JET BRAINS Kotlin

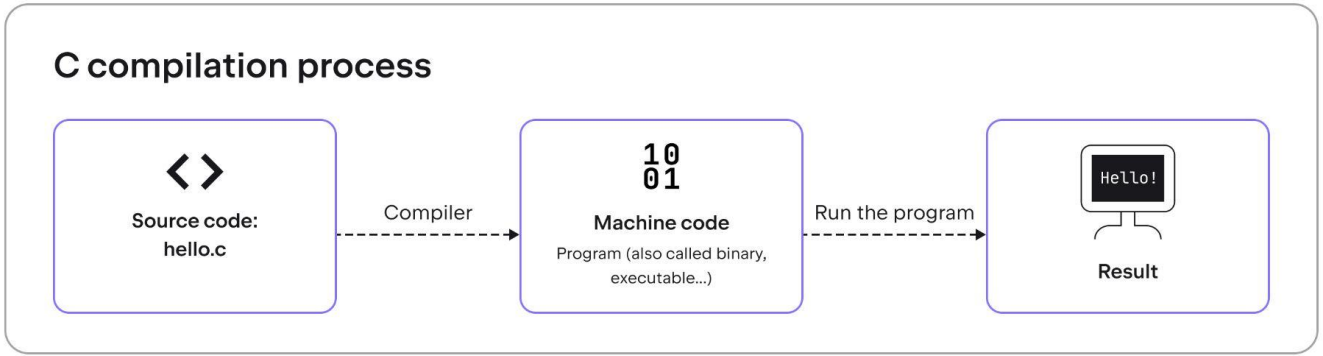
The Java Virtual Machine & the Kotlin Compiler

@kotlin | Developed by JetBrains

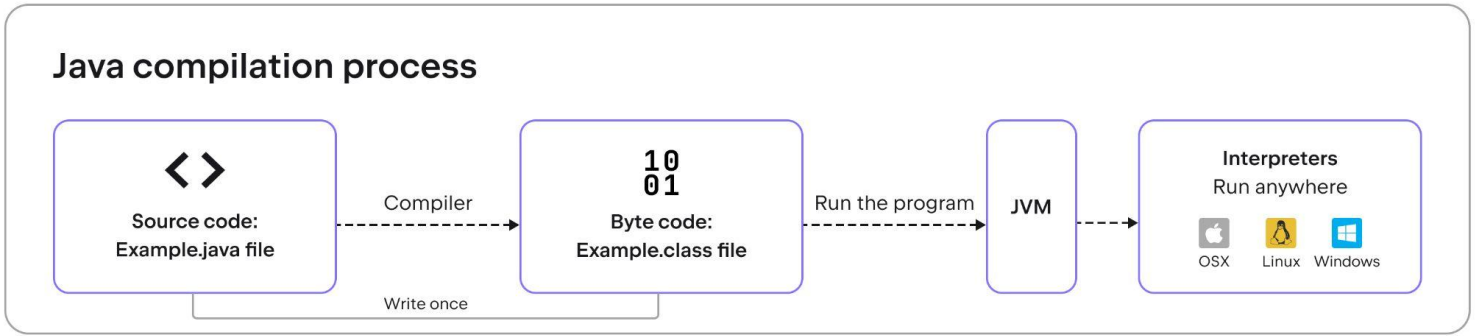
The Java language

- Was created in 1995.
- Is an OOP language with strong static typing.
- Has Just-in-time (JIT) compilation.
- Uses the Java Virtual Machine (JVM).
- Has a garbage collector, meaning you can allocate memory and it will be freed automatically.

Compilation process – Java vs C



VS



Java bytecode

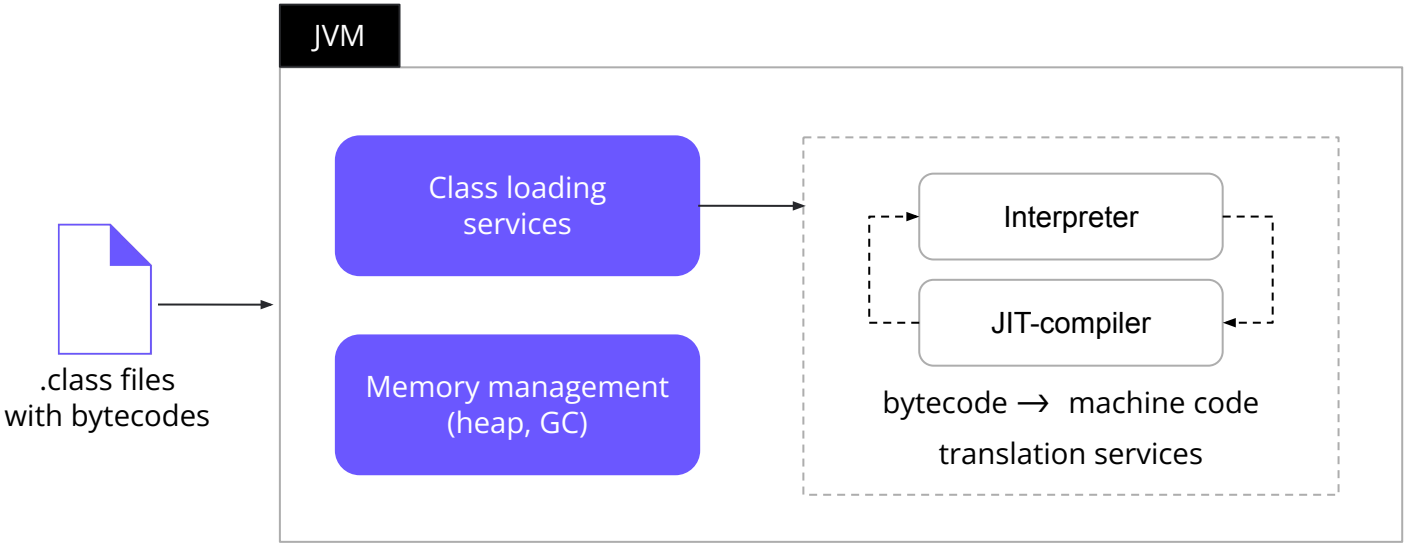
```java
public class Main {
    public static void main(String[] args) {
        System.out.print("Hello, World!");
    }
}
```

```
public class examples/Main {

  public <init>()V
   L0
    LINENUMBER 3 L0
    ALOAD 0
    INVOKESPECIAL java/lang/Object.<init> ()V
    RETURN
   L1
    LOCALVARIABLE this Lorg/examples/Main; L0 L1 0
    MAXSTACK = 1
    MAXLOCALS = 1

  public static main([Ljava/lang/String;)V
   L0
    LINENUMBER 5 L0
    GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
    LDC "Hello, World!"
    ICONST_0
    ANEWARRAY java/lang/Object
    INVOKEVIRTUAL java/io/PrintStream.print
           (Ljava/lang/String;[Ljava/lang/Object;)Ljava/io/PrintStream;
    POP
   L1
    LINENUMBER 6 L1
    RETURN
   L2
    LOCALVARIABLE args [Ljava/lang/String; L0 L2 0
    MAXSTACK = 3
    MAXLOCALS = 1
}
```

The JVM under the hood



Memory organisation

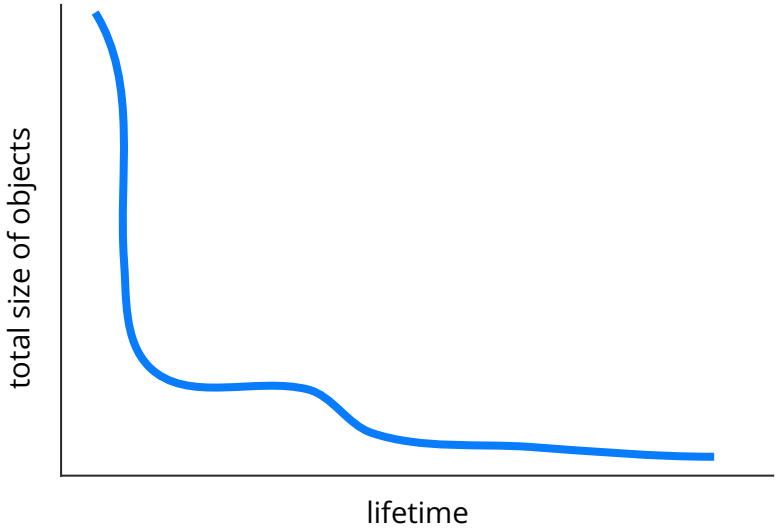
JVM memory is divided into two parts:

- Static memory, or non-heap memory, is created when the JVM starts, and it primarily stores class structures, fields, method data, and the code for methods and constructors.
    - Loaded classes
    - Stacks of all threads
    - Service memory for the JVM itself
    - Small – roughly 1024 KB for stacks
- Heap memory is the runtime data area shared among all JVM threads, and it is used to allocate memory for all Java objects.
    - All objects created during a program's execution
    - Large – from several MB up to several TB

Weak generational hypothesis

According to the weak generational hypothesis, objects tend to die young.

A related assumption is that, as an object lives longer, the likelihood will be that it will continue to live increases.



Generations

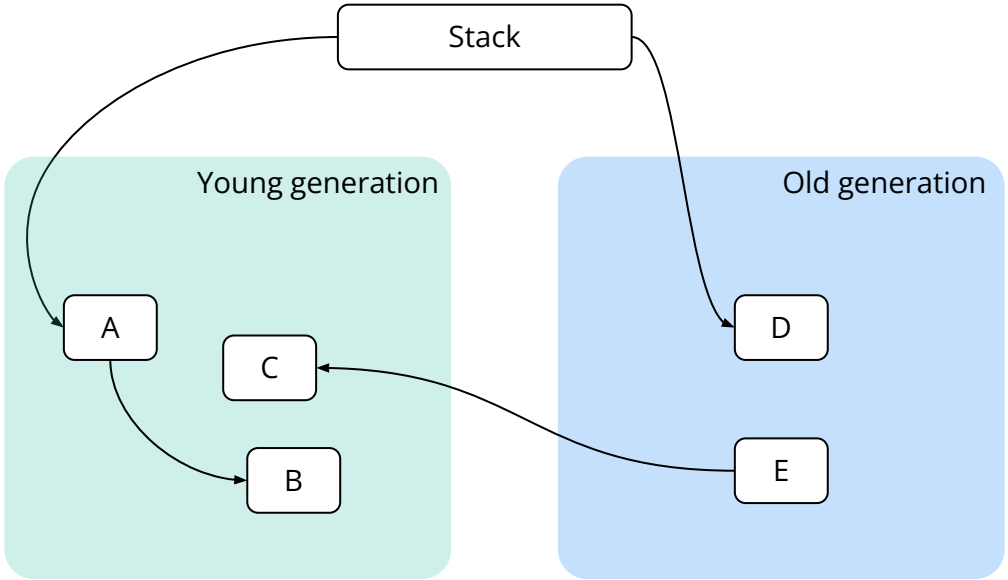
The memory of a program can be divided into two generations:

- Young
- Old

Garbage collection (GC) can be similarly divided:

- Small GC clears only objects from young generation
- Full GC clears objects from both generations

Dead objects

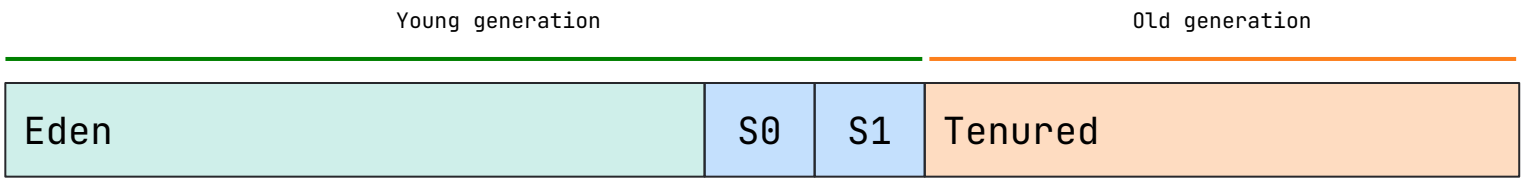


Serial garbage collector

The first garbage collector created was the serial garbage collector, which is single-threaded, and the parallel and CMS garbage collectors are based on it.

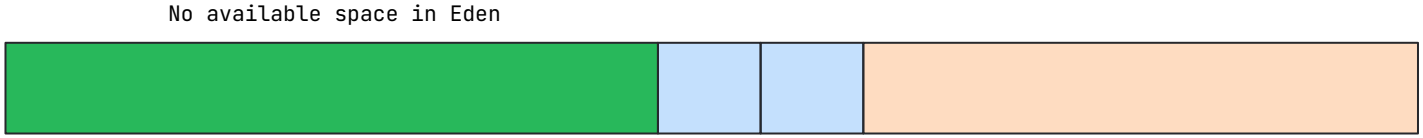
- In the serial garbage collector, the heap is divided into 4 areas:
  - Eden – Roughly 8/10 of the young generation
  - Survivor 0 – Roughly 1/10 of the young generation
  - Survivor 1 – Roughly 1/10 of the young generation
  - Tenured – Roughly 2/3 of the heap

- (Almost) All objects are created in the Eden area.

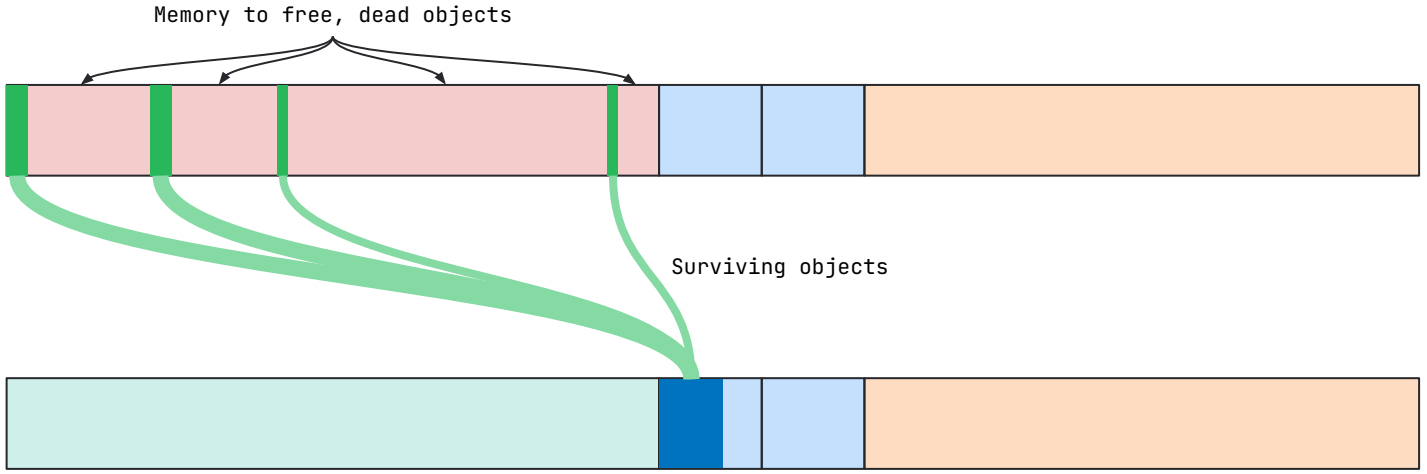


How garbage collection works

Current state:

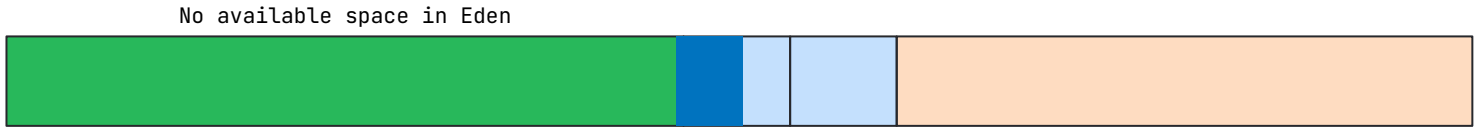


Before GC:

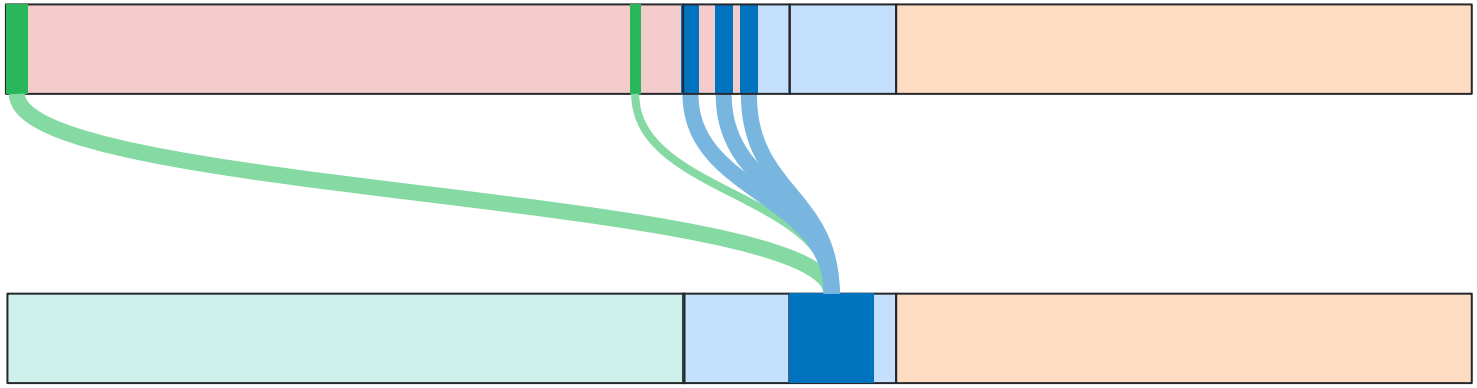


How garbage collection works

Current state:



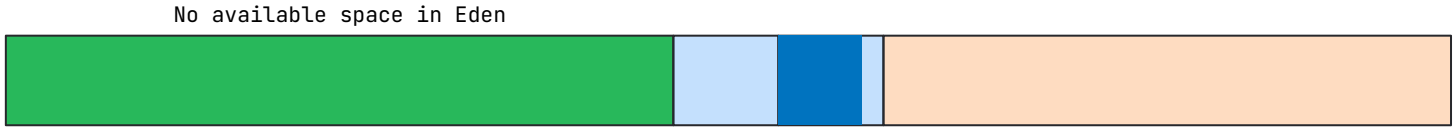
Before GC:



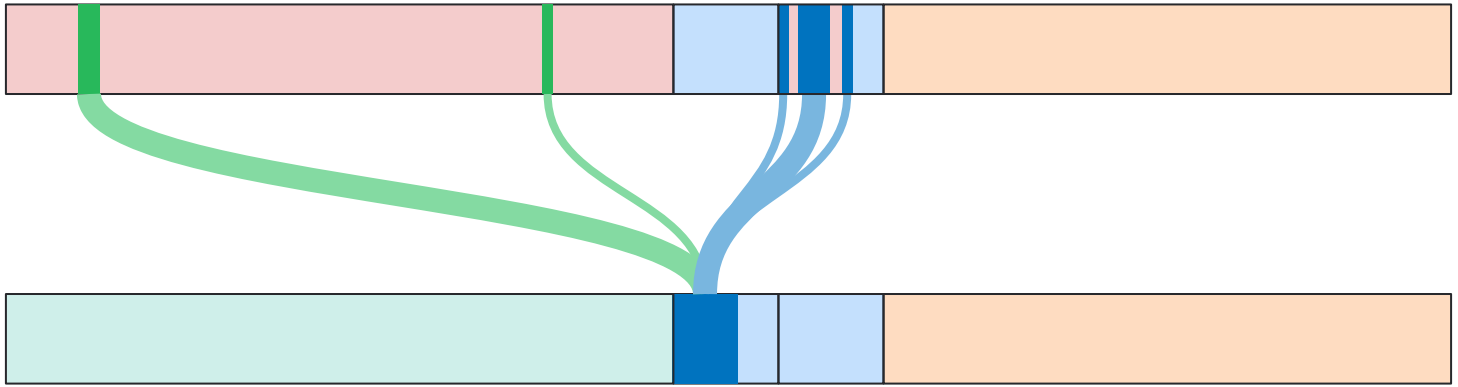
After GC:

How garbage collection works

Current state:



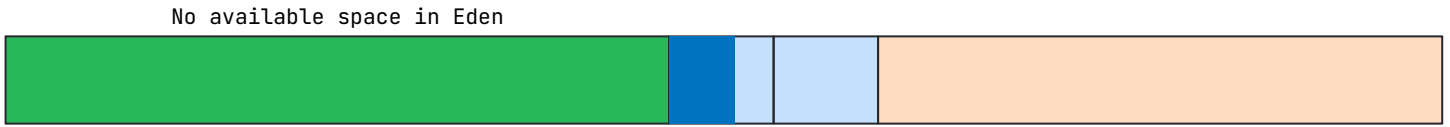
Before GC:



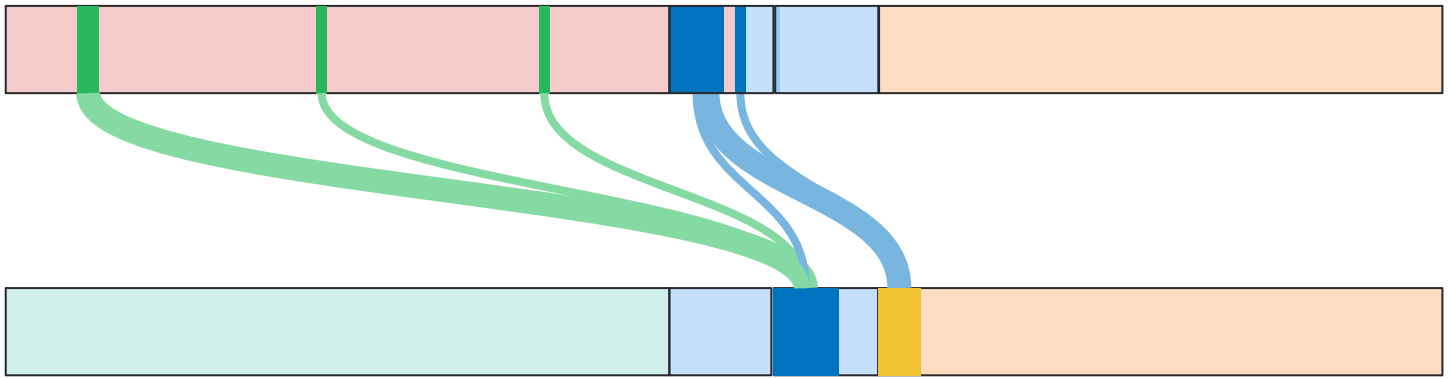
After GC:

How garbage collection works

Current state:



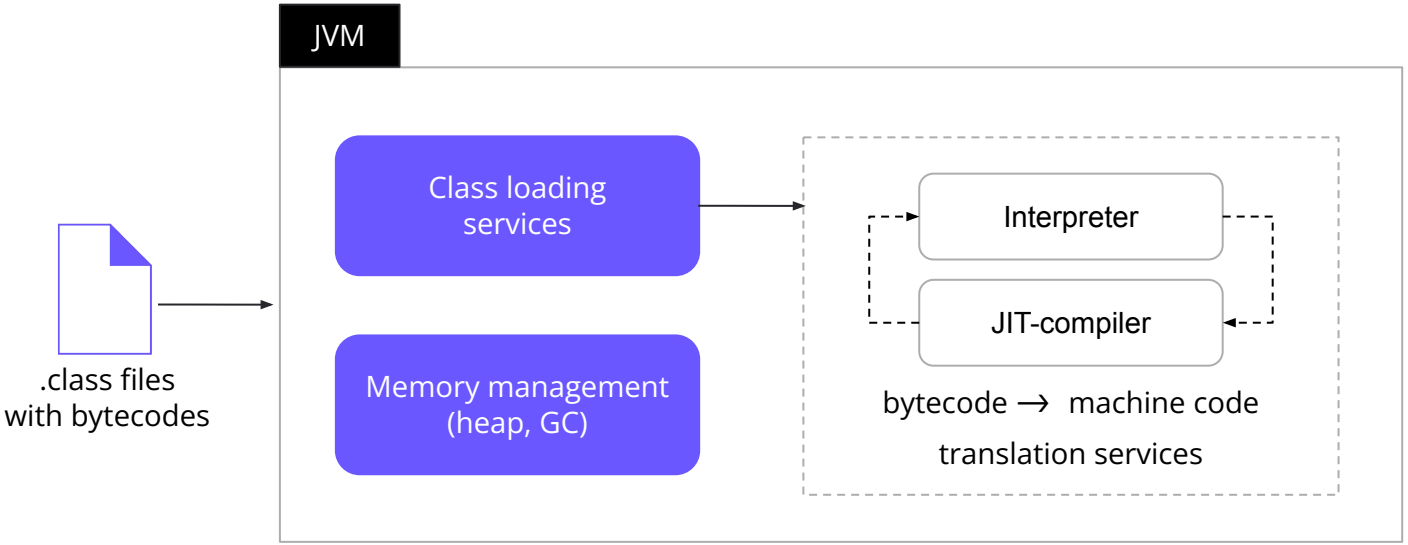
Before GC:



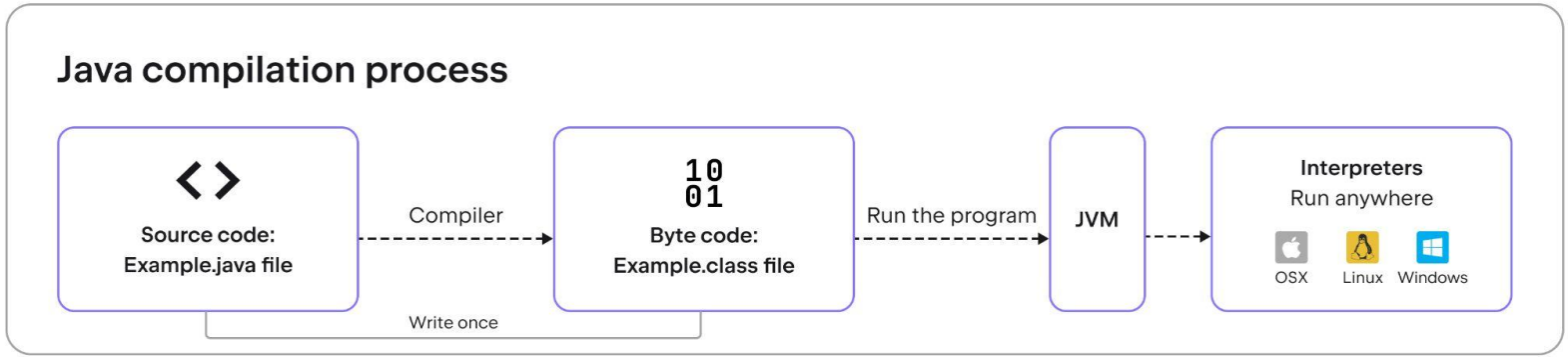
After GC:

Not enough space to move objects from Eden to Survivor 1

The JVM under the hood



Just-in-time compilation



- Program profiling occurs at runtime.
- Pieces of code are compiled for a specific platform to optimize the execution time.

Interpreting a command is much slower than executing it directly on the processor.

Why, then, do we need the interpreter?

Just-in-time compilation

Why, then, do we need the interpreter?

Interpreter

- Starts working almost instantly.
- The performance of the executable code is poor.

VS

JIT-compiler

- Kicks in after a long delay (needs time for optimizations).
- The performance of the executable (compiled) code is high.

Just-in-time compilation

What sorts of JIT code are worth compiling?

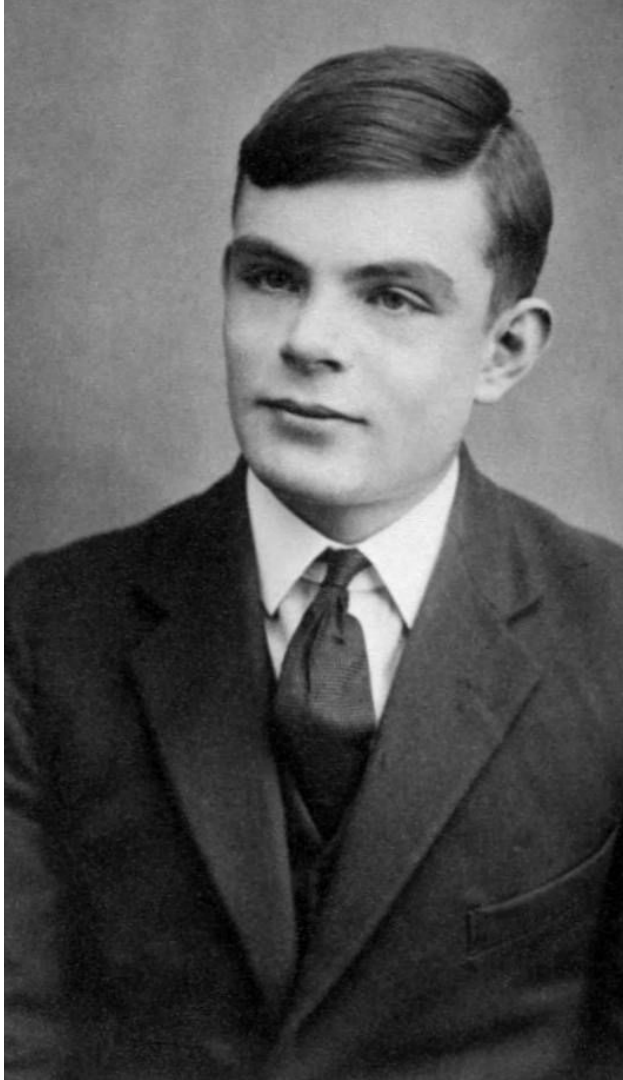
Code that will take a long time to run or code that runs frequently, because the compilation overhead will be covered by the profit from having optimized execution.

Just-in-time compilation

How can we understand which pieces of code will take a long time to execute?

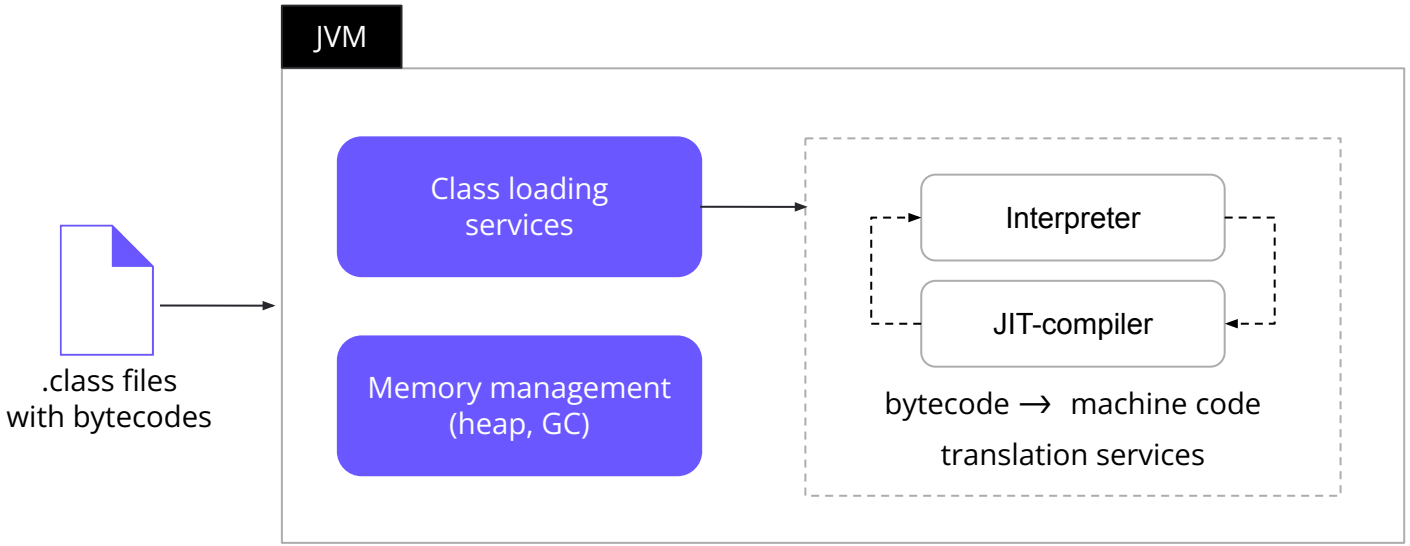
Some guy named Alan Turing said it is **impossible**.

But empirically it is possible. If a piece of code took a long enough time to execute once, then most likely the same will be true in the future.



The JVM under the hood

Why does the compiler need to access the interpreter?



The JVM under the hood

Why does the compiler need to access the interpreter?

```
class PiUtils {
    private static final double PI = 3.141592653589;

    public static double getPiSquared() {
        return PI * PI;
    }
}
```

```
public static double getPiSquared() {
    return 9.869604401084375;
}
```

The JVM under the hood

Why does the compiler need to access the interpreter?

```
class PiUtils {
    private static final double PI = 4;

    public static double getPiSquared() {
        return PI * PI;
    }
}
```

Reflection

```
public static double getPiSquared() {
    return 9.869604401084375;
}
```