbled. Laid low. I realize now that I am simply not smart at all. I made the mistake of thinking that I could understand generics. I simply cannot. I just can't. This is really depressing. It is the first time that I've ever not been able to understand something related to computers, in any domain, anywhere, period.”

"We simply cannot afford another wildcards”

— Joshua Bloch

Checked exceptions

“… requiring exception specifications could both enhance developer productivity and enhance code quality, but experience with large software projects suggests a different result – decreased productivity and little or no increase in code quality.”

— Bruce Eckel

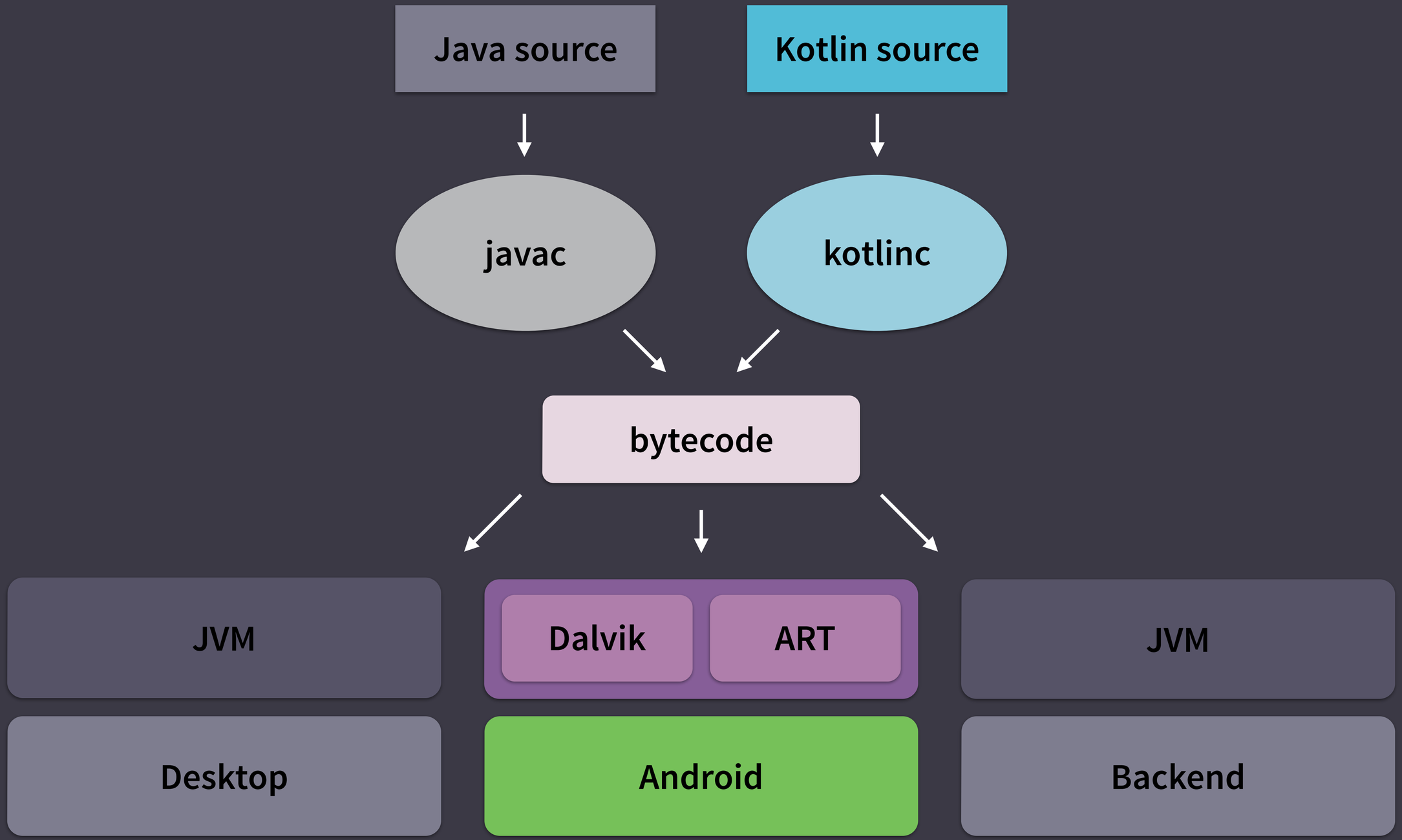
Kotlin to the rescue!

What Kotlin removes

- Checked exceptions
- Non-class primitive types
- Static members
- Non-private fields
- Wildcard types

What Kotlin adds

- Lambdas
- Data classes
- Function literals & inline functions
- Extension functions
- Null-safety
- Smart casts
- String templates
- Properties
- Primary constructors
- Class delegation
- Type inference
- Singletons
- Declaration-site variance
- Range expressions


Java source
Kotlin source
javac
kotlinc
bytecode
JVM
Dalvik
ART
JVM
Desktop
Android
Backend

Hello, Kotlin!

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

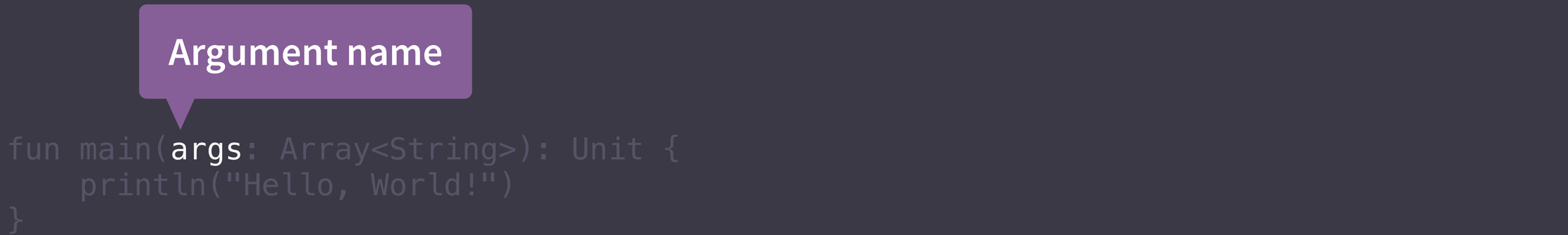
```
> Hello, World!
```

Function keyword

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

Function name

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


Argument name
fun main(args: Array<String>): Unit {
println("Hello, World!")
}

Argument type

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

Return type

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

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

`Unit` inferred

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

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

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

```
fun main(args: Array<String>) {
    var name = "World"
    println("Hello, $name!")
}
```

Variable declaration

```
fun main(args: Array<String>) {
    var name = "World"
    println("Hello, $name!")
}
```

String interpolation

```
fun main(args: Arr
    var name = "Worl
    println("Hello, $name!")
}
```

```
fun main(args: Array<String>) {
    var name = "World"
    if (args.isNotEmpty()) {
        name = args[0]
    }

    println("Hello, $name!")
}
```

```
fun main(args: Array<String>) {
    var name = "World"
    if (args.isNotEmpty()) {
        name = args[0]
    }

    println("Hello, $name!")
}
```

```
fun main(args: Array<String>) {
    val name = "World"
    if (args.isNotEmpty()) {
        name = args[0]
    }

    println("Hello, $name!")
}
```

Constant declaration

```
fun main(args: Array<String>) {
    val name = "World"
    if (args.isNotEmpty()) {
        name = args[0]
    }

    println("Hello, $name!")
}
```

```
fun ma
    va
    if (rgs.isNotEmpty()) {
        name = args[0]
    }

    println("Hello, $name!")
}
```

Val cannot be reassigned

```
fun main(args: Array<String>) {
    val name = "World"
    if (args.isNotEmpty()) {
        name = args[0]
    }

    println("Hello, $name!")
}
```

```
fun main(args: Array<String>) {
    val name = if (args.isNotEmpty()) {
        args[0]
    } else {
        "World"
    }

    println("Hello, $name!")
}
```

```
fun main(args: Array<String>) {
    val name = if (args.isNotEmpty()) {
        args[0]
    } else {
        "World"
    }

    println("Hello, $name!")
}
```

Conditional assignment block

```
fun main(args: Array<String>) {
    val name = if (args.isNotEmpty()) {
        args[0]
    } else {
        "World"
    }

    println("Hello, $name!")
}
```

```
fun main(args: Array<String>) {
    val name = if (args.isNotEmpty()) { args[0] } else { "World" }
    println("Hello, $name!")
}
```

```
fun main(args: Array<String>) {
    val name = if (args.isNotEmpty()) args[0] else "World"
    println("Hello, $name!")
}
```

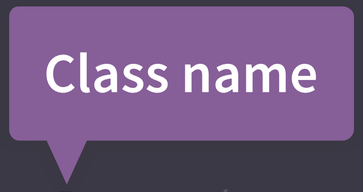
```
class Person(var name: String)

fun main(args: Array<String>) {
    val name = if (args.isNotEmpty()) args[0] else "World"
    println("Hello, $name!")
}
```

Class keyword

```
class Person(var name: String)

fun main(args: Array<String>) {
    val name = if (args.isNotEmpty()) args[0] else "World"
    println("Hello, $name!")
}
```




```
class Person(var name: String)

fun main(args: Array<String>) {
    val name = if (args.isNotEmpty()) args[0] else "World"
    println("Hello, $name!")
}
```

Primary constructor

```
class Person(var name: String)

fun main(args: Array<String>) {
    val name = if (args.isNotEmpty()) args[0] else "World"
    println("Hello, $name!")
}
```

Non-final class member

```
class Person(var name: String)

fun main(args: Array<String>) {
    val name = if (args.isNotEmpty()) args[0] else "World"
    println("Hello, $name!")
}
```

```
class Person(var name: String)

fun main(args: Array<String>) {
    val name = if (args.isNotEmpty()) args[0] else "World"
    println("Hello, $name!")
}
```

```
class Person(var name: String)

fun main(args: Array<String>) {

    println("Hello, $name!")
}
```

```
class Person(var name: String)

fun main(args: Array<String>) {
    val person = Person("Michael")
    println("Hello, $name!")
}
```

```
class Person(va

fun main(args: Array<String>) {
    val person = Person("Michael")
    println("Hello, $name!")
}
```

Instance declaration

```
class Person(var name: String)

fun main(args: Array<String>) {
    val person = Person("Michael")
    println("Hello, $name!")
}
```

```
class Person(var name: String)

fun main(args: Array<String>) {
    val person = Person("Michael")
    println("Hello, ${person.name}!")
}
```

```
> Hello, Michael!
```

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

class Person(var name: String)

fun main(args: Array<String>) {
    val person = Person("Michael")
    println("Hello, ${person.name}!")
}
```

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

class Person(var name: String)

fun main(args: Array<String>) {
    val person = Person("Michael")
    println("Hello, ${person.name}!")
}
```

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

class Person(var name: String, var lang: Language)

fun main(args: Array<String>) {
    val person = Person("Michael")
    println("Hello, ${person.name}!")
}
```

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

class Person(var name: String, var lang: Language = Language.EN)

fun main(args: Array<String>) {
    val person = Person("Michael")
    println("Hello, ${person.name}!")
}
```

Default value

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

class Person(var name: String, var lang: Language = Language.EN)

fun main(args: Array<String>) {
    val person = Person("Michael")
    println("Hello, ${person.name}!")
}
```

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

class Person(var name: String, var lang: Language = Language.EN) {
    fun greet() = println("${lang.greeting}, $name!")
}

fun main(args: Array<String>) {
    val person = Person("Michael")
    println("Hello, ${person.name}!")
}
```

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

class Person(var name: String, var lang: Language = Language.EN) {
    fun greet() = println("${lang.greeting}, $name!")
}

fun main(args: Array<String>) {
    val person = Person("Michael")
    println("Hello, ${person.name}!")
}
```

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

class Person(var name: String, var lang: Language = Language.EN) {
    fun greet() = println("${lang.greeting}, $name!")
}

fun main(args: Array<String>) {
    val person = Person("Michael")

}
```

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

class Person(var name: String, var lang: Language = Language.EN) {
    fun greet() = println("${lang.greeting}, $name!")
}

fun main(args: Array<String>) {
    val person = Person("Michael")
    person.greet()
}
```

```
> Hello, Michael!
```

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

class Person(var name: String, var lang: Language = Language.EN) {
    fun greet() = println("${lang.greeting}, $name!")
}

fun main(args: Array<String>) {


}
```

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

class Person(var name: String, var lang: Language = Language.EN) {
    fun greet() = println("${lang.greeting}, $name!")
}

fun main(args: Array<String>) {
    val people = listOf(
        Person("Michael"),
        Person("Miguel", Language.SP),
        Person("Michelle", Language.FR)
    )
}
```

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

class Person(var name: String, var lang: Language = Language.EN) {
    fun greet() = println("${lang.greeting}, $name!")
}

fun main(args: Array<String>) {
    val people = listOf(
        Person("Michael"),
        Person("Miguel", Language.SP),
        Person("Michelle", Language.FR)
    )

    for (person in people) {
        person.greet()
    }
}
```

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

class Person(var name: String, var lang: Language = Language.EN) {
    fun greet() = println("${lang.greeting}, $name!")
}

fun main(args: Array<String>) {
    val people = listOf(
        Person("Michael"),
        Person("Miguel", Language.SP),
        Person("Michelle", Language.FR)
    )

    people.forEach { person ->
        person.greet()
    }
}
```

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

class Person(var name: String, var lang: Language = Language.EN) {
    fun greet() = println("${lang.greeting}, $name!")
}

fun main(args: Array<String>) {
    val people = listOf(
        Person("Michael"),
        Person("Miguel", Language.SP),
        Person("Michelle", Language.FR)
    )

    people.forEach { it.greet() }
}
```

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

class Person(var name: String, var lang: Language = Language.EN) {
    fun greet() = println("${lang.greeting}, $name!")
}

fun main(args: Array<String>) {
    listOf(
        Person("Michael"),
        Person("Miguel", Language.SP),
        Person("Michelle", Language.FR)
    ).forEach { it.greet() }
}
```

```
> Hello, Michael!
> Hola, Miguel!
> Bonjour, Michelle!
```

Non-final

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

open class Person(var name: String, var lang: Language = Language.EN) {
    fun greet() = println("${lang.greeting}, $name!")
}

fun main(args: Array<String>) {
    listOf(
        Person("Michael"),
        Person("Miguel", Language.SP),
        Person("Michelle", Language.FR)
    ).forEach { it.greet() }.
}
```

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

open class Person(var name: String, var lang: Language = Language.EN) {
    fun greet() = println("${lang.greeting}, $name!")
}

class Hispanophone(name: String) : Person(name, Language.ES)
class Francophone(name: String) : Person(name, Language.FR)

fun main(args: Array<String>) {
    listOf(
        Person("Michael"),
        Person("Miguel", Language.SP),
        Person("Michelle", Language.FR)
    ).forEach { it.greet() }
}
```

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

open class Person(var name: String, var lang: Language = Language.EN) {
    fun greet() = println("${lang.greeting}, $name!")
}

class Hispanophone(name: String) : Person(name, Language.ES)
class Francophone(name: String) : Person(name, Language.FR)

fun main(args: Array<String>) {
    listOf(
        Person("Michael"),
        Hispanophone("Miguel"),
        Francophone("Michelle")
    ).forEach { it.greet() }
}
```

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

open class Person(var name: String, var lang: Language = Language.EN) {
    fun greet() = println("${lang.greeting}, $name!")
}

class Hispanophone(name: String) : Person(name, Language.ES)
class Francophone(name: String) : Person(name, Language.FR)

fun main(args: Array<String>) {
    listOf(
        Person("Michael"),
        Hispanophone("Miguel"),
        Francophone("Michelle")
    ).forEach { it.greet() }
}
```

What Kotlin Adds to Java

Type inference

Type inference

```
String string = "";
```

Type inference

```
var string: String = ""
```

Type inference

```
var string = ""
```

Type inference

```
var string = ""
var char = ' '
```

Type inference

```
var string = ""
var char = ' ‘

var int = 0
```

Type inference

```
var string = ""
var char = ' '

var int = 0
var long = 0L
```

Type inference

```
var string = ""
var char = ' '

var int = 0
var long = 0L
var float = 0F
```

Type inference

```
var string = ""
var char = ' '

var int = 0
var long = 0L
var float = 0F
var double = 0.0
```

Type inference

```
var string = ""
var char = ' '

var int = 0
var long = 0L
var float = 0F
var double = 0.0

var boolean = true
```

Type inference

```
var string = ""
var char = ' '

var int = 0
var long = 0L
var float = 0F
var double = 0.0

var boolean = true

var foo = MyFooType()
```

Null-safety

Null-safety

```
String a = null;
```

Null-safety

```
String a = null;
System.out.println(a.length());
```

Null-safety

```
String a = null;
System.out.println(a.length());
```

NullPointerException

Null-safety

```
val a: String = null
```

Null-safety

```
val a: String = null
```

Non-null type

Null-safety

```
val a: String? = null
```

Null-safety

```
val a: String? = null
```

Null-safety

```
val a: String? = null
println(a.length())
```

Null-safety

```
val a: String? = null
println(a.length())
```

Unsafe call

Null-safety

```
val a: String? = null
println(a?.length())
```

Null-safety

```
val a: String? = null
println(a?.length())
```

```
> null
```

Null-safety

```
int length = a != null ? a.length() : -1
```

Null-safety

```
int length = a != null ? a.length() : -1
```

Null check

Null-safety

```
int length = a != null ? a.length() : -1
```

Assignment selector

Null-safety

```
var length = if (a != null) a.length() else -1
```

Null-safety

```
var length = if (a != null) a.length() else -1
```

Null check

Null-safety

```
var length = if (a != null) a.length() else -1
```

Assignment selector

Null-safety

```
var length = a?.length() ?: -1
```

Null-safety

```
var length = a?.length() ?: -1
```

Null check

Null-safety

```
var length = a?.length() ?: -1
```

Assignment selector

Smart casts

Smart casts

```
if (x is String) {
    print(x.length())
}
```

Smart casts

```
if (x is String) {
    print(x.length())
}
```

Type check

Smart casts

```
if (x is String) {
    print(x.length())
}
```

Smart cast

Smart casts

```
if (x !is String) {
    return
}
print(x.size())
```

Smart casts

```
if (x !is String) {
    return
}
print
```

Type check

Smart casts

```
if (x !is String) {
    return
}
print(x.size())
```

Smart casts

```
if (x
}
print(x.size())
```

Smart cast

Smart casts

```
if (x !is String || x.size() == 0) {
    return
}
```

Smart casts

```
if (x !is String || x.size() == 0) {
    return
}
```

Type check

Smart casts

```
if (x !is String || x.size() == 0) {
    return
}
```

Smart casts

```
if (x !is String || x.size() == 0) {
    return
}
```

Smart cast

Smart casts

```
if (x is String && x.size() > 0) {
    print(x.size())
}
```

Smart casts

```
if (x is String && x.size() > 0) {
    print(x.size())
}
```

Type check

Smart casts

```
if (x is String && x.size() > 0) {
    print(x.size())
}
```

Smart casts

```
if (x is String && x.size() > 0) {
    print(x.size())
}
```

Smart cast

Smart casts

```
if (x is String && x.size() > 0) {
    print(x.size())
}
```

Smart cast

Smart casts

```
when (x) {
    is Int -> print(x + 1)
    is String -> print(x.size() + 1)
    is Array<Int> -> print(x.sum())
}
```

Smart casts

Type check

```
when (x) {
    is Int -> print(x + 1)
    is String -> print(x.size() + 1)
    is Array<Int> -> print(x.sum())
}
```

Smart casts

Smart cast

```
when (x) {
    is Int -> print(x + 1)
    is String -> print(x.size() + 1)
    is Array<Int> -> print(x.sum())
}
```

Smart casts

Type check

```
wh
    is Int -> print(x + 1)
    is String -> print(x.size() + 1)
    is Array<Int> -> print(x.sum())
}
```

Smart casts

Smart cast

```
when (x) {
    is Int -> print(x + 1)
    is String -> print(x.size() + 1)
    is Array<Int> -> print(x.sum())
}
```

Smart casts

Type check

```
wh
        nt(x + 1)
    is String -> print(x.size() + 1)
    is Array<Int> -> print(x.sum())
}
```

Smart casts

```
when (x) {
    is Int -> print(x + 1)
    is String -> print(x.size() + 1)
    is Array<Int> -> print(x.sum())
}
```

Smart cast

String templates

String templates

```
val apples = 4

println("I have " + apples + " apples.")
```

```
> I have 4 apples.
```

String templates

```
val apples = 4

println("I have $apples apples.")
```

```
> I have 4 apples.
> I have 4 apples.
```

String templates

```
val apples = 4
val bananas = 3

println("I have $apples apples and " + (apples + bananas) + " fruits.")
```

```
> I have 4 apples.
> I have 4 apples.
> I have 4 apples and 7 fruits.
```

String templates

```
val apples = 4
val bananas = 3

println("I have $apples apples and ${apples+bananas} fruits.")
```

```
> I have 4 apples.
> I have 4 apples and 7 fruits.
> I have 4 apples and 7 fruits.
```

Range expressions

Range expressions

```
if (1 <= i && i <= 10) {
    println(i)
}
```

Range expressions

```
if (1 <= i && i <= 10) {
    println(i)
}

if (IntRange(1, 10).contains(i)) {
    println(i)
}
```

Range expressions

```
if (1 <= i && i <= 10) {
    println(i)
}

if (1.rangeTo(10).contains(i)) {
    println(i)
}
```

Range expressions

```
if (1 <= i && i <= 10) {
    println(i)
}

if (i in 1..10) {
    println(i)
}
```

Range operator

Range expressions

```
for (i in 1..4) {
    print(i)
}
```

```
> 1234
```

Range expressions

```
for (i in 1..4 step 2) {
    print(i)
}
```

```
> 1234
> 13
```

Range expressions

```
for (i in 4 downTo 1 step 2) {
    print(i)
}
```

```
> 1234
> 13
> 42
```

Range expressions

```
for (i in 1.0..2.0) {
    print("$i ")
}
```

```
> 13
> 42
> 1.0 2.0
```

Range expressions

```
for (i in 1.0..2.0 step 0.3) {
    print("$i ")
}
```

```
> 42
> 1.0 2.0
> 1.0 1.3 1.6 1.9
```

Higher-order functions & lambdas

Higher-order functions & lambdas

```
public interface Function<T, R> {
    R call(T t);
}
```

Higher-order functions & lambdas

```
public interface Function<T, R> {
    R call(T t);
}

public static <T> List<T> filter(Collection<T> items, Function<T, Boolean> f) {
    final List<T> filtered = new ArrayList<T>();
    for (T item : items) if (f.call(item)) filtered.add(item);
    return filtered;
}
```

Higher-order functions & lambdas

```
public interface Function<T, R> {
    R call(T t);
}

public static <T> List<T> filter(Collection<T> items, Function<T, Boolean> f) {
    final List<T> filtered = new ArrayList<T>();
    for (T item : items) if (f.call(item)) filtered.add(item);
    return filtered;
}

filter(numbers, new Function<Integer, Boolean>() {
    @Override
    public Boolean call(Integer value) {
        return value % 2 == 0;
    }
});
```

Higher-order functions & lambdas

```
public interface Function<T, R> {
    R call(T t);
}

public static <T> List<T> filter(Collection<T> items, Function<T, Boolean> f) {
    final List<T> filtered = new ArrayList<T>();
    for (T item : items) if (f.call(item)) filtered.add(item);
    return filtered;
}

filter(numbers, new Function<Integer, Boolean>() {
    @Override
    public Boolean call(Integer value) {
        return value % 2 == 0;
    }
});
```

Higher-order functions & lambdas

Functional interface

```
public interface Function<T, R> {
    R call(T t);
}

public static <T> List<T> filter(Collection<T> items, Function<T, Boolean> f) {
    final List<T> filtered = new ArrayList<T>();
    for (T item : items) if (f.call(item)) filtered.add(item);
    return filtered;
}

filter(numbers, new Function<Integer, Boolean>() {
    @Override
    public Boolean call(Integer value) {
        return value % 2 == 0;
    }
});
```

Higher-order functions & lambdas

```
public interface Function<T, R> {
    R call(T t);
}

public static <T> List<T> filter(Collection<T> items, Function<T, Boolean> f) {
    final List<T> filtered = new ArrayList<T>();
    for (T item : items) if (f.call(item)) filtered.add(item);
    return filtered;
}

filter(numbers, new Function<Integer, Boolean>() {
    @Override
    public Boolean call(Integer value) {
        return value % 2 == 0;
    }
});
```

Interface argument

Higher-order functions & lambdas

```
public interface Function<T, R> {
    R call(T t);
}

public static <T> List<T> filt                    unction<T, Boolean> f) {
    final List<T> filtered = new ArrayList<T>();
    for (T item : items) if (f.call(item)) filtered.add(item);
    return filtered;
}

filter(numbers, new Function<Integer, Boolean>() {
    @Override
    public Boolean call(Integer value) {
        return value % 2 == 0;
    }
});
```

Interface function call

Higher-order functions & lambdas

```
public interface Function<T, R> {
    R call(T t);
}

public static <T> List<T> filter(Collection<T> items, Function<T, Boolean> f) {
    final List<T> filtered = new ArrayList<T>();
    for (T item : items) if (f.call(item)) filtered.add(item);
    return fil
}

filter(numbers, new Function<Integer, Boolean>() {
    @Override
    public Boolean call(Integer value) {
        return value % 2 == 0;
    }
});
```

Anonymous implementation

Higher-order functions & lambdas

```
fun <T> filter(items: Collection<T>, f: (T) -> Boolean): List<T> {
    val filtered = arrayListOf<T>()
    for (item in items) if (f(item)) filtered.add(item)
    return filtered
}
```

Higher-order functions & lambdas

```
fun <T> filter(items: Collection<T>, f: (T) -> Boolean): List<T> {
    val filtered = arrayListOf<T>()
    for (item in items) if (f(item)) filtered.add(item)
    return filtered
}

filter(numbers, { value ->
    value % 2 == 0
})
```

Higher-order functions & lambdas

```
fun <T> filter(items: Collection<T>, f: (T) -> Boolean): List<T> {
    val filtered = arrayListOf<T>()
    for (item in items) if (f(item)) filtered.add(item)
    return filtered
}

filter(numbers, { value ->
    value % 2 == 0
})
```

Higher-order functions & lambdas

```
fun <T> filter(items: Collection<T>, f: (T) -> Boolean): List<T> {
    val filtered = arrayListOf<T>()
    for (item in items) if (f(item)) fi
    return filtered
}

filter(numbers, { value ->
    value % 2 == 0
})
```

Function type

Higher-order functions & lambdas

```
fun <T> filter(items: Collection<T>, f: (T) -> Boolean): List<T> {
    val filtered = arrayListOf<T>()
    for (item in items) if (f(item)) filtered.add(item)
    return filtered
}

filter(numbers, { value ->
    value % 2 == 0
})
```

Function call

Higher-order functions & lambdas

```
fun <T> filter(items: Collection<T>, f: (T) -> Boolean): List<T> {
    val filtered = arrayListOf<T>()
    for (item
    return fil
}

filter(numbers, { value ->
    value % 2 == 0
})
```

Anonymous function

Higher-order functions & lambdas

```
fun <T> filter(items: Collection<T>, f: (T) -> Boolean): List<T> {
    val filtered = arrayListOf<T>()
    for (item in items) if (f(item)) filtered.add(item)
    return filtered
}

filter(numbers, { value ->
    value % 2 == 0
})
```

Higher-order functions & lambdas

```
fun <T> filter(items: Collection<T>, f: (T) -> Boolean): List<T> {
    val filtered = arrayListOf<T>()
    for (item in items) if (f(item)) filtered.add(item)
    return filtered
}

filter(numbers) { value ->
    value % 2 == 0
}
```

Higher-order functions & lambdas

```
fun <T> filter(items: Collection<T>, f: (T) -> Boolean): List<T> {
    val filtered = arrayListOf<T>()
    for (item in items) if (f(item)) filtered.add(item)
    return filtered
}

filter(numbers) {
    it % 2 == 0
}
```

Higher-order functions & lambdas

```
fun <T> filter(items: Collection<T>, f: (T) -> Boolean): List<T> {
    val filtered = arrayListOf<T>()
    for (item in items) if (f(item)) filtered.add(item)
    return filtered
}

filter(numbers) { it % 2 == 0 }
```

Extension functions

Extension functions

```
public fun isLollipopOrGreater(code: Int): Boolean {
    return code >= Build.VERSION_CODES.LOLLIPOP
}
```

Extension functions

```
public fun isLollipopOrGreater(code: Int): Boolean {
    return code >= Build.VERSION_CODES.LOLLIPOP
}
```

Extension functions

```
public fun Int.isLollipopOrGreater(code: Int): Boolean {
    return code >= Build.VERSION_CODES.LOLLIPOP
}
```

Extension functions

```
public fun Int.isLollipopOrGreater(): Boolean {
    return code >= Build.VERSION_CODES.LOLLIPOP
}
```

Extension functions

```
public fun Int.isLollipopOrGreater(): Boolean {
    return this >= Build.VERSION_CODES.LOLLIPOP
}
```

Extension functions

```
public fun Int.isLollipopOrGreater(): Boolean {.
    return this >= Build.VERSION_CODES.LOLLIPOP
}

if (Build.VERSION.SDK_INT.isLollipopOrGreater) {
    // ...
}
```

Extension functions

```
public fun Int.isLollipopOrGreater(): Boolean {.
    return this >= Build.VERSION_CODES.LOLLIPOP
}

if (16.isLollipopOrGreater) {
    // ...
}
```

Extension functions

```
final Function<Customer, Order> customerMapper = // ...
final Function<Order, Boolean> orderFilter = // ...
final Function<Order, Float> orderSorter = // ...

final List<Order> vipOrders = sortBy(filter(map(customers,
        customerMapper),
        orderFilter),
        orderSorter);
```

Extension functions

```
final Function<Customer, Order> customerMapper = // ...
final Function<Order, Boolean> orderFilter = // ...
final Function<Order, Float> orderSorter = // ...

final List<Order> vipOrders = sortBy(filter(map(customers,
        customerMapper),
        orderFilter),
        orderSorter);
```

Extension functions

```
final Function<Customer, Order> customerMapper = // ...
final Function<Order, Boolean> orderFilter = // ...
final Function<Order, Float> orderSorter = // ...

final List<Order> vipOrders = sortBy(filter(map(customers,
        customerMapper),
        orderFilter),
        orderSorter);
```

Extension functions

```
final Function<Customer, Order> customerMapper = // ...
final Function<Order, Boolean> orderFilter = // ...
final Function<Order, Float> orderSorter = // ...

final List<Order> vipOrders = sortBy(filter(map(customers,
        customerMapper),
        orderFilter),
        orderSorter);
```

Extension functions

```
final Function<Customer, Order> customerMapper = // ...
final Function<Order, Boolean> orderFilter = // ...
final Function<Order, Float> orderSorter = // ...

final List<Order> vipOrders = sortBy(filter(map(customers,
        customerMapper),
        orderFilter),
        orderSorter);
```

Extension functions

```
val vipOrders  = customers
        .map { it.lastOrder }
        .filter { it.total >= 500F }
        .sortBy { it.total }
```

Extension functions

```
val vipOrders  = customers
        .map { it.lastOrder }
        .filter { it.total >= 500F }
        .sortBy { it.total }
```

Extension functions

```
val vipOrders  = customers
        .map { it.lastOrder }
        .filter { it.total >= 500F }
        .sortBy { it.total }
```

Extension functions

```
val vipOrders  = customers
        .map { it.lastOrder }
        .filter { it.total >= 500F }
        .sortBy { it.total }
```

Extension functions

```
val vipOrders  = customers
        .map { it.lastOrder }
        .filter { it.total >= 500F }
        .sortBy { it.total }
```

Extension functions

```
val vipOrders  = customers
        .map { it.lastOrder }
        .filter { it.total >= 500F }
        .sortBy { it.total }
```

Properties

Properties

```
class Customer {
    private String firstName;
    private String lastName;
    private String email;

    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
    public String getEmail() { return email; }

    public void setFirstName(String firstName) { this.firstName = firstName }
    public void setLastName(String lastName) { this.lastName = lastName }
    public void setEmail(String email) { this.email = email }
}
```

Properties

```
class Customer {
    private String firstName;
    private String lastName;
    private String email;

    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
    public String getEmail() { return email; }

    public void setFirstName(String firstName) { this.firstName = firstName }
    public void setLastName(String lastName) { this.lastName = lastName }
    public void setEmail(String email) { this.email = email }
}
```

Properties

```
class Customer {
    private String firstName;
    private String lastName;
    private String email;

    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
    public String getEmail() { return email; }

    public void setFirstName(String firstName) { this.firstName = firstName }
    public void setLastName(String lastName) { this.lastName = lastName }
    public void setEmail(String email) { this.email = email }
}
```

Properties

```
class Customer {
    private String firstName;
    private String lastName;
    private String email;

    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
    public String getEmail() { return email; }

    public void setFirstName(String firstName) { this.firstName = firstName }
    public void setLastName(String lastName) { this.lastName = lastName }
    public void setEmail(String email) { this.email = email }
}
```

Properties

```
class Customer {
    var firstName: String = // ...
    var lastName: String = // ...
    var email: String = // ...
}
```

Properties

```
class Customer {
    var firstName: String = // ...
    var lastName: String = // ...
    var email: String = // ...
}

val customer = Customer()
customer.firstName = "Michael"
customer.lastName = "Pardo"
customer.email = "michael@michaelpardo.com"
```

Primary constructors

Primary constructors

```
class Customer {
    var firstName: String = // ...
    var lastName: String = // ...
    var email: String = // ...
}
```

Primary constructors

```
class Customer(firstName: String, lastName: String, email: String) {
    var firstName: String = // ...
    var lastName: String = // ...
    var email: String = // ...
}
```

Primary constructors

```
class Customer(firstName: String, lastName: String, email: String) {
    var firstName: String = firstName
    var lastName: String = lastName
    var email: String = email
}
```

Primary constructors

```
class Customer(firstName: String, lastName: String, email: String) {
    var firstName: String
    var lastName: String
    var email: String

    init {
        this.firstName = firstName
        this.lastName = lastName
        this.email = email
    }
}
```

Primary constructors

```
class Customer(firstName: String, lastName: String, email: String)
```

Primary constructors

```
class Customer(var firstName: String, var lastName: String, var email: String)
```

Primary constructors

```
class Customer(
    var firstName: String,
    var lastName: String,
    var email: String)
```

Primary constructors

```
class Customer(
    val firstName: String,
    val lastName: String,
    val email: String)
```

Singletons

Singletons

```
public class Singleton {
    private static volatile Singleton instance;
    private Singleton() { }

    public static Singleton getInstance() {
        if (instance == null ) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

Singletons

```
public class Singleton {
    private static volatile Singleton instance = null;
    private Singleton() { }

    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Singletons

```
public class Singleton {
    private static final Singleton INSTANCE = new Singleton();
    private Singleton() {}

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

Singletons

```
public class Singleton {
    private static final Singleton instance;

    static {
        try {
            instance = new Singleton();
        } catch (Exception e) {
            throw new RuntimeException("Darn, an error occurred!", e);
        }
    }

    public static Singleton getInstance() {
        return instance;
    }

    private Singleton() { }
}
```

Singletons

```
public class Singleton {
    private Singleton() { }

    private static class SingletonHolder {
        private static final Singleton INSTANCE = new Singleton();
    }

    public static Singleton getInstance() {
        return SingletonHolder.INSTANCE;
    }
}
```

Singletons

```
public enum Singleton {
    INSTANCE;

    public void execute (String arg) {
    }
}
```

Singletons

```
object Singleton
```

Singletons

```
object Logger {
    val tag = "TAG"

    fun d(message: String) {
        Log.d(tag, message)
    }
}
```

Companion objects

Companion objects

```
class LaunchActivity extends AppCompatActivity {
    public static final String TAG = LaunchActivity.class.getName();

    public static void start(Context context) {
        context.startActivity(new Intent(context, LaunchActivity.class));
    }
}
```

Companion objects

```
class LaunchActivity extends AppCompatActivity {
    public static final String TAG = LaunchActivity.class.getName();

    public static void start(Context context) {
        context.startActivity(new Intent(context, LaunchActivity.class));
    }
}
```

Companion objects

```
class LaunchActivity extends AppCompatActivity {
    public static final String TAG = LaunchActivity.class.getName();

    public static void start(Context context) {
        context.startActivity(new Intent(context, LaunchActivity.class));
    }
}
```

Companion objects

```
class LaunchActivity extends AppCompatActivity {
    public static final String TAG = LaunchActivity.class.getName();

    public static void start(Context context) {
        context.startActivity(new Intent(context, LaunchActivity.class));
    }
}

Timber.v("Starting activity %s", LaunchActivity.TAG);
```

Companion objects

```
class LaunchActivity extends AppCompatActivity {
    public static final String TAG = LaunchActivity.class.getName();

    public static void start(Context context) {
        context.startActivity(new Intent(context, LaunchActivity.class));
    }
}

Timber.v("Starting activity %s", LaunchActivity.TAG);

LaunchActivity.start(context);
```

Companion objects

```
class LaunchActivity {
    companion object {
        val TAG: String = LaunchActivity::class.simpleName

        fun start(context: Context) {
            context.startActivity(Intent(context, LaunchActivity::class))
        }
    }
}
```

Companion objects

```
class LaunchActivity {
    companion object {
        val TAG: String = LaunchActivity::class.simpleName

        fun start(context: Context) {
            context.startActivity(Intent(context, LaunchActivity::class))
        }
    }
}
```

Companion objects

```
class LaunchActivity {
    companion object {
        val TAG: String = LaunchActivity::class.simpleName

        fun start(context: Context) {
            context.startActivity(Intent(context, LaunchActivity::class))
        }
    }
}
```

Companion objects

```
class LaunchActivity {
    companion object {
        val TAG: String = LaunchActivity::class.simpleName

        fun start(context: Context) {
            context.startActivity(Intent(context, LaunchActivity::class))
        }
    }
}
```

Companion objects

```
class LaunchActivity {
    companion object {
        val TAG: String = LaunchActivity::class.simpleName

        fun start(context: Context) {
            context.startActivity(Intent(context, LaunchActivity::class))
        }
    }
}

Timber.v("Starting activity ${LaunchActivity.TAG}")
```

Companion objects

```
class LaunchActivity {
    companion object {
        val TAG: String = LaunchActivity::class.simpleName

        fun start(context: Context) {
            context.startActivity(Intent(context, LaunchActivity::class))
        }
    }
}

Timber.v("Starting activity ${LaunchActivity.TAG}")

LaunchActivity.start(context)
```

Class delegation

Class delegation

```
public class MyList<E> implements List<E> {
    private List<E> delegate;

    public MyList(delegate: List<E>) {
        this.delegate = delegate;
    }

    // ...

    public E get(int location) {
        return delegate.get(location)
    }

    // ...
}
```

Class delegation

```
public class MyList<E> implements List<E> {
    private List<E> delegate;

    public MyList(delegate: List<E>) {
        this.delegate = delegate;
    }

    // ...

    public E get(int location) {
        return delegate.get(location)
    }

    // ...
}
```

Class delegation

```
public class MyList<E> implements List<E> {
    private List<E> delegate;

    public MyList(delegate: List<E>) {
        this.delegate = delegate;
    }

    // ...

    public E get(int location) {
        return delegate.get(location)
    }

    // ...
}
```

Class delegation

```
public class MyList<E> implements List<E> {
    private List<E> delegate;

    public MyList(delegate: List<E>) {
        this.delegate = delegate;
    }

    // ...

    public E get(int location) {
        return delegate.get(location)
    }

    // ...
}
```

Class delegation

```
public class MyList<E> implements List<E> {
    private List<E> delegate;

    public MyList(delegate: List<E>) {
        this.delegate = delegate;
    }

    // ...

    public E get(int location) {
        return delegate.get(location)
    }

    // ...
}
```

Class delegation

```
class MyList<E>(list: List<E>) : List<E> by list
```

Class delegation

```
class MyList<E>(list: List<E>) : List<E> by list
```

Class delegation

```
class MyList<E>(list: List<E>) : List<E> by list
```

Class delegation

```
class MyList<E>(list: List<E>) : List<E> by list
```

Declaration-site variance

Declaration-site variance

```
String[] strings = { "hello", "world" };
Object[] objects = strings;
```

Declaration-site variance

```
List<String> strings = Arrays.asList("hello", "world");
List<Object> objects = strings;
```

Declaration-site variance

```
List<String> strings = Arrays.asList("hello", "world");
List<Object> objects = strings;
```

Error: incompatible types

Declaration-site variance

```
List<String> strings = Arrays.asList("hello", "world");
List<Object> objects = strings;
```

Declaration-site variance

```
List<String> strings = Arrays.asList("hello", "world");
List<? extends Object> objects = strings;
```

Declaration-site variance

```
public interface List<E> extends Collection<E> {
    public boolean addAll(Collection<? extends E> collection);

    public E get(int location);
}
```

Declaration-site variance

```
public interface List<out E> : Collection<E> {
    public fun get(index: Int): E
}

public interface MutableList<E> : List<E>, MutableCollection<E> {
    override fun addAll(c: Collection<E>): Boolean
}
```

Declaration-site variance

```
public interface List<out E> : Collection<E> {
    public fun get(index: Int): E
}

public interface MutableList<E> : List<E>, MutableCollection<E> {
    override fun addAll(c: Collection<E>): Boolean
}
```

Declaration-site variance

```
public interface List<out E> : Collection<E> {
    public fun get(index: Int): E
}

public interface MutableList<E> : List<E>, MutableCollection<E> {
    override fun addAll(c: Collection<E>): Boolean
}
```

Declaration-site variance

```
val strings: List<String> = listOf("hello", "world")
val objects: List<Any> = strings
```

Declaration-site variance

```
val strings: List<String> = arrayListOf("hello", "world")
val objects: List<Any> = strings
```

Declaration-site variance

```
val strings: MutableList<String> = arrayListOf("hello", "world")
val objects: List<Any> = strings
```

Declaration-site variance

```
val strings: MutableList<String> = arrayListOf("hello", "world")
val objects: MutableList<Any> = strings
```

Declaration-site variance

```
val strings: MutableList<String> = arrayListOf("hello", "world")
val objects: MutableList<Any> = strings
```

Type mismatch

Operator overloading

Operator overloading

```
enum class Coin(val cents: Int) {
    PENNY(1),
    NICKEL(5),
    DIME(10),
    QUARTER(25),
}
```

Operator overloading

```
enum class Coin(val cents: Int) {
    PENNY(1),
    NICKEL(5),
    DIME(10),
    QUARTER(25),
}

class Purse(var amount: Float)
```

Operator overloading

```
enum class Coin(val cents: Int) {
    PENNY(1),
    NICKEL(5),
    DIME(10),
    QUARTER(25)
}

class Purse(var amount: Float) {
    fun plusAssign(coin: Coin): Unit {
        amount += (coin.cents / 100f)
    }
}
```

Reserved function name

Operator overloading

```
enum class Coin(val cents: Int) {
    PENNY(1),
    NICKEL(5),
    DIME(10),
    QUARTER(25),
}

class Purse(var amount: Float) {
    fun plusAssign(coin: Coin): Unit {
        amount += (coin.cents / 100f)
    }
}

var purse = Purse(1.50f)
```

Operator overloading

```
enum class Coin(val cents: Int) {
    PENNY(1),
    NICKEL(5),
    DIME(10),
    QUARTER(25),
}

class Purse(var amount: Float) {
    fun plusAssign(coin: Coin): Unit {
        amount += (coin.cents / 100f)
    }
}

var purse = Purse(1.50f)
purse += Coin.QUARTER   // 1.75
```

Operator overloading

```
enum class Coin(val cents: Int) {
    PENNY(1),
    NICKEL(5),
    DIME(10),
    QUARTER(25),
}

class Purse(var amount: Float) {
    fun plusAssign(coin: Coin): Unit {
        amount += (coin.cents / 100f)
    }
}

var purse = Purse(1.50f)
purse += Coin.QUARTER   // 1.75
purse += Coin.DIME      // 1.85
```

Operator overloading

```
enum class Coin(val cents: Int) {
    PENNY(1),
    NICKEL(5),
    DIME(10),
    QUARTER(25),
}

class Purse(var amount: Float) {
    fun plusAssign(coin: Coin): Unit {
        amount += (coin.cents / 100f)
    }
}

var purse = Purse(1.50f)
purse += Coin.QUARTER   // 1.75
purse += Coin.DIME      // 1.85
purse += Coin.PENNY     // 1.86
```

Operator overloading

```
a + b
a - b
a * b
a / b
a % b
```

```
a += b
a -= b
a *= b
a /= b
a %= b
```

```
a++
a--

a[]
a()
```

```
a..b
a in b
a !in b
```

```
a == b
a != b
```

```
a > b
a < b
a >= b
a <= b
```