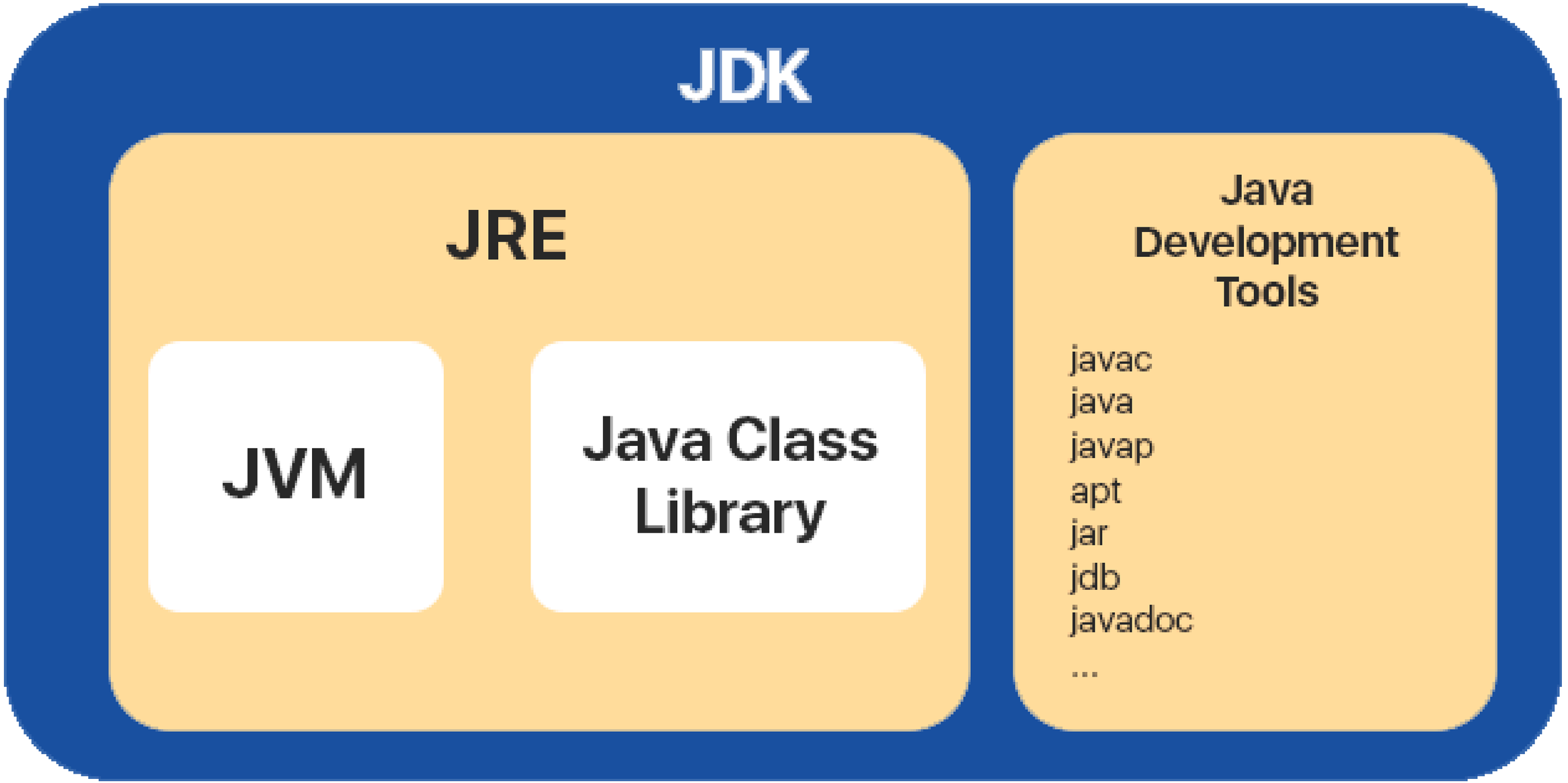
JVM: Виртуальная машина JAVA

"Write once, run anywhere"

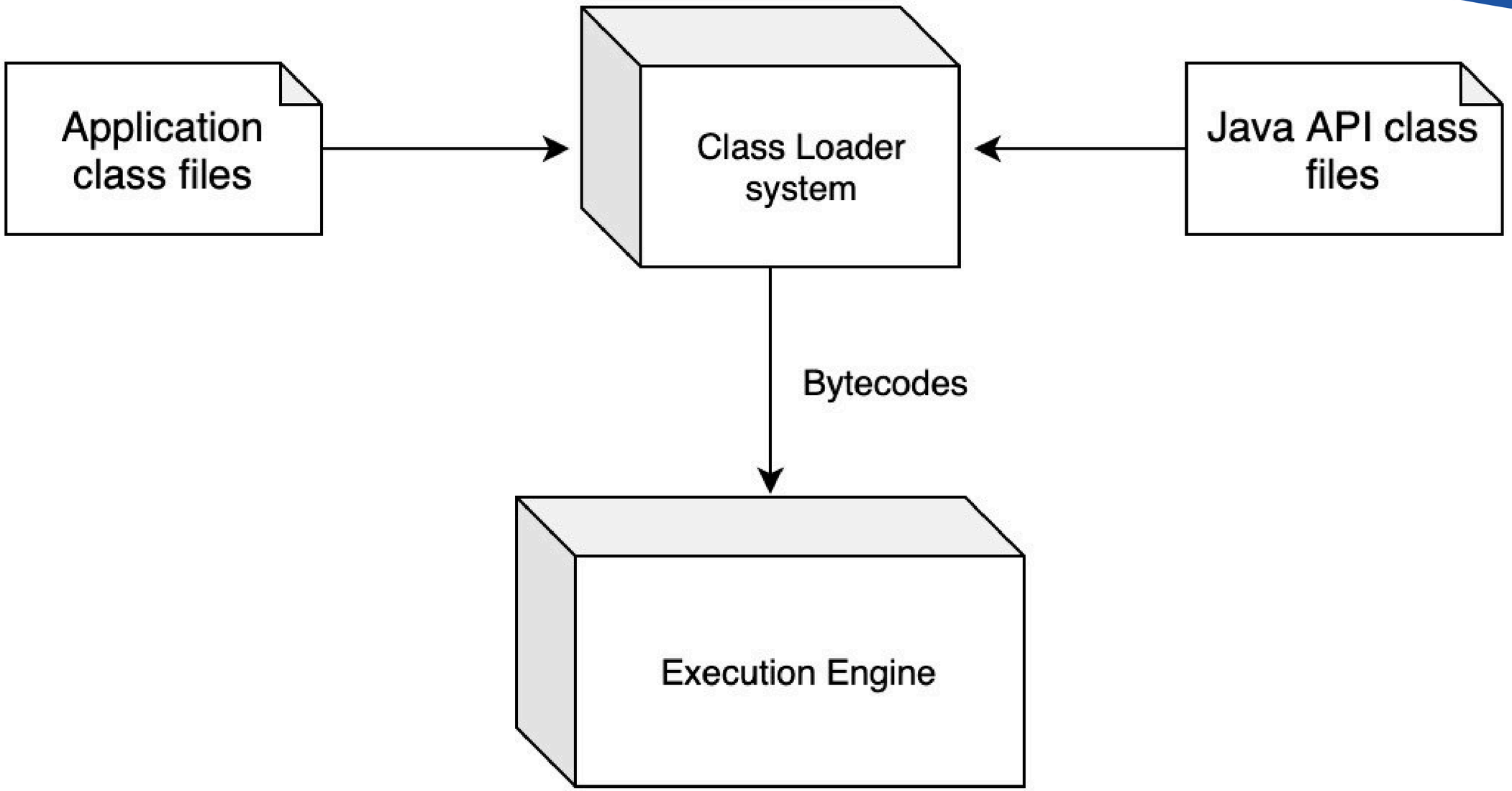
Введение


JDK
JRE
JVM
Java Class Library
Java Development Tools
javac
java
javap
apt
jar
jdb
javadoc
...

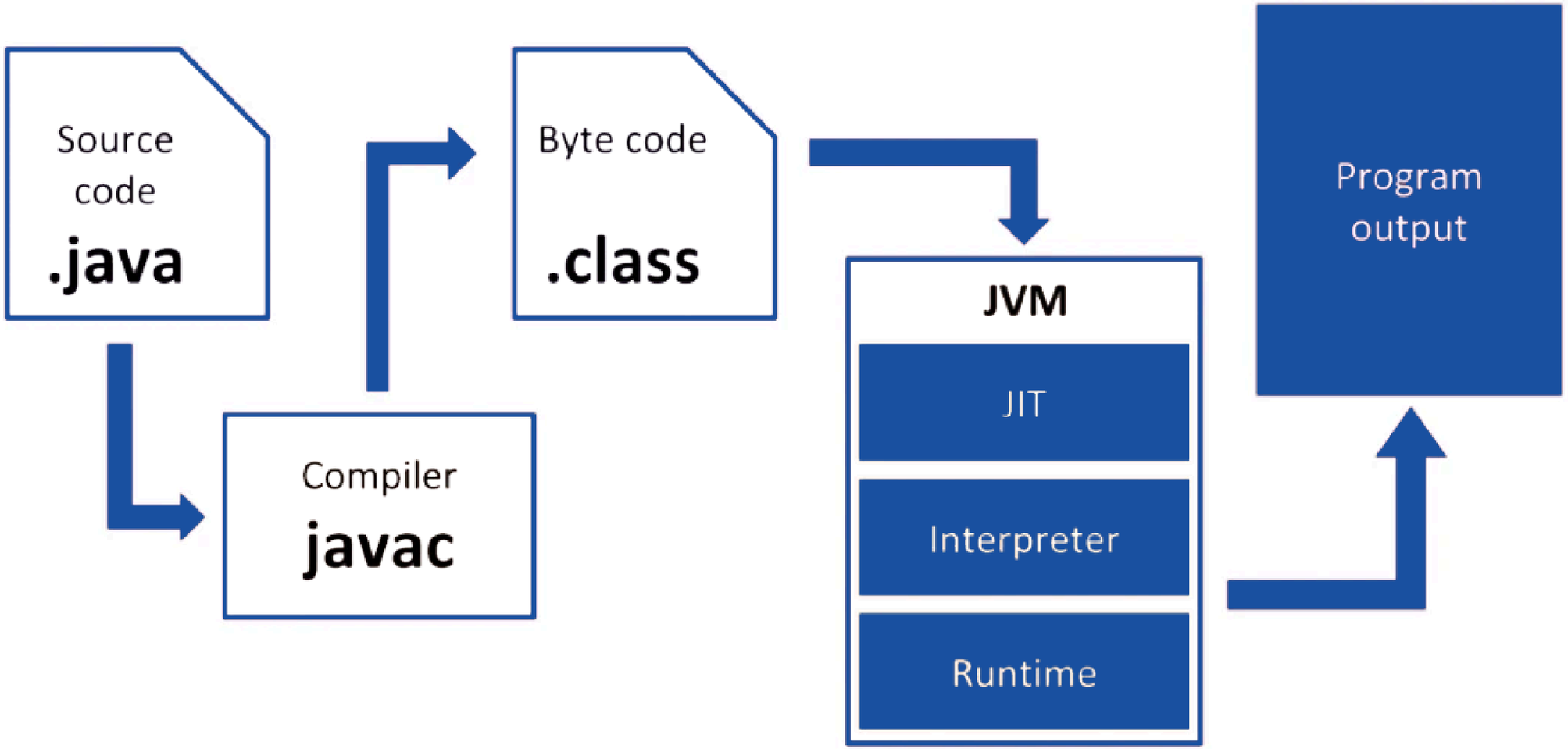
Компоненты JVM

- Загрузчик классов (Class Loader):
  - Загрузка классов в память.
  - Примеры: System ClassLoader
- Исполняющая среда (Execution Engine):
  - Интерпретация и JIT-компиляция (Just-In-Time Compilation).
- Управление памятью:
  - Стек и куча (Stack и Heap).
  - Garbage Collector (GC) и его основные алгоритмы.

Class loader



Компиляция в байт-код



Этапы загрузки классов

- Загрузка (loading)
- Связывание, линковка (linking)
  - Верификация (verification)
  - Подготовка (preparation)
  - Разрешение (resolution)
- Инициализация (initialization)

Пример

```java
package ru.javarush;

class Calculator {
    public static void main(String[] args){
        int a = Integer.valueOf(args[0]);
        int b = Integer.valueOf(args[1]);

        System.out.println(a + b);
    }
}
```

Компиляция класса в байт-код

```
javac -d bin src/ru/javarush/Calculator.java
```

Запуск скомпилированной программы

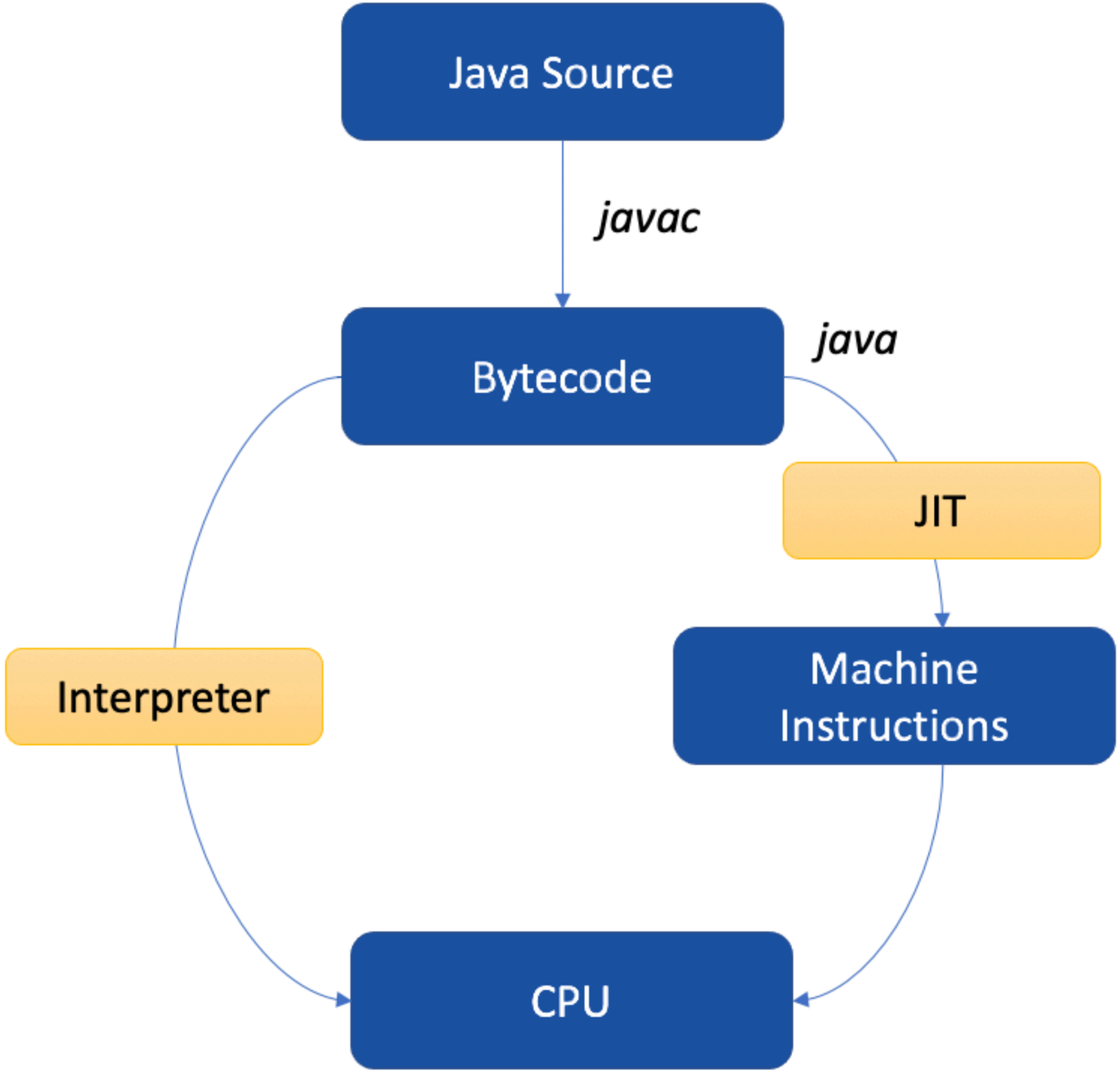
```
java -classpath ./bin ru.javarush.Calculator 1 2
```

```
├── src
│   └── ru
│       └── javarush
│           └── Calculator.java
└── bin
    └── ru
        └── javarush
            └── Calculator.class
```

JIT(Just-in-Time)

5 уровней компиляции:

- 0 — интерпретируемый код
- 1 — C1 с полной оптимизацией (без профилирования)
- 2 — C1 с учетом количества вызовов методов и итераций циклов
- 3 — C1 с профилированием
- 4 — C2



Куча(Heap)

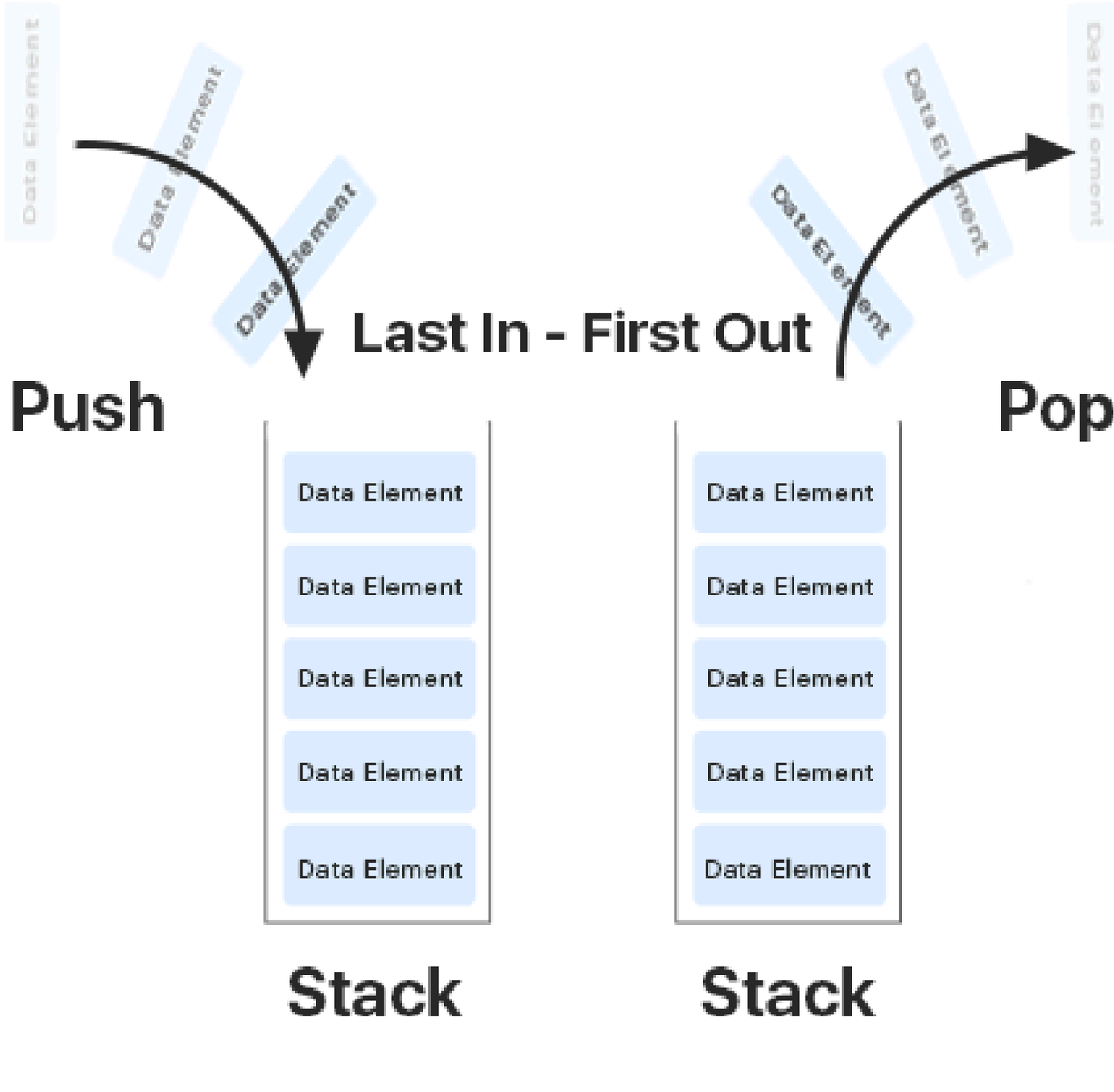
Куча делится на поколения для оптимизации сборки мусора:

- Молодое поколение (Young Generation): Сюда помещаются вновь созданные объекты. Это поколение далее делится на области: Eden Space и две Survivor Spaces (S0 и S1).

- Старое поколение (Old Generation): Содержит объекты, которые долго живут. Объекты перемещаются сюда из молодого поколения после того, как выживают несколько циклов сборки мусора.

Стек(Stack)

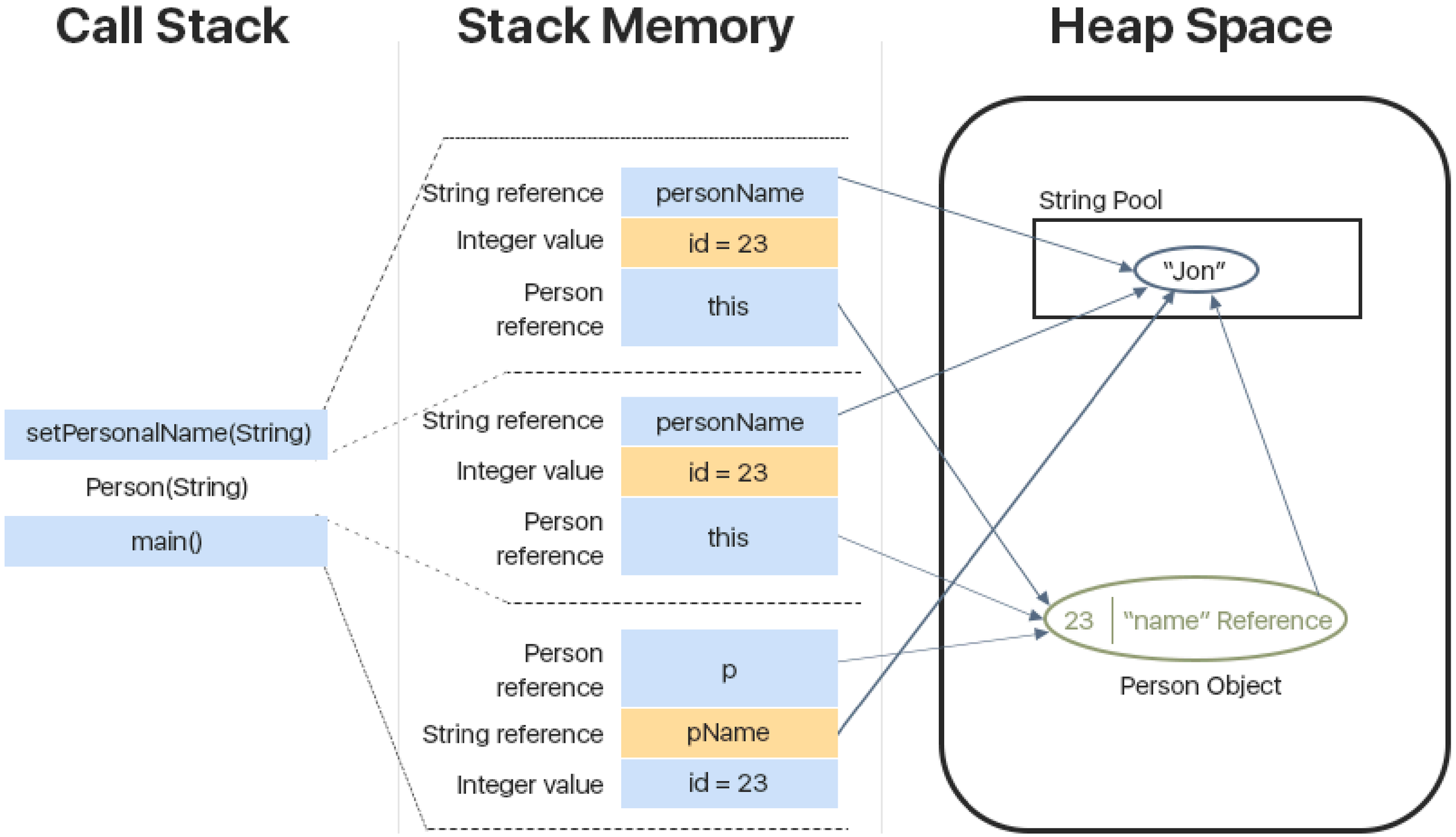
Каждый вызов метода создает новый фрейм стека, который содержит локальные переменные метода, параметры и информацию о выполнении. После завершения метода фрейм стека удаляется, освобождая память.



Пример управления памятью

```
public class App {
    public static void main(String[] args) {
        int id = 23;
        String pName = "Jon";
        Person p = null;
        p = new Person(id, pName);
    }
}
class Person {
    int pid;
    String name;

    // constructors, getters/setters
}
```


Call Stack
Stack Memory
Heap Space
setPersonalName(String)
Person(String)
main()
String reference
personName
Integer value
id = 23
Person reference
this
String reference
personName
Integer value
id = 23
Person reference
this
Person reference
p
String reference
pName
Integer value
id = 23
String Pool
"Jon"
23
"name" Reference
Person Object

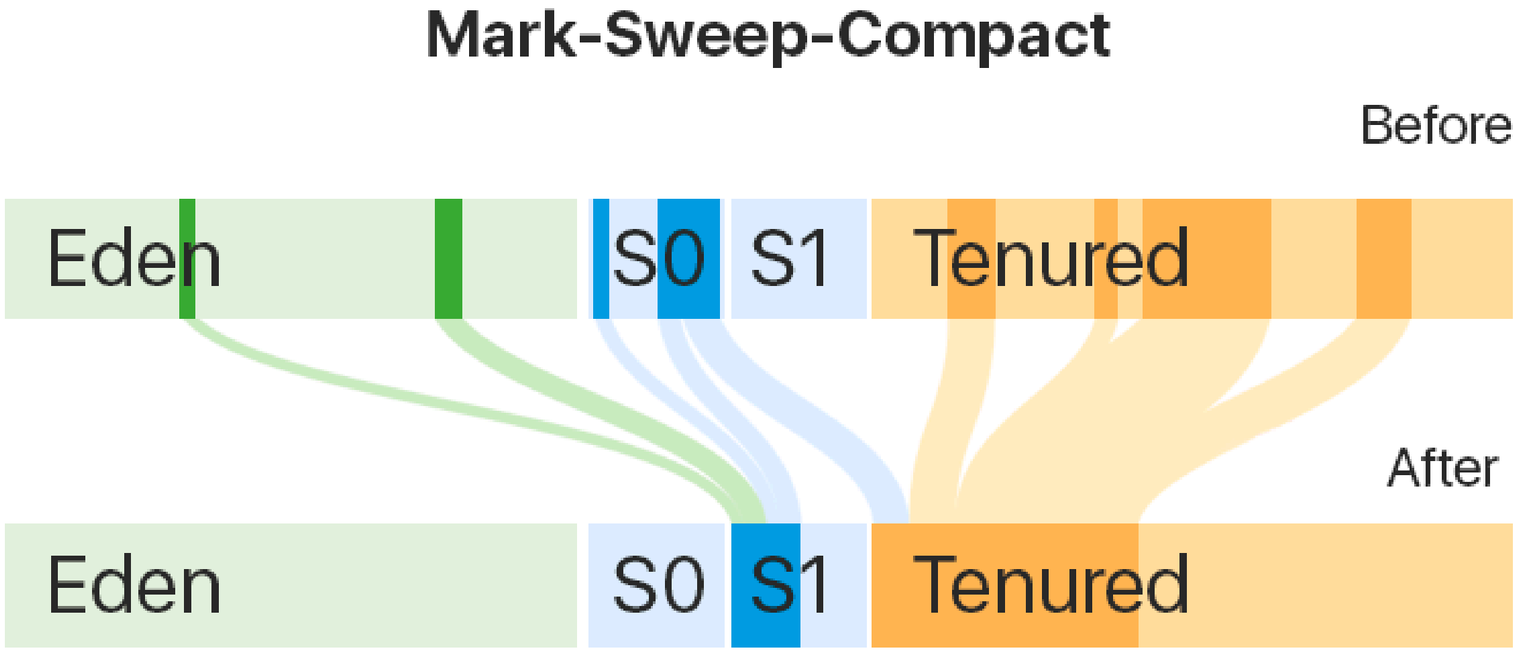
Garbage collector

Сборщик Мусора	Преимущества	Недостатки	Идеален для
Serial GC	Простота, эффективность для малых приложений	Низкая производительность на многопроцессорных системах	Малые приложения, ограниченные ресурсы
Parallel GC	Высокая пропускная способность	Длинные паузы GC	Серверные приложения, многопроцессорные системы
CMS	Низкие паузы GC	Возможная фрагментация памяти, сложность	Интерактивные приложения
G1 GC	Баланс между производительностью и задержкой, масштабируемость	Может требовать тонкой настройки	Большие серверные приложения
ZGC/Shenandoah	Минимальные паузы	Новизна, потенциальные ограничения в использовании	Приложения с требованиями к ультранизкой задержке

Serial GC

Пока на объект есть хотя бы одна сильная ссылка, он остается в памяти

Этот процесс может быть инициирован автоматически или вручную через вызов System.gc()



Заключение

- Компилятор javac преобразует исходный код программы в байт-код, который может быть выполнен на любой платформе, на которой установлена виртуальная машина Java;

- После компиляции JVM интерпретирует получившийся байт-код;

- Для ускорения работы Java-приложений, JVM использует механизм Just-In-Time компиляции, который преобразует наиболее часто выполняемые участки программы в машинный код и хранит их в памяти.

- JVM избавляется от объектов, к которым невозможно получить доступ с помощью garbage collector

Спасибо за внимание!