From Kotlin to Java

Alastair F. Donaldson

Isn’t it meant to be the other way around?!

Yes! Kotlin is a more modern language than Java

Lots of demand for Java → Kotlin migration

- JetBrains offer strong tool support for this
- Several books on the topic

Demand for Kotlin → Java migration? Not so much

My aim: teach you some Java because **Java is important**
Kotlin → Java conversion will help illustrate the differences

Aims of this lecture

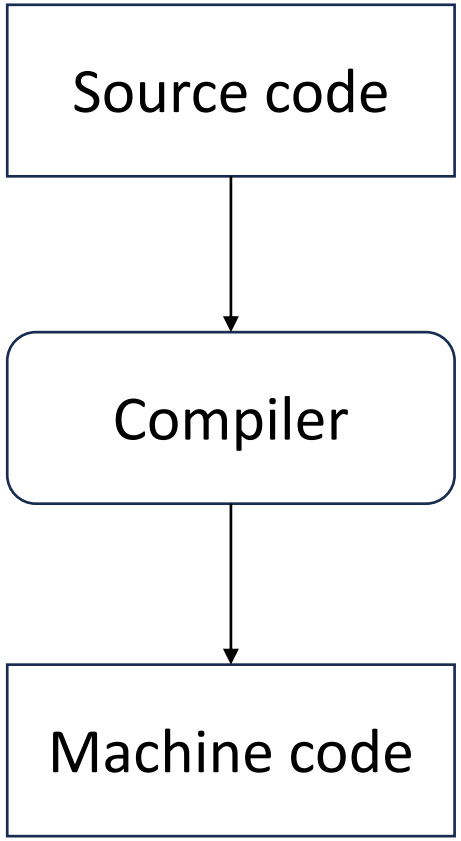
- Discuss compilation, interpretation, just-in-time compilation
- Introduce the Java Virtual Machine on which Java and Kotlin code runs
- Briefly explain garbage collection
- Explain backing fields in Kotlin
- Demo some Kotlin → Java conversion
- Introduce Java via the differences these demos expose

Coding demos

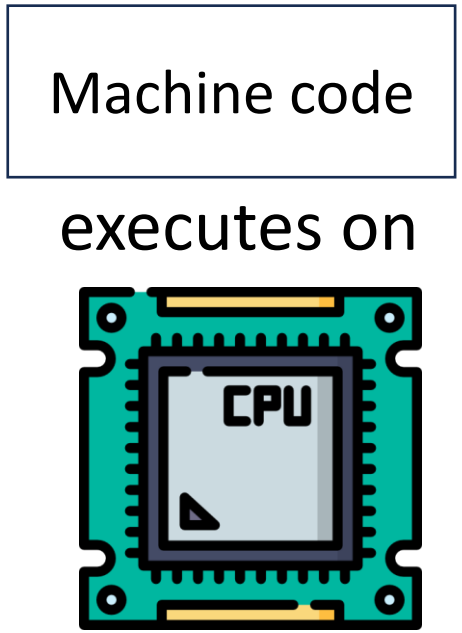
https://gitlab.doc.ic.ac.uk/afd/kotlin-2024-java-conversion-demos

Static compilation

At compile time (static):



At runtime (dynamic):



Benefits of static compilation

Direct execution of machine code at runtime is **fast**

The static compiler can perform **optimisations** to generate efficient machine code

Static compiler may be able to **identify bugs** in your code before you execute it

Drawbacks of static compilation

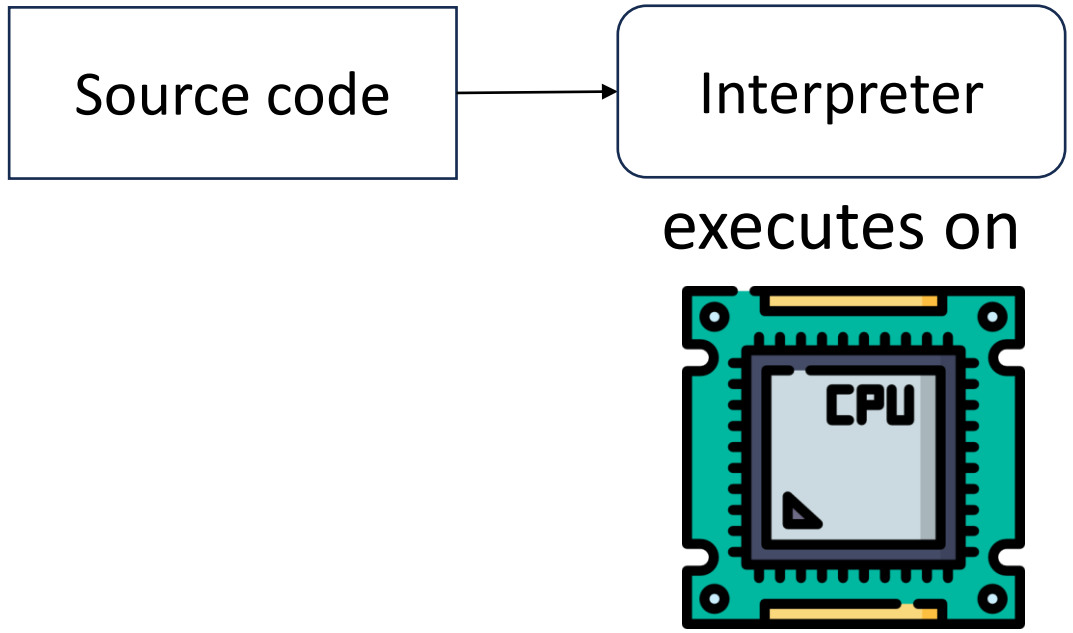
Compiling **takes time** – annoying if you need quick feedback about how your program runs

Machine code is **not portable** between CPUs with different architectures (e.g. x86 vs. ARM)

- A binary compiled for x86 cannot executed directly on ARM

Interpretation

At runtime (dynamic):



Benefits of interpretation

Quick turnaround if you need to run your program again and again, changing it between runs

- No waiting for it to compile – known as read-eval-print loop (REPL)

Portable: source code can be interpreted on different architectures if an interpreter is available for each architecture

- E.g. the same source code can be interpreted to run on x86 or ARM

Drawbacks of interpretation

Slow execution:

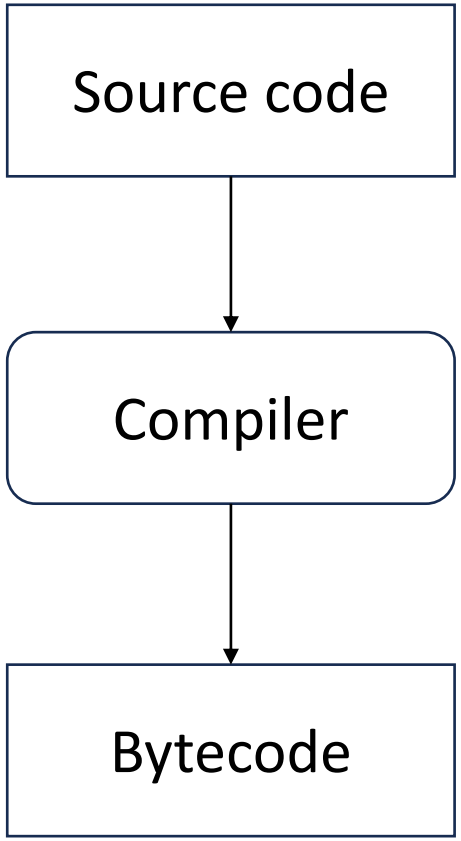
- The interpreter is a machine code program
- It can be seen as simulating the statements of the source program
- Typically **much slower** than executing a machine code version of the source program

Usually little or **no static error checking** – bug finding delayed until runtime

Not fundamental to interpretation, but interpreted languages are usually **dynamically typed**

Just-in-time compilation

At compile time (static):

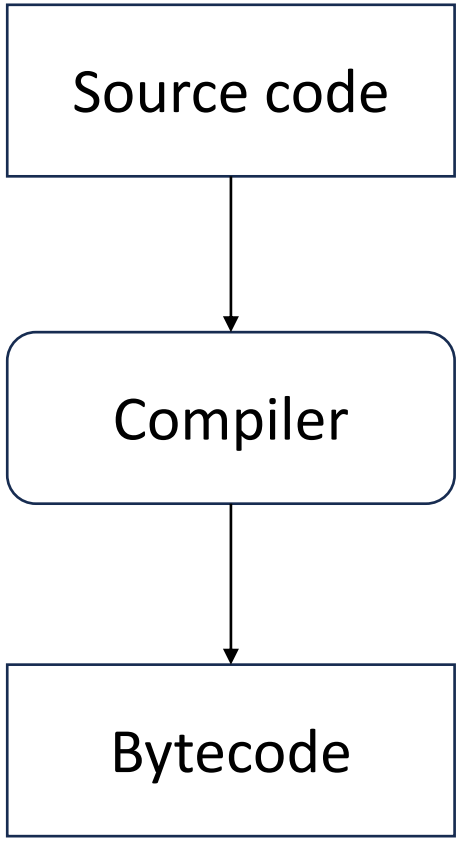


Bytecode is:

- Lower level than source code
- Higher level than machine code
- Not tied to any particular CPU

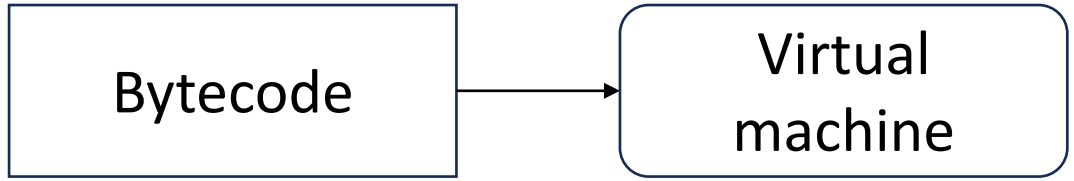
Just-in-time (JIT) compilation

At compile time (static):

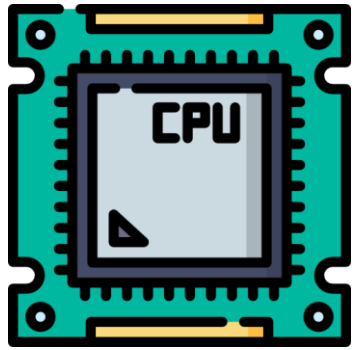


This is called an **ahead-of-time** compiler

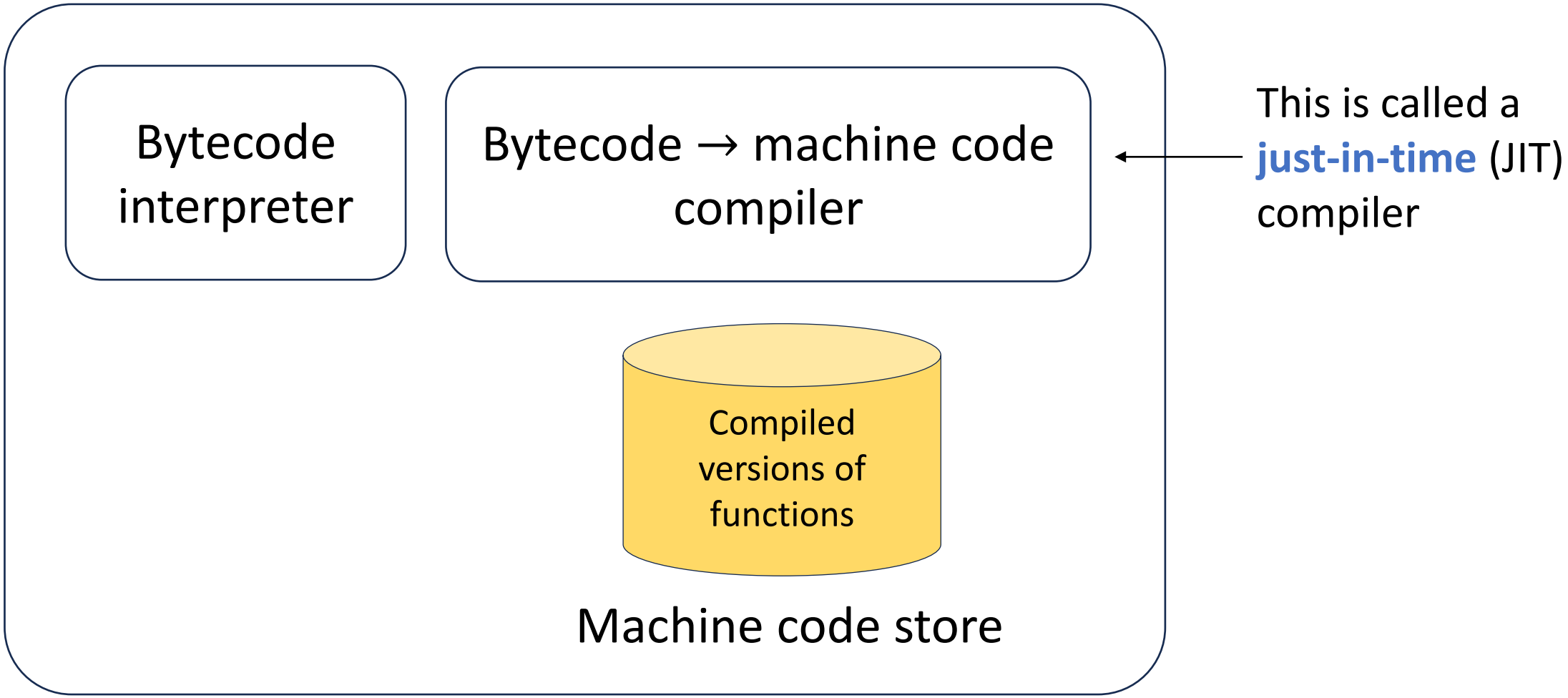
At runtime (dynamic):



executes on



Inside the virtual machine (VM)



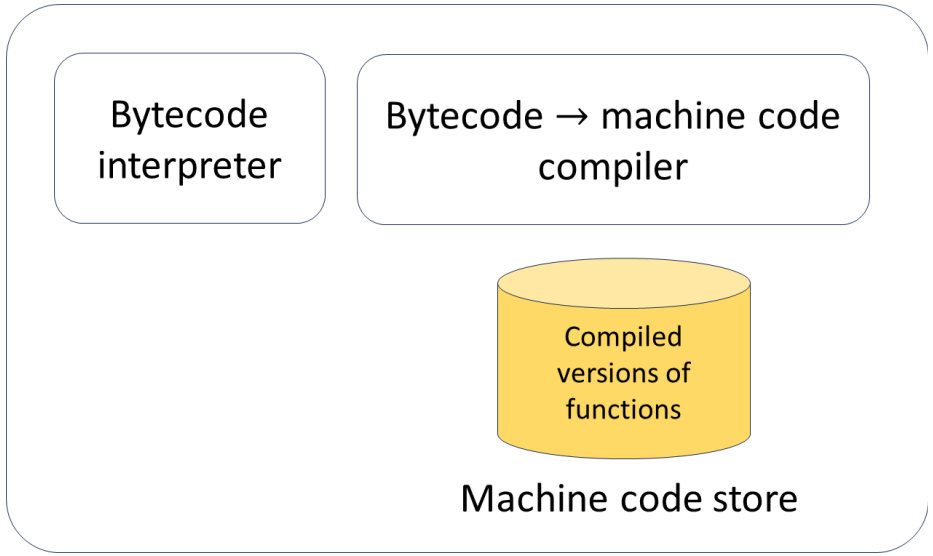
Inside the virtual machine (VM)

Initially, all bytecode is **interpreted**

At runtime, VM looks for **hotspots**: functions that are executed a lot

Hotspots are compiled to machine code in the background – called **just-in-time (JIT) compilation**

Once a function is available as machine code, it does not need to be interpreted



Benefits of JIT compilation

Portable: Bytecode can run on any platform where a VM is available

- Bytecode can run on x86 and ARM, as long as an x86 VM exists and an ARM VM exists

Lots of scope for **optimisation**:

- **Static optimisation** by the ahead-of-time compiler
- **Dynamic optimisation** by the JIT compiler, based on **profiling data** about where the running program spends its time

Performance of program gets faster as more functions become available as machine code

Drawbacks of JIT compilation

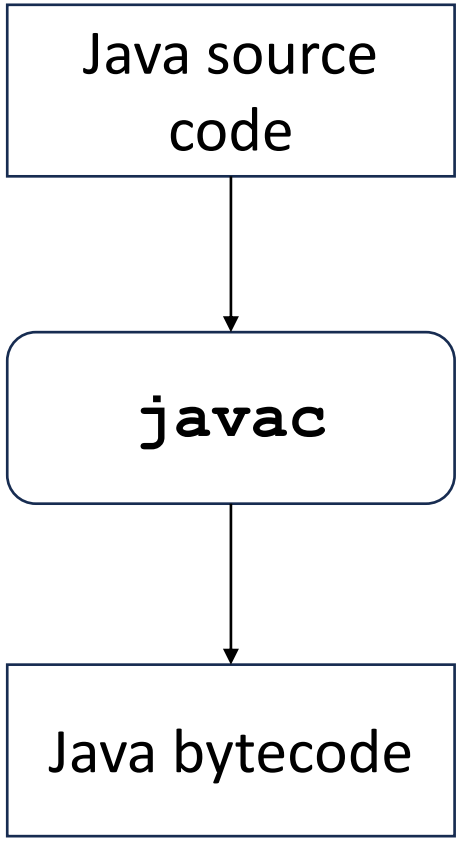
Unpredictable performance:

- Runtime is initially slow due to bytecode interpretation
- Performance improves over time
- Performance can suddenly leap due to a key function being JIT-compiled
- JIT-compiler may make bad long-term decisions based on short-term profiling information

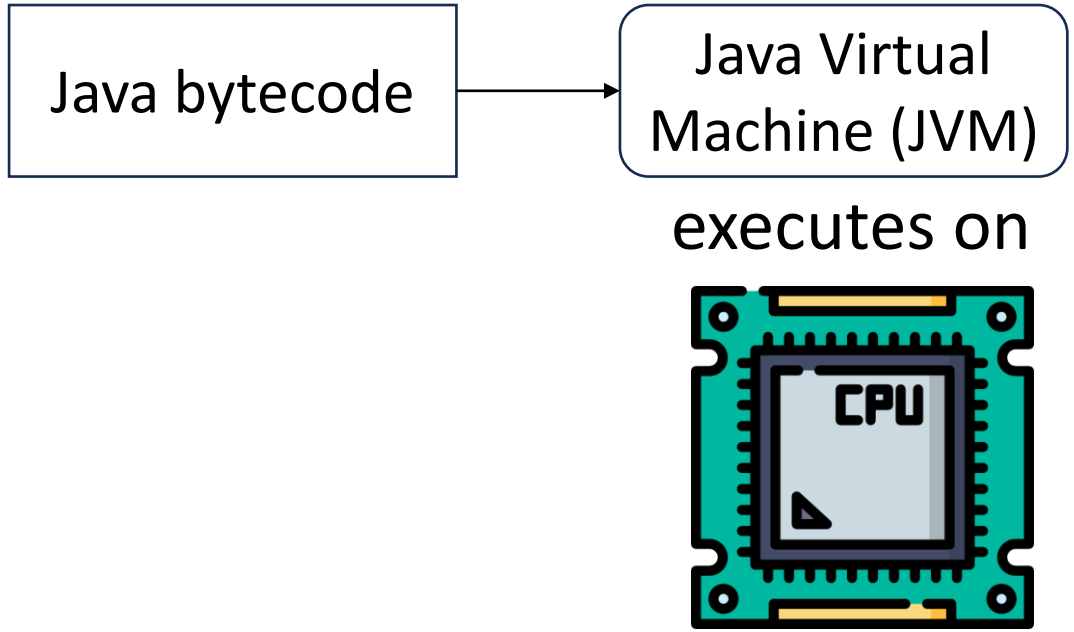
Bloat: A virtual machine is a large and complex application

Java uses just-in-time compilation

At compile time (static):

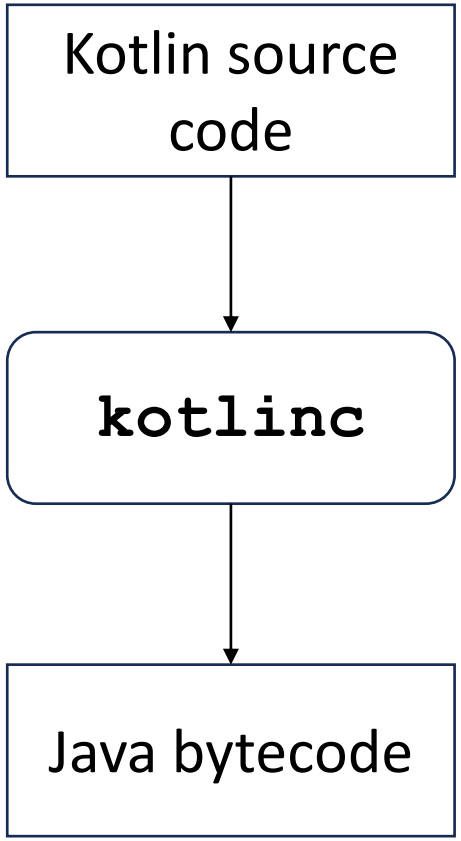


At runtime (dynamic):

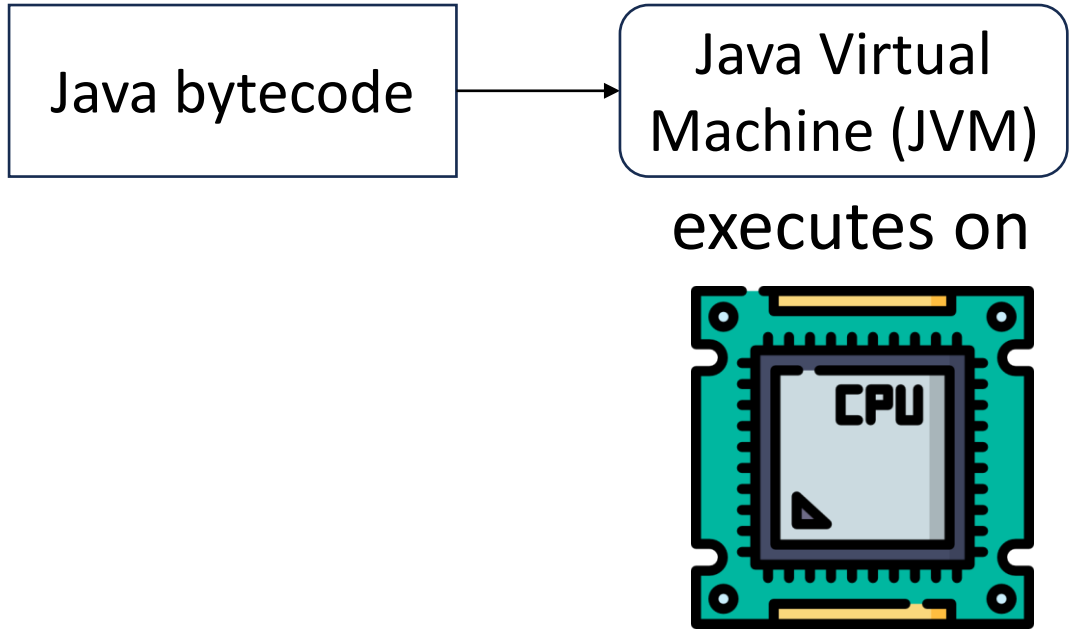


Kotlin uses just-in-time compilation

At compile time (static):

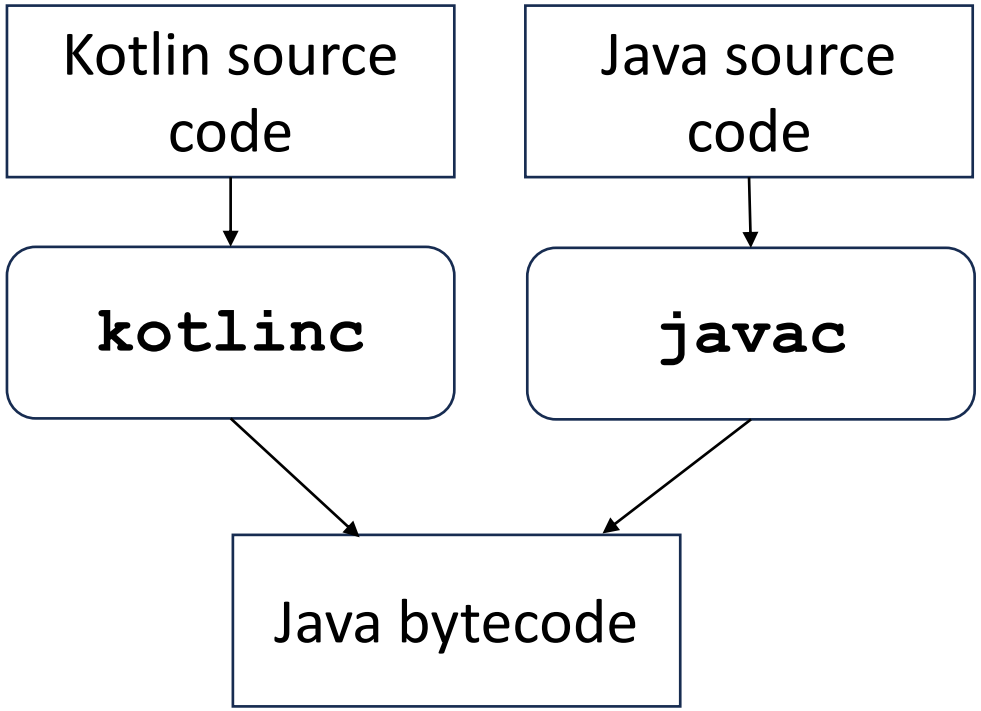


At runtime (dynamic):

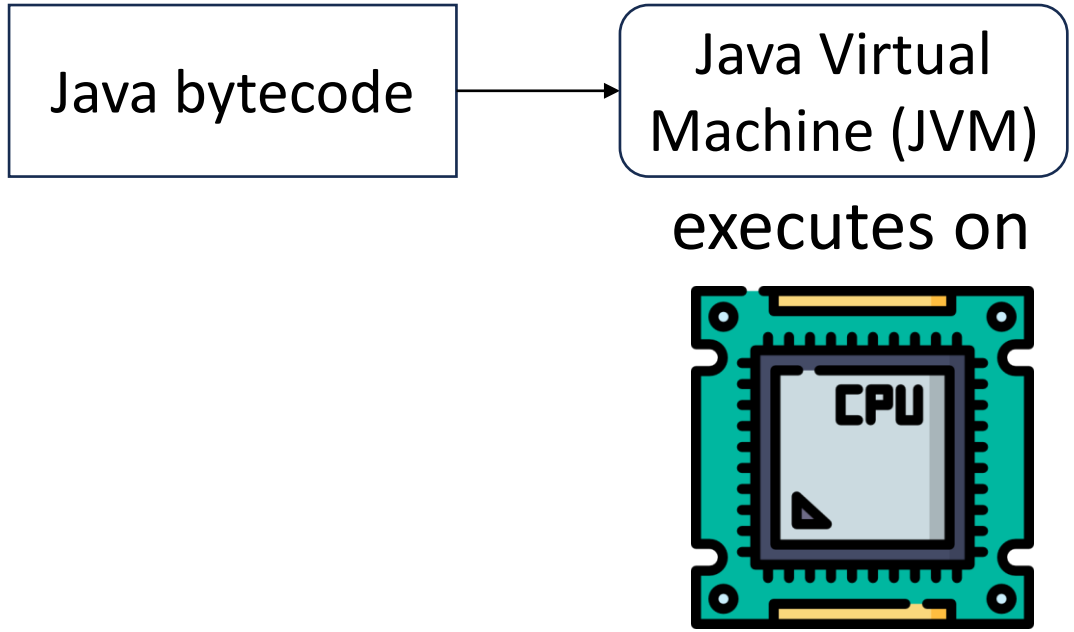


Kotlin and Java work well together

At compile time (static):



At runtime (dynamic):



Aside: garbage collection

The JVM take care of deallocating objects that can no longer be used by a program

Program **roots**: stacks of all threads + top-level properties

If an object cannot be reached from a program root, it is **garbage**

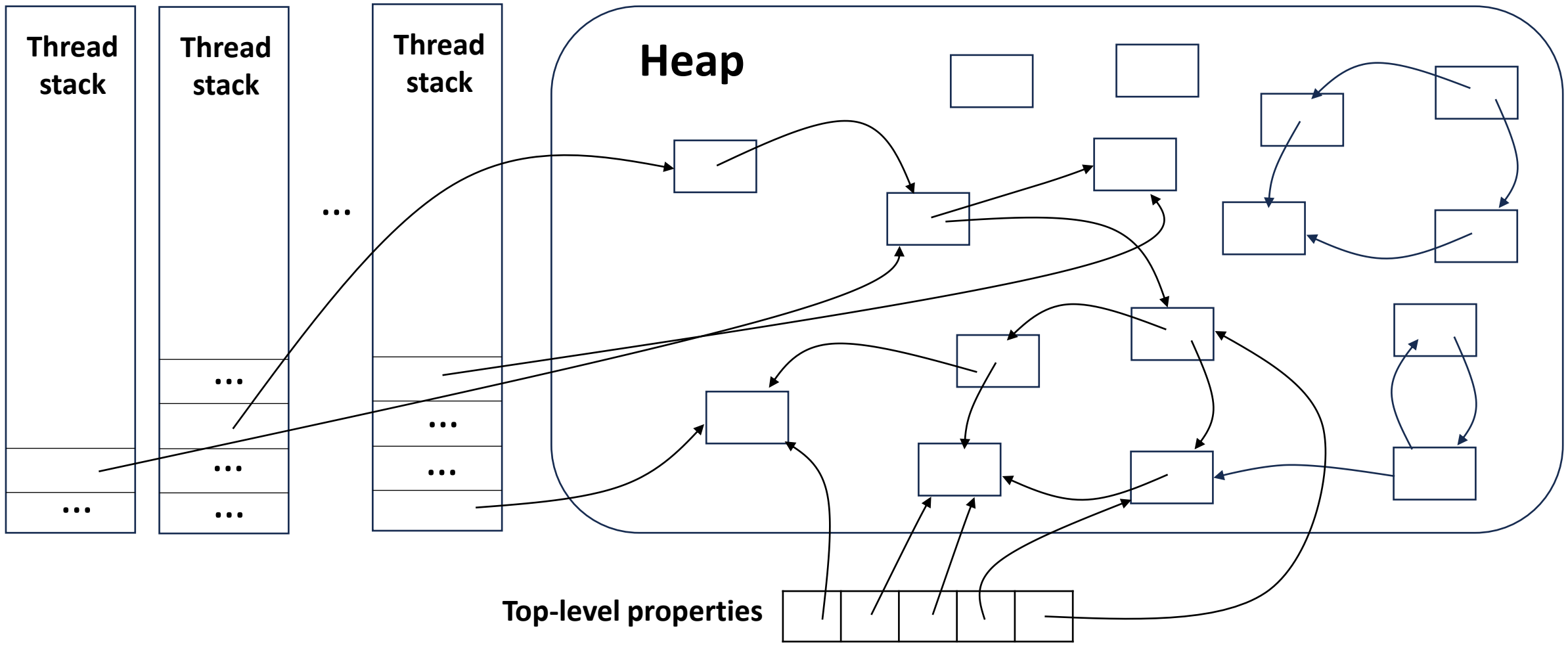
The **garbage collector** identifies garbage and frees the associated memory

Without garbage collection your program would run out of memory!

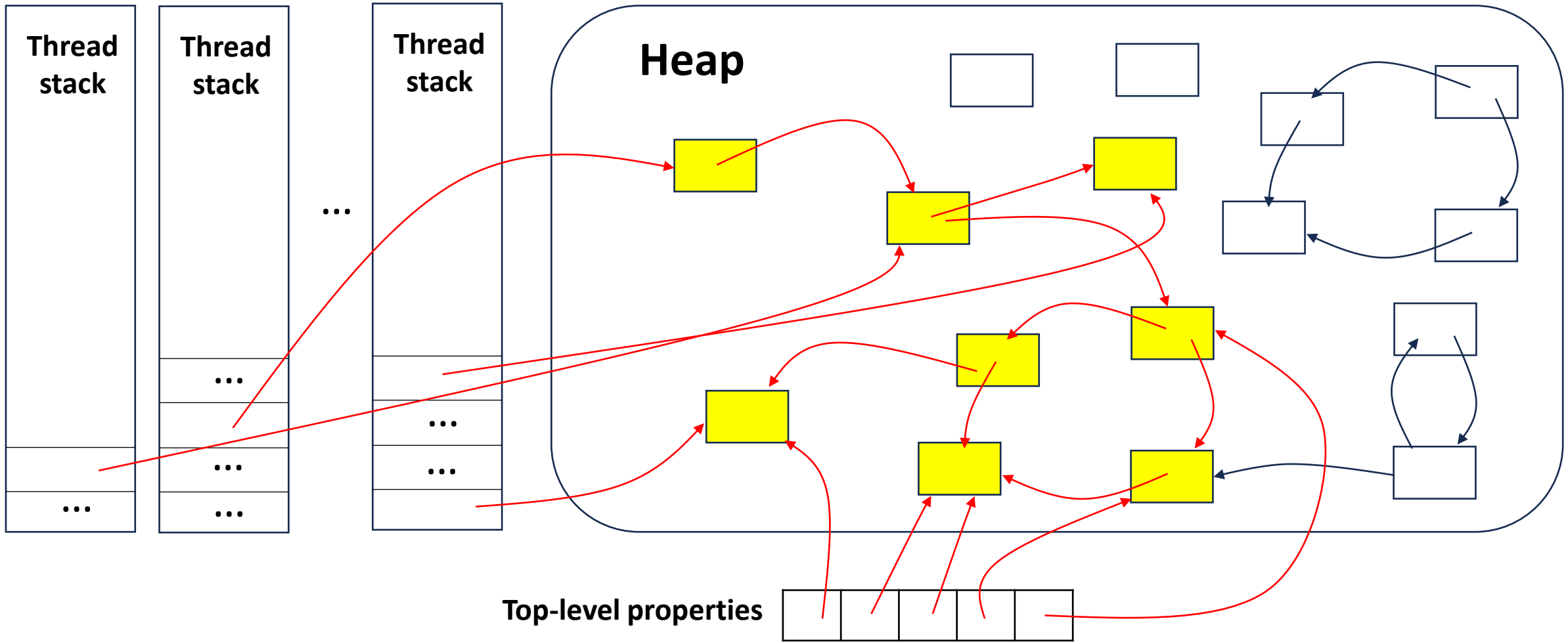
Many languages are garbage-collected (e.g. Haskell, Python, C#)

Some languages require manual deallocation (e.g. C, C++)

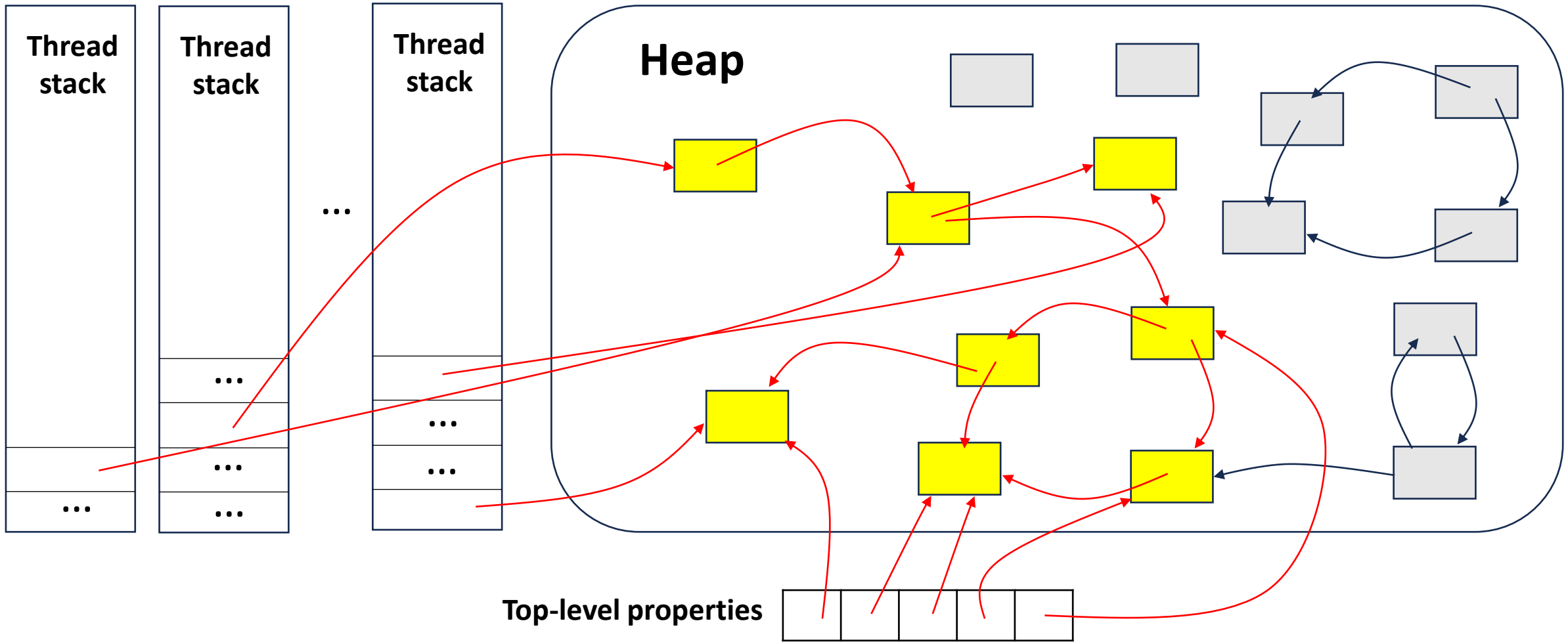
Mark and sweep garbage collection



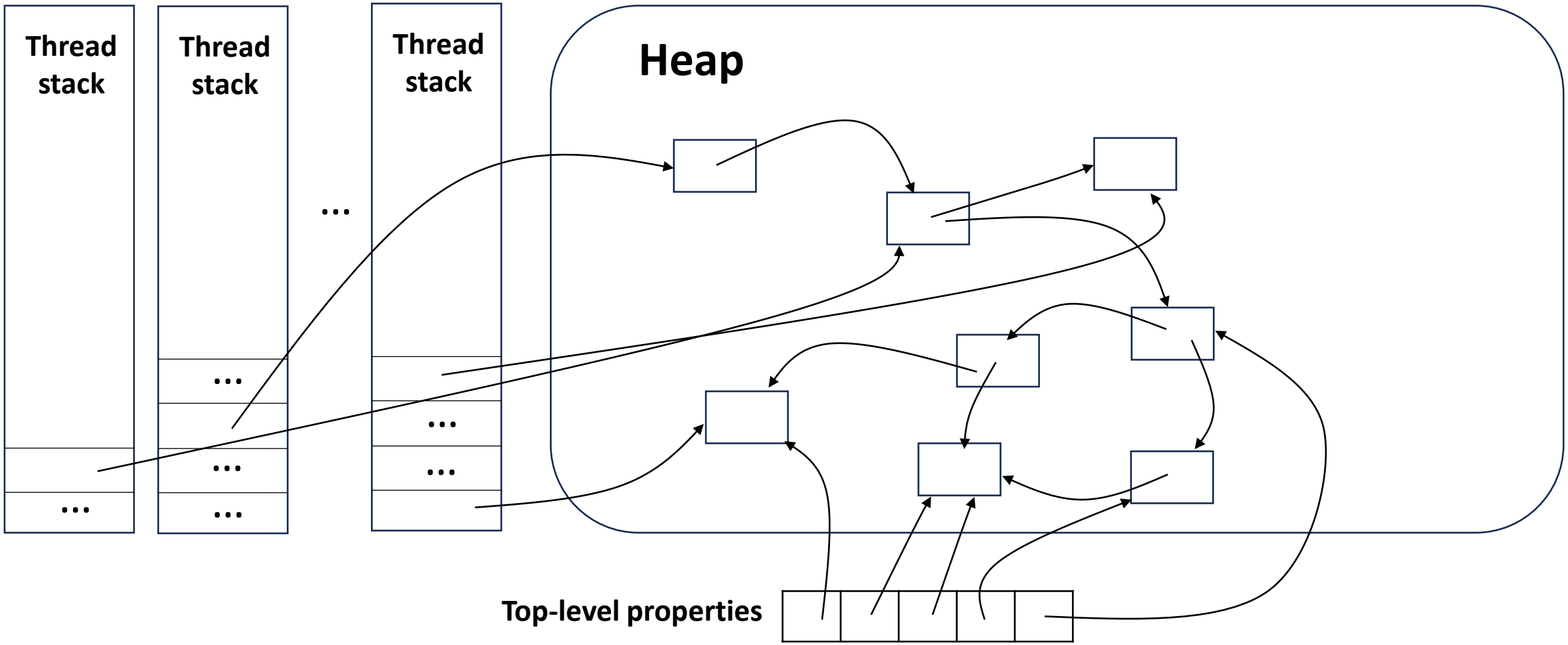
Mark all objects reachable from roots



Sweep away unmarked objects



Sweep away unmarked objects



Backing fields in Kotlin

Java classes centre around the concept of **fields**

Kotlin has fields, but we haven’t talked about them so far!

Computed and non-computed properties

Non-computed properties – they hold a value that can be retrieved

```
data class Rectangle(val width: Int, val height: Int) {
    val area: Int
        get() = width * height
}
```

A computed property – its value is a function of (zero or more) other properties

Computed and non-computed properties

```
class Silly {
    val five: Int
        get() = 5

    val twenty: Int
        get() = 20
}
```

Computed properties

```
class Counter {
    var value: Int = 0
        private set
    val negativeValue: Int
        get() = -value

    fun inc(): Int = ++value
}
```

Non-computed property

Computed property

Changing `value` leads to a change in the result that `negativeValue` will yield

Non-computed properties have **backing fields**

A non-computed value needs to store its value somewhere in memory

The value is stored in a **backing field**

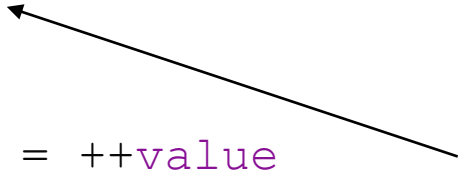
Let’s see backing fields in the IntelliJ debugger

Demo

Referring to backing fields directly

```
class NonNegativeCounter {
    var value: Int = 0
        set(newValue) {
            if (newValue < 0) throw UnsupportedOperationException()
            value = newValue
        }

    fun inc(): Int = ++value
}
```

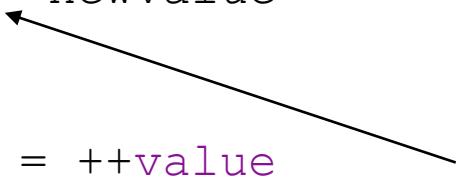


Wrong! This invokes set again – infinite recursion

Referring to backing fields directly

```
class NonNegativeCounter {
    var value: Int = 0
        set(newValue) {
            if (newValue < 0) throw UnsupportedOperationException()
            field = newValue
        }

    fun inc(): Int = ++value
}
```



Correct. This assigns the new value to the backing field

Intro to Java by converting some Kotlin to Java

Kotlin

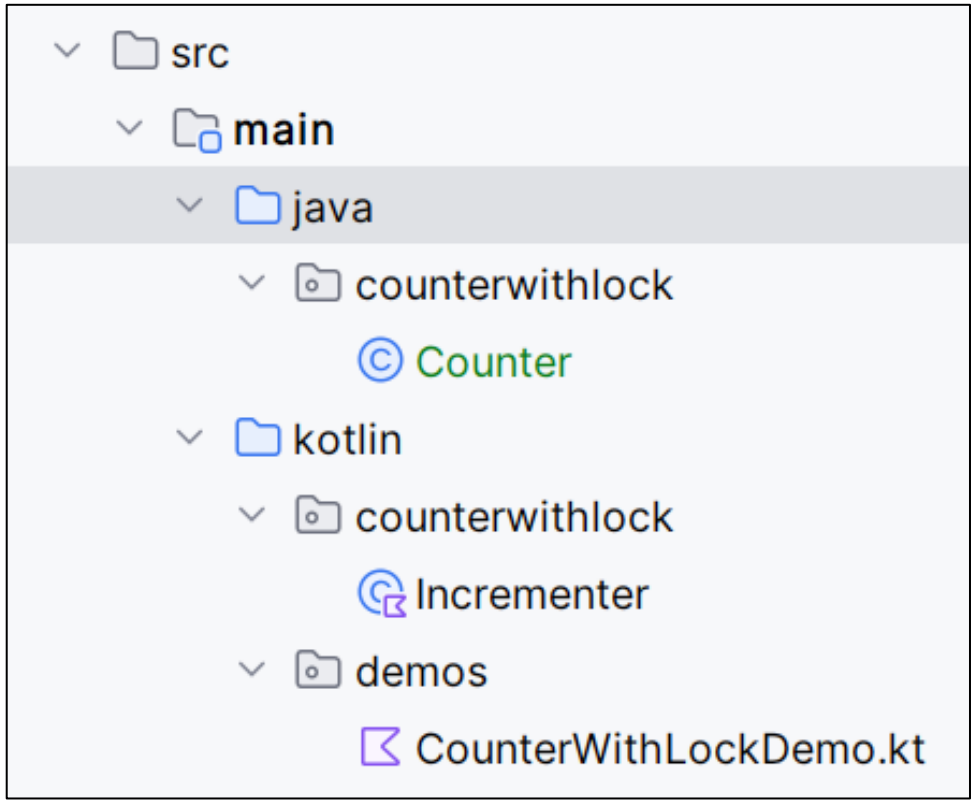
Java

Co-locating Java and Kotlin code in a project

Java code for package `mypackage` lives in `src/main/java/mypackage`

Kotlin code for package `mypackage` lives in `src/main/kotlin/mypackage`

Similar for tests



Kotlin to Java demo 1

The `Counter` class from our concurrency demo

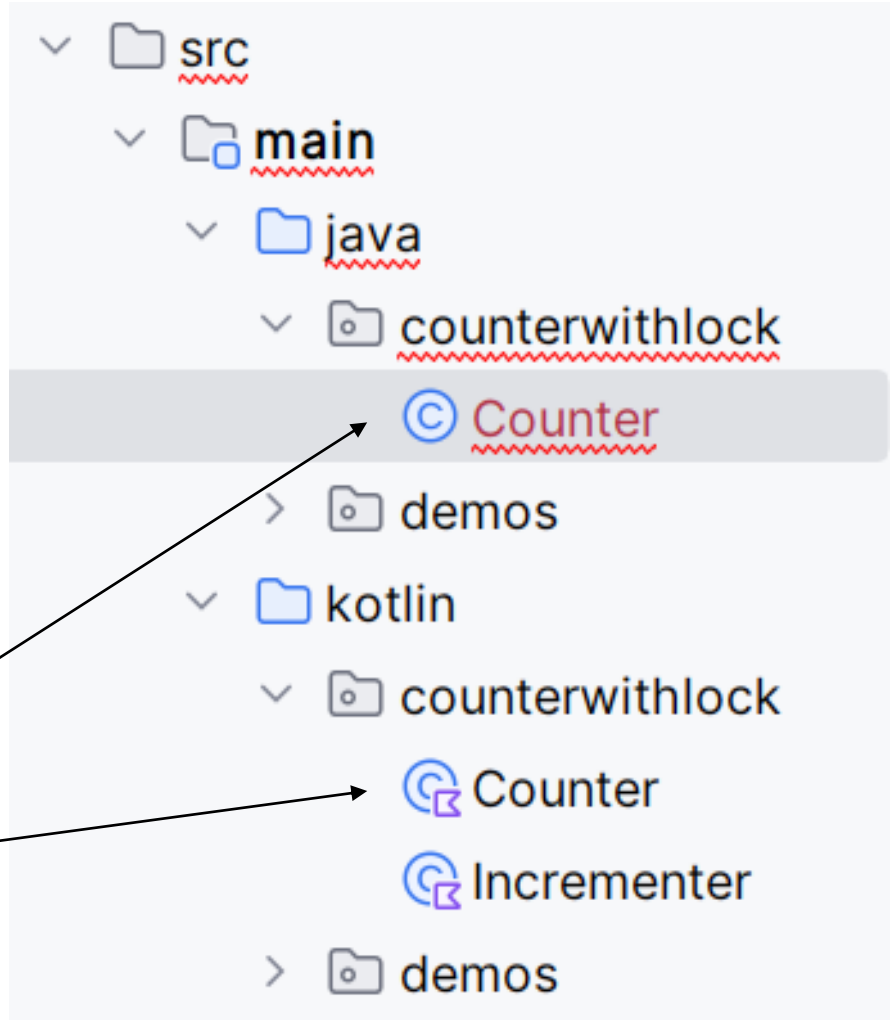
Kotlin and Java files in a project share a namespace

We cannot have a Java and Kotlin classes with same name in same package

After compilation, there would be two bytecode files with same name and nothing to distinguish them

Not clear which one should be used, hence not allowed

Compile error: Duplicate class found



In Java, public is not the default visiblity

```
class Counter { …
```

```
public class Counter { …
```

Equivalent in Kotlin

Not equivalent in Java

In Java, public is not the default visiblity

Java: default visibility is package-visible

No keyword to specify this – it's what you get if you do not specify visibility

A package-visible declaration is visible to all code in same package

Kotlin: does not have package visibility

Advanced: Kotlin visibility and Java visibility are misaligned in some other ways – look them up if interested!

Comparing a Kotlin property with an equivalent Java field

```kotlin
class Counter {
    private val lock: Lock = ReentrantLock()
```

```java
public class Counter {
    private final Lock lock = new ReentrantLock();
```

A **field** declaration

Comparing a Kotlin property with an equivalent Java field

```kotlin
class Counter {
    private val lock: Lock = ReentrantLock()
```

```java
public class Counter {
    private final Lock lock = new ReentrantLock();
```

Same meaning as in Kotlin

Comparing a Kotlin property with an equivalent Java field

```kotlin
class Counter {
    private val lock: Lock = ReentrantLock()
```

```java
public class Counter {
    private final Lock lock = new ReentrantLock();
```


`final` means that field cannot be modified after construction

Comparing a Kotlin property with an equivalent Java field

```kotlin
class Counter {
    private val lock: Lock = ReentrantLock()
```

```java
public class Counter {
    private final Lock lock = new ReentrantLock();
```

Type names come before field name

Comparing a Kotlin property with an equivalent Java field

```kotlin
class Counter {
    private val lock: Lock = ReentrantLock()
```

```java
public class Counter {
    private final Lock lock = new ReentrantLock();
```

`new` is mandatory when invoking a constructor

Aside: Ally likes `new` – it make the points where objects are constructed explicit

A Kotlin property, vs. a Java field + getter

```kotlin
class Counter {
    …
    var value: Int = 0
        private set
```

```java
public class Counter {
    …
    private int value = 0;

    public int getValue() {
        return value;
}
```

A field declaration

A getter method

Java has **primitive** types

```
class Counter {
    …
    var value: Int = 0
        private set
```

```
public class Counter {
    …
    private int value = 0;

    public int getValue() {
        return value;
}
```

The `int` type represents a plain integer value, not a reference to an integer object

Other primitive types include `float` and `double`

Method return type comes before method name

```
class Counter {
    …
    var value: Int = 0
        private set
```

```
public class Counter {
    …
    private int value = 0;

    public int getValue() {
        return value;
}
```

How do we mimic `private set`?

```
class Counter {
    …
    var value: Int = 0
        private set
```

```
public class Counter {
    …
    private int value = 0;

    public int getValue() {
        return value;
}
```

1. Make `value` field `private`
2. Do not provide a `setValue` method

Java fields should always be private

Best practice:

- Make all fields private
- Only provide a public “getter” method if reading the field value is part of the service your class should provide
- Make fields final if possible
- Only provide a public “setter” for a non-final field if changing the value of the field is part of the service your class should provide

This approach maximises **encapsulation**

Java does not have expression bodies

```
class Counter {
    …
    var value: Int = 0
        private set
```

```
public class Counter {
    …
    private int value = 0;

    public int getValue() {
        return value;
}
```

A (non-abstract) method body is always a block of code

The `try` … `finally` pattern

```
fun inc(): Int {
    lock.withLock {
        val result = value
        value++
        return result
    }
}
```

```
public int inc() {
    try {
        lock.lock();
        int result = value;
        value++;
        return result;
    } finally {
        lock.unlock();
    }
}
```

A `finally` block always gets executed, regardless of what the code in the try block does

Aside: Kotlin also has `try` … `finally`, and `withLock` uses this patten

```
fun <T> Lock.withLock(action: () -> T): T {
    lock()
    try {
        return action()
    } finally {
        unlock()
    }
}
```

Kotlin to Java demo 2

The `Incrementer` class from our concurrency demo

Java: use `implements` when implementing an interface

```
class Incrementer(
    private val counter: Counter,
    private val numIncrements: Int,
) : Runnable {
    …
```

```
public class Incrementer implements Runnable {

    private final Counter counter;
    private final int numIncrements;
    …
    public Incrementer(Counter counter, int numIncrements) {
        this.counter = counter;
        this.numIncrements = numIncrements;
    }
    …
```

Java constructor: looks like a method with no return type, and with the class as its name

```
class Incrementer(
    private val counter: Counter,
    private val numIncrements: Int,
) : Runnable {
    …
```

```
public class Incrementer implements Runnable {

    private final Counter counter;
    private final int numIncrements;
    …
    public Incrementer(Counter counter, int numIncrements) {
        this.counter = counter;
        this.numIncrements = numIncrements;
    }
    …
```

A constructor

Refers to field

Refers to constructor parameter

Java has no notion of “primary” constructor

Java has `int` primitive type and `Integer` class type

```
private val _observedValues: MutableSet<Int> = mutableSetOf()
```

`Integer` is an immutable class that wraps a primitive int value

```
private final Set<Integer> observedValues = new HashSet<>();
```

`Set<int>` not allowed: generic collections can only be instantiated using class / interface types (not primitives)

Advanced: investigate `int` vs. `Integer`

Some questions to investigate (not examinable):

- What is auto-boxing and auto-unboxing in Java and how does it work?
- Why doesn't Kotlin distinguish explicitly between primitive and non-primitive integer types?
- What is the difference between `Array<Int>` and `IntArray` in Kotlin?

Java collections: `List` and `Set` offer a **mutable** interface

```kotlin
private val _observedValues: MutableSet<Int> = mutableSetOf()

val observedValues: Set<Int>
    get() = _observedValues
```

Kotlin's `Set` interface: does not offer mutator methods like `add` and `remove`

Java's `Set` interface: does offer these and other mutator methods

```java
private final Set<Integer> observedValues = new HashSet<>();

public Set<Integer> getObservedValues() {
    return Collections.unmodifiableSet(observedValues);
}
```

Java collections: `List` and `Set` offer a **mutable** interface

Kotlin has `Set` and `MutableSet`, `List` and `MutableList`, etc.

Java just has `Set`, `List`, etc., which all offer a **mutable** service

```
private final Set<Integer> observedValues = new HashSet<>();

public Set<Integer> getObservedValues() {
    return Collections.unmodifiableSet(observedValues);
}
```

To disallow mutable operations, return a protective wrapper

Exercise: what are the downsides of relying on protective wrappers?

Java `void` type is similar to Kotlin's `Unit`

```
override fun run() {
    …
}
```

equivalent to:

```
override fun run(): Unit {
    …
}
```

Kotlin: `Unit` indicates that the special unit value is returned

```
public void run() {
    …
}
```

Java: `void` indicates that **no value** is returned

Java does not have an override keyword

Instead you can (and should) use the `@Override` annotation when overriding a superclass method or implementing an interface method

```
override fun run() {
    …
}
```

```
@Override
public void run() {
    …
}
```

Java has “C-style” for loops

```
override fun run() {
    for (i in 1..numIncrements) {
        _observedValues.add(counter.inc())
    }
}
```

```
public void run() {
    for (int i = 1; i <= numIncrements; i++) {
        observedValues.add(counter.inc());
    }
}
```

Homework: find out what this syntax means

Kotlin to Java demo 3

The `CounterWithLockDemo` class from our concurrency demo

Java does not have top-level functions or properties

```
fun main() {
    …
}
```

In Java, all methods, including main methods, must occur inside classes or interfaces

```
public class CounterWithLockDemo {
    public static void main(String[] args) {
        …
    }
    …
}
```

`static` means that we do not call the method on an instance of the class

Java forces the programmer to explicitly handle (or explicitly ignore) certain exceptions

```
thread1.join()
thread2.join()
```

An `InterruptedException` is possible in both languages

In Kotlin you are allowed to implicitly ignore this possibility

```
try {
    thread1.join();
    thread2.join();
} catch (InterruptedException exception) {
    // Ignore, or do something in response
}
```

Kotlin has convenience functions for printing

```
println(counter.value)
```

```
System.out.println(counter.getValue());
```

Kotlin's `println` simply calls Java's `System.out.println`

Java does not have infix functions and lacks many convenience methods

```
println(
    incrementer1.observedValues intersect incrementer2.observedValues,
)
```

Kotlin provides methods like **intersect** via extension methods

```
final Set<Integer> intersection =
    new HashSet<>(incrementer1.getObservedValues());
intersection.retainAll(incrementer2.getObservedValues());
System.out.println(intersection);
```

Advanced: study how the Kotlin standard library implements `intersect`

Java lacks many convenience methods

```
println(incrementer1.observedValues.sorted()[0])
```

```
System.out.println(incrementer1.getObservedValues()
        .stream()
        .sorted()
        .toList()
        .get(0));
```

More work needed in Java to get the values as a sorted list!

Java does not support operator overloading

```
println(incrementer1.observedValues.sorted()[0])
```

We can use `[]` to index a list in Kotlin, which leads to `get` being called

```
System.out.println(incrementer1.getObservedValues()
        .stream()
        .sorted()
        .toList()
        .get(0));
```

In Java we must call the `get` method by name