Introduction to Kotlin

Feature comparison between J & K

@iChanSek

www.chansek.com

SPAM !!!

About Me

About Me

- Working as an Android Engineer since last 6 years

About Me

- Working as an Android Engineer since last 6 years
- Created few apps like Chanse Games, Chanse Shops

About Me

- Working as an Android Engineer since last 6 years
- Created few apps like Chanse Games, Chanse Shops
- A failure Entrepreneur

About Me

- Working as an Android Engineer since last 6 years
- Created few apps like Chanse Games, Chanse Shops
- A failure Entrepreneur
- Author of an about to be released Kotlin book

About Me

- Working as an Android Engineer since last 6 years
- Created few apps like Chanse Games, Chanse Shops
- A failure Entrepreneur
- Author of an about to be released Kotlin book
- Only know upto Java-7

About Me

- Working as an Android Engineer since last 6 years
- Created few apps like Chanse Games, Chanse Shops
- A failure Entrepreneur
- Author of an about to be released Kotlin book
- Only know upto Java-7
- Worked with Kotlin since last 1 year

About Me

- Working as an Android Engineer since last 6 years
- Created few apps like Chanse Games, Chanse Shops
- A failure Entrepreneur
- Author of an about to be released Kotlin book
- Only know upto Java-7
- Worked with Kotlin since last 1 year
- Organizing Bengaluru’s Kotlin Group - BlrKotlin

About Me

- Working as an Android Engineer since last 6 years
- Created few apps like Chanse Games, Chanse Shops
- A failure Entrepreneur
- Author of an about to be released Kotlin book
- Only know upto Java-7
- Worked with Kotlin since last 1 year
- Organizing Bengaluru’s Kotlin Group - BlrKotlin
- meetup.com/BlrKotlin & blrkotlin.herokuapp.com

Kotlin - The selling point

Highlights of Kotlin

Highlights of Kotlin

- Statically typed language like Java

Highlights of Kotlin

- Statically typed language like Java
- Billion Dollar mistake is no more as **Null** is in type system and treated specially

Highlights of Kotlin

- Statically typed language like Java
- Billion Dollar mistake is no more as **Null** is in type system and treated specially
- 100% interoperable with Java

Highlights of Kotlin

- Statically typed language like Java
- Billion Dollar mistake is no more as **Null** is in type system and treated specially
- 100% interoperable with Java
- Believes in “Sharing is Caring” by supporting multi-platform

Highlights of Kotlin

- Statically typed language like Java
- Billion Dollar mistake is no more as **Null** is in type system and treated specially
- 100% interoperable with Java
- Believes in “Sharing is Caring” by supporting multi-platform
- Much more concise than Java

Why should you try Kotlin?

Why should you try Kotlin?

- No need to refactor the whole project.

Why should you try Kotlin?

- No need to refactor the whole project.
- Write Less - Maintain Less - Spend Less

Why should you try Kotlin?

- No need to refactor the whole project.
- Write Less - Maintain Less - Spend Less
- No NPE - Millions of Profit - More Team Budget *

Why should you try Kotlin?

- No need to refactor the whole project.
- Write Less - Maintain Less - Spend Less
- No NPE - Millions of Profit - More Team Budget *
- Lot of awesome features to write neat code.

Java vs Kotlin

final vs val

final vs val

```
class Test {
    private String tag = "Test";

    public void test() {
        Log.d(tag, "Testing...");
    }
}
```

final vs val

```java
class Test {
    private String tag = "Test";

    public void test() {
        Log.d(tag, "Testing...");
    }
}
```

```java
class Test {
    private var tag = "Test";

    fun test() {
        Log.d(tag, "Testing...");
    }
}
```

final vs val

```java
class Test {
    private String tag = "Test";

    public void test() {
        Log.d(tag, "Testing...");
    }
}
```

```java
class Test {
    private var tag = "Test";

    fun test() {
        Log.d(tag, "Testing...");
    }
}
```

- **final** is ignored in Java

final vs val

```java
class Test {
    private String tag = "Test";

    public void test() {
        Log.d(tag, "Testing...");
    }
}
```

```kotlin
class Test {
    private var tag = "Test";

    fun test() {
        Log.d(tag, "Testing...");
    }
}
```

- **final** is ignored in Java
- **var** is an evil keyword in Kotlin

final vs val

```java
class Test {
    private String tag = "Test";

    public void test() {
        Log.d(tag, "Testing...");
    }
}
```

```kotlin
class Test {
    private var tag = "Test";

    fun test() {
        Log.d(tag, "Testing...");
    }
}
```

- **final** is ignored in Java
- **var** is an evil keyword in Kotlin
- Using **var** and not updating the value? You will be discouraged until you use **val**.

final vs val

```java
class Test {
    private String tag = "Test";

    public void test() {
        Log.d(tag, "Testing...");
    }
}
```

```kotlin
class Test {
    private val tag = "Test";

    fun test() {
        Log.d(tag, "Testing...");
    }
}
```

- **final** is ignored in Java
- **var** is an evil keyword in Kotlin
- Using **var** and not updating the value? You will be discouraged until you use **val**.

final vs open

final vs open

```
class Parent { }

class Child extends Parent { }
```

final vs open

```
class Parent { }

class Child extends Parent { }
```

```
class Parent

class Child extends Parent()
```

final vs open

```
class Parent { }

class Child extends Parent { }
```

```
class Parent

class Child extends Parent()
```

- All classes are final by default.

final vs open

```
class Parent { }

class Child extends Parent { }
```

```
class Parent

class Child extends Parent()
```

- All classes are final by default.
- If you want to inherit, you have to plan and design the class accordingly.

final vs open

```
class Parent { }

class Child extends Parent { }
```

```
open class Parent

class Child extends Parent()
```

- All classes are final by default.
- If you want to inherit, you have to plan and design the class accordingly.
- **open** keyword does the thing for you.

private means private

private means private

```java
public class JavaEncapsulation {

    private class Animal { }

    private Animal animal;

    public Animal getAnimal() {
        return animal;
    }

    public void setAnima(Animal anima) {
        this.animal = anima;
    }
}
```

private means private

```java
public class JavaEncapsulation {

    private class Animal { }

    private Animal animal;

    public Animal getAnimal() {
        return animal;
    }

    public void setAnima(Animal anima) {
        this.animal = anima;
    }
}
```

```kotlin
class KtEncapsulation {

    private inner class Animal { }

    private var animal = Animal()

    fun getAnimal() = animal

    fun setAnimal(newAnimal: Animal) {
        animal = newAnimal
    }
}
```

private means private

```java
public class JavaEncapsulation {

    private class Animal { }

    private Animal animal;

    public Animal getAnimal() {
        return animal;
    }

    public void setAnima(Animal anima) {
        this.animal = anima;
    }
}
```

```kotlin
class KtEncapsulation {

    private inner class Animal { }

    private var animal = Animal()

    fun getAnimal() = animal

    fun setAnimal(newAnimal: Animal) {
        animal = newAnimal
    }
}
```

Animal type is private to this class

private means private

```java
public class JavaEncapsulation {

    private class Animal { }

    private Animal animal;

    public Animal getAnimal() {
        return animal;
    }

    public void setAnima(Animal anima) {
        this.animal = anima;
    }
}
```

```kotlin
class KtEncapsulation {

    private inner class Animal { }

    private var animal = Animal()

    fun getAnimal() = animal

    fun setAnimal(newAnimal: Animal) {
        animal = newAnimal
    }
}
```

getAnimal() is public function

private means private

```java
public class JavaEncapsulation {

    private class Animal { }

    private Animal animal;

    public Animal getAnimal() {
        return animal;
    }

    public void setAnima(Animal anima) {
        this.animal = anima;
    }
}
```

```kotlin
class KtEncapsulation {

    private inner class Animal { }

    private var animal = Animal()

    fun getAnimal() = animal

    fun setAnimal(newAnimal: Animal) {
        animal = newAnimal
    }
}
```

Can not expose a private type Animal from public functions getAnimal() and setAnimal()

private means private

```java
public class JavaEncapsulation {

    private class Animal { }

    private Animal animal;

    public Animal getAnimal() {
        return animal;
    }

    public void setAnima(Animal anima) {
        this.animal = anima;
    }
}
```

```kotlin
class KtEncapsulation {

    inner class Animal { }

    private var animal = Animal()

    fun getAnimal() = animal

    fun setAnimal(newAnimal: Animal) {
        animal = newAnimal
    }
}
```

Compiles fine as no violation of Encapsulation

concat vs template

concat vs template

```
public String toString() {
    return "User{" +
        "name='" + name + '\'' +
        ", email='" + email + '\'' +
        ", phone='" + phone + '\'' +
    '}';
}
```

concat vs template

```java
public String toString() {
    return "User{" +
        "name='" + name + '\'' +
        ", email='" + email + '\'' +
        ", phone='" + phone + '\'' +
    '}';
}
```

```fun
fun toString(): String {
    return "User(name='$name',
email='$email', phone='$phone')"
}
```

concat vs template

```java
public String toString() {
    return "User{" +
        "name='" + name + '\'' +
        ", email='" + email + '\'' +
        ", phone='" + phone + '\'' +
    '}';
}
```

```fun
fun toString(): String {
    return "User(name='$name',
email='$email', phone='$phone')"
}
```

- String Templating allows to write any expression within the String.

concat vs template

```java
public String toString() {
    return "User{" +
        "name='" + name + '\'' +
        ", email='" + email + '\'' +
        ", phone='" + phone + '\'' +
    '}';
}
```

```fun
fun toString(): String {
    return "User(name='$name',
email='$email', phone='$phone')"
}
```

- String Templating allows to write any expression within the String.
- A simple variable can be referred using **$** symbol as prefix.

concat vs template

```java
public String toString() {
    return "User{" +
        "name='" + name + '\'' +
        ", email='" + email + '\'' +
        ", phone='" + phone + '\'' +
    '}';
}
```

```fun
fun toString(): String {
    return "User(name='$name',
email='$email', phone='$phone')"
}
```

- String Templating allows to write any expression within the String.
- A simple variable can be referred using **$** symbol as prefix.
- A complex operation can also be done using **${operation}** syntax. For example - **${email.toUpperCase()}**

overloading vs default parameters

overloading vs default parameters

```
int add(int a, int b) {
    return a + b;
}

int add(int a, int b, int c) {
    return a + b + c;
}

public void testAdd() {
    add(10, 20);
    add(10, 20, 30);
}
```

overloading vs default parameters

```java
int add(int a, int b) {
    return a + b;
}

int add(int a, int b, int c) {
    return a + b + c;
}

public void testAdd() {
    add(10, 20);
    add(10, 20, 30);
}
```

```fun
fun add(a: Int, b: Int, c: Int = 0) =
a + b + c

fun testAdd() {
    add(10, 20)
    add(10, 20, 30)
}
```

overloading vs default parameters

```java
int add(int a, int b) {
    return a + b;
}

int add(int a, int b, int c) {
    return a + b + c;
}

public void testAdd() {
    add(10, 20);
    add(10, 20, 30);
}
```

```fun
fun add(a: Int, b: Int, c: Int = 0) =
a + b + c

fun testAdd() {
    add(10, 20)
    add(10, 20, 30)
}
```

- **a** and **b** are usual parameters.

overloading vs default parameters

```java
int add(int a, int b) {
    return a + b;
}

int add(int a, int b, int c) {
    return a + b + c;
}

public void testAdd() {
    add(10, 20);
    add(10, 20, 30);
}
```

```fun
fun add(a: Int, b: Int, c: Int = 0) =
a + b + c

fun testAdd() {
    add(10, 20)
    add(10, 20, 30)
}
```

- **a** and **b** are usual parameters.
- **c** is having a default value as 0.

overloading vs default parameters

```java
int add(int a, int b) {
    return a + b;
}

int add(int a, int b, int c) {
    return a + b + c;
}

public void testAdd() {
    add(10, 20);
    add(10, 20, 30);
}
```

```fun add(a: Int, b: Int, c: Int = 0) =
a + b + c

fun testAdd() {
    add(10, 20)
    add(10, 20, 30)
}
```

- **a** and **b** are usual parameters.
- **c** is having a default value as 0.
- When no value passed to **c**, 0 will be considered.

Type Cast vs Smart Cast

Type Cast vs Smart Cast

```java
public void print(Animal animal) {
    if (animal instanceof Dog) {
        Dog dog = (Dog) animal;
        S.out.println(dog.toDog());
    } else if (animal instanceof Cat) {
        Cat cat = (Cat) animal;
        S.out.println(cat.toCat());
    }
}
```

Type Cast vs Smart Cast

```java
public void print(Animal animal) {
    if (animal instanceof Dog) {
        Dog dog = (Dog) animal;
        S.out.println(dog.toDog());
    } else if (animal instanceof Cat) {
        Cat cat = (Cat) animal;
        S.out.println(cat.toCat());
    }
}
```

```fun
fun print(animal: Animal) {
    if (animal is Dog) {
        println(animal.toDog())
    } else if (animal is Cat) {
        println(animal.toCat())
    }
}
```

Type Cast vs Smart Cast

```java
public void print(Animal animal) {
    if (animal instanceof Dog) {
        Dog dog = (Dog) animal;
        S.out.println(dog.toDog());
    } else if (animal instanceof Cat) {
        Cat cat = (Cat) animal;
        S.out.println(cat.toCat());
    }
}
```

```fun
fun print(animal: Animal) {
    if (animal is Dog) {
        println(animal.toDog())
    } else if (animal is Cat) {
        println(animal.toCat())
    }
}
```

- **is** more readable than **instanceof**

Type Cast vs Smart Cast

```java
public void print(Animal animal) {
    if (animal instanceof Dog) {
        Dog dog = (Dog) animal;
        S.out.println(dog.toDog());
    } else if (animal instanceof Cat) {
        Cat cat = (Cat) animal;
        S.out.println(cat.toCat());
    }
}
```

```fun
fun print(animal: Animal) {
    if (animal is Dog) {
        println(animal.toDog())
    } else if (animal is Cat) {
        println(animal.toCat())
    }
}
```

- **is** more readable than **instanceof**
- if - **animal** is already a **Dog**

Type Cast vs Smart Cast

```java
public void print(Animal animal) {
    if (animal instanceof Dog) {
        Dog dog = (Dog) animal;
        S.out.println(dog.toDog());
    } else if (animal instanceof Cat) {
        Cat cat = (Cat) animal;
        S.out.println(cat.toCat());
    }
}
```

```fun
fun print(animal: Animal) {
    if (animal is Dog) {
        println(animal.toDog())
    } else if (animal is Cat) {
        println(animal.toCat())
    }
}
```

- **is** more readable than **instanceof**
- if - **animal** is already a **Dog**
- else if - **animal** is already a **Cat**

POJO vs data class

POJO vs data class

```java
class User {
    private String name;
    private String email;
    private String phone;

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

    public String getEmail() { return email; }

    public void setEmail(String email) { this.email = email; }

    public String getPhone() { return phone; }

    public void setPhone(String phone) { this.phone = phone; }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        User user = (User) o;
        return Objects.equals(name, user.name) &&
                Objects.equals(email, user.email) &&
                Objects.equals(phone, user.phone);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, email, phone);
    }

    @Override
    public String toString() {
        return "User{" +
```

POJO vs data class

```java
class User {
    private String name;
    private String email;
    private String phone;

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

    public String getEmail() { return email; }

    public void setEmail(String email) { this.email = email; }

    public String getPhone() { return phone; }

    public void setPhone(String phone) { this.phone = phone; }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        User user = (User) o;
        return Objects.equals(name, user.name) &&
                Objects.equals(email, user.email) &&
                Objects.equals(phone, user.phone);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, email, phone);
    }

    @Override
    public String toString() {
        return "User{" +
```

```fortran
data class User(var name: String, var email: String, var phone: String)
```

POJO vs data class

```java
class User {
    private String name;
    private String email;
    private String phone;

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

    public String getEmail() { return email; }

    public void setEmail(String email) { this.email = email; }

    public String getPhone() { return phone; }

    public void setPhone(String phone) { this.phone = phone; }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        User user = (User) o;
        return Objects.equals(name, user.name) &&
                Objects.equals(email, user.email) &&
                Objects.equals(phone, user.phone);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, email, phone);
    }

    @Override
    public String toString() {
        return "User{" +
```

```fun
data class User(var name: String, var email: String, var phone: String)
```

- Just prefix with **data** keyword and your POJO is ready.

POJO vs data class

```java
class User {
    private String name;
    private String email;
    private String phone;

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

    public String getEmail() { return email; }

    public void setEmail(String email) { this.email = email; }

    public String getPhone() { return phone; }

    public void setPhone(String phone) { this.phone = phone; }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        User user = (User) o;
        return Objects.equals(name, user.name) &&
                Objects.equals(email, user.email) &&
                Objects.equals(phone, user.phone);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, email, phone);
    }

    @Override
    public String toString() {
        return "User{" +
```

```fun
data class User(var name: String, var email: String, var phone: String)
```

- Just prefix with **data** keyword and your POJO is ready.

POJO vs data class

```java
class User {
    private String name;
    private String email;
    private String phone;

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

    public String getEmail() { return email; }

    public void setEmail(String email) { this.email = email; }

    public String getPhone() { return phone; }

    public void setPhone(String phone) { this.phone = phone; }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        User user = (User) o;
        return Objects.equals(name, user.name) &&
                Objects.equals(email, user.email) &&
                Objects.equals(phone, user.phone);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, email, phone);
    }

    @Override
    public String toString() {
        return "User{" +
```

```fr
data class User(var name: String, var email: String, var phone: String)
```

- Just prefix with **data** keyword and your POJO is ready.
- You get **getters**, **setters**, **equals()**, **hashCode()** and **toString()** for free.

POJO vs data class

```java
class User {
    private String name;
    private String email;
    private String phone;

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

    public String getEmail() { return email; }

    public void setEmail(String email) { this.email = email; }

    public String getPhone() { return phone; }

    public void setPhone(String phone) { this.phone = phone; }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        User user = (User) o;
        return Objects.equals(name, user.name) &&
                Objects.equals(email, user.email) &&
                Objects.equals(phone, user.phone);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, email, phone);
    }

    @Override
    public String toString() {
        return "User{" +
```

```java
data class User(var name: String, var email: String, var phone: String)
```

- Just prefix with **data** keyword and your POJO is ready.
- You get **getters**, **setters**, **equals()**, **hashCode()** and **toString()** for free.
- A **copy()** also in addition to all.

POJO vs data class

```java
class User {
    private String name;
    private String email;
    private String phone;

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

    public String getEmail() { return email; }

    public void setEmail(String email) { this.email = email; }

    public String getPhone() { return phone; }

    public void setPhone(String phone) { this.phone = phone; }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        User user = (User) o;
        return Objects.equals(name, user.name) &&
                Objects.equals(email, user.email) &&
                Objects.equals(phone, user.phone);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, email, phone);
    }

    @Override
    public String toString() {
        return "User{" +
```

```java
data class User(var name: String, var email: String, var phone: String)
```

- Just prefix with **data** keyword and your POJO is ready.
- You get **getters**, **setters**, **equals()**, **hashCode()** and **toString()** for free.
- A **copy()** also in addition to all.
- **var** - both getters and setters

POJO vs data class

```java
class User {
    private String name;
    private String email;
    private String phone;

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

    public String getEmail() { return email; }

    public void setEmail(String email) { this.email = email; }

    public String getPhone() { return phone; }

    public void setPhone(String phone) { this.phone = phone; }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        User user = (User) o;
        return Objects.equals(name, user.name) &&
                Objects.equals(email, user.email) &&
                Objects.equals(phone, user.phone);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, email, phone);
    }

    @Override
    public String toString() {
        return "User{" +
```

```fr
data class User(var name: String, var email: String, var phone: String)
```

- Just prefix with **data** keyword and your POJO is ready.
- You get **getters**, **setters**, **equals()**, **hashCode()** and **toString()** for free.
- A **copy()** also in addition to all.
- **var** - both getters and setters

POJO vs data class

```java
class User {
    private String name;
    private String email;
    private String phone;

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

    public String getEmail() { return email; }

    public void setEmail(String email) { this.email = email; }

    public String getPhone() { return phone; }

    public void setPhone(String phone) { this.phone = phone; }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        User user = (User) o;
        return Objects.equals(name, user.name) &&
                Objects.equals(email, user.email) &&
                Objects.equals(phone, user.phone);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, email, phone);
    }

    @Override
    public String toString() {
        return "User{" +
```

```gradle
data class User(var name: String, val email: String, var phone: String)
```

- Just prefix with **data** keyword and your POJO is ready.
- You get **getters**, **setters**, **equals()**, **hashCode()** and **toString()** for free.
- A **copy()** also in addition to all.
- **var** - both getters and setters
- **val** - only getters

Features of Kotlin

Features that are not present in Java

Type System

Nullable Types

Nullable Types

- Kotlin’s solution to billion dollar mistake.

Nullable Types

- Kotlin’s solution to billion dollar mistake.
- By default everything in Kotlin is **non-nullable**.

Nullable Types

- Kotlin’s solution to billion dollar mistake.
- By default everything in Kotlin is **non-nullable**.
- If you want to assign null to a variable, you have to declare that as a **nullable**.

Nullable Types

- Kotlin’s solution to billion dollar mistake.
- By default everything in Kotlin is **non-nullable**.
- If you want to assign null to a variable, you have to declare that as a **nullable**.
- Syntax:
  - non-nullable - Int, String, User
  - nullable - Int?, String?, User?

Nullable Types

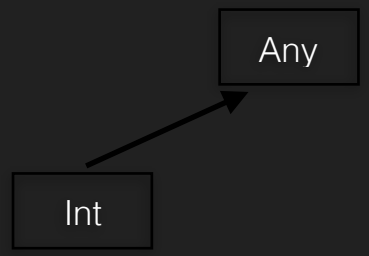
- Kotlin’s solution to billion dollar mistake.
- By default everything in Kotlin is **non-nullable**.
- If you want to assign null to a variable, you have to declare that as a **nullable**.
- Syntax:
  - non-nullable - Int, String, User
  - nullable - Int?, String?, User?
- **User?** is super class of **User** as it can hold null additionally.

Type Hierarchy

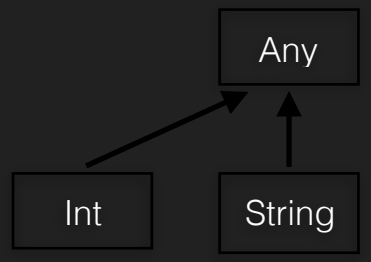
Type Hierarchy

Any

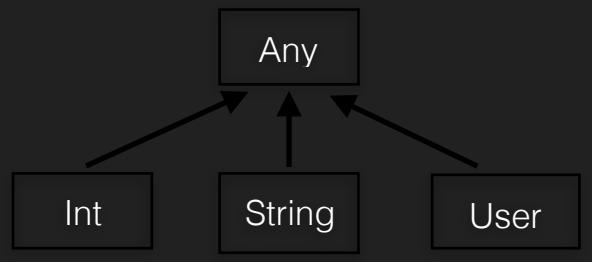
Type Hierarchy



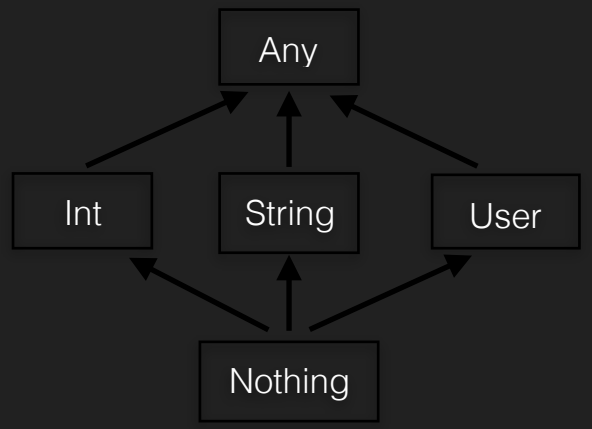
Type Hierarchy



Type Hierarchy

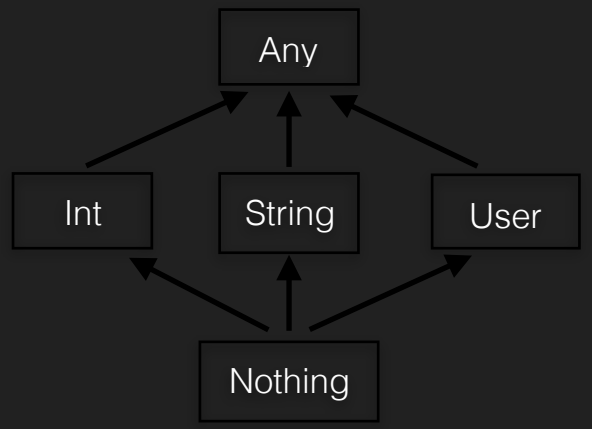


Type Hierarchy

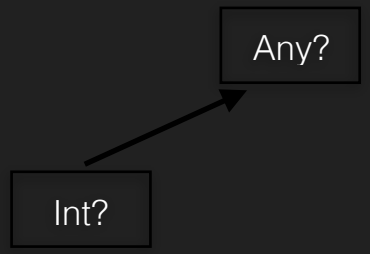


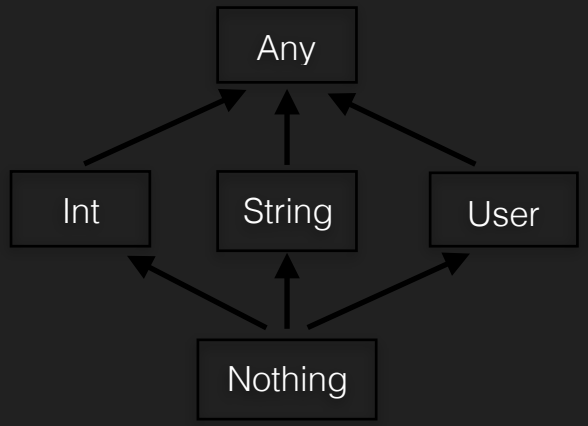
Type Hierarchy

Any?

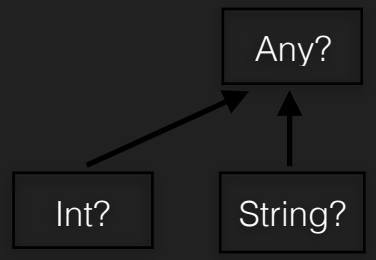


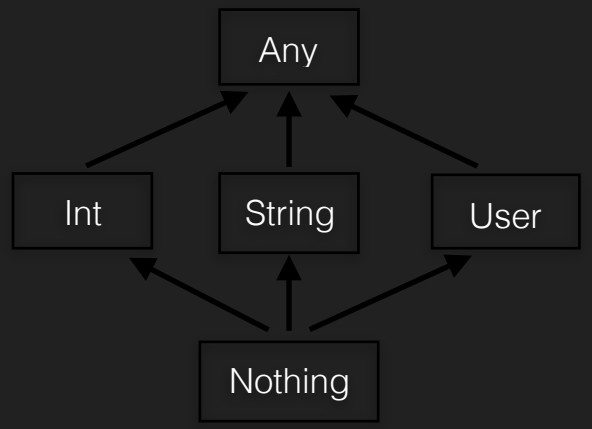
Type Hierarchy



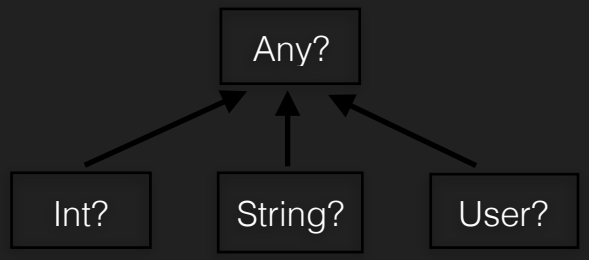


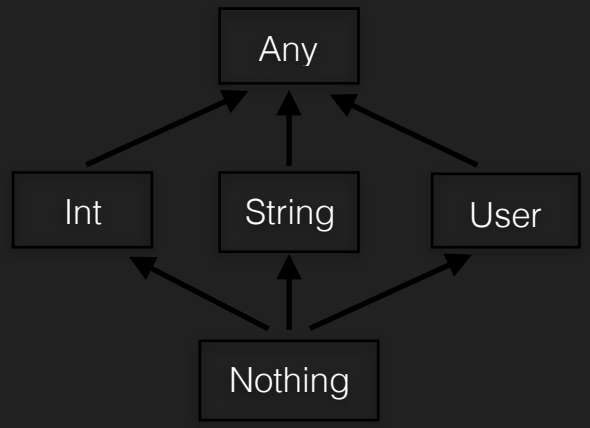
Type Hierarchy



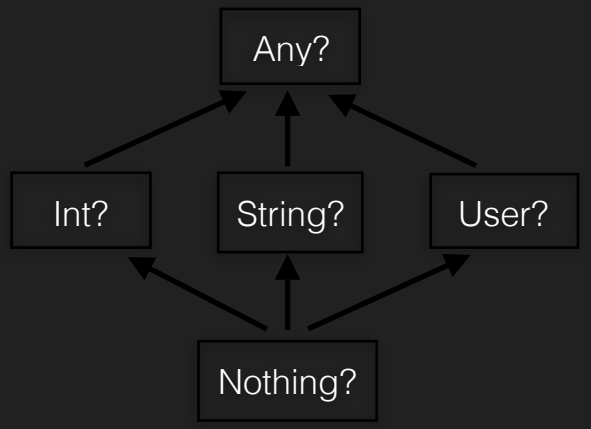


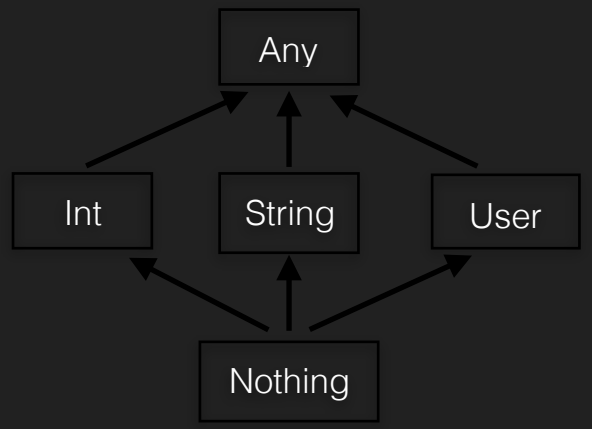
Type Hierarchy



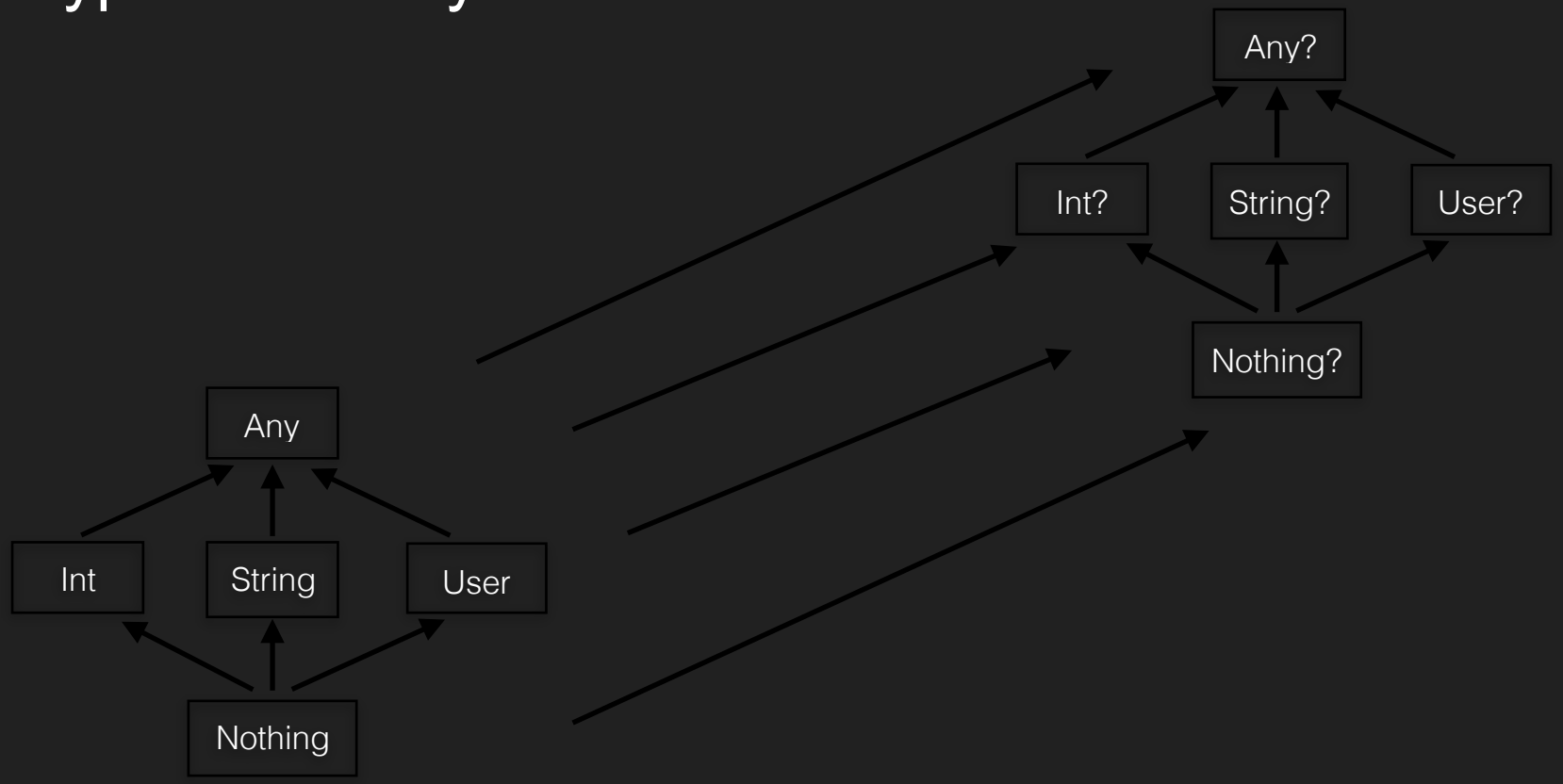


Type Hierarchy





Type Hierarchy



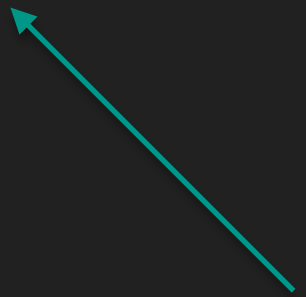
Examples - Nullable Types

Examples - Nullable Types

```
var nullableName: String = null
```

Examples - Nullable Types

```
var nullableName: String = null
```



Compilation Error: Can't assign null to a nun-null String

Examples - Nullable Types

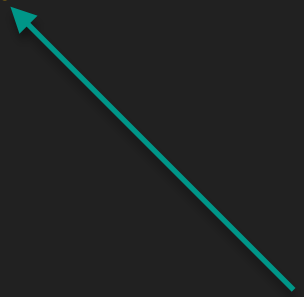
```
var nullableName: String = null

var nullableName: String? = null
```

Examples - Nullable Types

```
var nullableName: String = null

var nullableName: String? = null
```



Compiles fine

Examples - Nullable Types

```
var nullableName: String = null

var nullableName: String? = null

var name: String = "BangaloreJUG"
```

Examples - Nullable Types

```
var nullableName: String = null

var nullableName: String? = null

var name: String = "BangaloreJUG"
```

Type - Non Nullable

Examples - Nullable Types

```
var nullableName: String = null

var nullableName: String? = null

var name: String = "BangaloreJUG"
```

Type - Non Nullable

Value should not be null

Examples - Nullable Types

```
var nullableName: String = null

var nullableName: String? = null

var name: String = "BangaloreJUG"

nullableName = name
```

Examples - Nullable Types

```
var nullableName: String = null

var nullableName: String? = null

var name: String = "BangaloreJUG"

nullableName = name
```

Compiles fine: String is sub-class of String?

Examples - Nullable Types

```
var nullableName: String = null

var nullableName: String? = null

var name: String = "BangaloreJUG"

nullableName = name

name = nullableName
```

Examples - Nullable Types

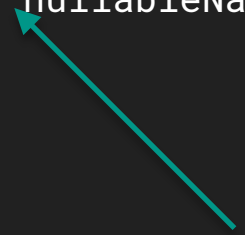
```
var nullableName: String = null

var nullableName: String? = null

var name: String = "BangaloreJUG"

nullableName = name

name = nullableName
```



Compilation Error: You know why

Examples - Nullable Types

```
var nullableName: String = null

var nullableName: String? = null

var name: String = "BangaloreJUG"

nullableName = name

name = nullableName
```

Destructuring Declaration

Let’s return multiple values from a function

Let’s return multiple values from a function

```
data class User(var name: String,
                var email: String,
                var phone: String)
```

Let’s return multiple values from a function

```
data class User(var name: String,
                var email: String,
                var phone: String)



fun getUser() = User("BangaloreJUG",
            "bangalorejug@gmail.com",
            "0000000000")
```

Let’s return multiple values from a function

```
data class User(var name: String,
                var email: String,
                var phone: String)
```

```
fun getUser() = User("BangaloreJUG",
            "bangalorejug@gmail.com",
            "0000000000")
```

```
val (name, email, phone) =
                User("Chandra Sekhar Nayak",
                        "chansek@live.com",
                        "8792629767")
println("Name is - $name")
println("Email is - $email")
println("Phone is - $phone")
```

Let’s return multiple values from a function

```
data class User(var name: String,
                var email: String,
                var phone: String)
```

```
fun getUser() = User("BangaloreJUG",
            "bangalorejug@gmail.com",
            "0000000000")
```

```
val (name, email, phone) =
                User("Chandra Sekhar Nayak",
                        "chansek@live.com",
                        "8792629767")
println("Name is - $name")
println("Email is - $email")
println("Phone is - $phone")
```

```
val (group, _, contact) = getUser()
println("Meetup Group name - $group")
println("Contact number - $contact")
```

Let’s return multiple values from a function

```
data class User(var name: String,
                var email: String,
                var phone: String)
```

```
fun getUser() = User("BangaloreJUG",
            "bangalorejug@gmail.com",
            "0000000000")
```

```
for ((key, value) in map) {
   // do something with key and value
}
```

```
val (name, email, phone) =
                User("Chandra Sekhar Nayak",
                        "chansek@live.com",
                        "8792629767")
println("Name is - $name")
println("Email is - $email")
println("Phone is - $phone")
```

```
val (group, _, contact) = getUser()
println("Meetup Group name - $group")
println("Contact number - $contact")
```

Ranges

The world of .. operator

The world of .. operator

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

The world of .. operator

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

```
for (i in 1..4) print(i) // "1234"
```

The world of .. operator

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

```
for (i in 1..4) print(i) // "1234"
```

```
for (i in 4..1) print(i) // No Output
for (i in 4 downTo 1)
    print(i)             // "4321"
```

The world of .. operator

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

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

```
for (i in 1..4) print(i) // "1234"
```

```
for (i in 4..1) print(i) // No Output
for (i in 4 downTo 1)
    print(i)             // "4321"
```

The world of .. operator

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

```
for (i in 1..4) print(i) // "1234"
```

```
for (i in 4..1) print(i) // No Output
for (i in 4 downTo 1)
    print(i)             // "4321"
```

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

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

The world of .. operator

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

```
for (i in 1..4) print(i) // "1234"
```

```
for (i in 4..1) print(i) // No Output
for (i in 4 downTo 1)
    print(i)             // "4321"
```

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

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

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

Operator Overloading

Unary Operations

Unary Operations

```
data class Point(val x: Int, val y: Int)
```

Unary Operations

```
data class Point(val x: Int, val y: Int)

operator fun Point.unaryMinus() = Point(-x, -y)
```

Unary Operations

```
data class Point(val x: Int, val y: Int)

operator fun Point.unaryMinus() = Point(-x, -y)

fun main(args: Array<String>) {
    val point = Point(10, 20)
    println(-point)  // prints "(-10, -20)"
}
```

Unary Operations

```
data class Point(val x: Int, val y: Int)

operator fun Point.unaryMinus() = Point(-x, -y)

fun main(args: Array<String>) {
    val point = Point(10, 20)
    println(-point)  // prints "(-10, -20)"
}
```

- 3 operators are supported.

Unary Operations

```
data class Point(val x: Int, val y: Int)

operator fun Point.unaryMinus() = Point(-x, -y)

fun main(args: Array<String>) {
    val point = Point(10, 20)
    println(-point)  // prints "(-10, -20)"
}
```

- 3 operators are supported.
- Function names should not be changed.

Unary Operations

```
data class Point(val x: Int, val y: Int)

operator fun Point.unaryMinus() = Point(-x, -y)

fun main(args: Array<String>) {
    val point = Point(10, 20)
    println(-point)  // prints "(-10, -20)"
}
```

- 3 operators are supported.
- Function names should not be changed.
- Like this there are some other functions for different operators.

Unary Operations

```
data class Point(val x: Int, val y: Int)

operator fun Point.unaryMinus() = Point(-x, -y)

fun main(args: Array<String>) {
    val point = Point(10, 20)
    println(-point)  // prints "(-10, -20)"
}
```

Expression	Function
+a	a.unaryPlus()
-a	a.unaryMinus()
!a	a.not()

- 3 operators are supported.
- Function names should not be changed.
- Like this there are some other functions for different operators.

Increment and Decrement Operations

Increment and Decrement Operations

Expression	Function
a++	a.inc()
a—	a.dec()

Increment and Decrement Operations

Expression	Function
a++	a.inc()
a—	a.dec()

Binary Operations

Expression	Function
a++	a.inc()
a—	a.dec()

Expression	Function
a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.times(b)
a / b	a.div(b)
a % b	a.rem(b)
a..b	a.rangeTo(b)

Binary Operations

Expression	Function
a++	a.inc()
a—	a.dec()

Expression	Function
a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.times(b)
a / b	a.div(b)
a % b	a.rem(b)
a..b	a.rangeTo(b)

‘in’ Operator

Expression	Function
a++	a.inc()
a—	a.dec()

Expression	Function
a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.times(b)
a / b	a.div(b)
a % b	a.rem(b)
a..b	a.rangeTo(b)

Expression	Function
a in b	b.contains(a)
a !in b	!b.contains(a)

‘in’ Operator

Expression	Function
a++	a.inc()
a—	a.dec()

Expression	Function
a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.times(b)
a / b	a.div(b)
a % b	a.rem(b)
a..b	a.rangeTo(b)

Expression	Function
a in b	b.contains(a)
a !in b	!b.contains(a)

Indexed Access Operator

Expression	Function
a++	a.inc()
a—	a.dec()

Expression	Function
a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.times(b)
a / b	a.div(b)
a % b	a.rem(b)
a..b	a.rangeTo(b)

Expression	Function
a in b	b.contains(a)
a !in b	!b.contains(a)

Expression	Function
a[i]	a.get(i)
a[i, j]	a.get(i, j)
a[i1, i2, .., in]	a.get(i1, i2, .., in)
a[i] = b	a.set(i, b)
a[i, j] = b	a.set(i, j, b)
a[i1, i2, .., in] = b	a.set(i1, i2, .., in, b)

Indexed Access Operator

Expression	Function
a++	a.inc()
a—	a.dec()

Expression	Function
a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.times(b)
a / b	a.div(b)
a % b	a.rem(b)
a..b	a.rangeTo(b)

Expression	Function
a in b	b.contains(a)
a !in b	!b.contains(a)

Expression	Function
a[i]	a.get(i)
a[i, j]	a.get(i, j)
a[i1, i2, .., in]	a.get(i1, i2, .., in)
a[i] = b	a.set(i, b)
a[i, j] = b	a.set(i, j, b)
a[i1, i2, .., in] = b	a.set(i1, i2, .., in, b)

Invoke Operator

Expression	Function
a++	a.inc()
a—	a.dec()

Expression	Function
a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.times(b)
a / b	a.div(b)
a % b	a.rem(b)
a..b	a.rangeTo(b)

Expression	Function
a in b	b.contains(a)
a !in b	!b.contains(a)

Expression	Function
a()	a.invoke()
a(i)	a.invoke(i)
a(i, j)	a.invoke(i, j)

Expression	Function
a[i]	a.get(i)
a[i, j]	a.get(i, j)
a[i1, i2, .., in]	a.get(i1, i2, .., in)
a[i] = b	a.set(i, b)
a[i, j] = b	a.set(i, j, b)
a[i1, i2, .., in] = b	a.set(i1, i2, .., in, b)

Invoke Operator

Expression	Function
a++	a.inc()
a—	a.dec()

Expression	Function
a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.times(b)
a / b	a.div(b)
a % b	a.rem(b)
a..b	a.rangeTo(b)

Expression	Function
a in b	b.contains(a)
a !in b	!b.contains(a)

Expression	Function
a[i]	a.get(i)
a[i, j]	a.get(i, j)
a[i1, i2, .., in]	a.get(i1, i2, .., in)
a[i] = b	a.set(i, b)
a[i, j] = b	a.set(i, j, b)
a[i1, i2, .., in] = b	a.set(i1, i2, .., in, b)

Expression	Function
a()	a.invoke()
a(i)	a.invoke(i)
a(i, j)	a.invoke(i, j)

Extension Functions

Extending an existing class

Extending an existing class

- The most powerful feature of Kotlin

Extending an existing class

- The most powerful feature of Kotlin
- No utility class is required

Extending an existing class

- The most powerful feature of Kotlin
- No utility class is required
- IDE can auto suggest

Extending an existing class

- The most powerful feature of Kotlin
- No utility class is required
- IDE can auto suggest
- Accidentally multiple utility function for same doesn’t get created

Let’s extend the functionality of a library class

Let’s extend the functionality of a library class

```
fun ArrayList<String>.swap(index1: Int, index2: Int) {
    val tmp = this[index1]
    this[index1] = this[index2]
    this[index2] = tmp
}
```

Let’s extend the functionality of a library class

```
fun ArrayList<String>.swap(index1: Int, index2: Int) {
    val tmp = this[index1]
    this[index1] = this[index2]
    this[index2] = tmp
}


fun main(args: Array<String>) {
    val list = arrayListOf("BangaloreJUG", "BlrKotlin", "BlrDroid")
    list.swap(0, 1)
    print(list)
}
```

Inline Functions

Inline Functions

```
var value = 0

fun operate(a: Int, func: () -> Unit) {
    value = a
    func()
}
```

Inline Functions

```
var value = 0

fun operate(a: Int, func: () -> Unit) {
    value = a
    func()
}
```

```
fun main(args: Array<String>) {
    assignAndOperate(2) {
        value *= 5
    }
    println(value)

    operate(2) {
        value *= 10
    }
    println(value)
}
```

Inline Functions

```
var value = 0

inline fun operate(a: Int, func: () -> Unit) {
    value = a
    func()
}
```

```
fun main(args: Array<String>) {
    assignAndOperate(2) {
        value *= 5
    }
    println(value)

    operate(2) {
        value *= 10
    }
    println(value)
}
```

Inline Functions

```
var value = 0

inline fun operate(a: Int, func: () -> Unit) {
    value = a
    func()
}
```

inline tells compiler to copy the function body to each calling place.

```
fun main(args: Array<String>) {
    assignAndOperate(2) {
        value *= 5
    }
    println(value)

    operate(2) {
        value *= 10
    }
    println(value)
}
```

Infix Functions

Infix Functions

```
class Couple(private val first: String, private val second: String)
```

Infix Functions

```
class Couple(private val first: String, private val second: String)

fun getOldCouples() = arrayListOf(
        Couple("Virat", “Anushka"),
        Couple("Rekha", “Amitabh")
)
```

Infix Functions

```
class Couple(private val first: String, private val second: String)

fun getOldCouples() = arrayListOf(
        Couple("Virat", “Anushka"),
        Couple("Rekha", “Amitabh")
)

infix fun String.loves(that: String) = Couple(this, that)
```

Infix Functions

```
class Couple(private val first: String, private val second: String)

fun getOldCouples() = arrayListOf(
        Couple("Virat", “Anushka"),
        Couple("Rekha", “Amitabh")
)

infix fun String.loves(that: String) = Couple(this, that)

fun getModernCouples() = arrayListOf(
        "Virat" loves "Anushka",
        "Rekha" loves "Amitabh"
)
```

Infix Functions

```
class Couple(private val first: String, private val second: String)

fun getOldCouples() = arrayListOf(
        Couple("Virat", “Anushka"),
        Couple("Rekha", “Amitabh")
)

infix fun String.loves(that: String) = Couple(this, that)

fun getModernCouples() = arrayListOf(
        "Virat" loves "Anushka",
        "Rekha" loves "Amitabh"
)

val languages = mapOf(1 to "Java", 2 to "Kotlin", 3 to "Scala")
```

Java interoperability

Calling Java from Kotlin

Calling Java from Kotlin

```java
public class Student {
    private String name;
    private String rollNo;

    public String getName() { return name; }

    public void setName(String name) {
        this.name = name;
    }

    public String getRollNo() { return rollNo; }

    public void setRollNo(String rollNo) {
        this.rollNo = rollNo;
    }
}
```

Calling Java from Kotlin

```java
public class Student {
    private String name;
    private String rollNo;

    public String getName() { return name; }

    public void setName(String name) {
        this.name = name;
    }

    public String getRollNo() { return rollNo; }

    public void setRollNo(String rollNo) {
        this.rollNo = rollNo;
    }
}
```

```kotlin
fun main(args: Array<String>) {
    val student = Student()
    student.name = "Rahim"
    student.rollNo = "R1"

    val name = student.name
    val rollNo = student.rollNo
}
```

Calling Java from Kotlin

```java
public class Student {
    private String name;
    private String rollNo;

    public String getName() { return name; }

    public void setName(String name) {
        this.name = name;
    }

    public String getRollNo() { return rollNo; }

    public void setRollNo(String rollNo) {
        this.rollNo = rollNo;
    }
}
```

```kotlin
fun main(args: Array<String>) {
    val student = Student()
    student.name = "Rahim"
    student.rollNo = "R1"

    val name = student.name
    val rollNo = student.rollNo
}
```

same as student.setName(“Rahim”)

Calling Java from Kotlin

```java
public class Student {
    private String name;
    private String rollNo;

    public String getName() { return name; }

    public void setName(String name) {
        this.name = name;
    }

    public String getRollNo() { return rollNo; }

    public void setRollNo(String rollNo) {
        this.rollNo = rollNo;
    }
}
```

```kotlin
fun main(args: Array<String>) {
    val student = Student()
    student.name = "Rahim"
    student.rollNo = "R1"

    val name = student.name
    val rollNo = student.rollNo
}
```

same as student.getRollNo()

Calling Kotlin from Java

Calling Kotlin from Java

```kotlin
class Meetup {
    var name: String = ""
    var location: String = ""
}
```

Calling Kotlin from Java

```kotlin
class Meetup {
    var name: String = ""
    var location: String = ""
}
```

```java
public class Main {
    public static void main(String... args) {
        Meetup meetup = new Meetup();
        meetup.setName("BangaloreJUG");
        meetup.setLocation("Oracle");

        System.out.println(meetup.getName());
        System.out.println(meetup.getLocation());
    }
}
```

Calling Kotlin from Java

```kotlin
class Meetup {
    var name: String = ""
    var location: String = ""
}
```

```java
public class Main {
    public static void main(String... args) {
        Meetup meetup = new Meetup();
        meetup.setName("BangaloreJUG");
        meetup.setLocation("Oracle");

        System.out.println(meetup.getName());
        System.out.println(meetup.getLocation());
    }
}
```

same as meetup.name = “Rahim”

Calling Kotlin from Java

```kotlin
class Meetup {
    var name: String = ""
    var location: String = ""
}
```

```java
public class Main {
    public static void main(String... args) {
        Meetup meetup = new Meetup();
        meetup.setName("BangaloreJUG");
        meetup.setLocation("Oracle");

        System.out.println(meetup.getName());
        System.out.println(meetup.getLocation());
    }
}
```

same as meetup.location

Coroutines

A small example

A small example

```
fun main(args: Array<String>) {
    launch {
        delay(1000L)
        println("World!")
    }
    println("Hello,")
    Thread.sleep(2000L)
}
```

A small example

```
fun main(args: Array<String>) {
    launch {
        delay(1000L)
        println("World!")
    }
    println("Hello,")
    Thread.sleep(2000L)
}
```

launch new coroutine in background and continue

A small example

```
fun main(args: Array<String>) {
    launch {
        delay(1000L)
        println("World!")
    }
    println("Hello,")
    Thread.sleep(2000L)
}
```

non-blocking delay for 1 second (default time unit is ms)

A small example

```
fun main(args: Array<String>) {
    launch {
        delay(1000L)
        println("World!")
    }
    println("Hello,")
    Thread.sleep(2000L)
}
```

print after delay

A small example

```
fun main(args: Array<String>) {
    launch {
        delay(1000L)
        println("World!")
    }
    println("Hello,")
    Thread.sleep(2000L)
}
```

main thread continues while coroutine is delayed

Thank You

Join BlrKotlin for continuing your journey in Kotlin

meetup.com/blrkotlin

blrkotlin.herokuapp.com

@iChanSek

www.chansek.com