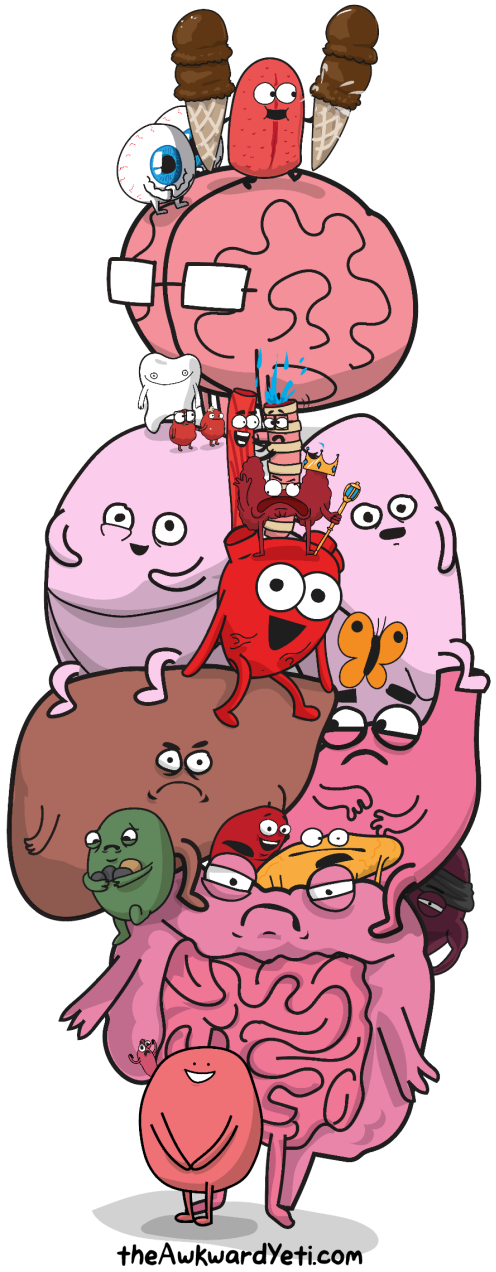
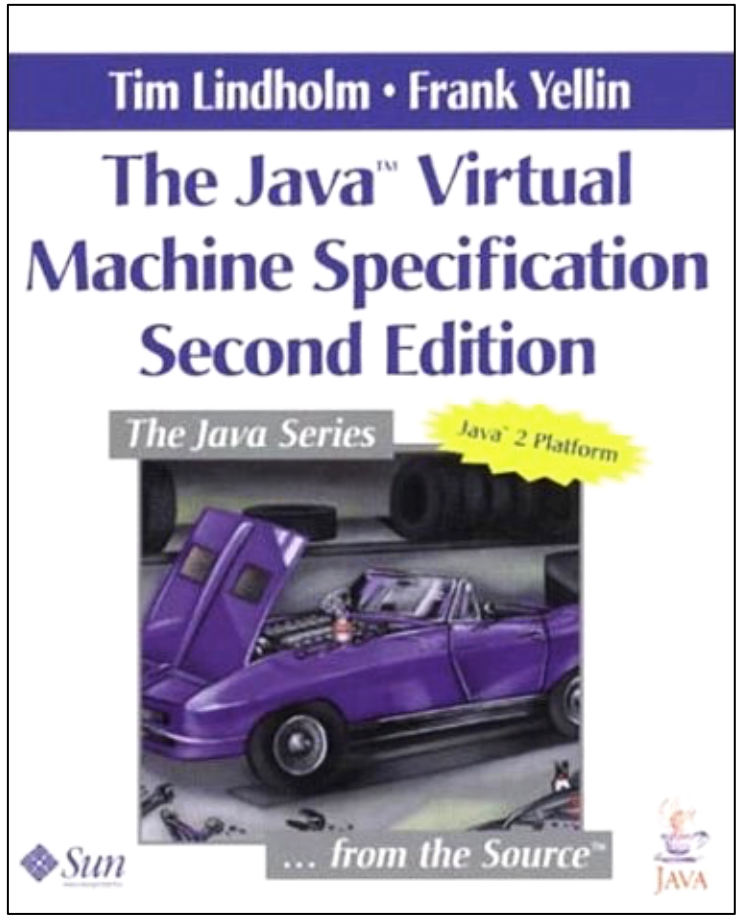
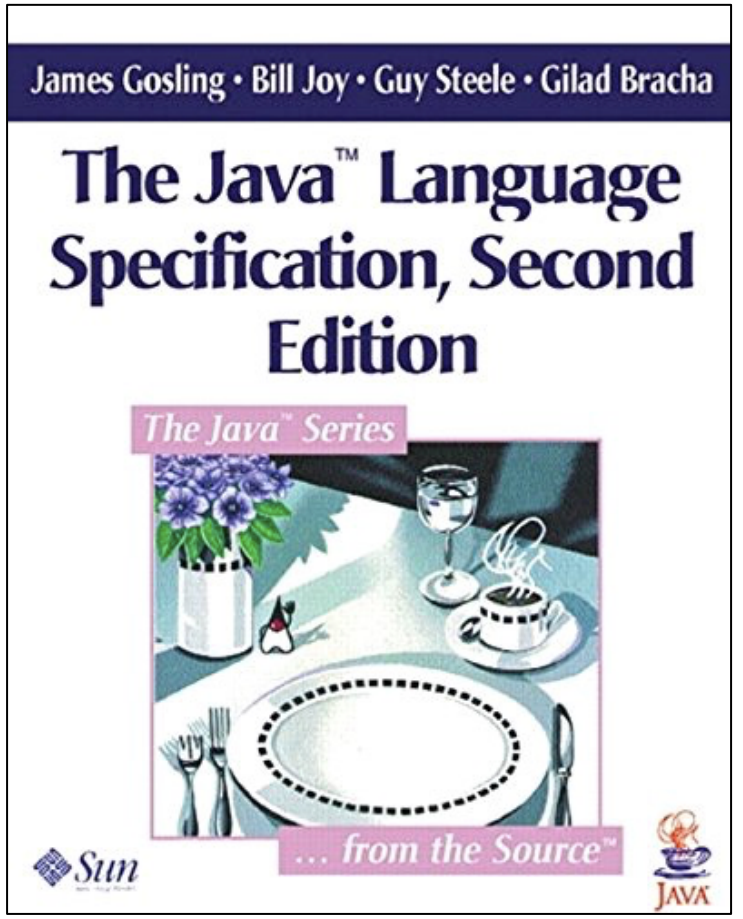


JVM: краткий курс общей анатомии

Никита Липский, Excelsior LLC

Владимир Иванов, Oracle Corp.


Tim Lindholm • Frank Yellin
The Java™ Virtual Machine Specification Second Edition
The Java Series
Java 2 Platform
Sun
... from the Source™
JAVA


James Gosling • Bill Joy • Guy Steele • Gilad Bracha
The Java™ Language Specification, Second Edition
The Java™ Series
Sun
... from the Source™
JAVA

EXCELSIOR

Excelsior JET.

The Java™ Real Machine.

EXCELSIOR

Никита Липский

- Более 20 лет профессиональной карьеры
- Инициатор проекта Excelsior JET
  - работал над проектом более 16 лет
  - как идейный вдохновитель
  - как компиляторный инженер
  - как руководитель
  - и много в каких еще ролях
- Open source проекты WebFX и Java ReStart
  - в свободное от работы время
- Twitter: @pjBooms

EXCELSIOR

Владимир Иванов

- Написал первую Java программу в 2002
- С 2005 работает в компании Sun/Oracle
- Инженер HotSpot JVM Compiler
  - JIT-компиляторы
  - Поддержка динамических языков на JVM
  - Работа с платформенным кодом
- Twitter: @iwan0www

EXCELSIOR

План доклада

- Java class file and bytecode
- Classloading engine
- Execution engine: interpretators, JIT, AOT
- Meta information access subsystem: reflection, indy, JNI
- Threading, exception handling, synchronization
- Memory management: heap, allocation, GC
- Manageability and Monitoring

Java class file & bytecode

EXCELSIOR

Java class file

- 1 класс <–> 1 класс-файл
- Constant Pool
  - числа, строки
  - указатели на классы, методы, поля
- Описание класса
  - имя
  - модификаторы
  - супер класс
  - супер интерфейсы
  - поля
  - методы
  - атрибуты

EXCELSIOR

Java class file

- Поля, методы тоже имеют атрибуты (например, значения константных полей)

- Главный атрибут метода – это его код: Java байт-код

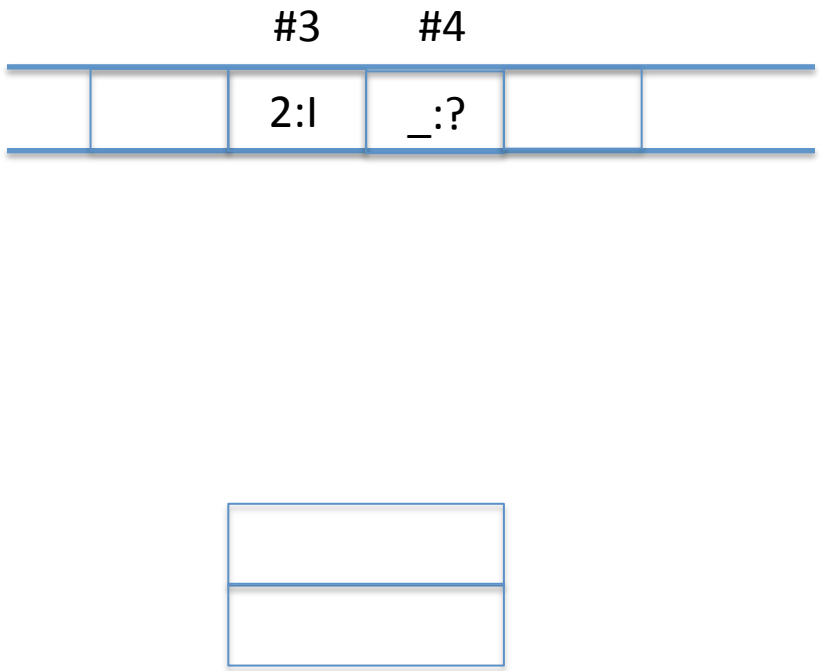
Java bytecode

- Массив инструкций
- Стэк операндов инструкций метода
- Массив локальных переменных (аргументы метода, локальные переменные)

Java bytecode

Инструкция берет свои операнды со стэка и кладет результат на стэк. Пример:

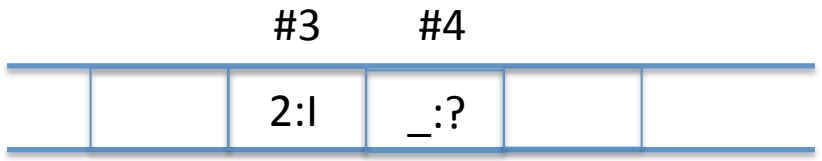
```
0: iload 3
2: bipush 5
4: iadd
5: istore 4
7: …
```

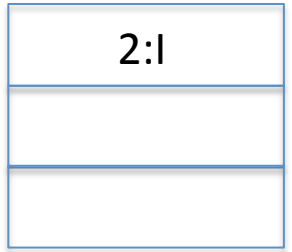


Java bytecode

Инструкция берет свои операнды со стэка и кладет результат на стэк. Пример:

```
0: iload 3
2: bipush 5
4: iadd
5: istore 4
7: …
```

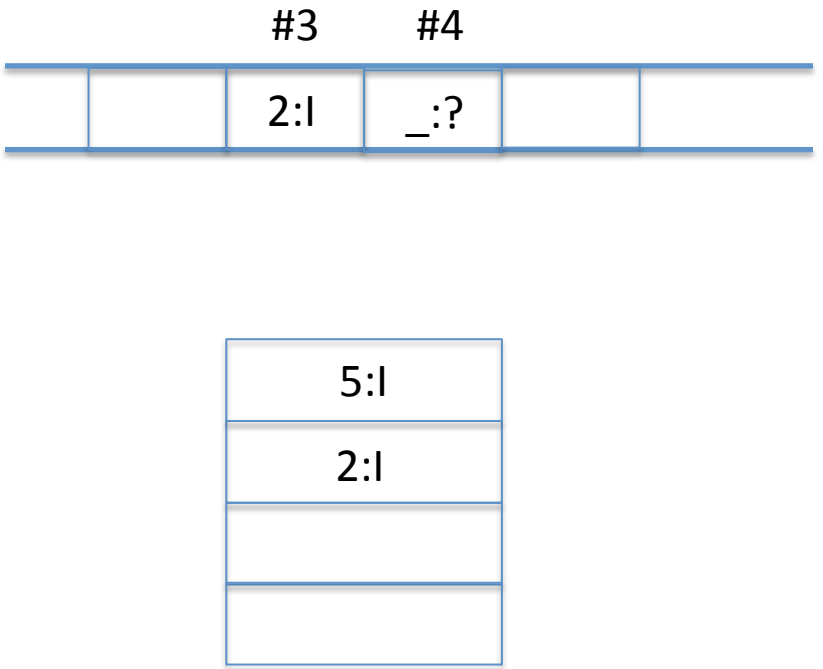




Java bytecode

Инструкция берет свои операнды со стэка и кладет результат на стэк. Пример:

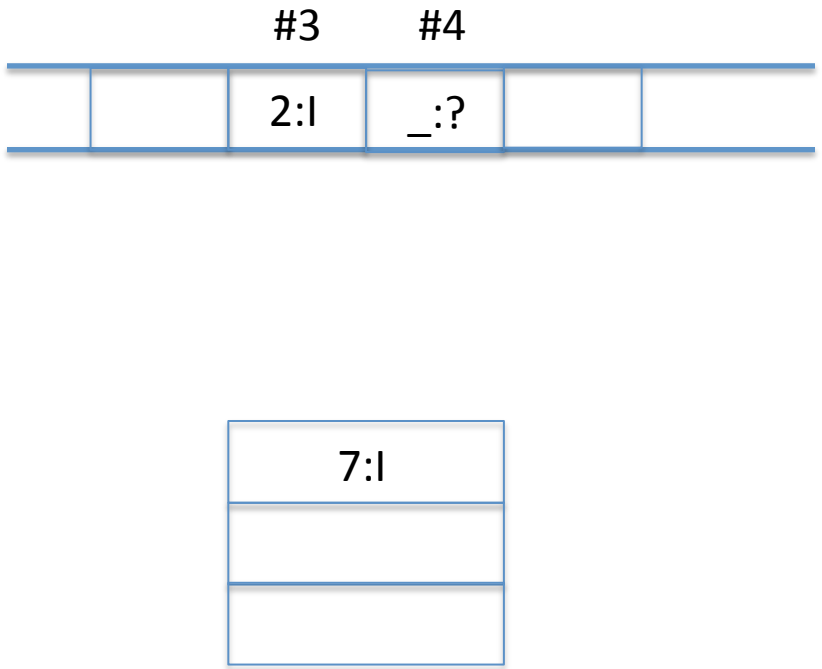
```
0: iload 3
2: bipush 5
4: iadd
5: istore 4
7: …
```



Java bytecode

Инструкция берет свои операнды со стэка и кладет результат на стэк. Пример:

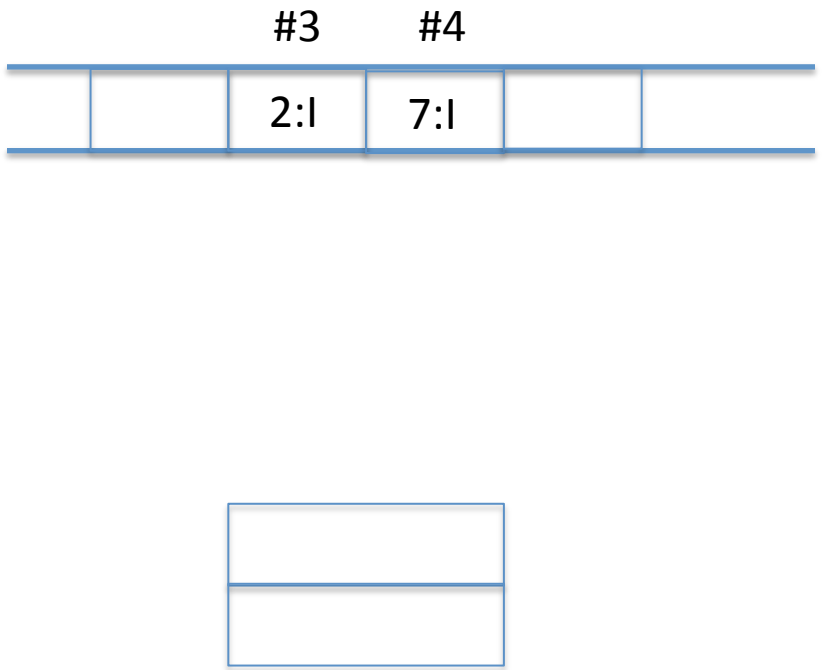
```
0: iload 3
2: bipush 5
4: iadd
5: istore 4
7: …
```



Java bytecode

Инструкция берет свои операнды со стэка и кладет результат на стэк. Пример:

```
0: iload 3
2: bipush 5
4: iadd
5: istore 4
7: …
```



JVM: предисловие

EXCELSIOR

Программа для JVM

Любая программа исполняемая на JVM имеет?

Программа для JVM

Любая программа исполняемая на JVM имеет:

- main класс

```
public static void main(String[] args)
```

- classpath – список директорий и архивов (jar файлов)

- В мире веб приложений программа для JVM – это веб сервер
  - Tomcat, GlassFish и т.п.

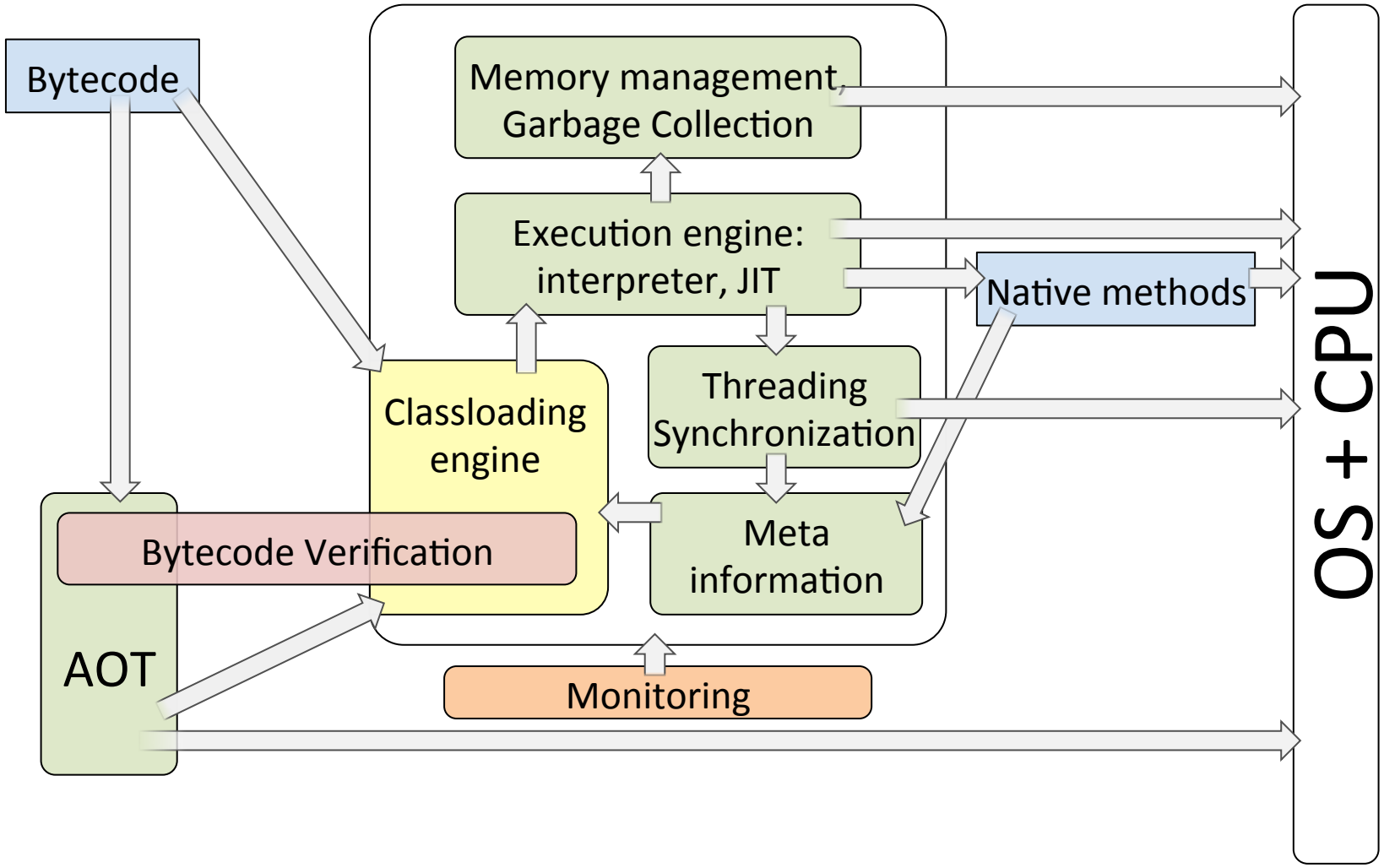
EXCELSIOR

Java Runtime

Для исполнения программы на JVM одной JVM не достаточно. Нужен Java Runtime Environment:

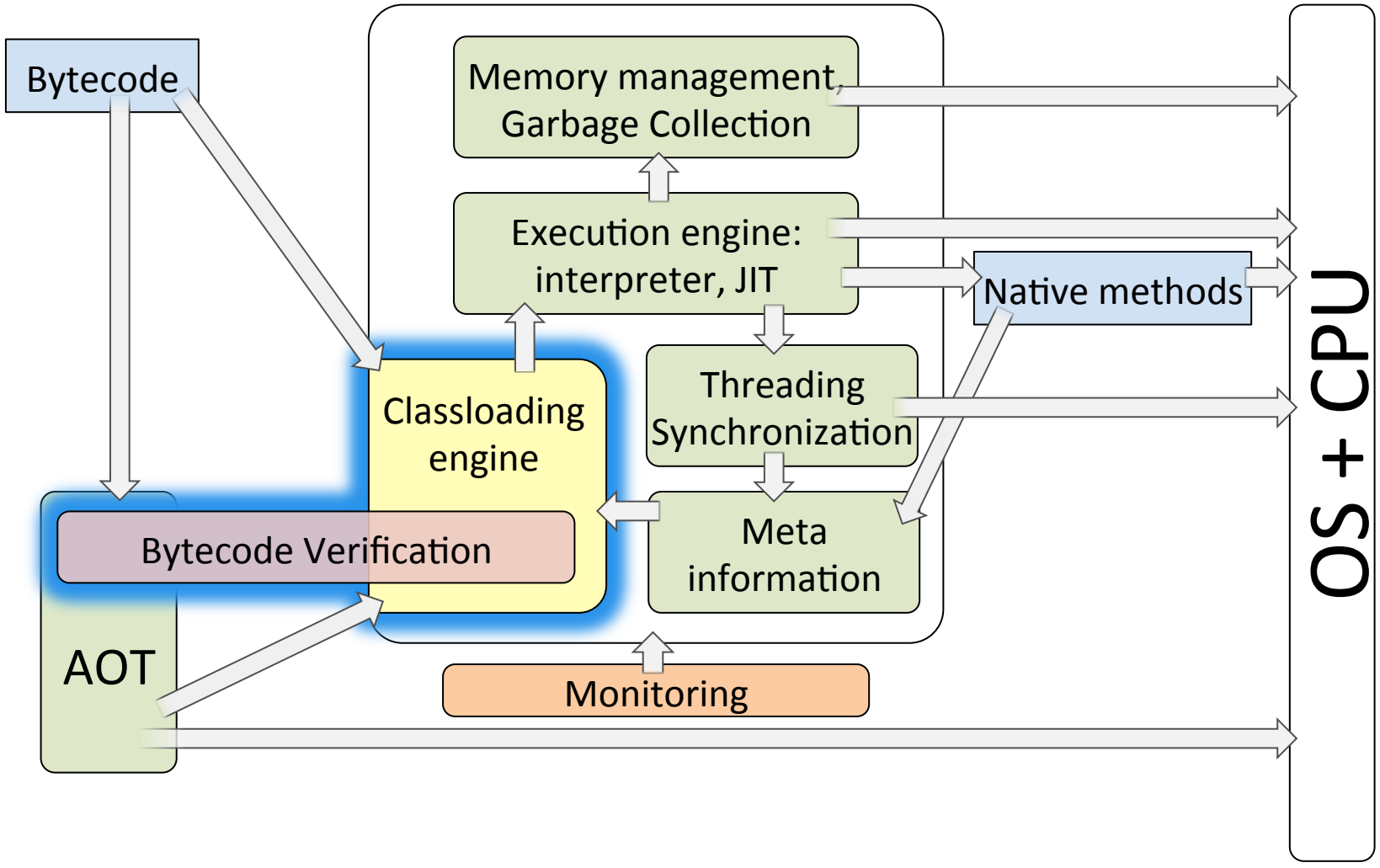
- JVM
- Платформенные классы
  - core классы (j.l.Object, j.l.String и т.д.)
  - Java standard APIs (IO, NET, NIO, AWT/Swing и т.д)
- Реализация нативных методов платформенных классов (OS specific)
- Вспомогательные файлы (описатели временных зон, медиа ресурсы и т.п.)

Анатомия JVM



EXCELSIOR

Classloading engine



EXCELSIOR

Загрузка классов

- Где JVM берет классы для исполнения:
  - Из Java Runtime (платформенные классы)
  - Из classpath приложения
  - Авто-сгенеренные на лету (Proxy, Reflection accessors, реализация invoke dynamic)
  - Предоставленные самим приложением

Загрузка классов

- Каждый класс грузится каким-то загрузчиком классов:
  - Платформенные классы грузятся bootstrap загрузчиком
  - Классы из classpath приложения – системным загрузчиком (AppClassLoader)
  - Классы приложения могут создавать свои загрузчики, которые будут грузить классы

Загрузка классов

- Каждый класс грузится каким-то загрузчиком классов:
  - Платформенные классы грузятся bootstrap загрузчиком
  - Классы из classpath приложения – системным загрузчиком (AppClassLoader)
  - Классы приложения могут создавать свои загрузчики, которые будут грузить классы
- Загрузчик классов образует уникальное пространство имен классов

EXCELSIOR

Старт JVM

- Грузится main класс системным загрузчиком (из classpath приложения)
  - Провоцирует загрузку части платформенных классов (core)

- Исполняется метод main(String[] args)

Процесс загрузки класса (создание класса)

- Читается class file
  - Проверяется корректность формата (может выбросить ClassFormatError)
- Создается ран-тайм представление класса в выделенной области памяти
  - runtime constant pool in Method Area aka **Meta Space** aka Permanent Generation
- Грузятся суперкласс, суперинтерфейсы

Линковка

- Верификация байт-кода
- Подготовка
- Разрешение символьных ссылок

```
0: bipush 2
2: bipush 2
4: iadd
5: goto 0
```

Что произойдет при исполнении этого байт-кода?

A: Программа зациклится

B: VerifyError

C: StackOverflowError

D: Байт-код не исполнится

Верификация байт-кода

- Происходит с классом один раз
- Проверка корректности инструкций (корректности переходов)
- Проверка выхода за пределы стэка операндов и локальных переменных
- Проверка совместимости типов

EXCELSIOR

Ну а если все-таки запустить без верификации?

```
$ java –Xverify:none test
```

```
# A fatal error has been detected by the Java Runtime Environment:
Exception in thread "main" java.lang.StackOverflowError
        at test.foo(test.j)
        at test.main(test.j:3)
#
#  Internal Error (javaCalls.cpp:53), pid=8012, tid=16396
#  guarantee(!thread->is_Compiler_thread()) failed: cannot make java calls
from the compiler
#
# If you would like to submit a bug report, please visit:
#   http://bugreport.java.com/bugreport/crash.jsp
```

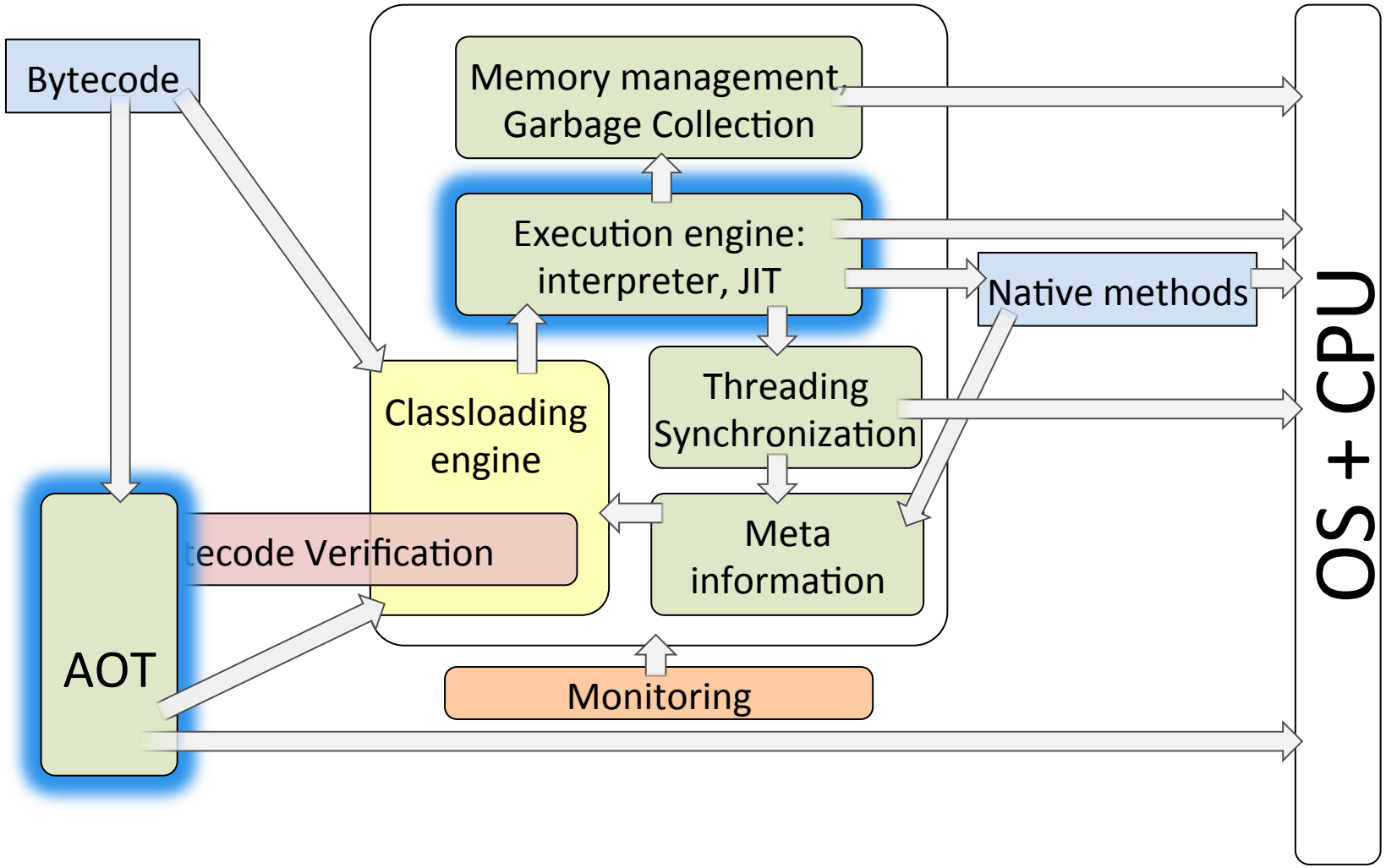
EXCELSIOR

Инициализация класса

- Вызов статического инициализатора класса
- Случается при first use:
  - new
  - доступ до статического поля
  - вызов статического метода
- Провоцирует инициализацию супер-класса и супер-интерфейсов с default методами

EXCELSIOR

Execution engine



EXCELSIOR

Исполнение Java байт-кода

JVM может исполнять байт-код двумя способам:

- Интерпретировать
- Транслировать в машинный код, который будет исполняться непосредственно на CPU

Интерпретатор

```
pc = 0;
do {
    fetch opcode at pc;
    if (operands) fetch operands;
    execute the opcode;
    calculate pc;
} while (there is more to do);
```

EXCELSIOR

Компиляторы

- Неоптимизирующие
  - “что вижу, то пою”
- Простые оптимизирующие
  - пример: HotSpot Client
- Сложные оптимизирующие
  - пример: HotSpot Server

Компиляторы

- Динамические (Just-In-Time – **JIT**).
  - Трансляция в машинный код происходит во время исполнения программы

- Статические (Ahead-Of-Time – **AOT**)
  - Трансляция происходит до исполнения программы

Динамические компиляторы (JIT)

- Работают одновременно с исполняемой программой
- Компилируют **горячий** код
- Горячий код вычисляется с помощью динамического профилировщика
- Используют информацию времени исполнения для оптимизаций

EXCELSIOR

Статические компиляторы (AOT)

- Не ограничены в ресурсах для оптимизации программ
- Компилируют каждый метод программы применяя самые агрессивные оптимизации
- На оптимизацию не тратятся ресурсы во время исполнения программы (быстрее старт)

EXCELSIOR

Статические компиляторы (AOT)

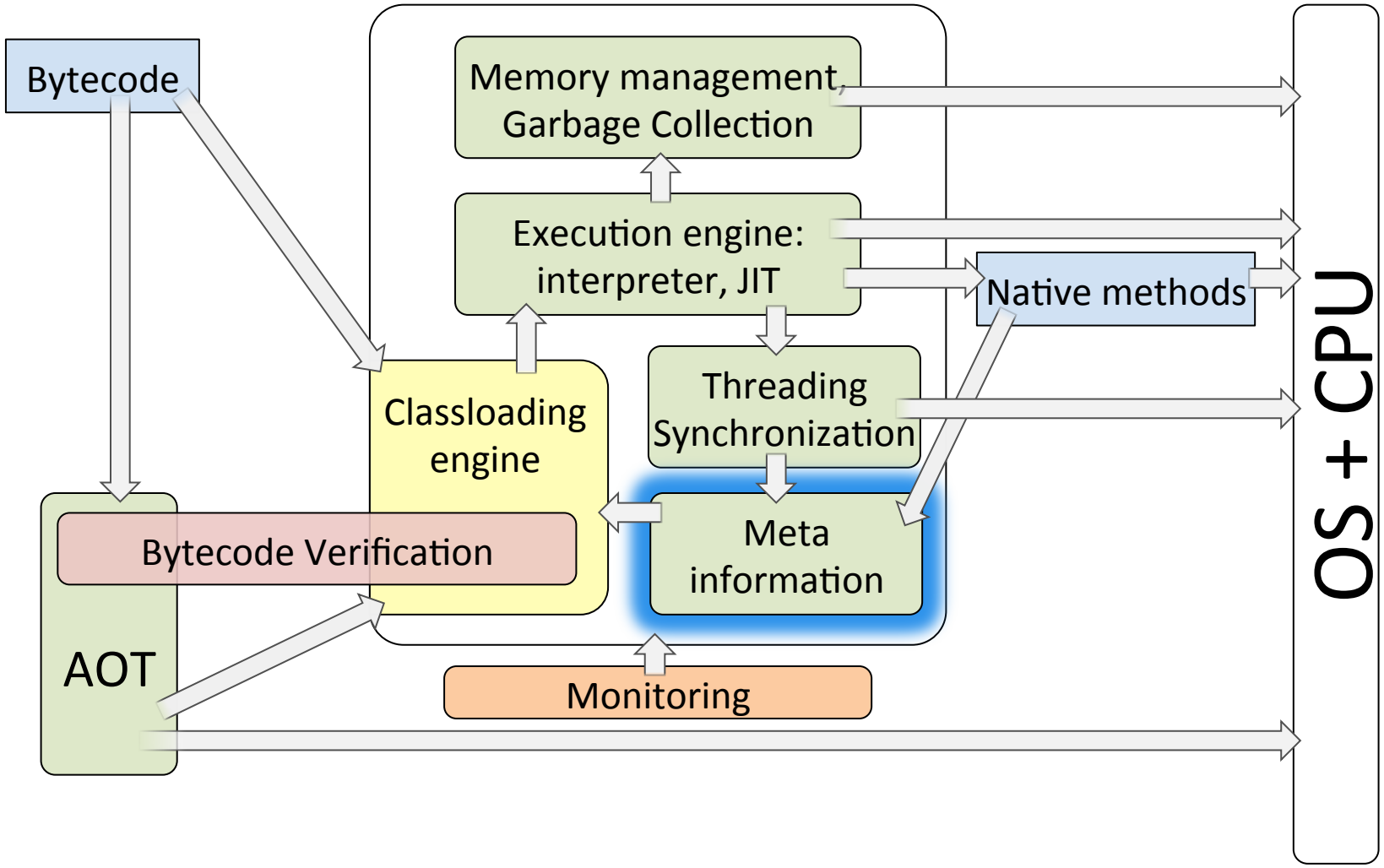
Интересный факт:

Во время работы статически скомпилированной Java программы может не исполниться ни одного Java байт-кода.

Вопрос: *а где JVM?*

EXCELSIOR

Meta information subsystem



EXCELSIOR

Reflection

- Позволяет доступаться до классов, полей, методов по имени из Java программы
- Реализуется в JVM через доступ в Meta space
- Ключевая возможность Java для многих популярных фреймворков и реализаций языков на JVM

Method Handles and invokedynamic (JSR-292, indy)

- Indy: программируемый вызов
  - для эффективной реализации динамических языков на JVM

- MethodHandle – целевой объект вызова через invokedynamic
  - может быть доступом к полю, методу
  - комбинацией других MethodHandle
  - может использоваться отдельно от indy: Reflection 2.0

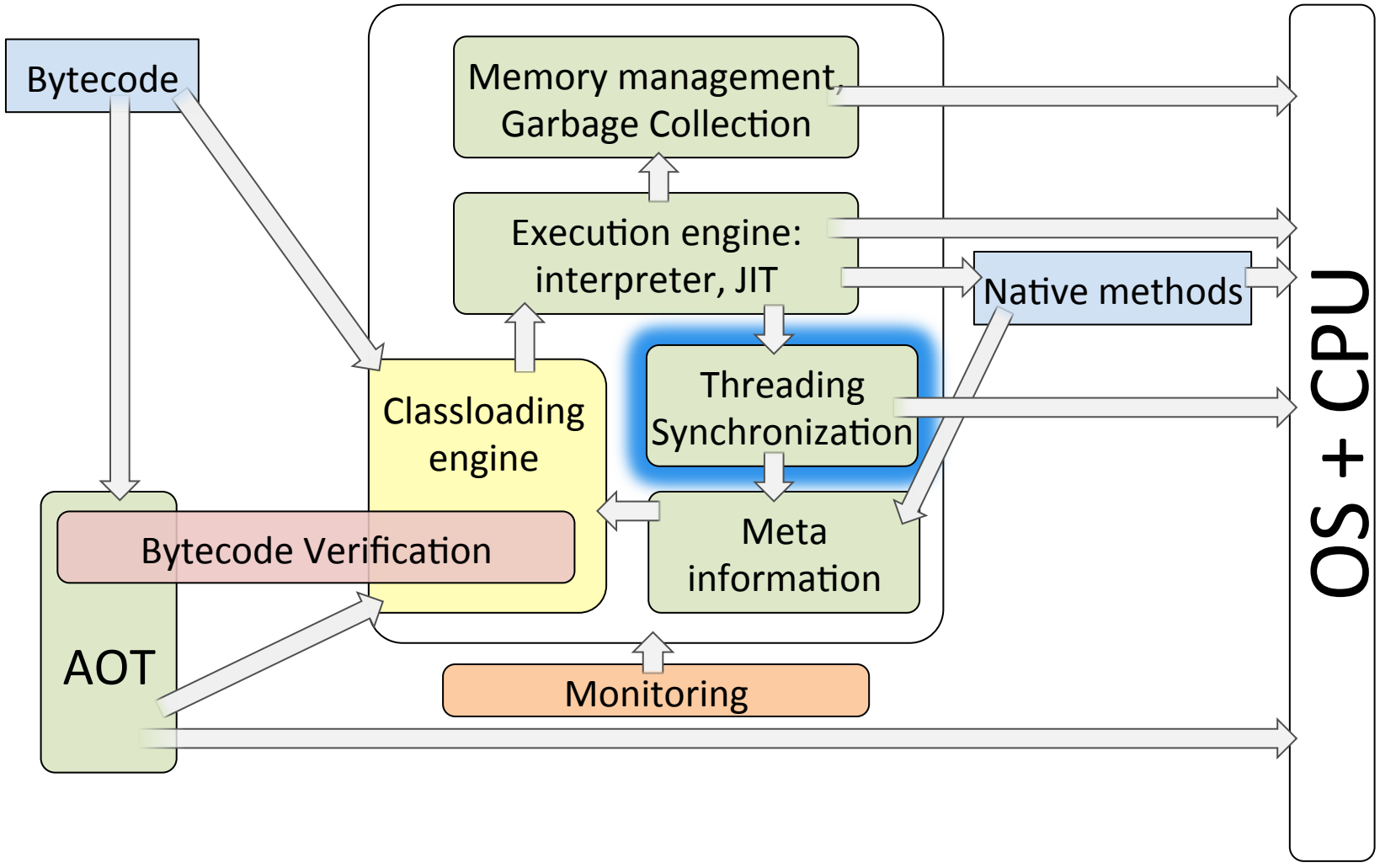
EXCELSIOR

Java native interface (JNI)

- Связывает JVM с внешним миром (OS)
- Си интерфейс к JVM
  - Не зависит от реализации JVM
  - Используется для реализации native методов на языке C (или другом системном языке)
  - С помощью JNI написаны платформенно-зависимые реализации Java SE API: IO, NET, AWT
- Реализуется в JVM как доступ к Meta space

EXCELSIOR

Threading and synchronization



EXCELSIOR

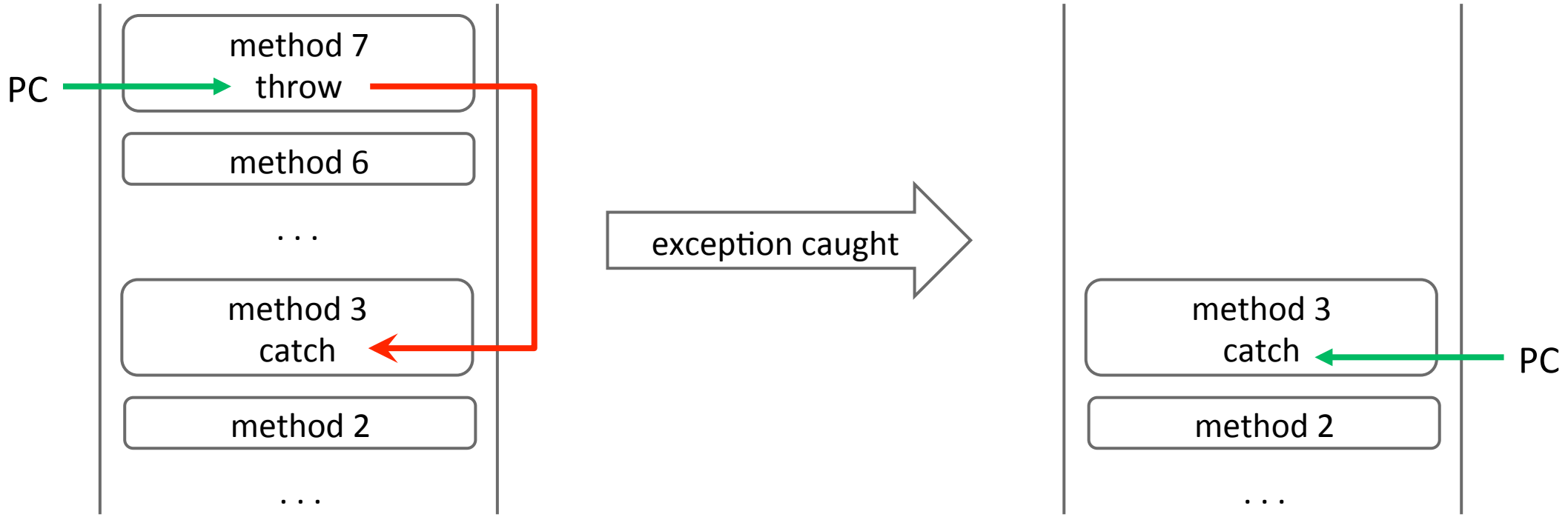
java.lang.Thread

- Java поток мапируется на нативный поток в соотношение 1-1
- С потоком связана память используемая для локальных переменных и стэка операндов методов (фреймов методов): стэк (stack)
  - Размер стэка – параметр JVM: -Xss
- Имеет информацию о стэке вызовов методов потока (stack trace)
  - В любой момент может о нем рассказать

EXCELSIOR

Обработка исключений

Знание о стэке вызовов помогает в обработке исключений:



Потоки и Java Memory Model

```
// Thread 1:
Shared.data = getData();

Shared.ready = true;
```

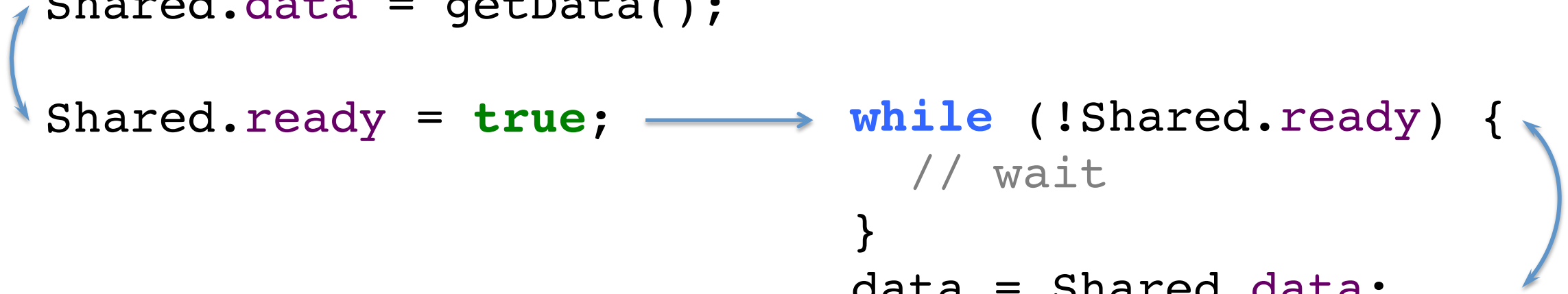
```
// Thread 2:


while (!Shared.ready) {
   // wait
}
data = Shared.data;
```

EXCELSIOR

Потоки и Java Memory Model

```
// Thread 1:
Shared.data = getData();

Shared.ready = true;
```

```
// Thread 2:

while (!Shared.ready) {
   // wait
}
data = Shared.data;
```

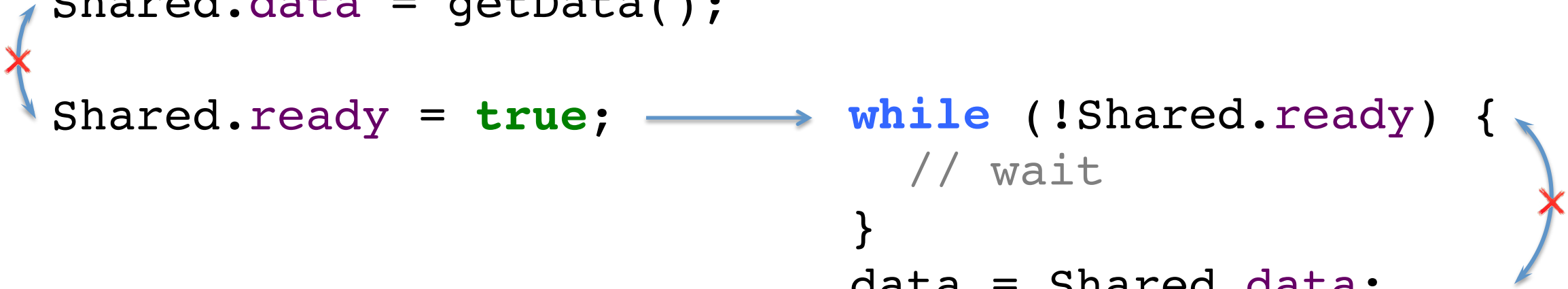


EXCELSIOR

Потоки и Java Memory Model

```
// Thread 1:
Shared.data = getData();

Shared.ready = true;
```

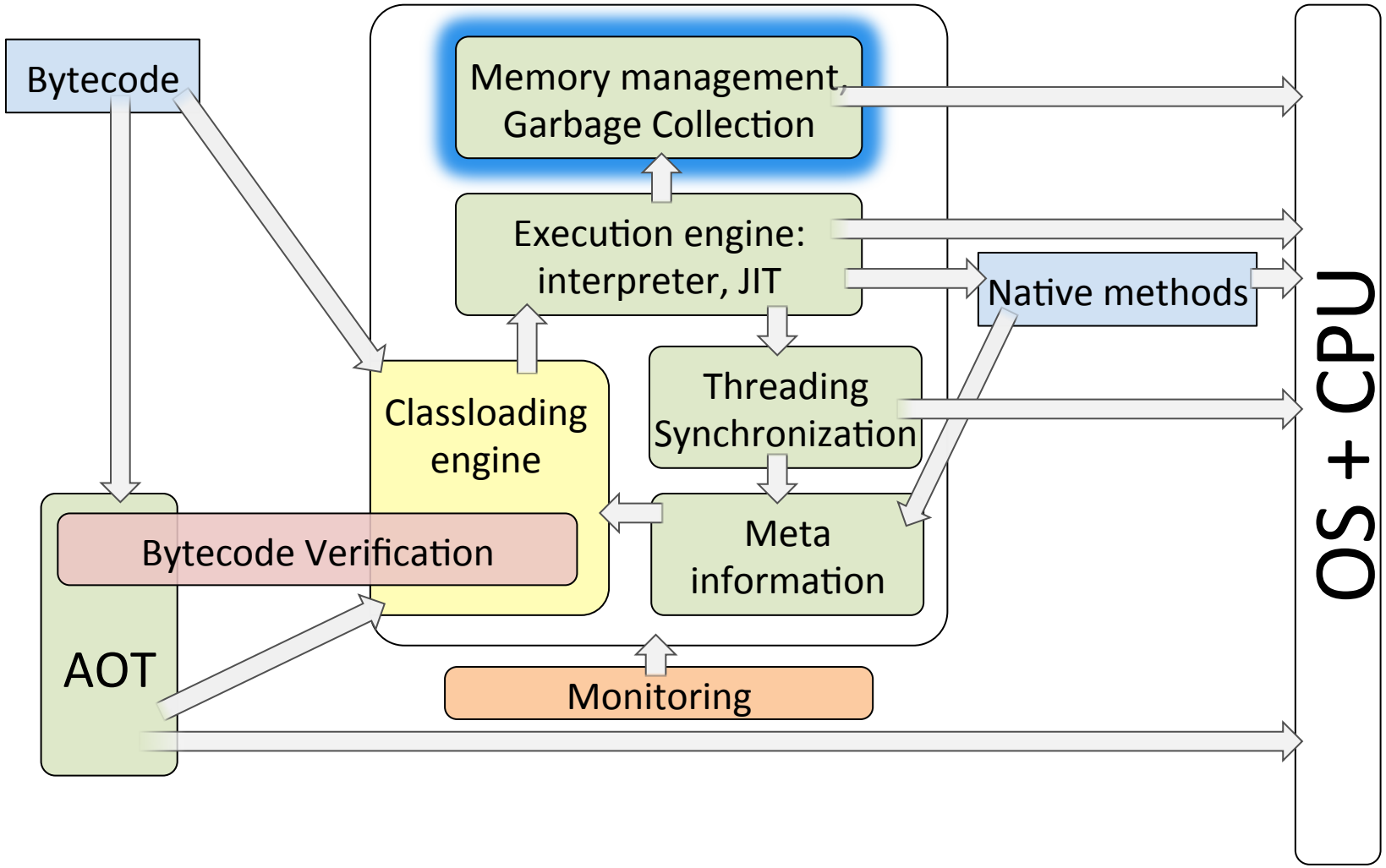
```
// Thread 2:

while (!Shared.ready) {
    // wait
}
data = Shared.data;
```



Синхронизация

- Для безопасного доступа к разделяемой памяти между потоками
- В наивной реализации используются средства ОС
  - ОС монитор есть в каждом Java объекте как скрытое поле
- Оптимизирована когда конкуренция за ресурс происходит много реже, чем вход в synchronized

Memory management



EXCELSIOR

Выделение памяти

- Реализация оператора **new**

- Объекты выделенные с помощью оператора **new** располагаются в т.н. куче (Java heap)
- Организация Java heap JVM-специфична
- Разметка (layout) Java объекта тоже JVM специфична.

Аллокация объектов

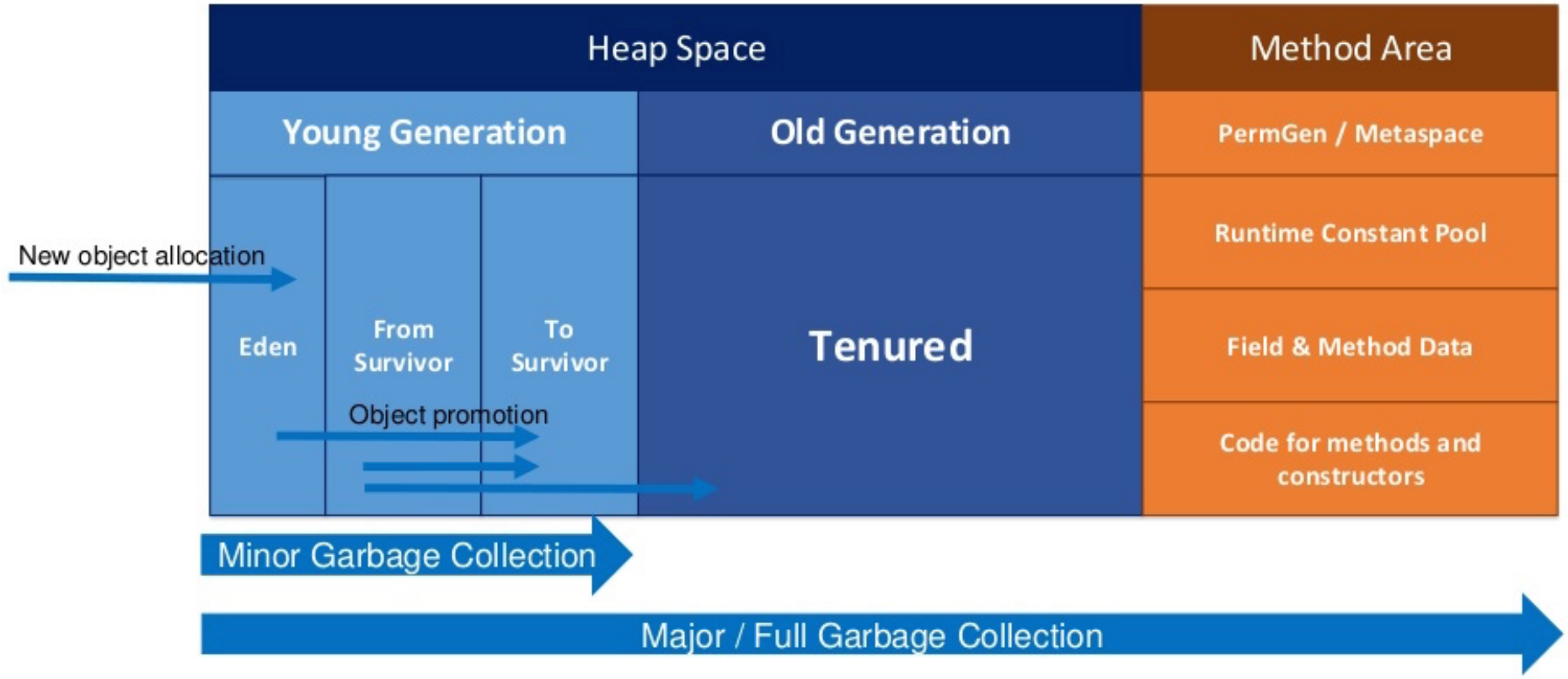
- Должна быть быстрой
  - JVM запрашивает у ОС память не под один объект, а сразу на много
  - Аллокация методом продвижения границы
- Потоко-безопасной (thread-safe), но при этом параллельной (не блокирующей)
  - Thread local heaps: каждый поток “грызет” свой кусок памяти

Layout Java объекта

Не специфицируется JVM, но по факту требует:

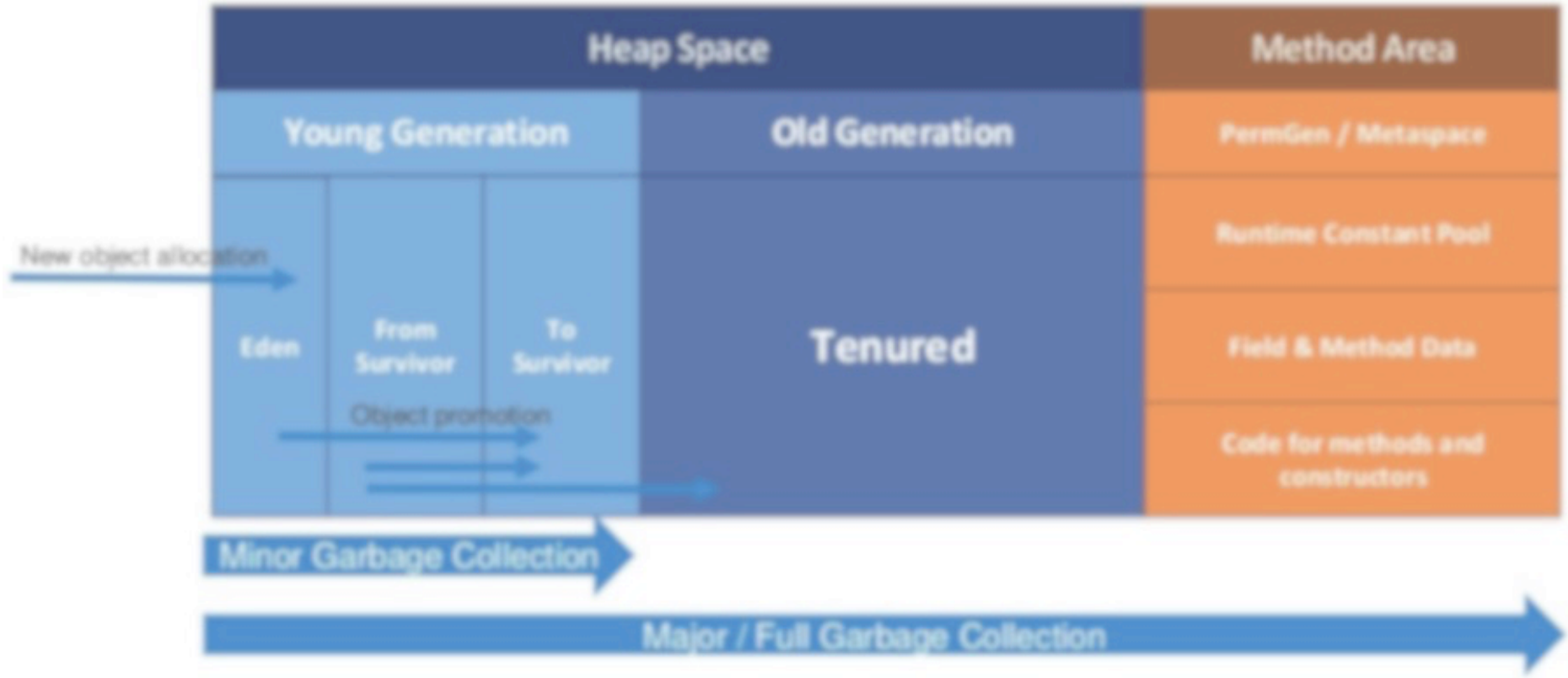
- Java Object header
  - Указатель на класс
  - Монитор (lock)
  - Identity hashcode
  - Флаги для GC
- Поля
  - Могут быть переупорядочены из соображений экономии размера, выравнивания, особенностей целевой архитектуры

Сборка мусора



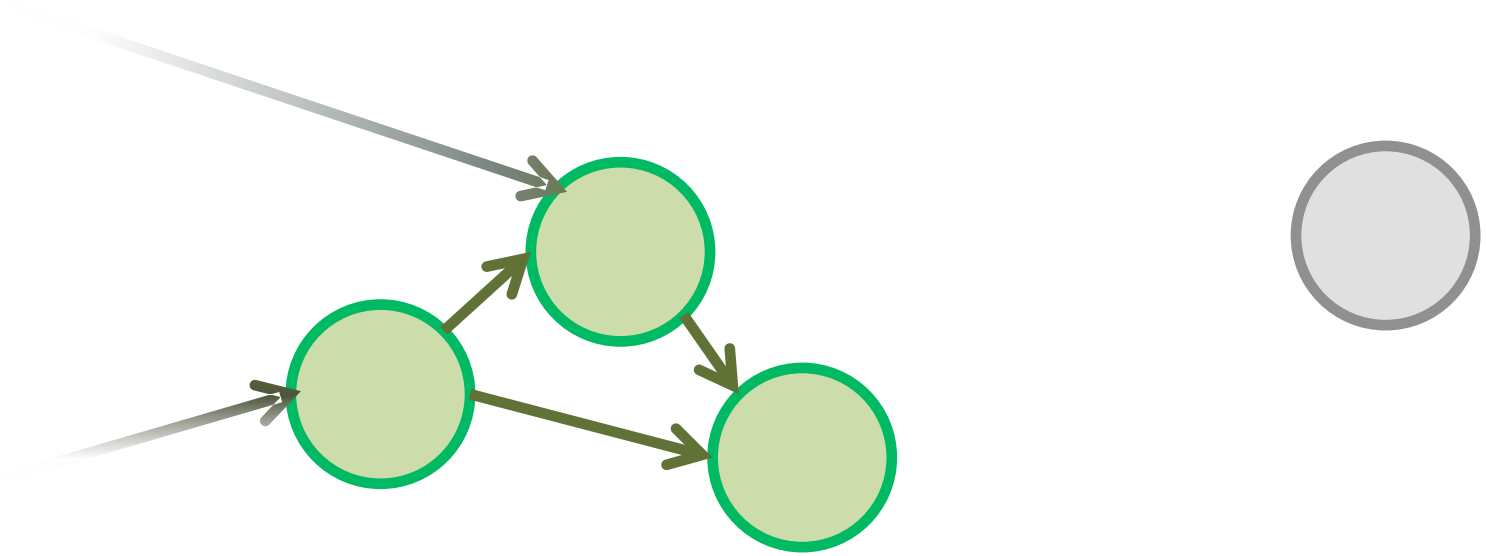
EXCELSIOR

Сборка мусора



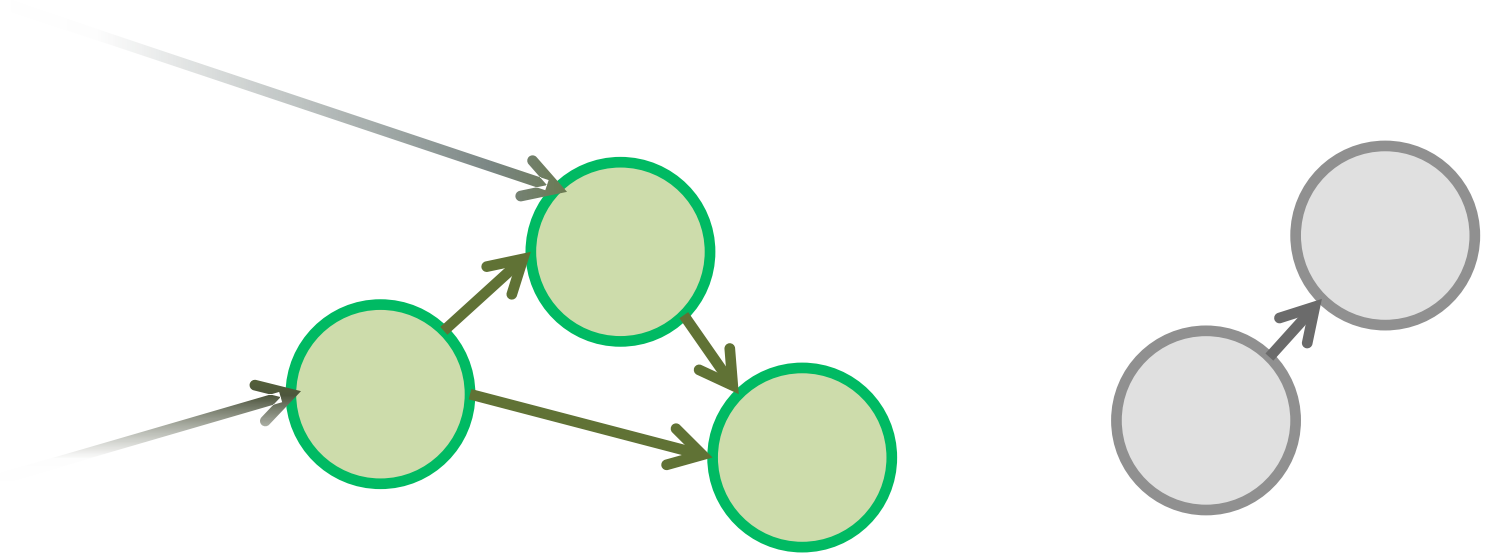
EXCELSIOR

Что такое мусор?



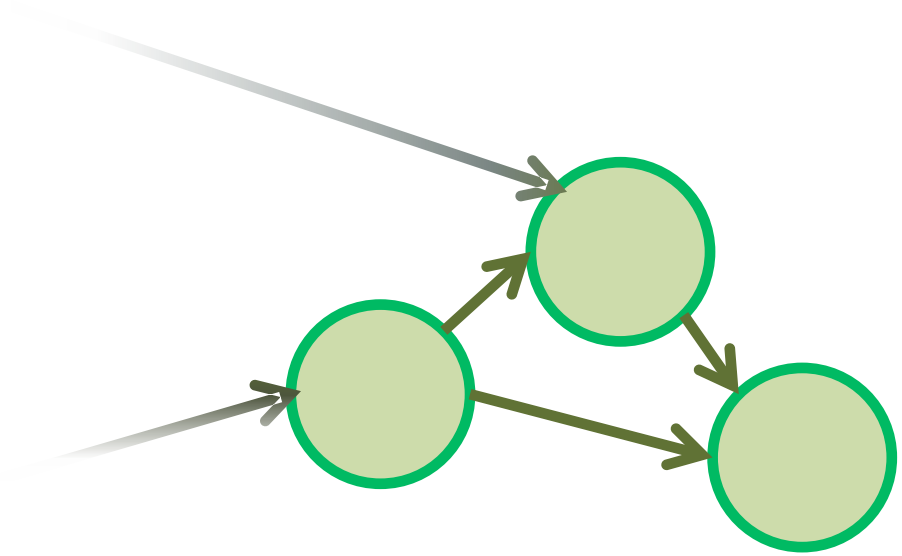
EXCELSIOR

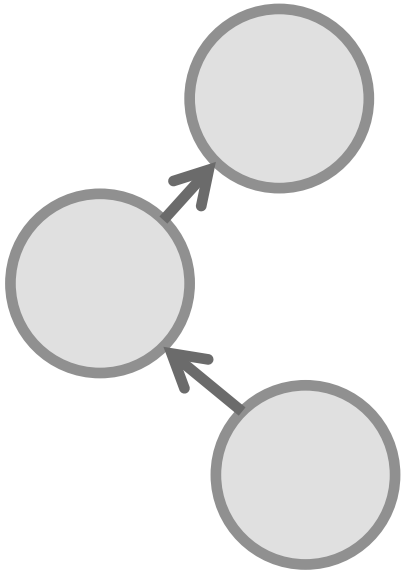
Что такое мусор?



EXCELSIOR

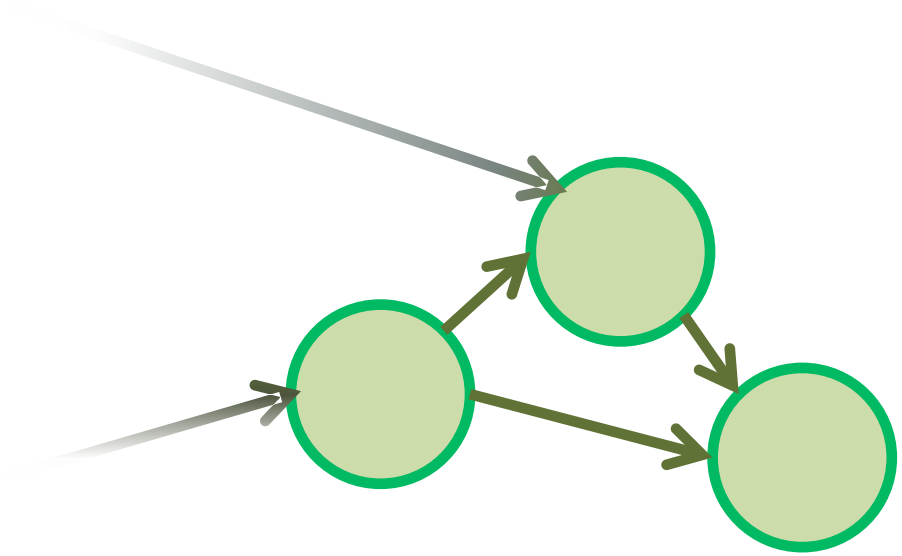
Что такое мусор?

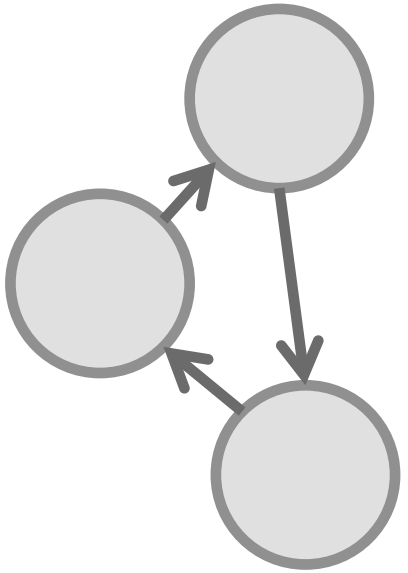




EXCELSIOR

Что такое мусор?





EXCELSIOR

Мусор

Мусором являются объекты, которые не могут использоваться программой

Вопрос: А какие объекты могут использоваться?

Мусор

Мусором являются объекты, которые не могут использоваться программой

Вопрос: А какие объекты могут использоваться?

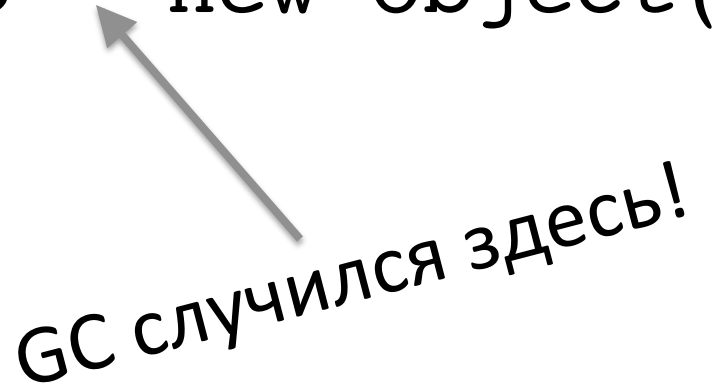
Ответ: Не мусор!

Не мусор

1. Объекты в статических полях классов
2. В локальных переменных
3. … всё?

Не мусор

```
Object o = new Object();
...
```



EXCELSIOR

Не мусор 2

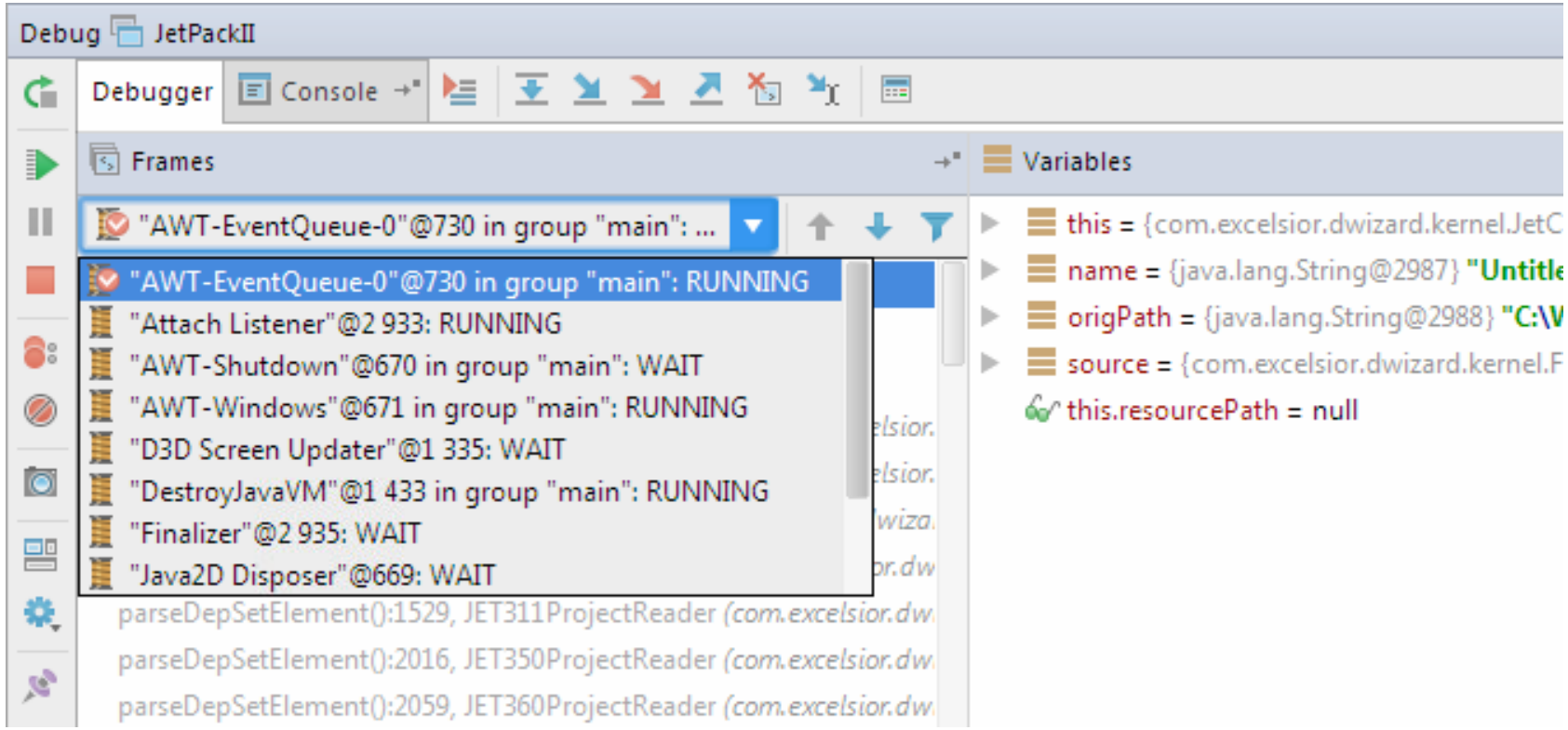
1. Объекты в статических полях классов

2. ~~В локальных переменных~~
   Доступных из фрейма метода (локальные переменные и стэк операндов)

Какого метода?

EXCELSIOR

Не мусор



EXCELSIOR

Не мусор 2 возвращается

1. Объекты в статических полях классов

2. ~~В локальных переменных~~
   Доступных из ~~фрейма метода~~ фреймов методов (локальные переменные и стэк операндов) стэка вызовов всех Java потоков.

3. Объекты, на которые ссылается “не мусор”

EXCELSIOR

Корневое множество объектов (GC roots)

1. Объекты в статических полях классов
2. Объекты доступные со стэка Java потоков
3. Объекты из JNI ссылок в native методах

EXCELSIOR

Не мусор 3

Не мусор aka *живые объекты* – это:

1. Объекты из корневого множества

2. Объекты, на которые ссылаются живые объекты

Все остальное – мусор.

Трассирующие сборщики

- Mark-and-sweep
  - Помечает живые объекты (mark), “выметает” (удаляет) мусор (sweep)

- Stop-and-copy
  - Копирует живые объекты в специальное место (copy)
  - Освободившиеся место (мусор и места где были живые объекты) может использоваться для новой аллокации

Stop the World

- Живые объекты определены для определенного момента исполнения программы
  - При исполнении множество меняется

- Чтобы собрать мусор в общем случае нужно остановить потоки, чтобы определить где мусор (*STW пауза*)

Stop the World

Одна из основных задач современных сборщиков мусора – это уменьшение времени STW паузы. Методы уменьшения:

- Инкрементальный
  - собирать не весь мусор в паузе
- Параллельный
  - собирать мусор во многих потоках в паузе
- Одновременный (concurrent)
  - собирать мусор одновременно с работой программы (не останавливая потоки)

EXCELSIOR

Поколенная сборка мусора

Слабая гипотеза о поколениях:

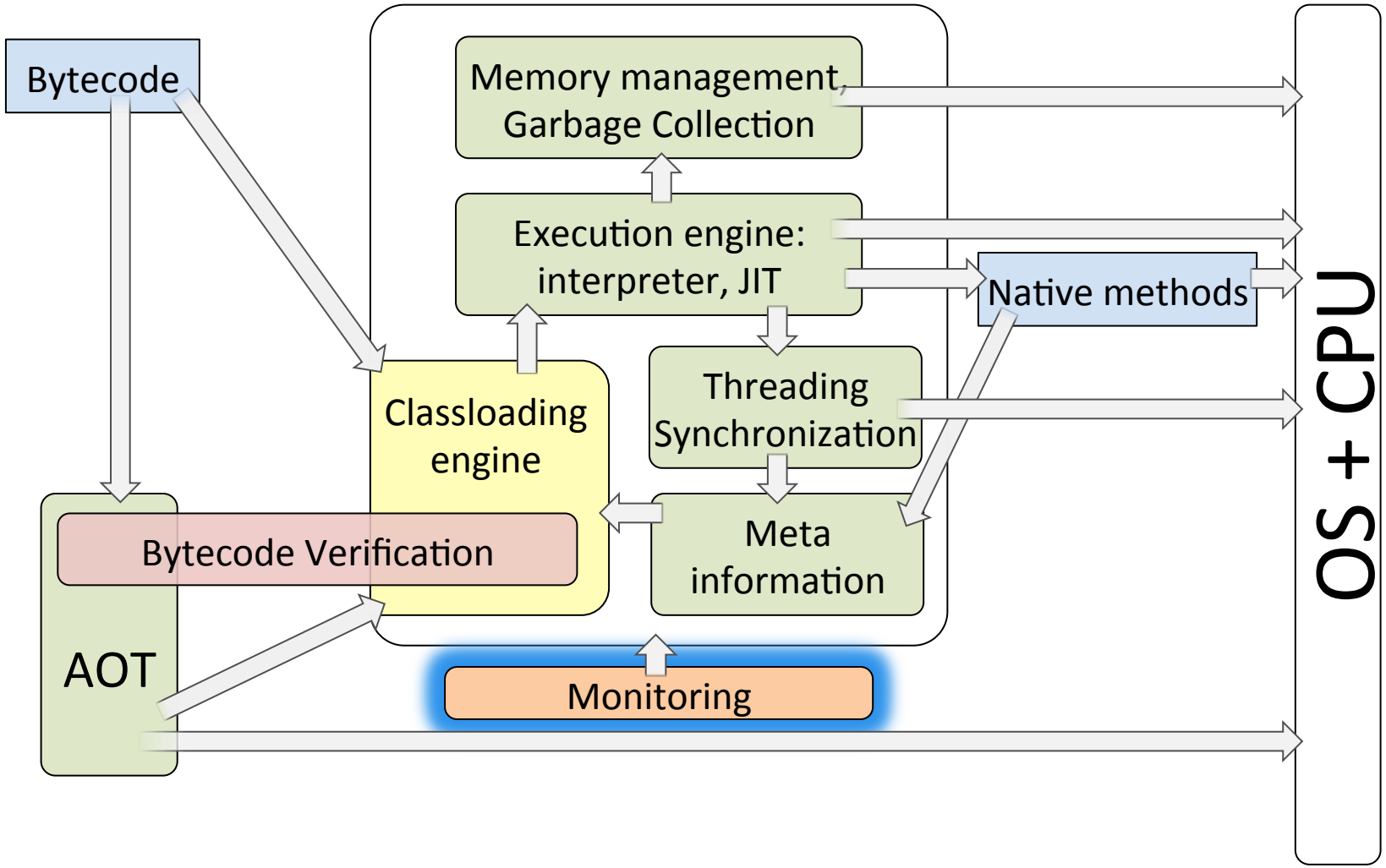
- *большинство объектов умирает молодыми*
- *старые объекты редко ссылаются на молодые*

Поколенный (generational) GC:

- частный вид инкрементального
- во время т.н. *малых* сборок удаляем мусор среди молодых объектов
- объекты, пережившие одну или несколько сборок, перемещаем в область старого поколения

EXCELSIOR

Manageability & Monitoring



EXCELSIOR

Manageability & Monitoring

JVM знает про вашу программу всё:

- про все загруженные классы
- про все живые объекты
- про все потоки
- про все исполняемые методы потоков

Почему бы не поделиться с вами этой информацией во время исполнения?

EXCELSIOR

Manageability & Monitoring

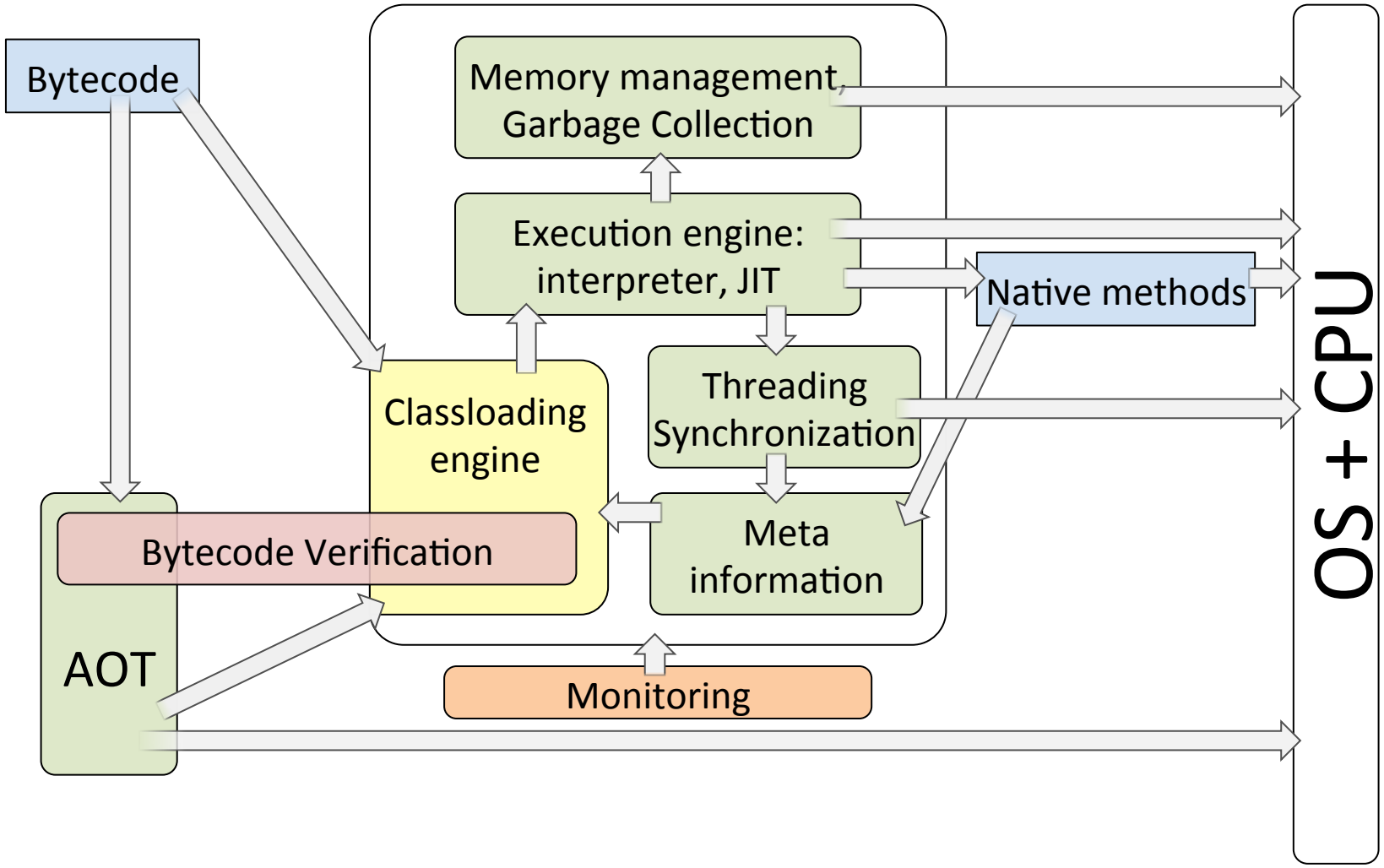
JVM Tool Interface (JVM TI):

- отладчики
- профилировщики

Java Management Beans:

- Инструменты мониторинга запущенных приложений
  - JConsole, JMX console, AMC
  - Visual VM
  - Java Mission Control

Абстрактная JVM



EXCELSIOR

Реализации JVM

Совместимые с Java SE спецификацией:

- Oracle HotSpot
- Oracle JRockit (**RIP**)
- IBM J9
- Excelsior JET
- Azul (HotSpot based, но свой GC)
- SAP, RedHat (свои порты HotSpot на разные платформы)

EXCELSIOR

Заключение

- JVM сложная, но жутко интересная штука
- Java – золотая середина современных IT технологий:
  - Подробно специфицирована
  - Эффективность помноженная на гибкость
- Все реализации JVM в постоянном развитии на острие науки и технологий

EXCELSIOR

Вопросы и ответы

Никита Липский,
Excelsior

nlipsky@excelsior-usa.com
twitter: @pjBooms

Владимир Иванов,
Oracle

vladimir.x.ivanov@oracle.com
twitter: @iwan0www

EXCELSIOR