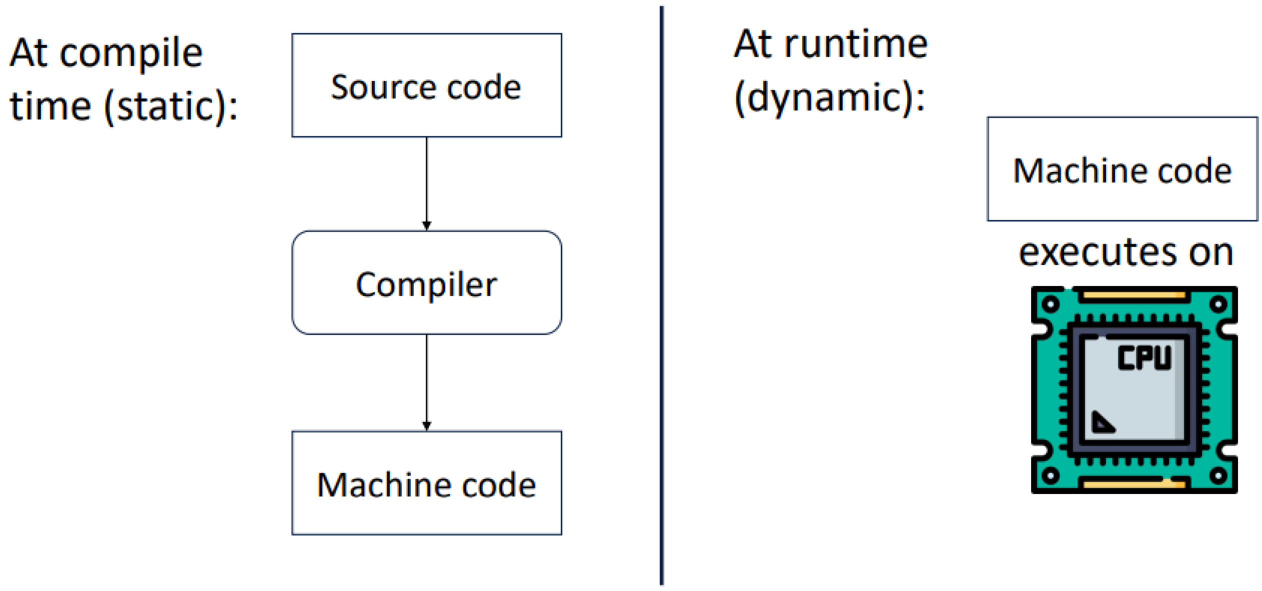
# From Kotlin to Java

## Static compilation




## 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

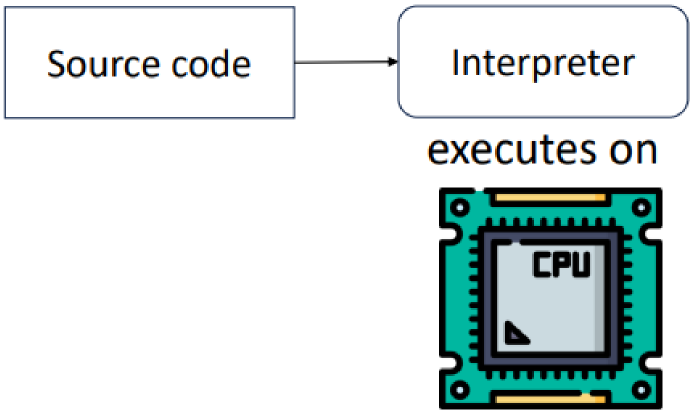
## Drawbacks of static compilation

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

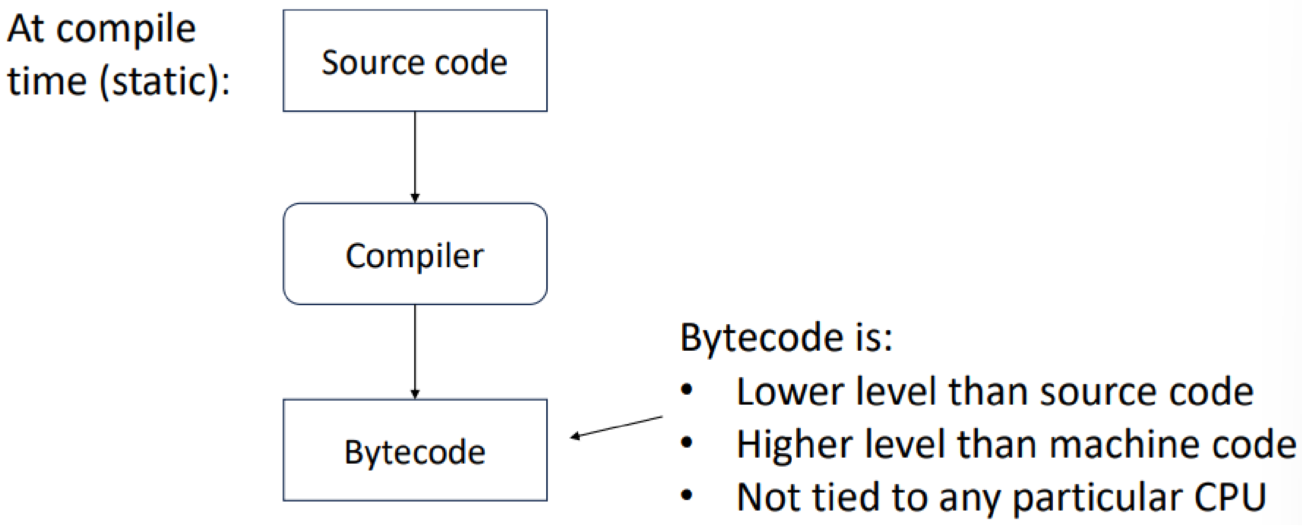
Slow execution:

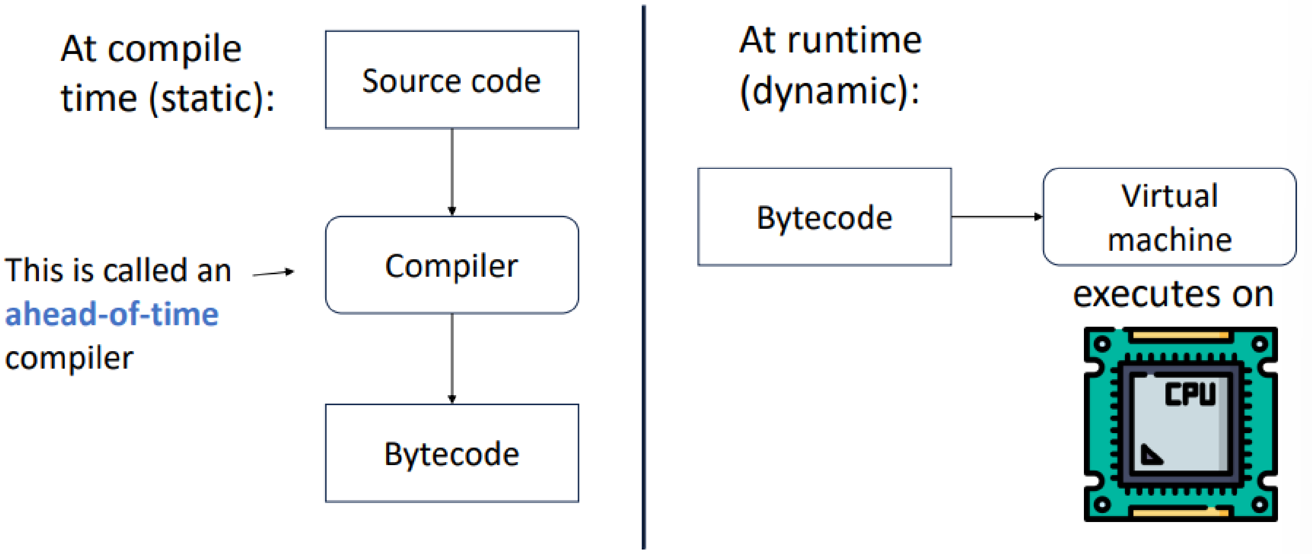
- The interpreter is a machine code program
- It can be seen as simulating the statements of the source program
- Typically much slower than executing a machine code version of the source program

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

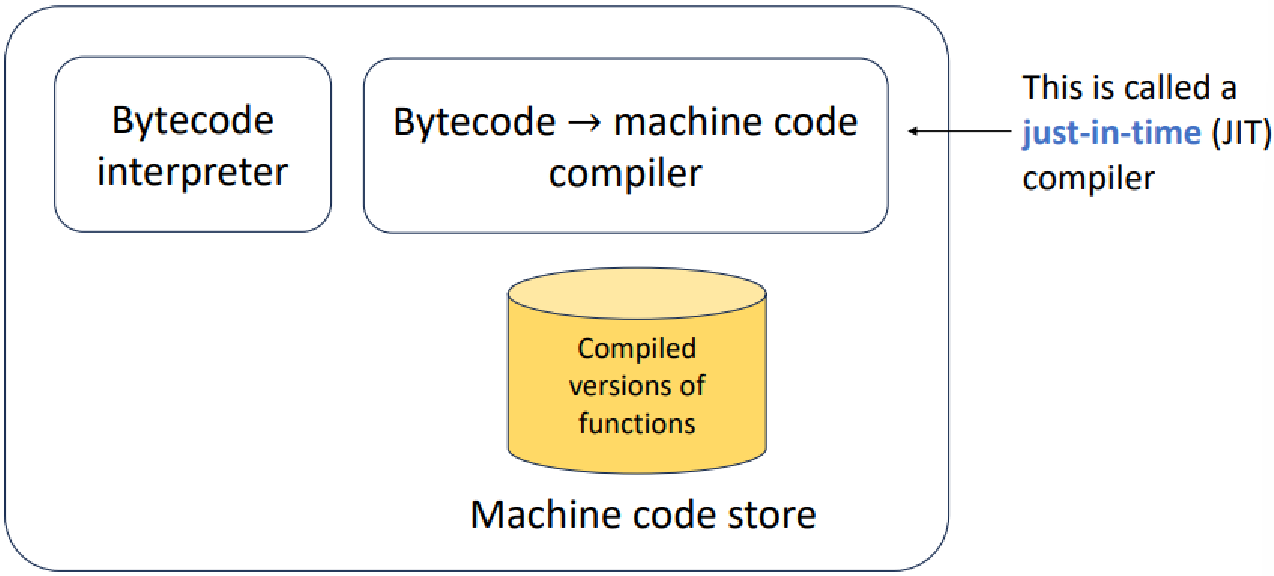
No static error checking is not fundamental to interpretation, but interpreted languages are usually dynamically typed

## Just-in-time (JIT) compilation



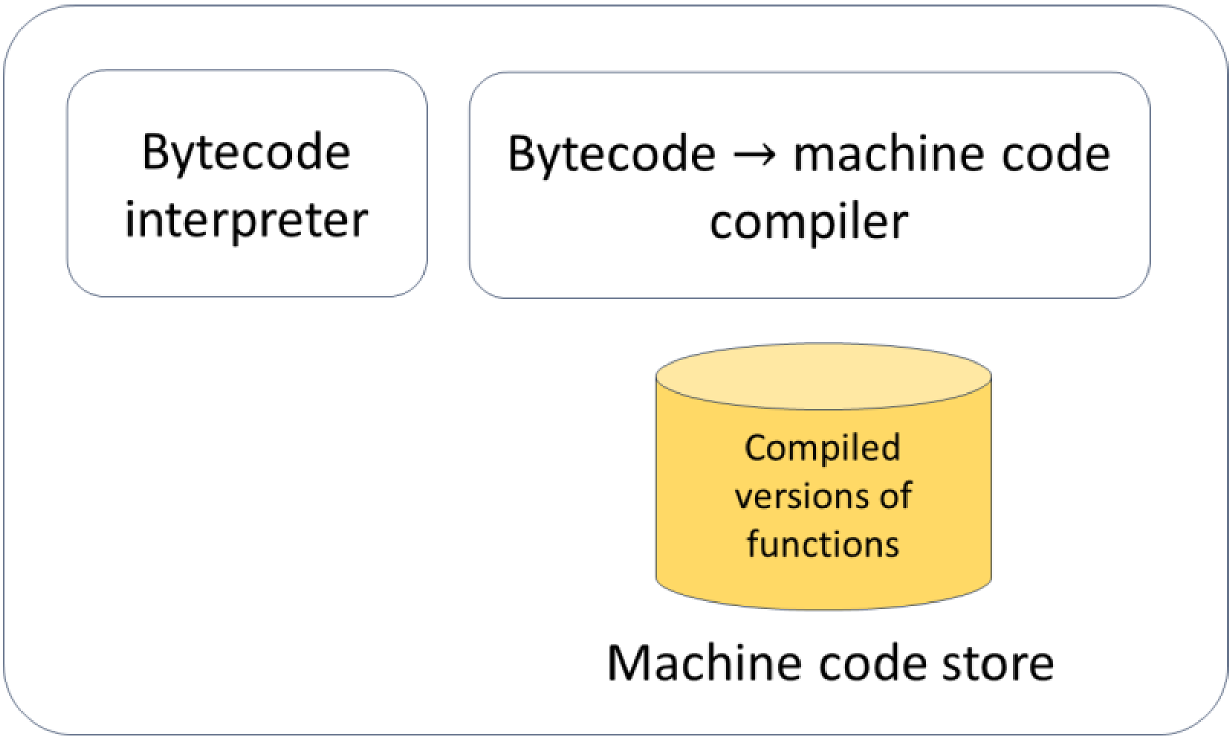




# Inside the virtual machine (VM)



# 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

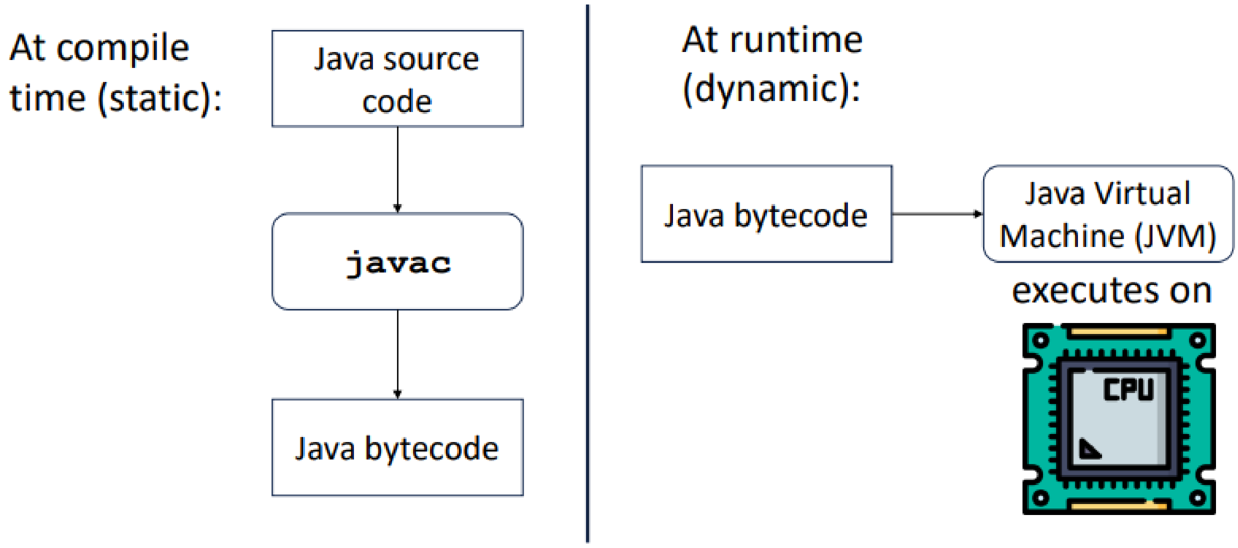
# Drawbacks of JIT compilation

Unpredictable performance:

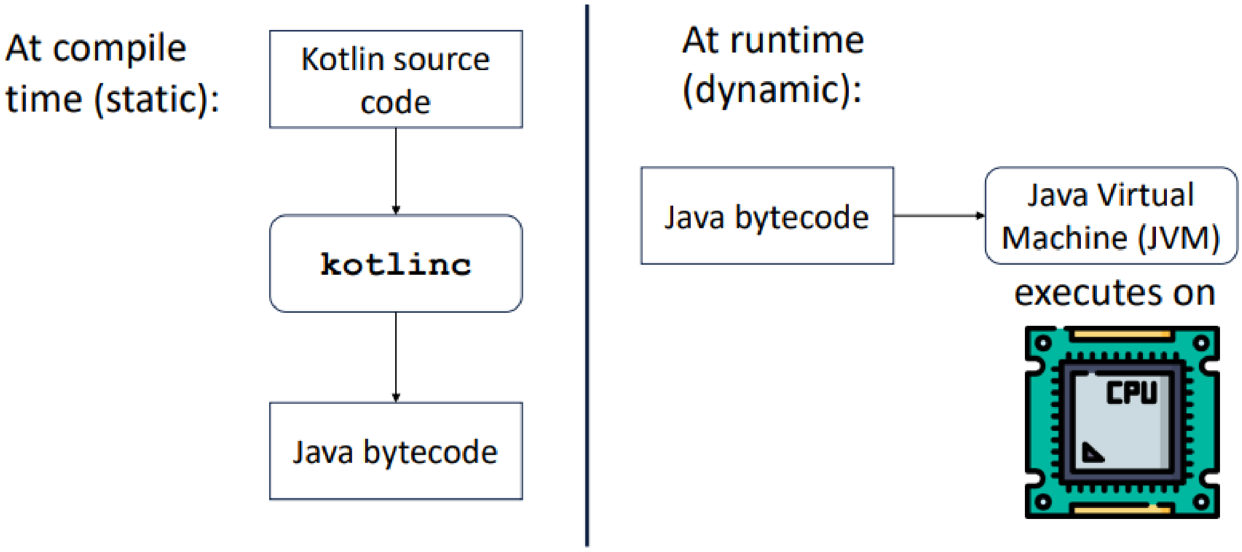
- Runtime is initially slow due to bytecode interpretation
- Performance improves over time
- Performance can suddenly leap due to a key function being JITcompiled
- JIT-compiler may make bad long-term decisions based on short-term profiling information

Bloat: A virtual machine is a large and complex application

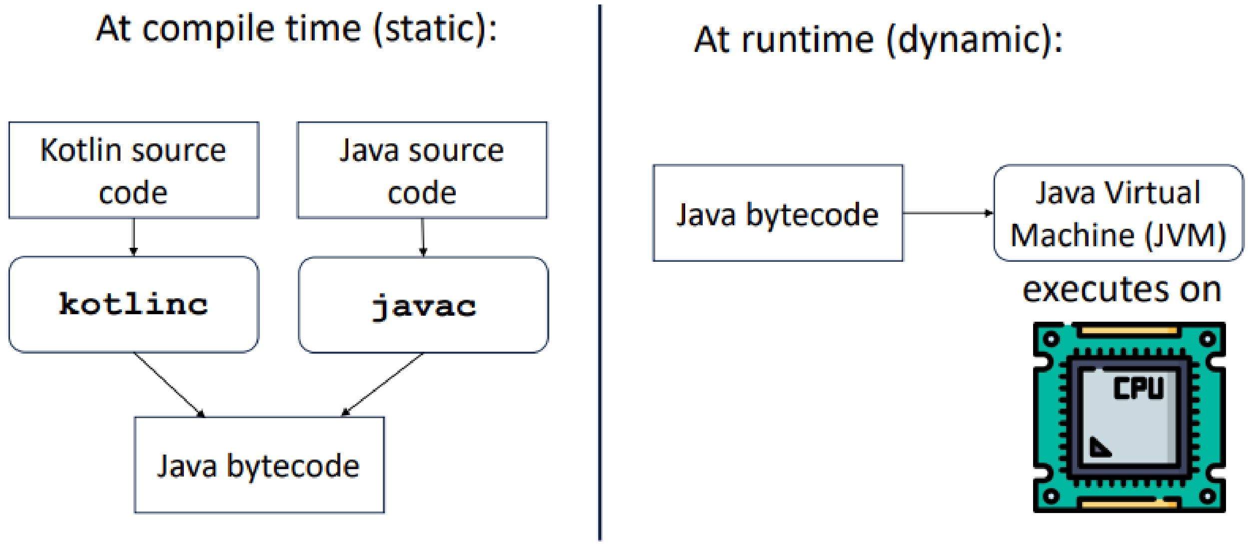
# Java uses just-in-time compilation




# Kotlin uses just-in-time compilation



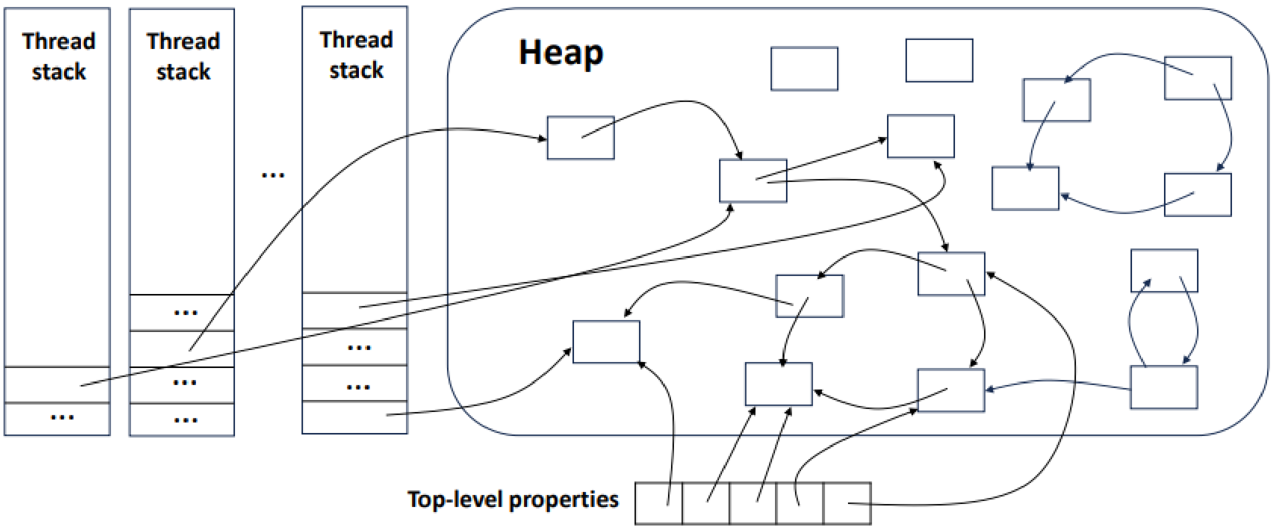

# Kotlin and Java work well together



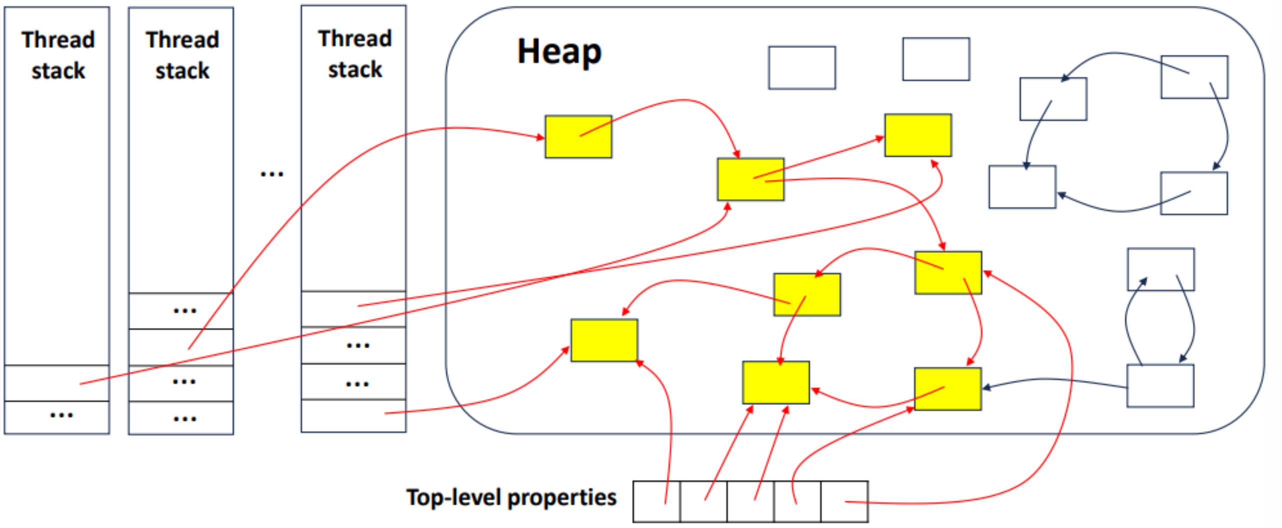

# Aside: garbage collection

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++)

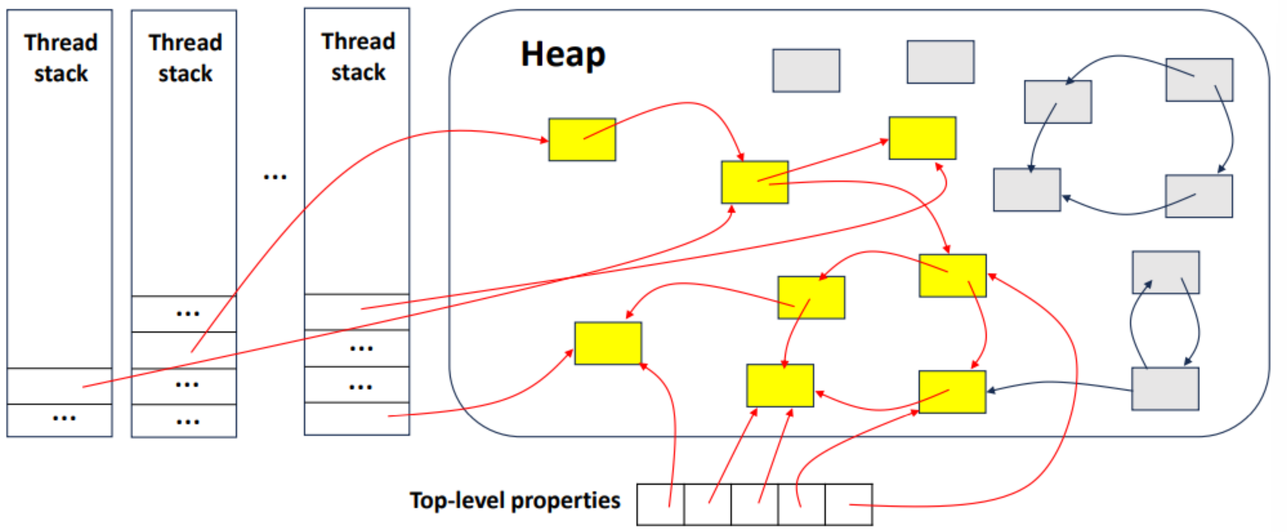
# Mark and sweep garbage collection

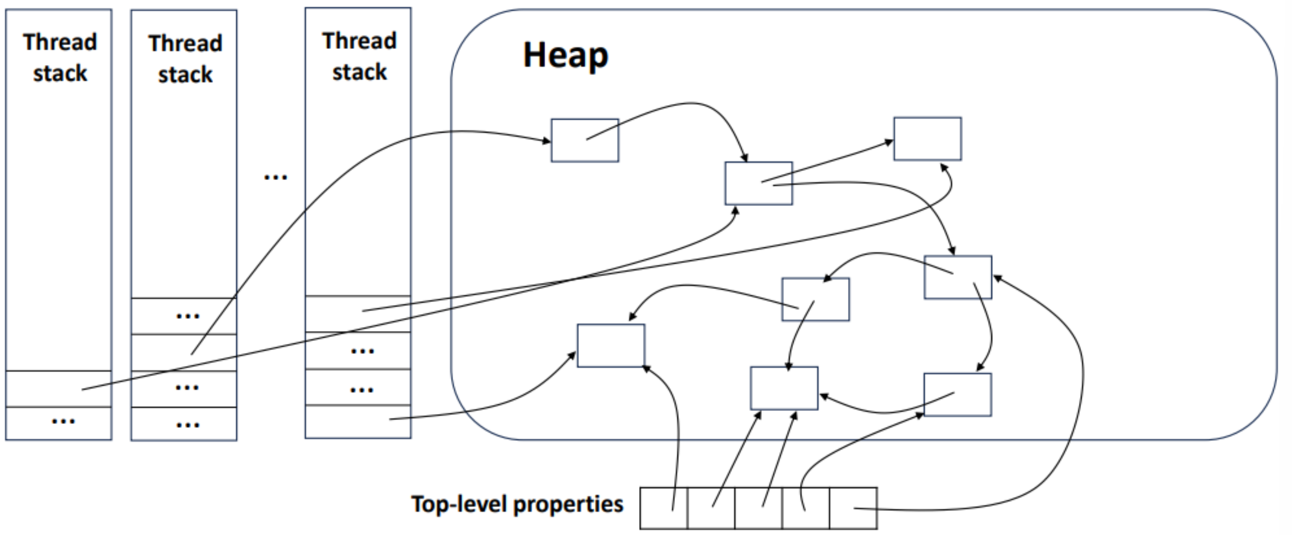



# Mark all objects reachable from roots




# Sweep away unmarked objects




# 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 properties

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

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

Non-computed property

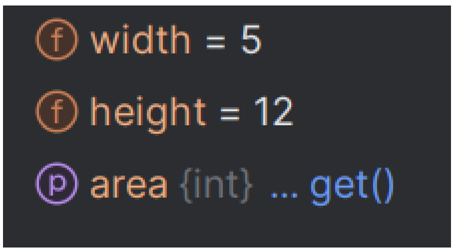
```
class Counter {
    var value: Int = 0
        private set
    val negativeValue: Int
        get() = -value

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

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.




f = Non-computed value
There is a backing field for each 'f' value

p = Computed value

No backing field needed

## 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

```
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

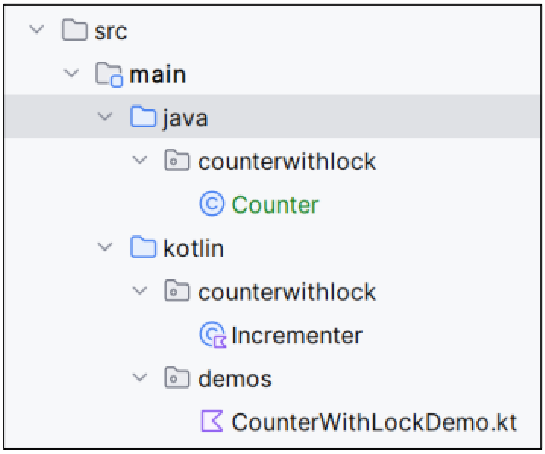
# Intro to Java by converting some Kotlin to Java

## 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

```java
package counterwithlock;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
public class Counter {
    private final Lock lock = new ReentrantLock();
    private int value = 0;
    public int getValue() {
        return value;
    }
    int inc() {
        try {
            lock.lock();
            final int result = value;
            value++;
            return result;
        } finally {
            lock.unlock();
        }
    }
}
```

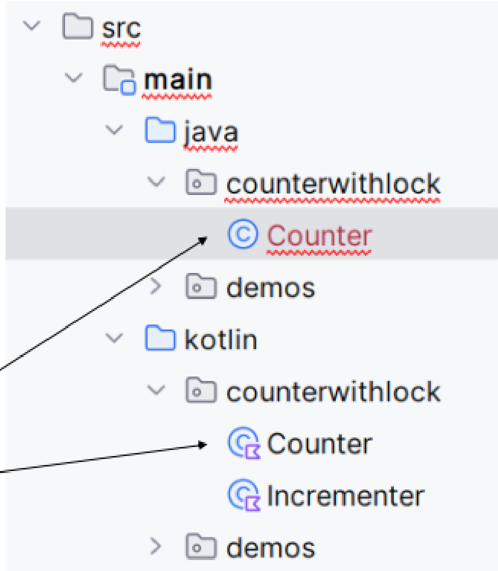
# Kotlin and Java files in a project share a namespace

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 visibility

```
class Counter { …
```

```
public class Counter { …
```

Equivalent in Kotlin

Not equivalent in Java

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

# 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

`private final`

Same meaning as in Kotlin

`private final Lock lock`

`final` means that field cannot be modified after construction

`inal Lock lock`

Type names come before field name

`lock = new Reentra`

`new` is mandatory when invoking a constructor

## A Kotlin property, vs. a Java field + getter

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

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

    public int getValue() {                   A getter method
        return value;
}
```

## Java has primitive types

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

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

```java
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
```

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

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

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

# 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

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

```
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();
    }
}
```

# 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

```
package counterwithlock;
import java.util.Collections;
import java.util.HashSet;
import java.util.Set;
public class Incrementer implements Runnable {
    private final Counter counter;
    private final int numIncrements;
    private final Set<Integer> observedValues;
    // A Java constructor
    public Incrementer(Counter counter, int numIncrements) {
        this.counter = counter;
        this.numIncrements = numIncrements;
        this.observedValues = new HashSet<>();
        // More code would go here - the code from a Kotlin "init"
block.
    }
      public Set<Integer> getObservedValues() {
            return Collections.unmodifiableSet(observedValues);
      }
      @Override
      public void run() {
        for(int i = 1; i <= numIncrements; i++) {
```

```
                observedValues.add(counter.inc());
            }
        }
    }
```

## 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 {
    …
```

A constructor

```
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 has no notion of “primary” constructor

Refers to field

Refers to constructor parameter

## 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)

## `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);
}
```

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

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

```java
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`

```kotlin
override fun run() {
    …
}
```

equivalent to:

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

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

```java
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

```
package demos;
import counterwithlock.Counter;
import counterwithlock.Incrementer;
```

```
import java.util.HashSet;
import java.util.Set;
class CounterWithLockDemo {
    public static void main(String[] args) {
        final Counter counter = new Counter();
        final Incrementer incrementer1 = new Incrementer(counter,
500);
        final Incrementer incrementer2 = new Incrementer(counter,
500);
        final Thread thread1 = new Thread(incrementer1);
        final Thread thread2 = new Thread(incrementer2);
        thread1.start();
        thread2.start();
        try {
            thread1.join();
            thread2.join();
        } catch (InterruptedException exception) {
            // Either do something about it, or ignore it.
            // Here we will just ignore.
        }
        System.out.println(counter.getValue());
        final Set<Integer> intersection = new HashSet<>(
                incrementer1.getObservedValues()
        );
        intersection.retainAll(incrementer2.getObservedValues());
        System.out.println(intersection);
        System.out.println(incrementer1.getObservedValues()
                .stream()
                .sorted()
                .toList()
                .get(0));
        System.out.println(incrementer2.getObservedValues()
                .stream()
                .sorted()
                .toList()
                .get(0));
    }
}
```

## 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

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

```java
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

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

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

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

## Java lacks many convenience methods

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

```java
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

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

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

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

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