Introduction to Kotlin Workshop for Intermediate Android Developers

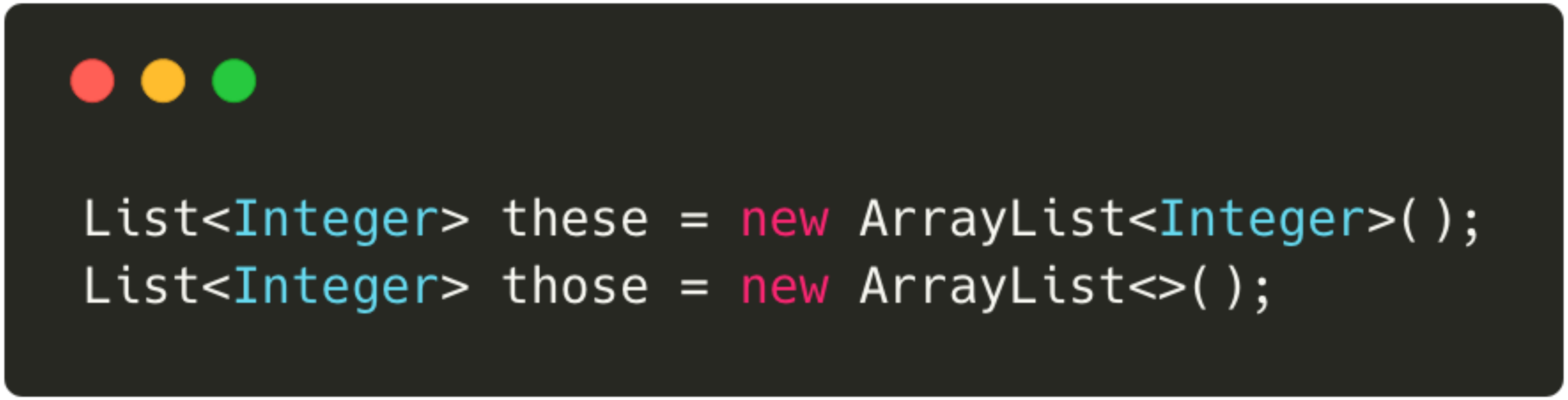
June 9 + 10, 2018

@velosmobile

Generics

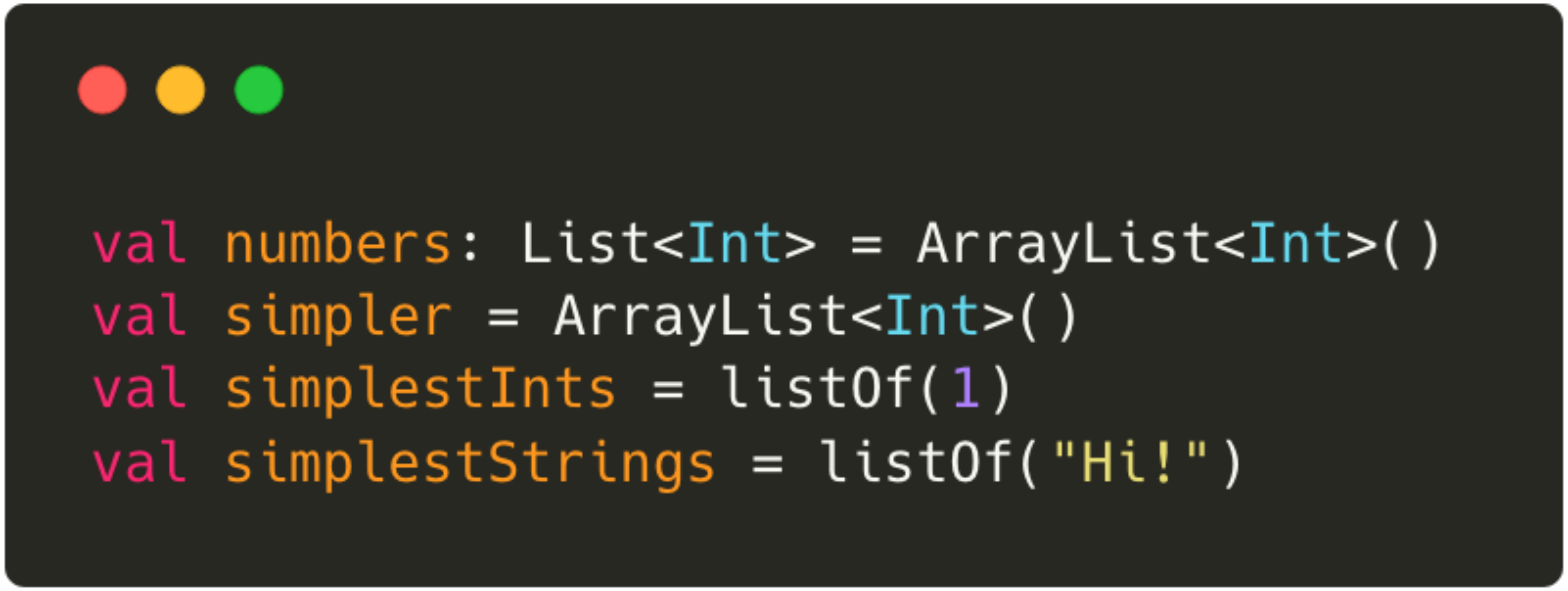
Invariance

Java Generics



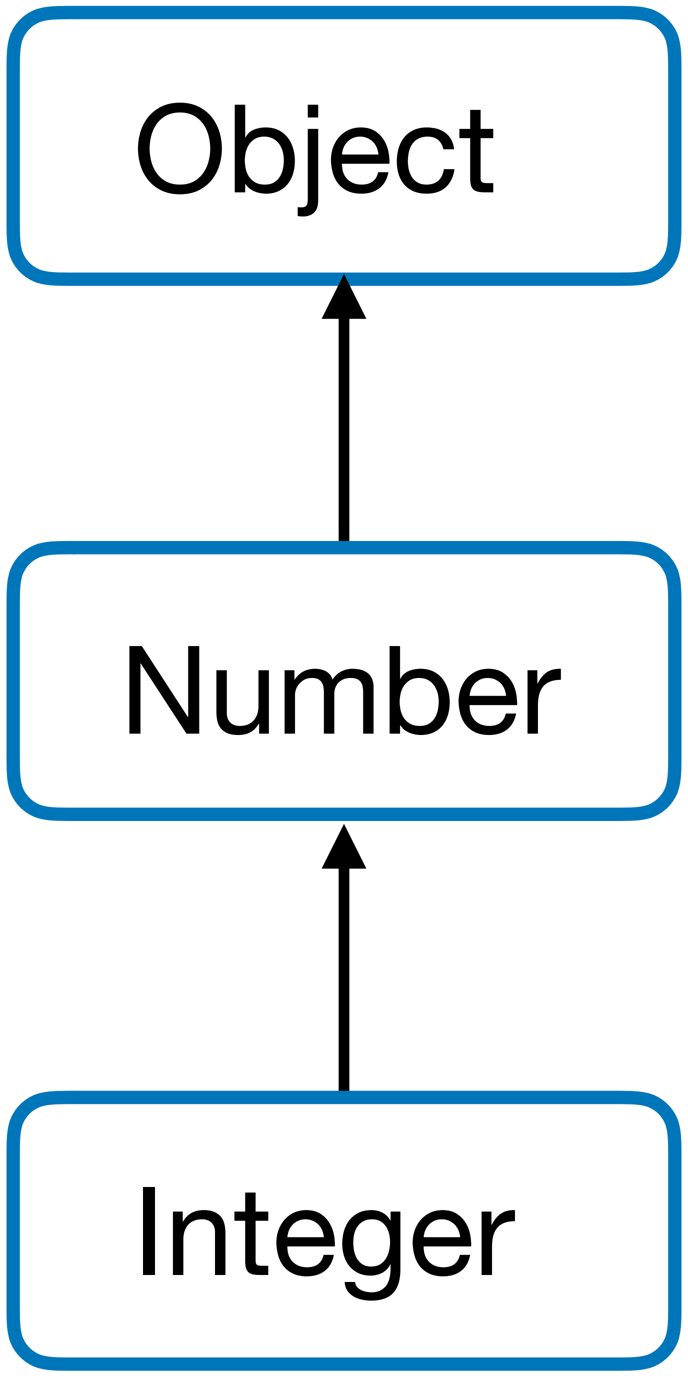
```
List<Integer> these = new ArrayList<Integer>();
List<Integer> those = new ArrayList<>();
```

Kotlin Generics

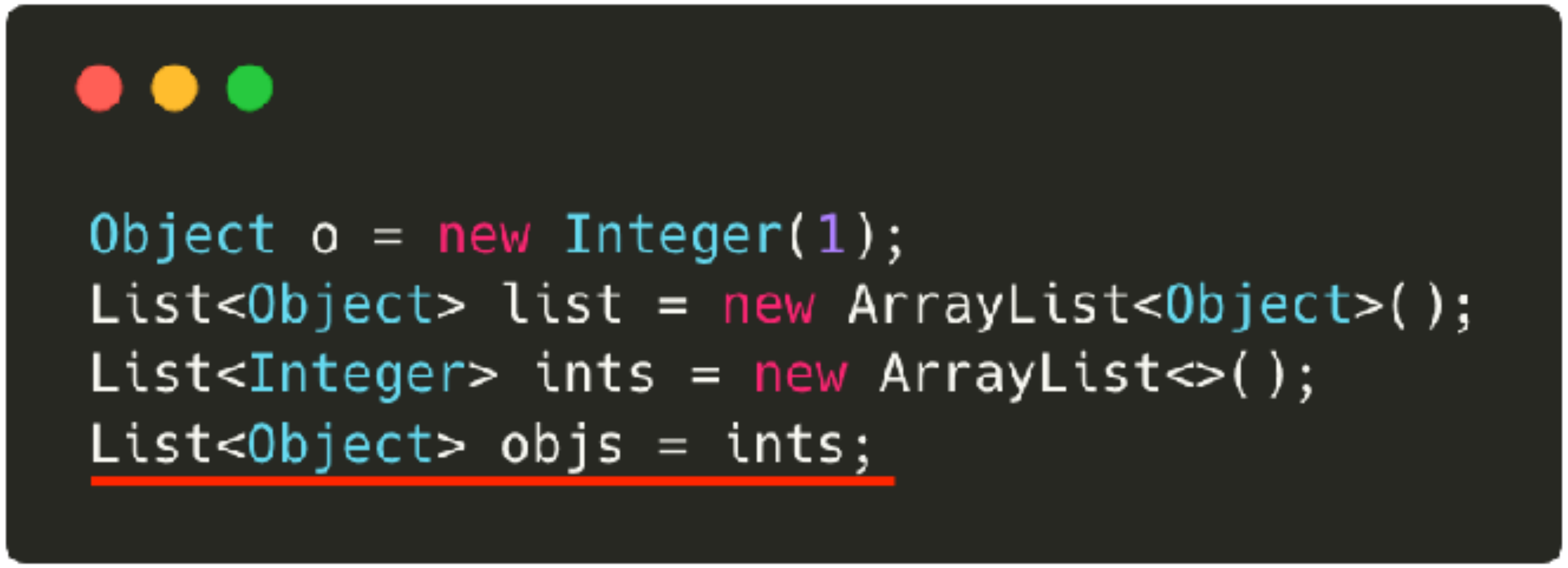


```kotlin
val numbers: List<Int> = ArrayList<Int>()
val simpler = ArrayList<Int>()
val simplestInts = listOf(1)
val simplestStrings = listOf("Hi!")
```

Java Inheritance Hierarchy

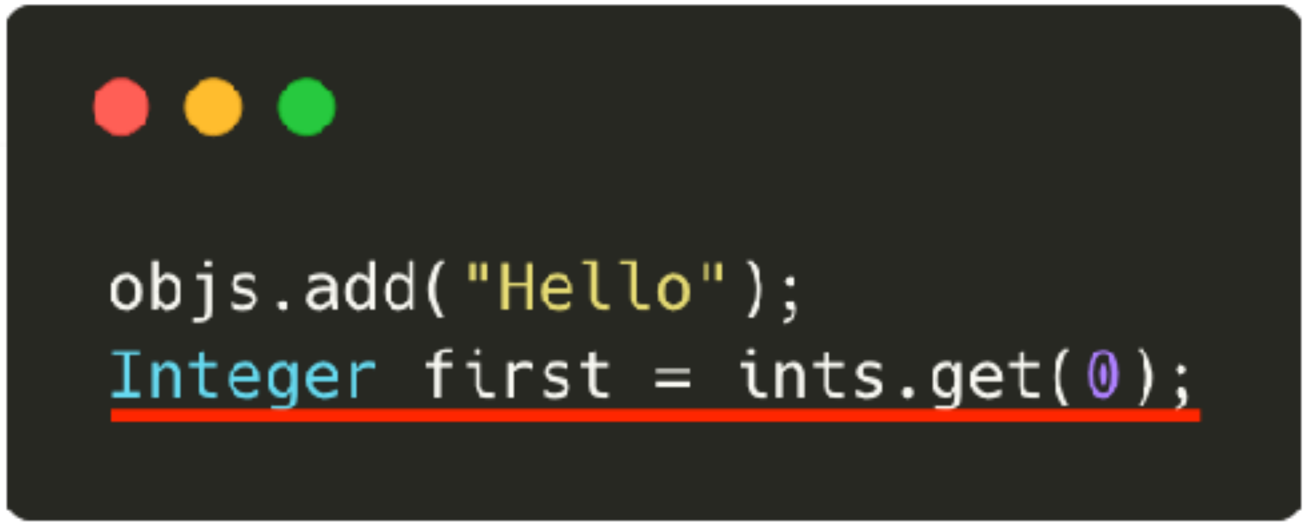


Invariance



```java
Object o = new Integer(1);
List<Object> list = new ArrayList<Object>();
List<Integer> ints = new ArrayList<>();
List<Object> objs = ints;
```

If that compiled,
we could do this:

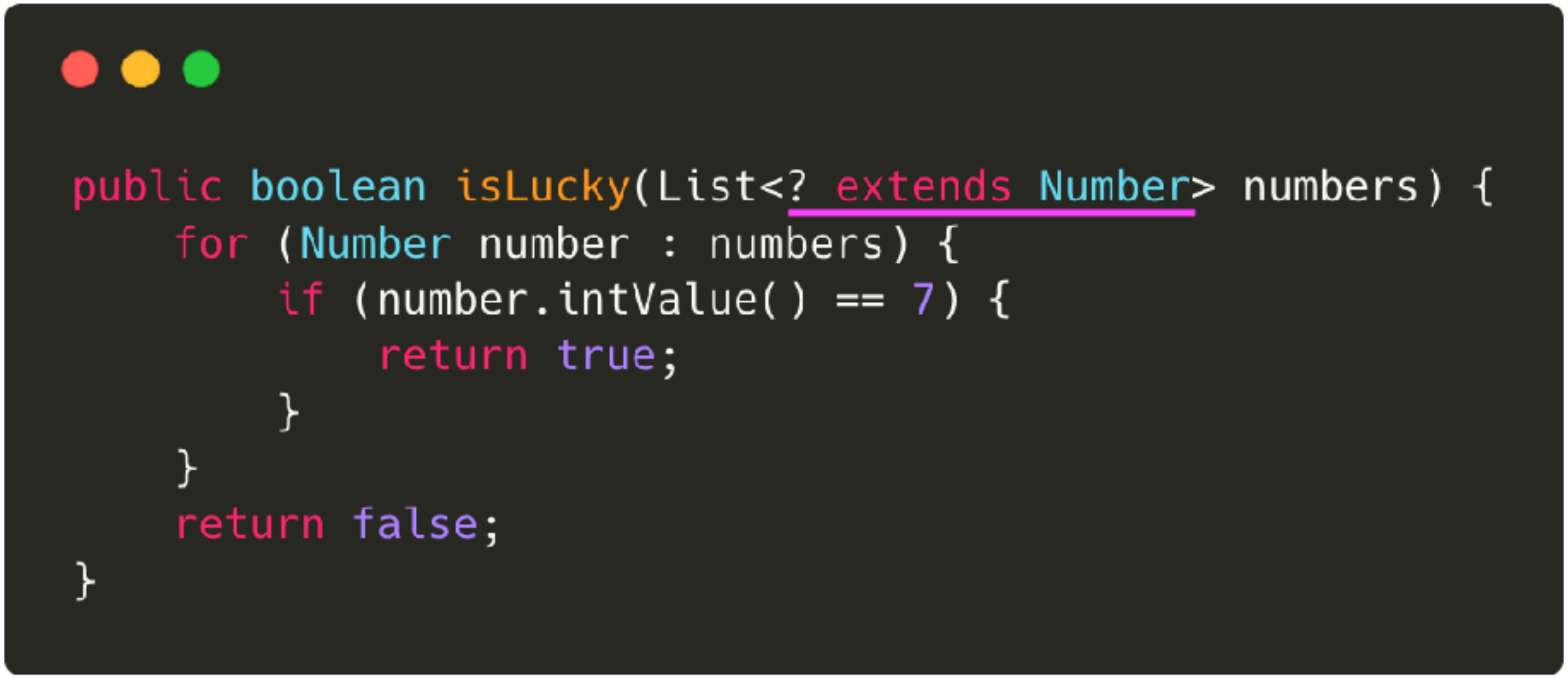


```java
objs.add("Hello");
Integer first = ints.get(0);
```

Generics are *invariant.*

Covariance

Java Covariance

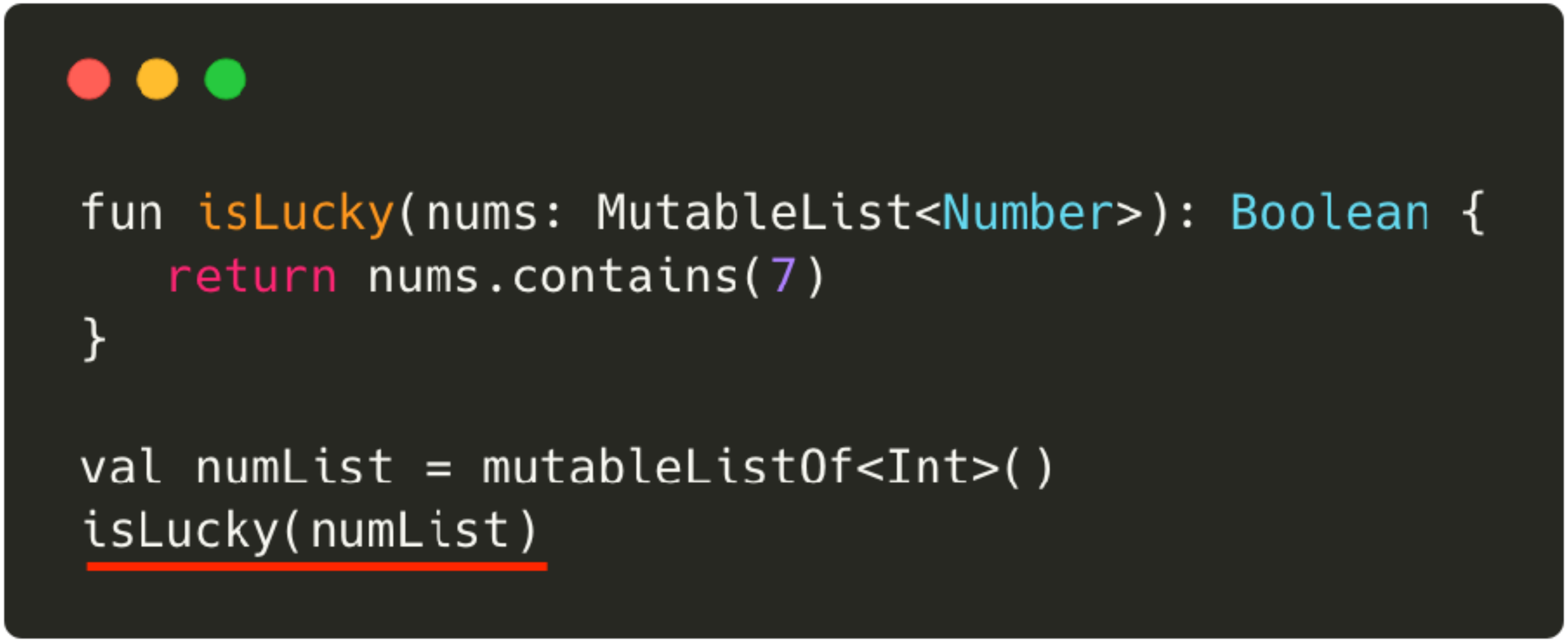


```java
public boolean isLucky(List<? extends Number> numbers) {
    for (Number number : numbers) {
        if (number.intValue() == 7) {
            return true;
        }
    }
    return false;
}
```

Covariance

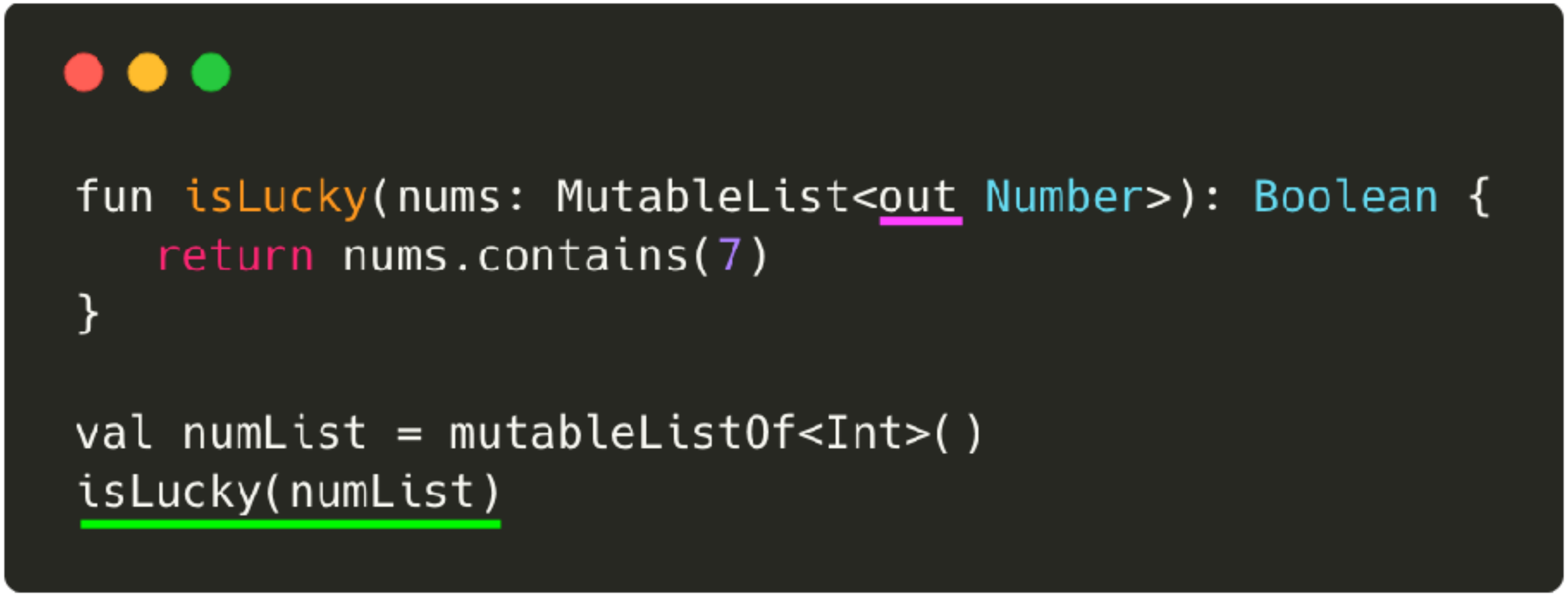
- Accept a subtype
- Can read from the generic
- Cannot write to the generic

Kotlin Variance



```kotlin
fun isLucky(nums: MutableList<Number>): Boolean {
    return nums.contains(7)
}

val numList = mutableListOf<Int>()
isLucky(numList)
```

Kotlin Covariance



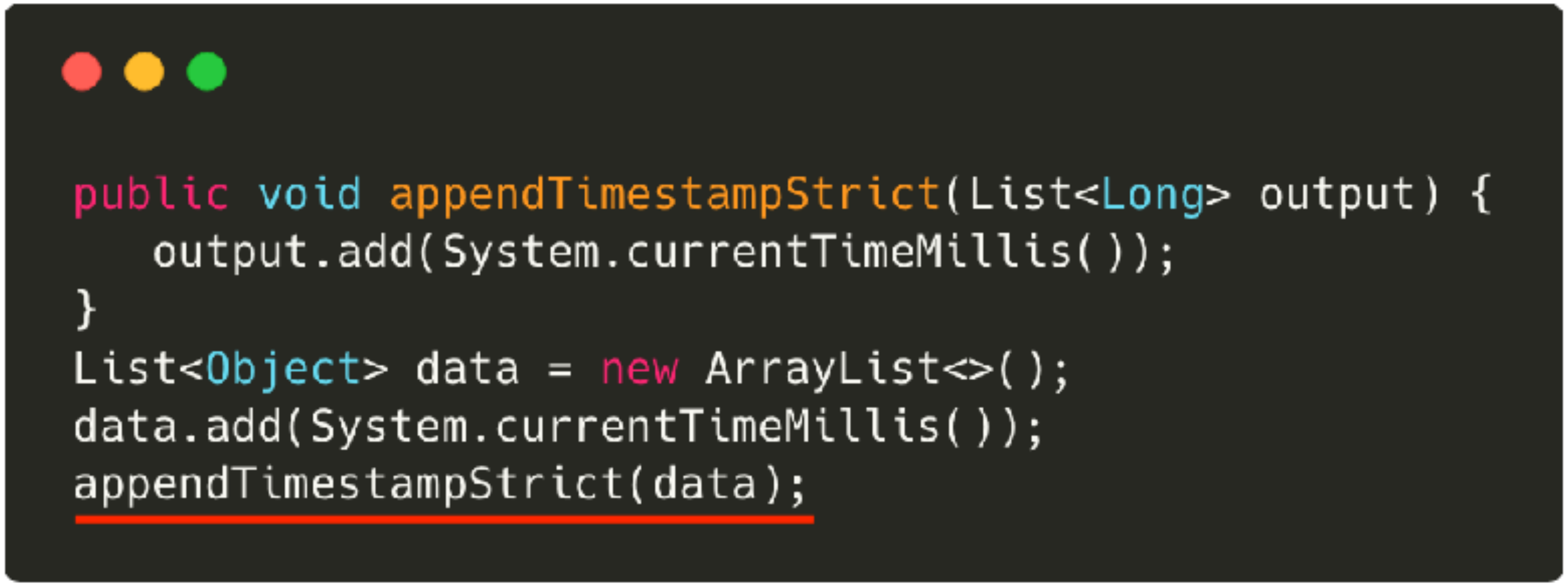
```kotlin
fun isLucky(nums: MutableList<out Number>): Boolean {
    return nums.contains(7)
}

val numList = mutableListOf<Int>()
isLucky(numList)
```

Contravariance

What Is Contravariance?

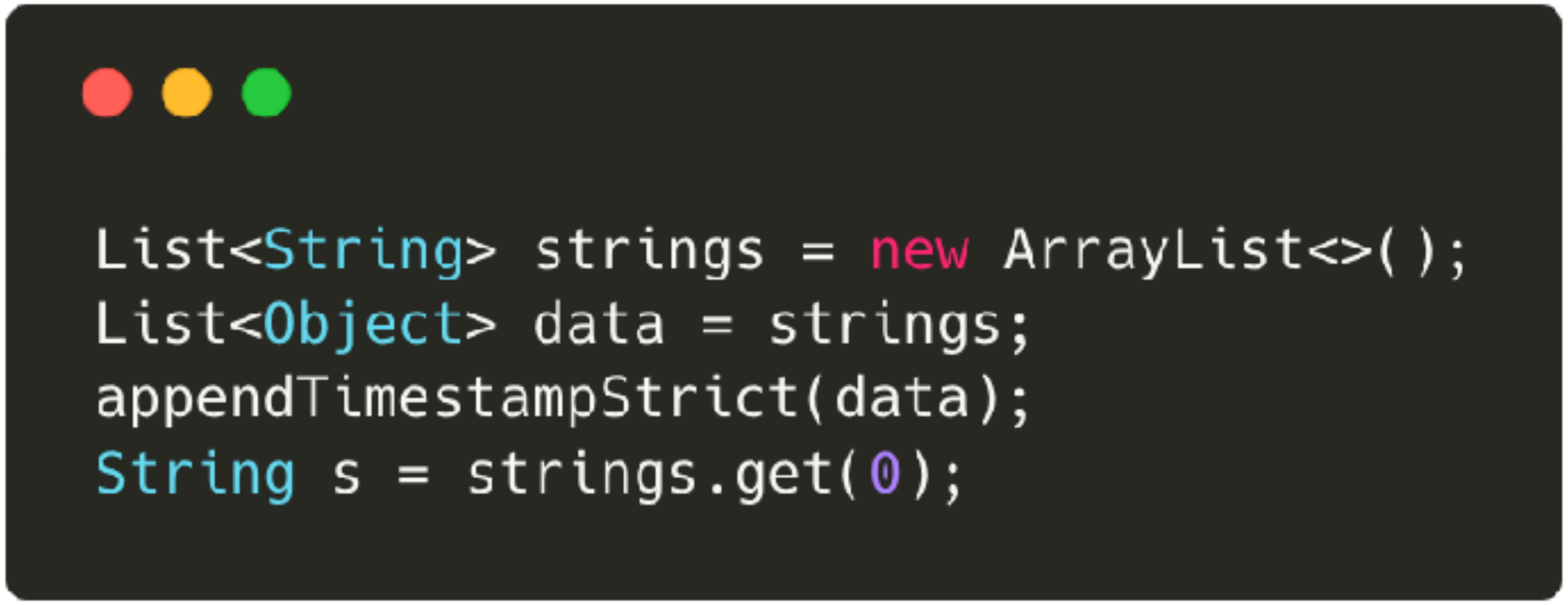
- Accepts a supertype
- Can write to the generic
- Cannot read from the generic

Java Motivation



```
public void appendTimestampStrict(List<Long> output) {
   output.add(System.currentTimeMillis());
}
List<Object> data = new ArrayList<>();
data.add(System.currentTimeMillis());
appendTimestampStrict(data);
```

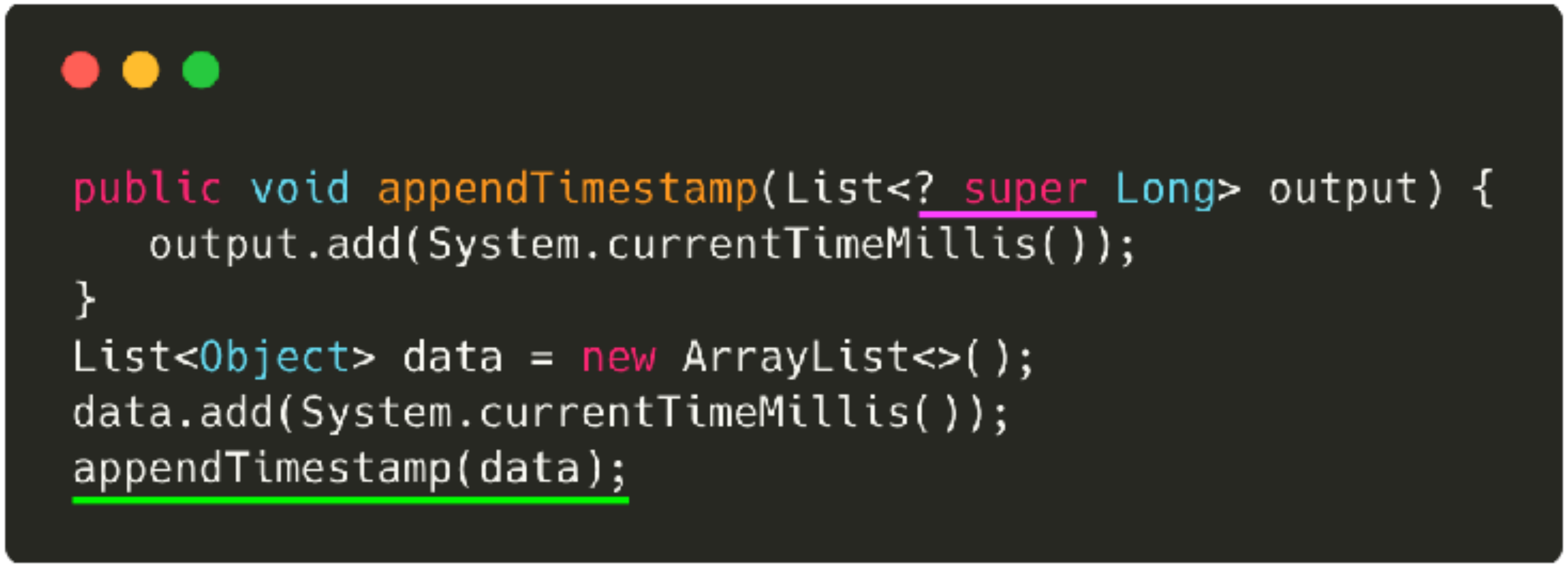
If that was allowed, we could do this:



```
List<String> strings = new ArrayList<>();
List<Object> data = strings;
appendTimestampStrict(data);
String s = strings.get(0);
```

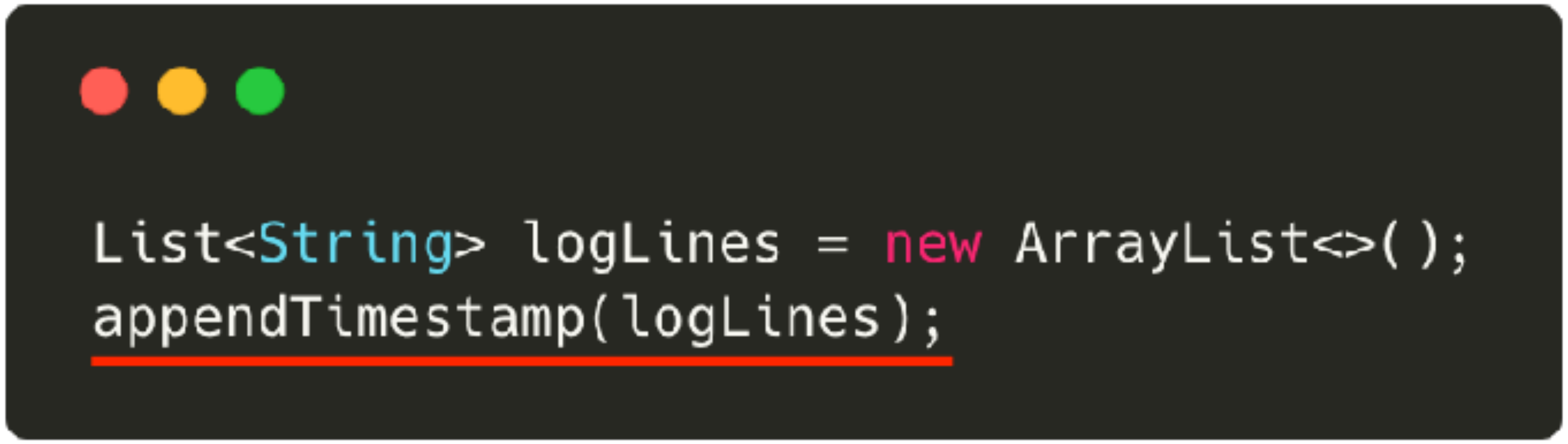
Java Contravariance

Succeeds: Object is a supertype of Long.



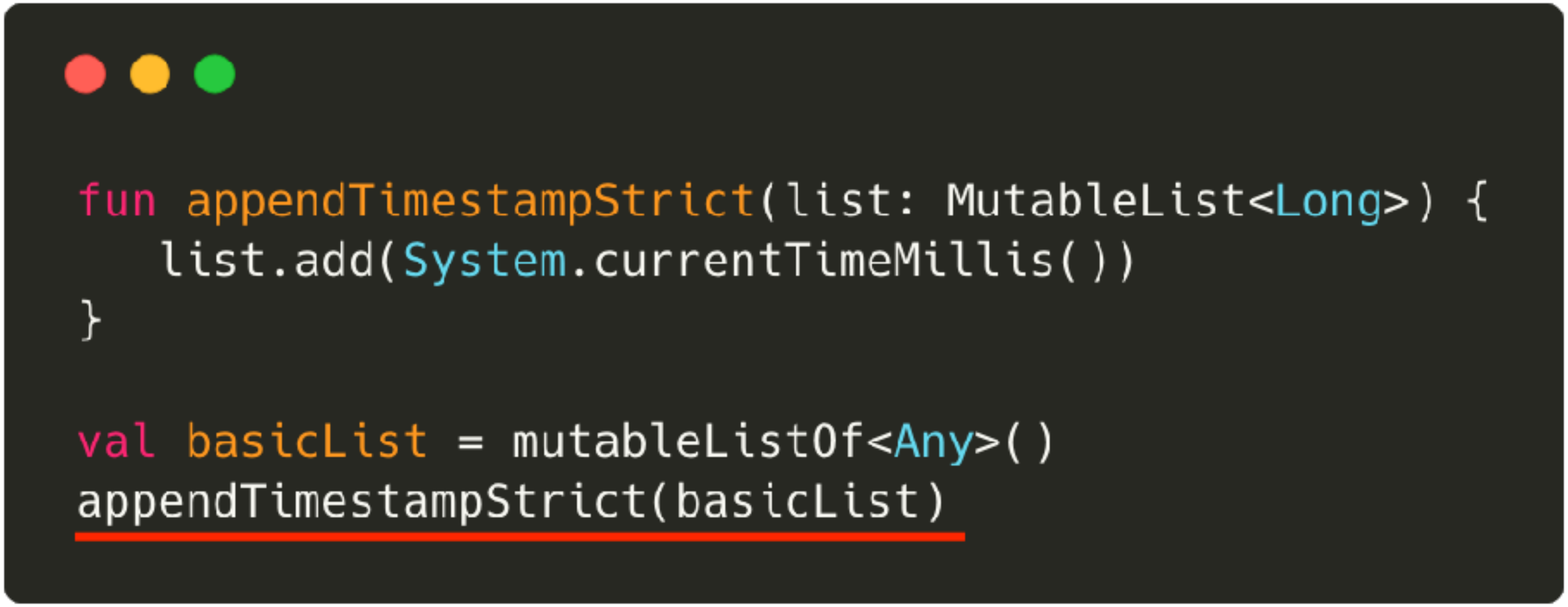
```java
public void appendTimestamp(List<? super Long> output) {
   output.add(System.currentTimeMillis());
}
List<Object> data = new ArrayList<>();
data.add(System.currentTimeMillis());
appendTimestamp(data);
```

Properly fails: String is not a supertype of Long.

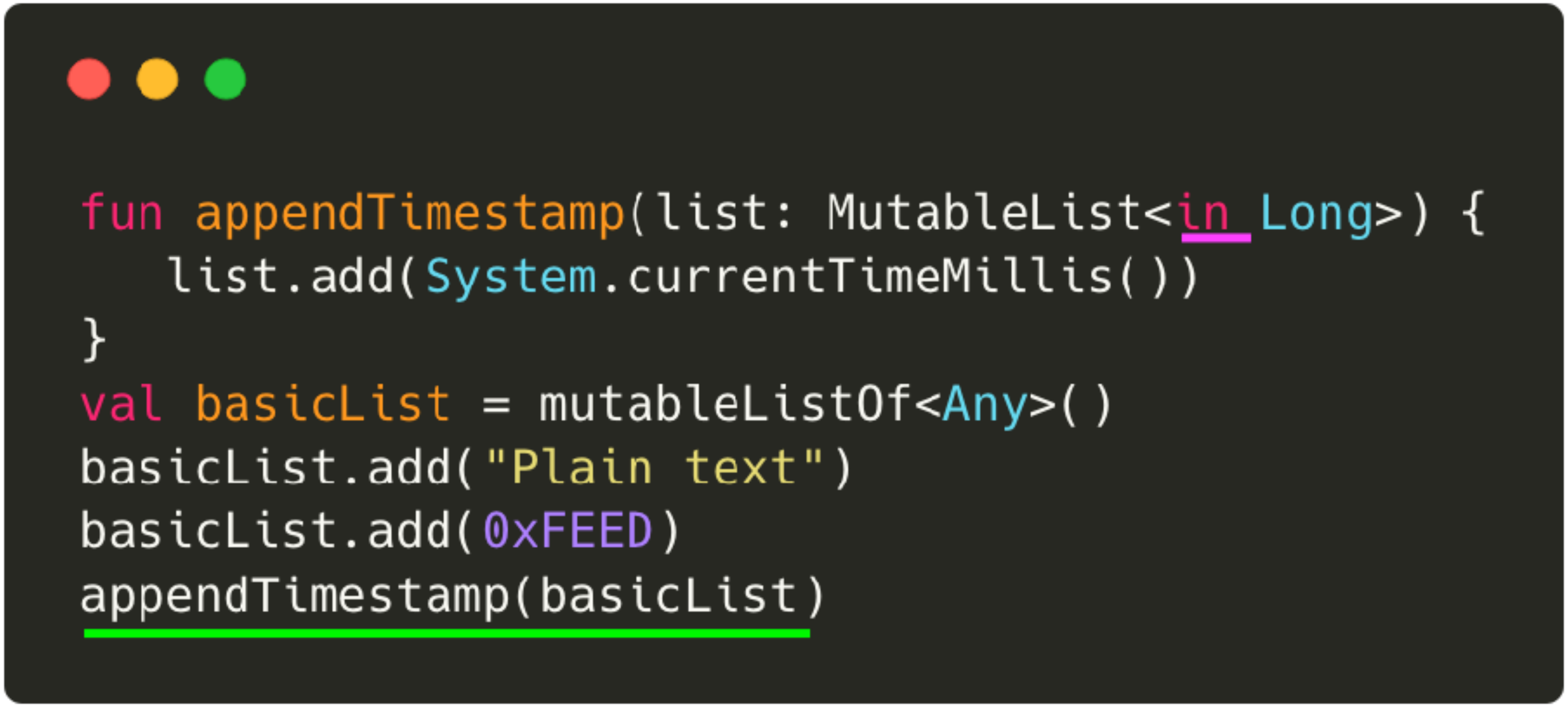


```java
List<String> logLines = new ArrayList<>();
appendTimestamp(logLines);
```

Kotlin Motivation


```kotlin
fun appendTimestampStrict(list: MutableList<Long>) {
    list.add(System.currentTimeMillis())
}

val basicList = mutableListOf<Any>()
appendTimestampStrict(basicList)
```

Kotlin Contravariance



```
fun appendTimestamp(list: MutableList<in Long>) {
    list.add(System.currentTimeMillis())
}
val basicList = mutableListOf<Any>()
basicList.add("Plain text")
basicList.add(0xFEED)
appendTimestamp(basicList)
```

@velosmobile

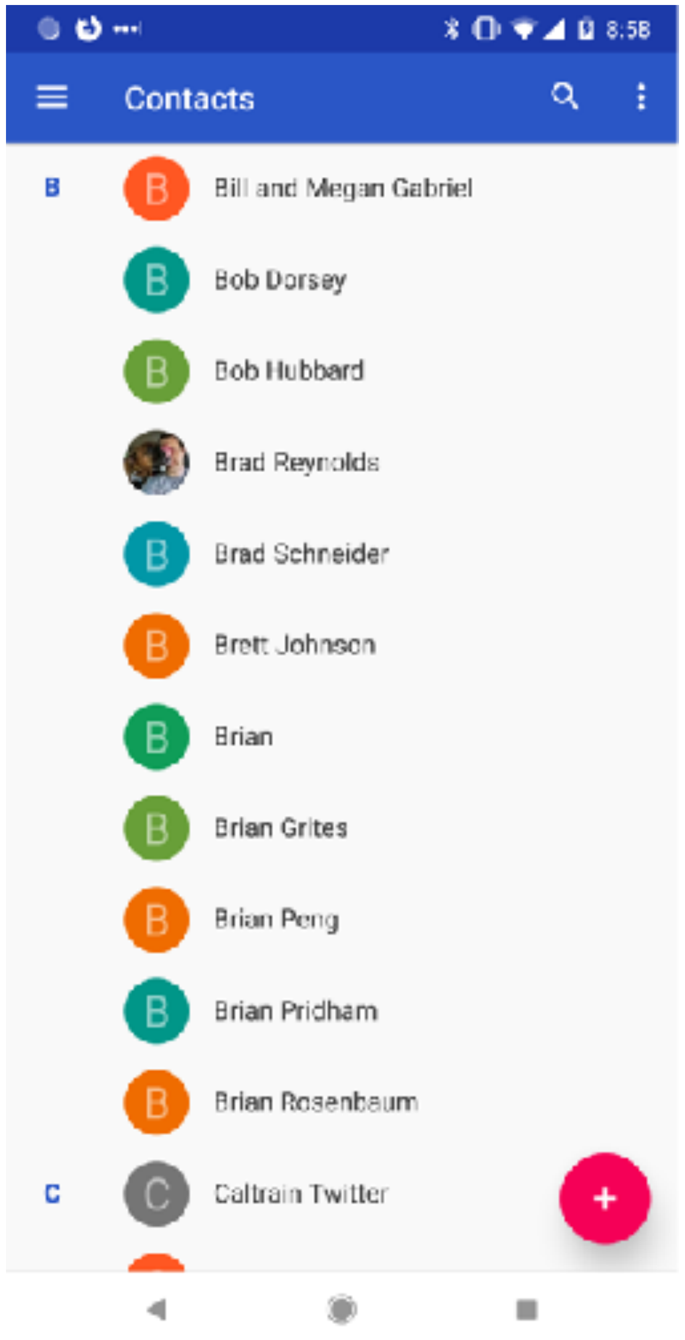
Should I Care?

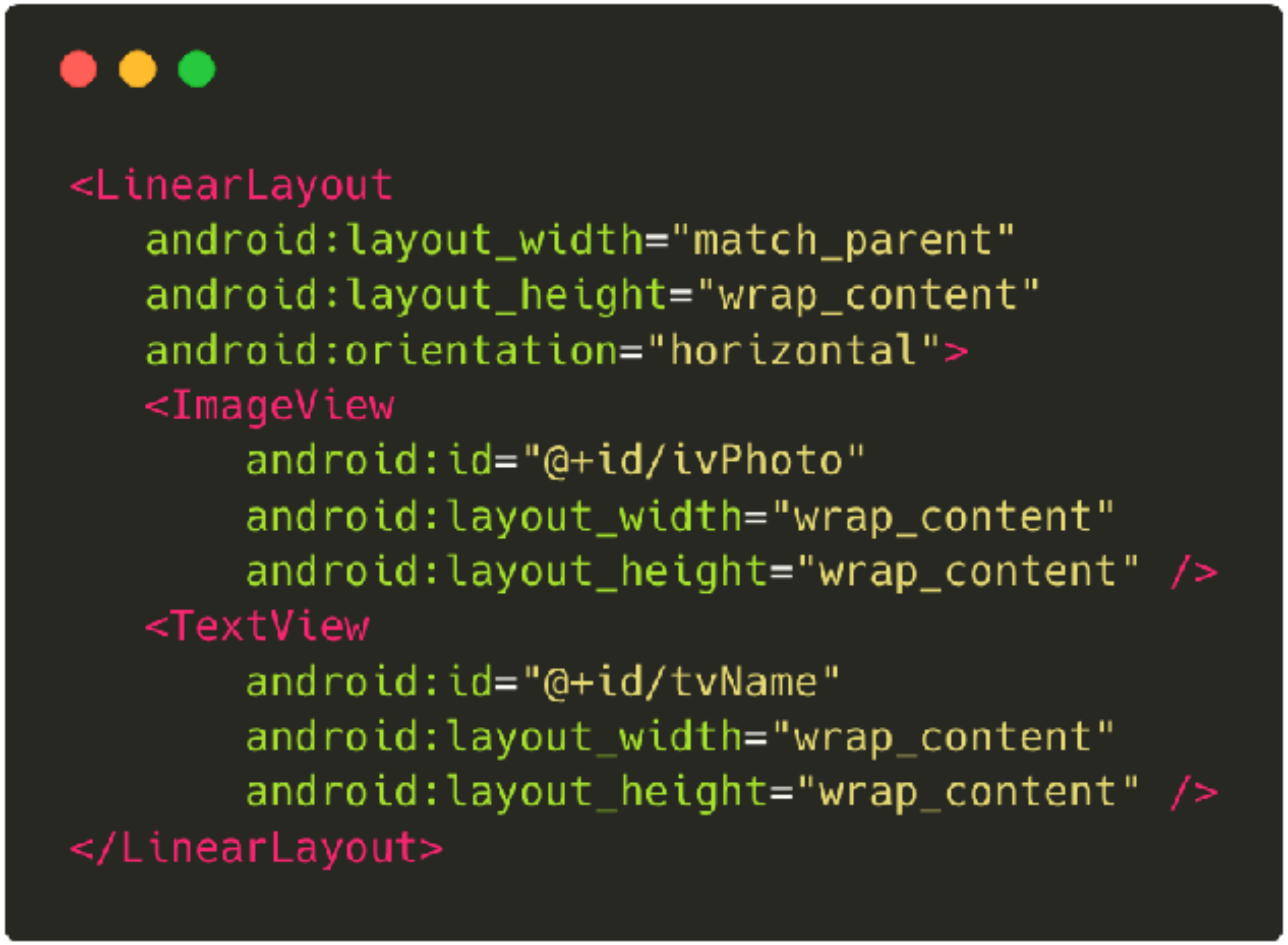
- Significant factor when creating APIs
- Especially important for collections
- Will often see for auto-translated Java code
- Handy to know
- Probably not a daily skill

Custom Views and Companion Objects

Nested Views

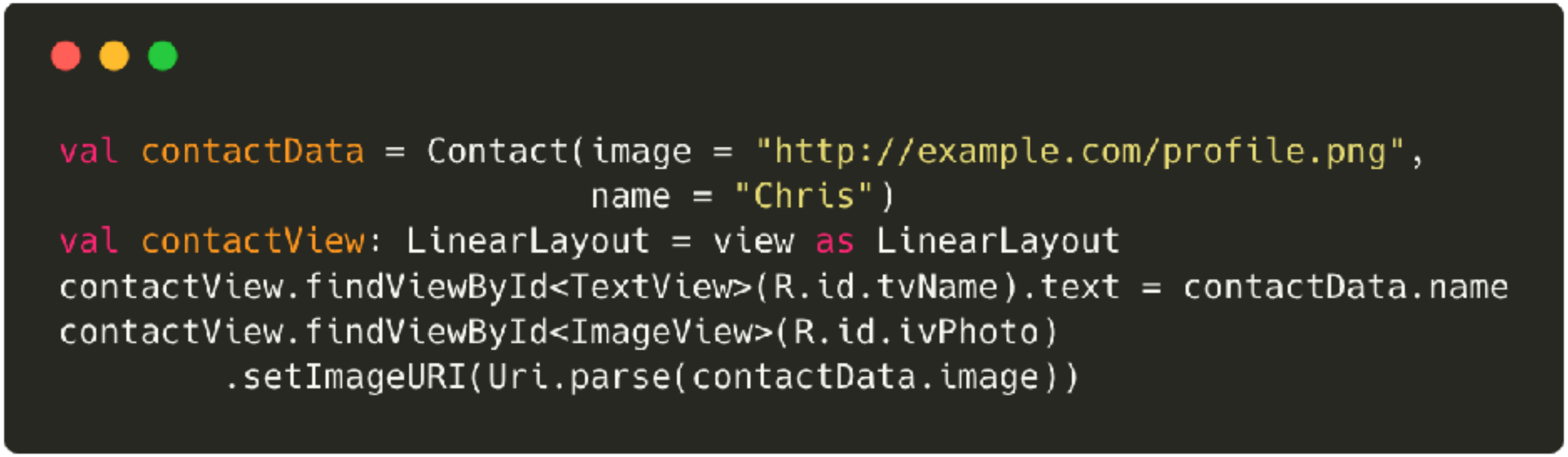
Contacts Example





```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">
    <ImageView
        android:id="@+id/ivPhoto"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
    <TextView
        android:id="@+id/tvName"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</LinearLayout>
```

Standard Configuration



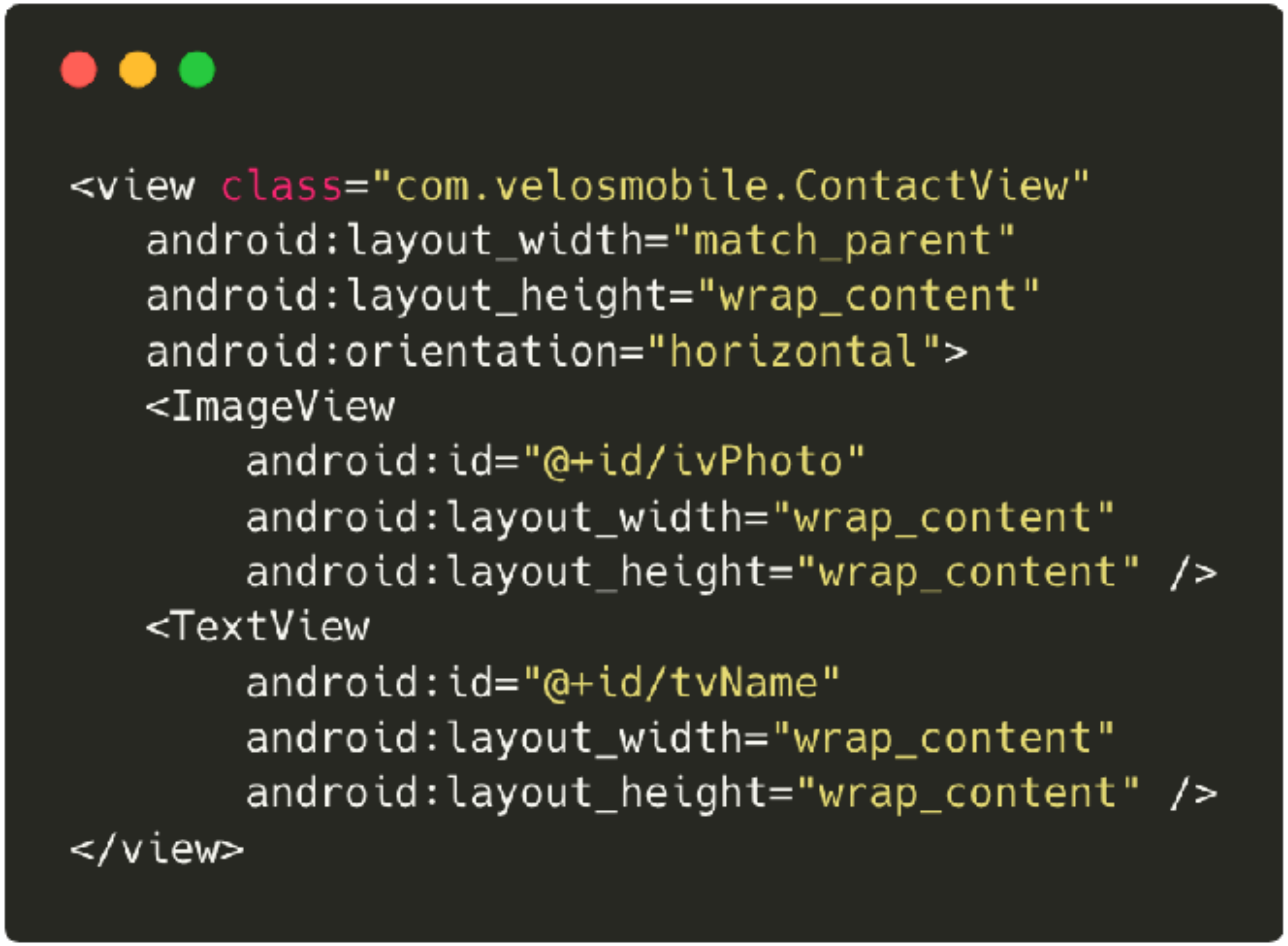
```kotlin
val contactData = Contact(image = "http://example.com/profile.png",
                          name = "Chris")
val contactView: LinearLayout = view as LinearLayout
contactView.findViewById<TextView>(R.id.tvName).text = contactData.name
contactView.findViewById<ImageView>(R.id.ivPhoto)
        .setImageURI(Uri.parse(contactData.image))
```

Standard Problems

- Leads to code bloat
- Very long Activity and Adapter files
- View logic mixed with application logic
- Grows less manageable as views become more complex

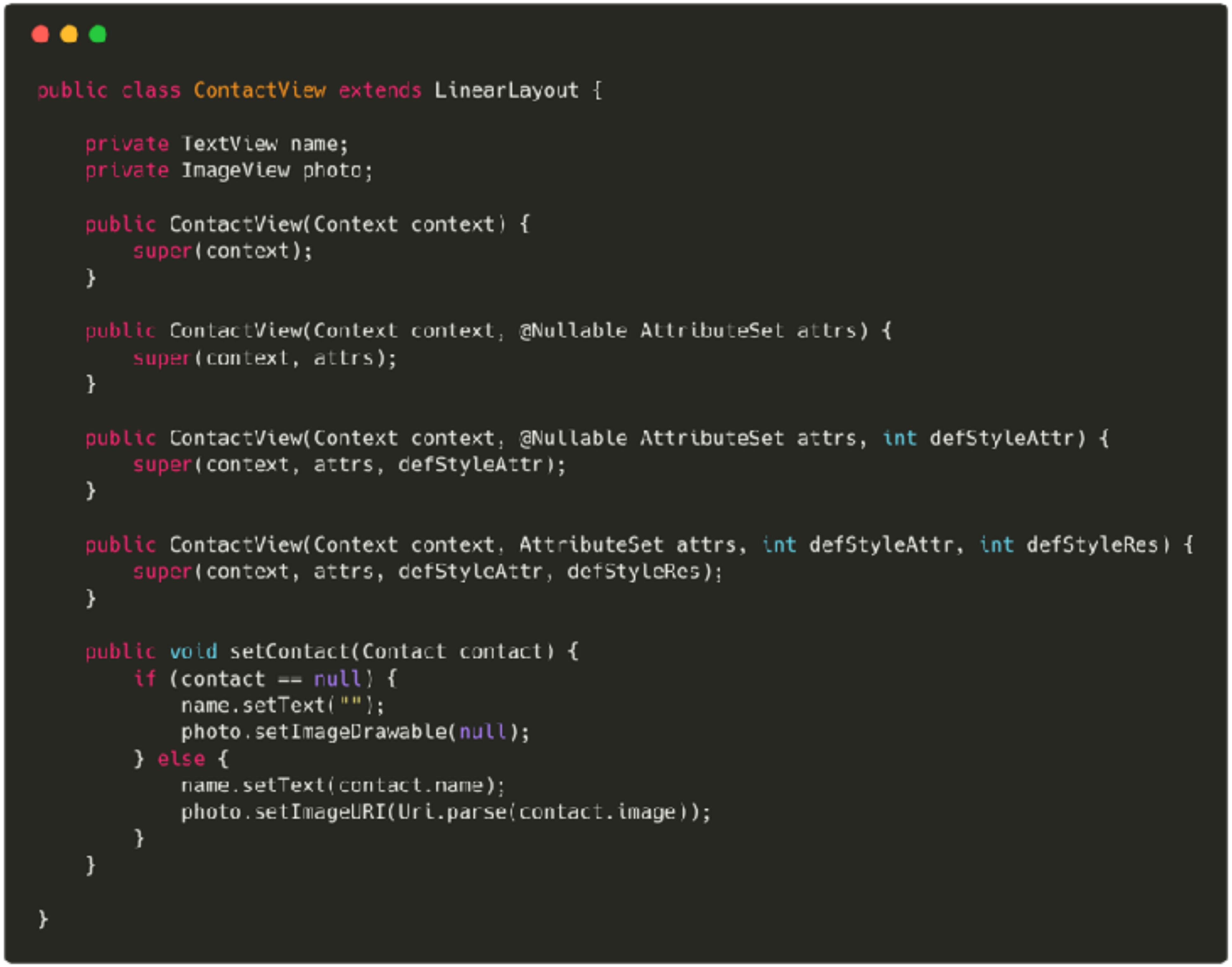
Custom Views

Custom View Layout



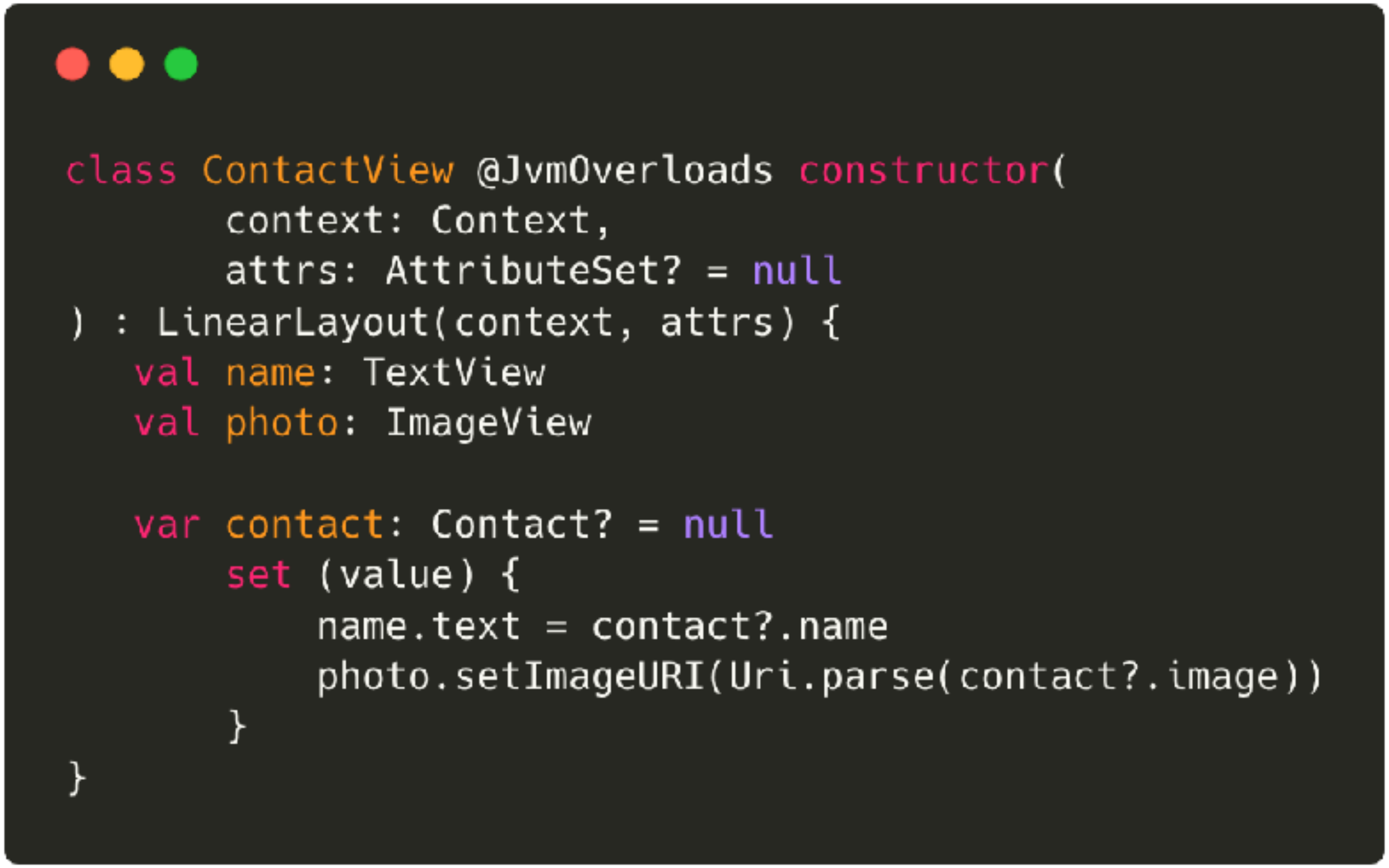
```
<view class="com.velosmobile.ContactView"
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:orientation="horizontal">
   <ImageView
       android:id="@+id/ivPhoto"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content" />
   <TextView
       android:id="@+id/tvName"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content" />
</view>
```

Java Custom View



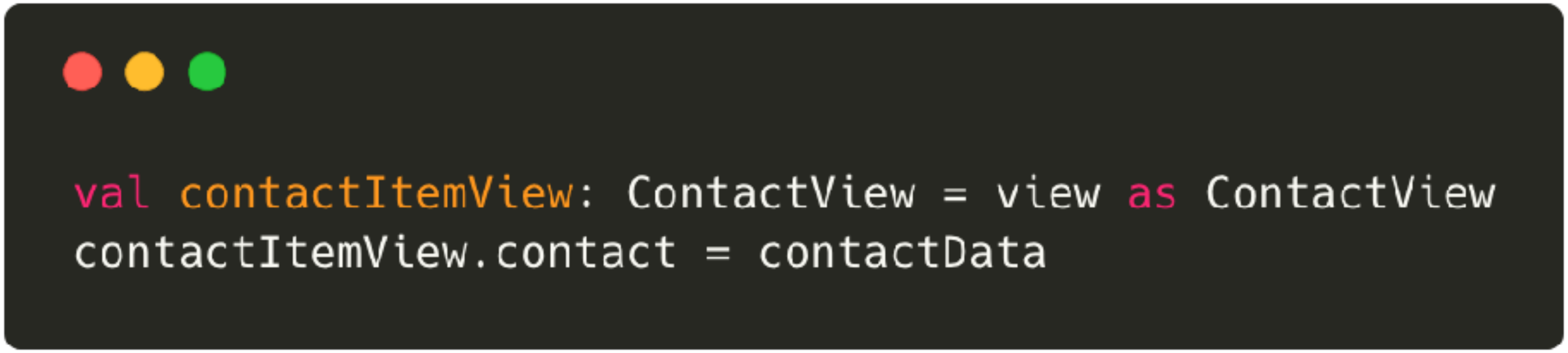
```java
public class ContactView extends LinearLayout {

    private TextView name;
    private ImageView photo;

    public ContactView(Context context) {
        super(context);
    }

    public ContactView(Context context, @Nullable AttributeSet attrs) {
        super(context, attrs);
    }

    public ContactView(Context context, @Nullable AttributeSet attrs, int defStyleAttr) {
        super(context, attrs, defStyleAttr);
    }

    public ContactView(Context context, AttributeSet attrs, int defStyleAttr, int defStyleRes) {
        super(context, attrs, defStyleAttr, defStyleRes);
    }

    public void setContact(Contact contact) {
        if (contact == null) {
            name.setText("");
            photo.setImageDrawable(null);
        } else {
            name.setText(contact.name);
            photo.setImageURI(Uri.parse(contact.image));
        }
    }

}
```

Kotlin Custom View



```kotlin
class ContactView @JvmOverloads constructor(
        context: Context,
        attrs: AttributeSet? = null
) : LinearLayout(context, attrs) {
    val name: TextView
    val photo: ImageView

    var contact: Contact? = null
        set (value) {
            name.text = contact?.name
            photo.setImageURI(Uri.parse(contact?.image))
        }
}
```

Configuring Custom View



```kotlin
val contactItemView: ContactView = view as ContactView
contactItemView.contact = contactData
```

Custom View Benefits

- Separation of concerns
- 1-to-1 mapping of model to view
- Elegantly supports collections and composition
- Classes are smaller and more focused

Advanced Custom Views

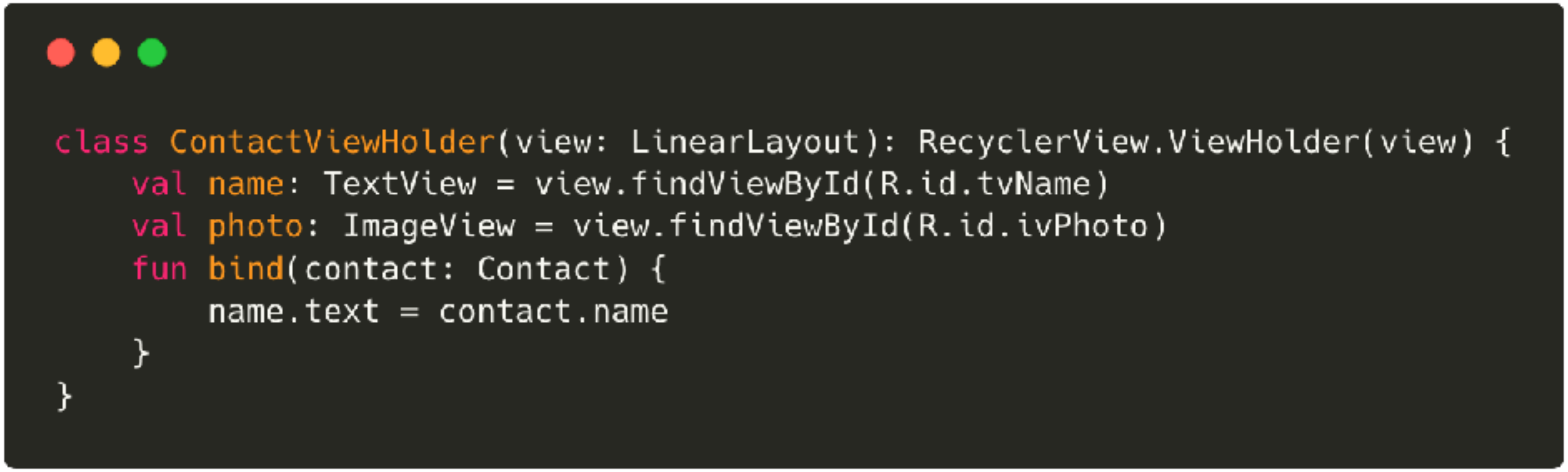
- New functionality
- Accessing touch events
- Unique layouts
- Highly customized drawing

ViewHolder

Motivation

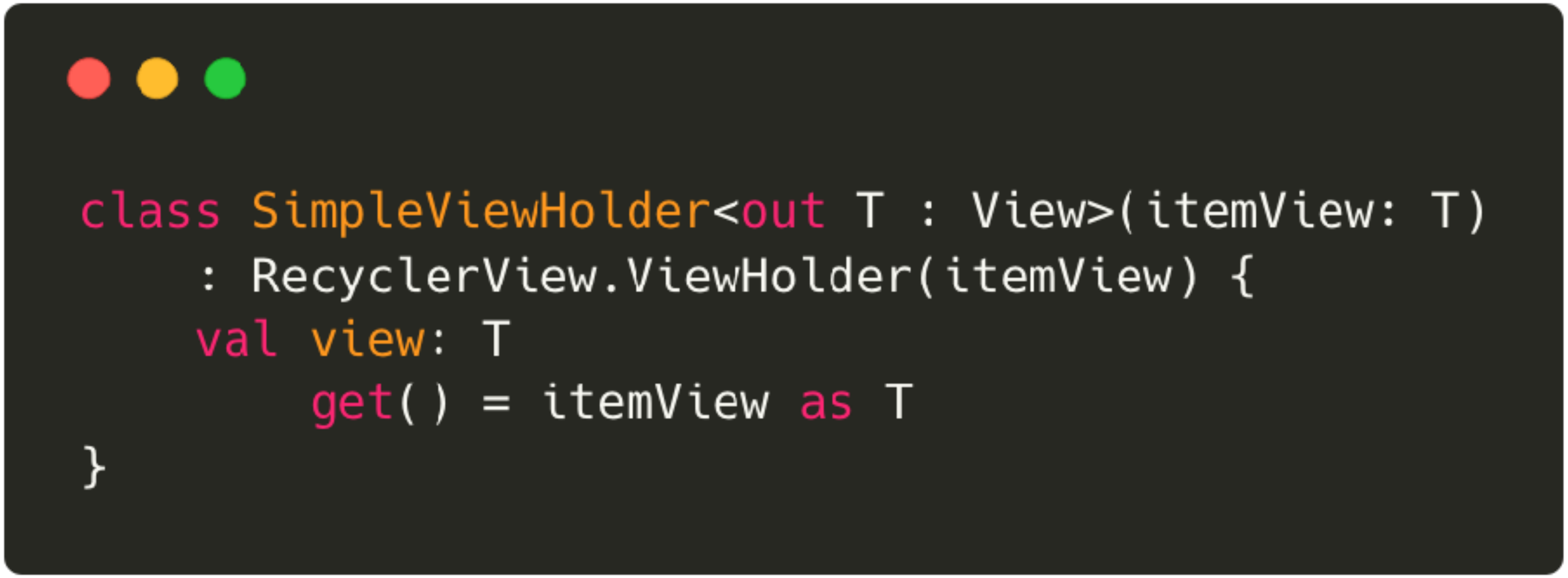
- RecyclerView needs to configure views many times as the user scrolls.
- Finding views is fairly fast, but adds up.
- We can speed things up by finding the child views in advance.

Naive Implementation



```
class ContactViewHolder(view: LinearLayout): RecyclerView.ViewHolder(view) {
    val name: TextView = view.findViewById(R.id.tvName)
    val photo: ImageView = view.findViewById(R.id.ivPhoto)
    fun bind(contact: Contact) {
        name.text = contact.name
    }
}
```

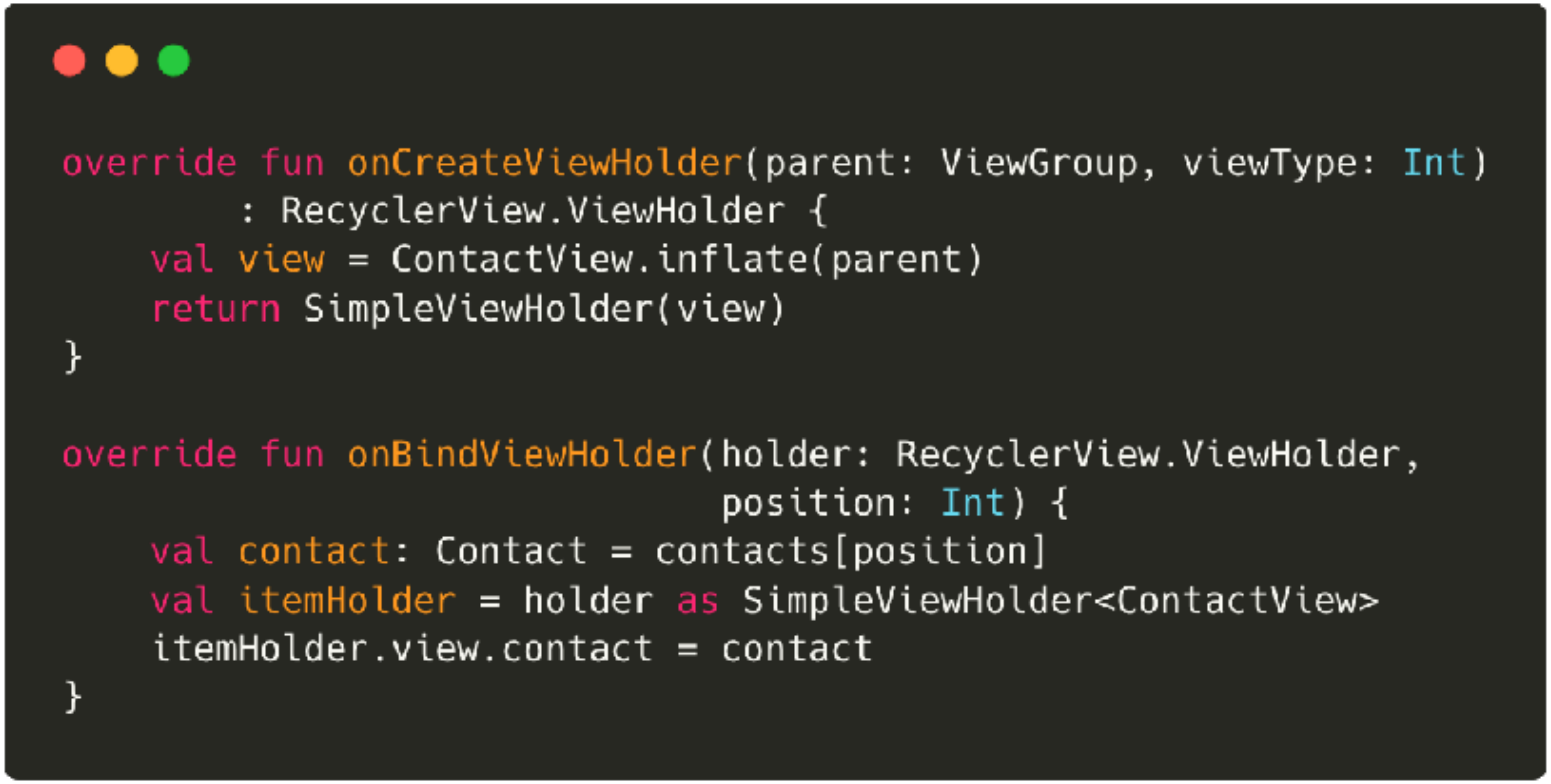
Using Generics



```
class SimpleViewHolder<out T : View>(itemView: T)
    : RecyclerView.ViewHolder(itemView) {
    val view: T
        get() = itemView as T
}
```

- Declares type of view linked to this ViewHolder
- Do not need to create a new ViewHolder for each view type

SimpleViewHolder in Action

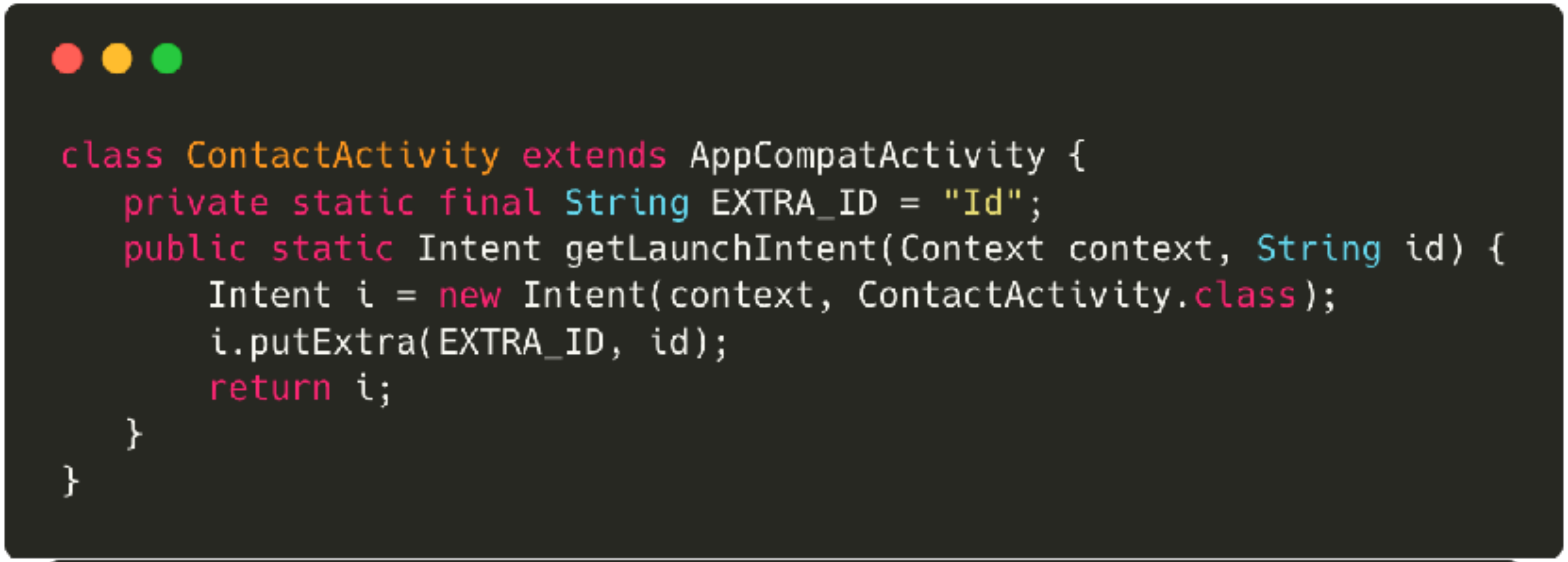


```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
        : RecyclerView.ViewHolder {
    val view = ContactView.inflate(parent)
    return SimpleViewHolder(view)
}

override fun onBindViewHolder(holder: RecyclerView.ViewHolder,
                              position: Int) {
    val contact: Contact = contacts[position]
    val itemHolder = holder as SimpleViewHolder<ContactView>
    itemHolder.view.contact = contact
}
```

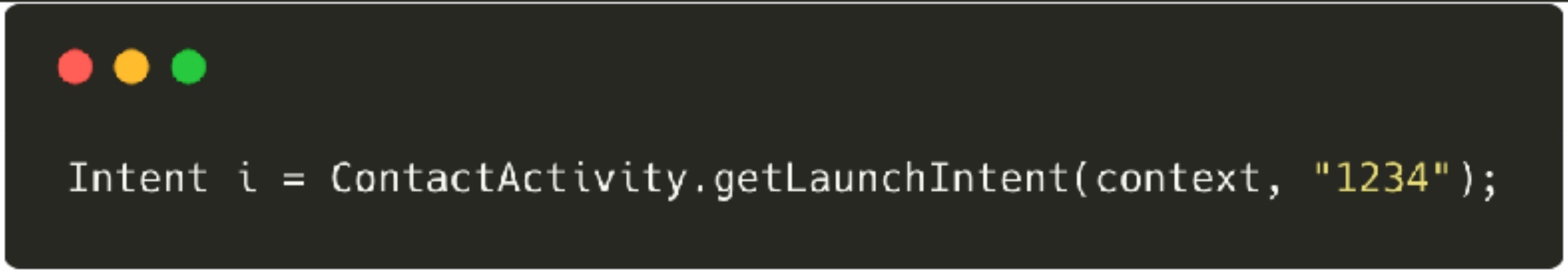
- Can access elements through the custom view
- Adapters no longer need to know details of item views

Companion Objects

Java Statics



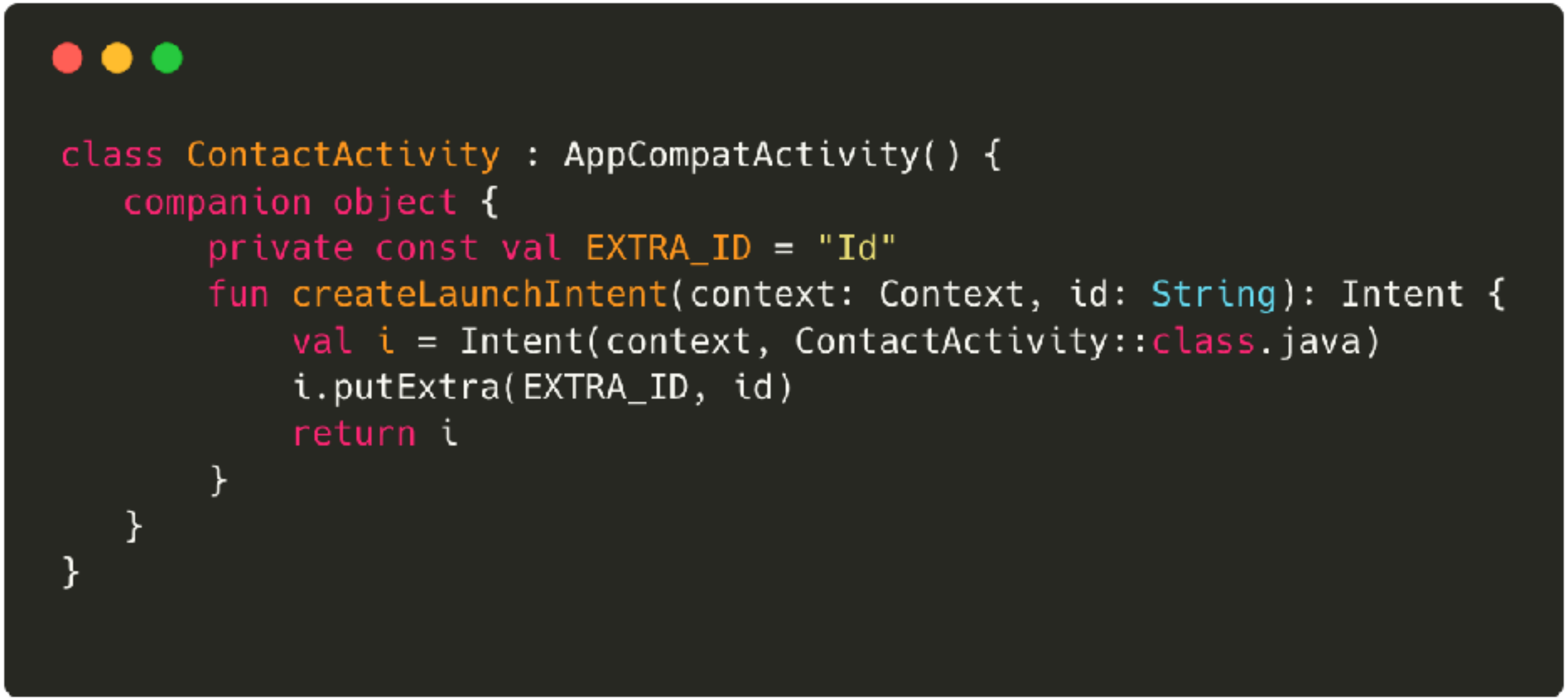
```java
class ContactActivity extends AppCompatActivity {
   private static final String EXTRA_ID = "Id";
   public static Intent getLaunchIntent(Context context, String id) {
       Intent i = new Intent(context, ContactActivity.class);
       i.putExtra(EXTRA_ID, id);
       return i;
   }
}
```



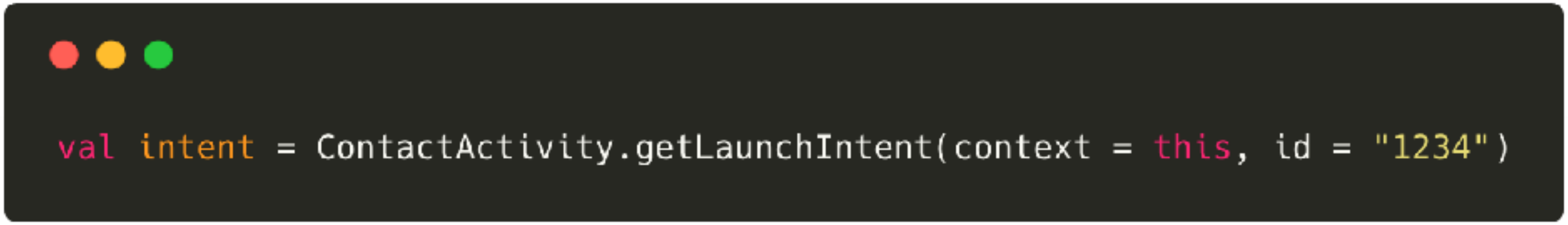
```java
Intent i = ContactActivity.getLaunchIntent(context, "1234");
```

- Accessible through class, not object
- Only one instance for all objects

Kotlin Companion Object



```kotlin
class ContactActivity : AppCompatActivity() {
    companion object {
        private const val EXTRA_ID = "Id"
        fun createLaunchIntent(context: Context, id: String): Intent {
            val i = Intent(context, ContactActivity::class.java)
            i.putExtra(EXTRA_ID, id)
            return i
        }
    }
}
```



```kotlin
val intent = ContactActivity.getLaunchIntent(context = this, id = "1234")
```

@velosmobile

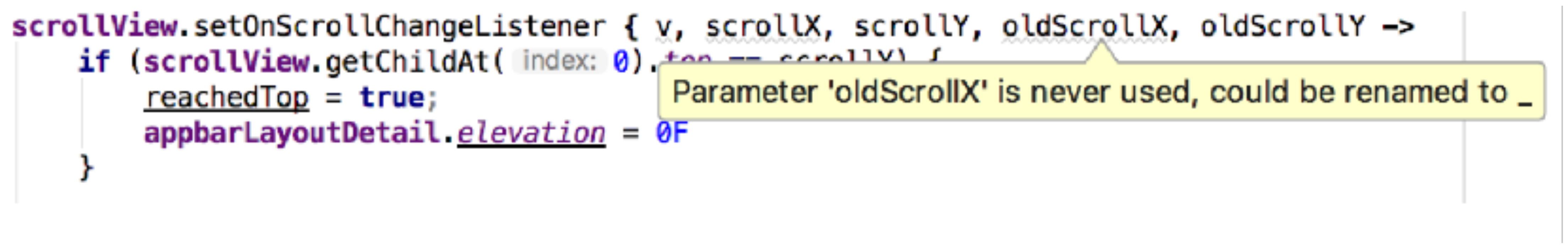
Best Practices

Introducing Kotlin

- Try to make new projects 100% Kotlin
- Consider migrating existing apps to Kotlin
    - Prefer converting entire modules over individual files
    - But prefer converting individual files to nothing

Learning and Teaching

- Code review everything
    - Include Java developers on Kotlin pull requests
    - Tag a friend on solo projects
- Rewrite classes as you learn more
- Try to eliminate all gray underlines in Android Studio.



```
scrollView.setOnScrollChangeListener { v, scrollX, scrollY, oldScrollX, oldScrollY ->
    if (scrollView.getChildAt( index: 0).top == scrollY) {
        reachedTop = true;
        appbarLayoutDetail.elevation = 0F
    }
```

Parameter 'oldScrollX' is never used, could be renamed to _

Reliable Code

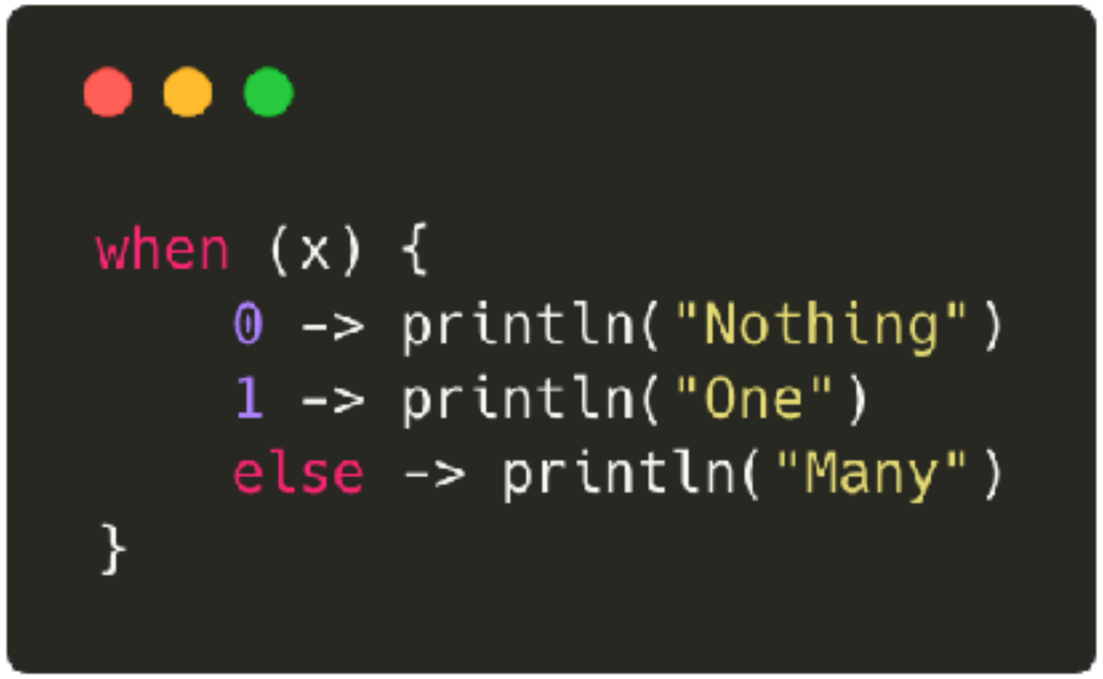
- Invest time up front to determine nullability. Study API documents and talk with server developers.
- Prefer using val to var. Reserve var for things that will change multiple times.
- Prefer List to MutableList. MutableList can often be replaced with functional operations like filter and map.
- Prefer enums to constant integers.

Structuring Code

- Prefer extension functions to utility functions.
- Group extension functions by the class they extend.
- Thoughtfully use top-level declarations. Consider namespaces.
- Follow Java structure in mixed code environments.

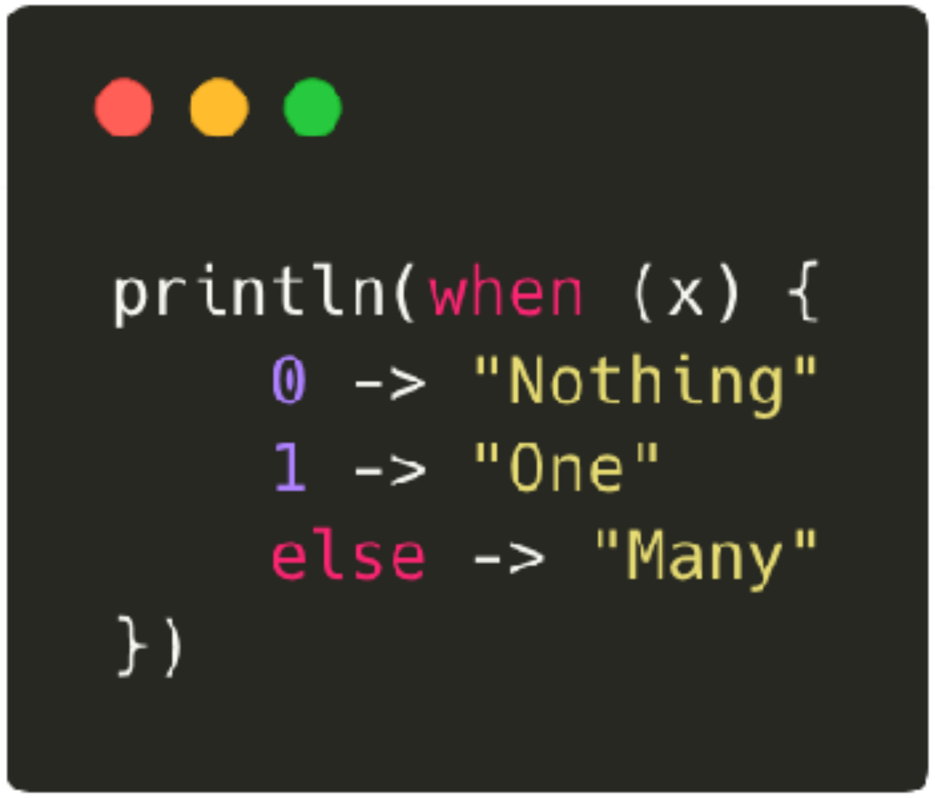
Expressive Code

Within reason, remove duplicated code.



```
when (x) {
    0 -> println("Nothing")
    1 -> println("One")
    else -> println("Many")
}
```

Rewrite and consolidate.



```
println(when (x) {
    0 -> "Nothing"
    1 -> "One"
    else -> "Many"
})
```

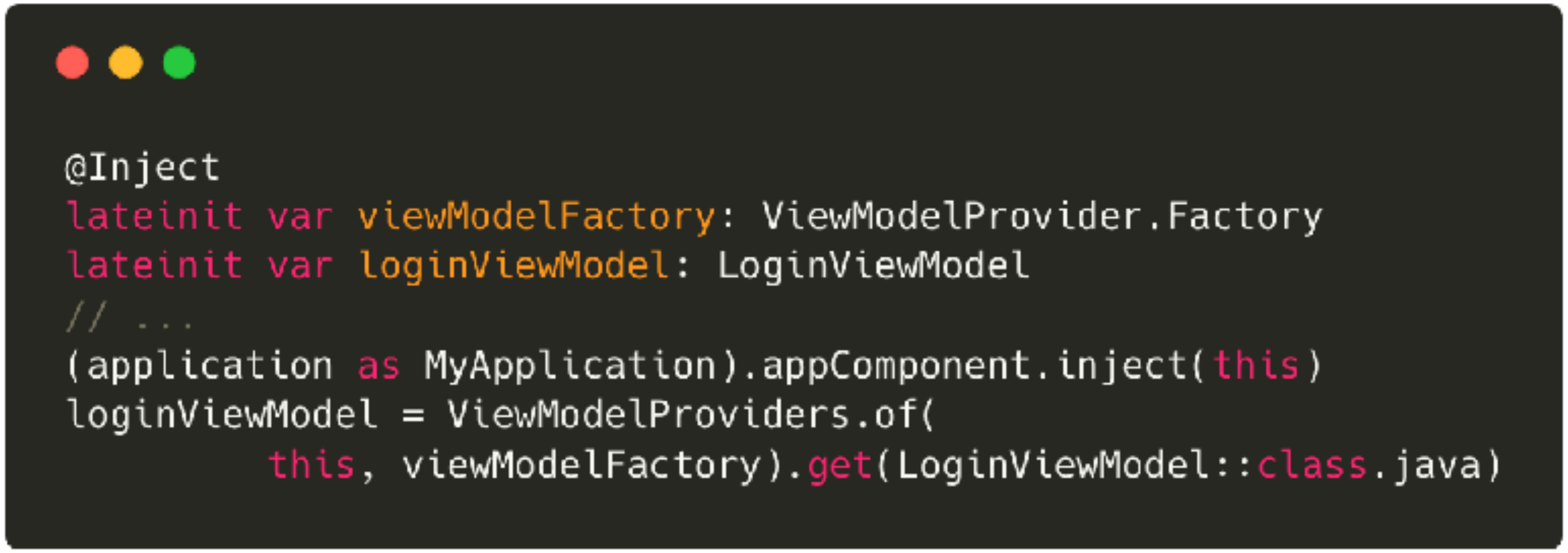
Code Style

- Expressive, terse, but not confusing.
- Keep semantic density high while remaining readable.
- Use Android Studio auto-formatting (Cmd + Opt + L).
  - Apply default style or set one for your company.

Our Favorite Libraries

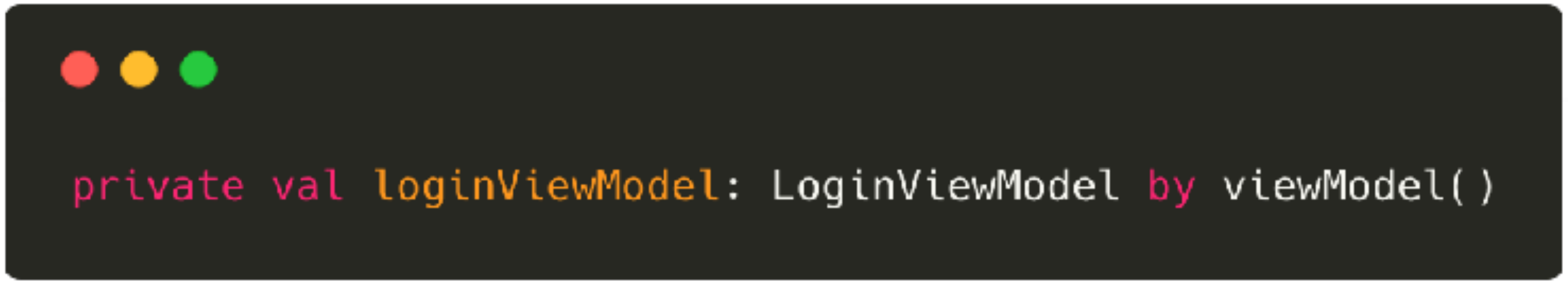
Dependency Injection: Koin

Before



```
@Inject
lateinit var viewModelFactory: ViewModelProvider.Factory
lateinit var loginViewModel: LoginViewModel
// ...
(application as MyApplication).appComponent.inject(this)
loginViewModel = ViewModelProviders.of(
         this, viewModelFactory).get(LoginViewModel::class.java)
```

After



```
private val loginViewModel: LoginViewModel by viewModel()
```

https://github.com/Ekito/koin

Other Libraries

- View binding with Kotterknife: https://github.com/JakeWharton/kotterknife
- Android extensions with KTX (from Jetpack): https://github.com/android/android-ktx
- Reactive programming with RxKotlin: https://github.com/ReactiveX/RxKotlin

Recap: Day 2

- Generics: Variance
- Custom Views
- Companion Objects
- Kotlin-Android Best Practices
- Android Tasks 2-4

Feedback

https://bit.ly/2sQmmZ5