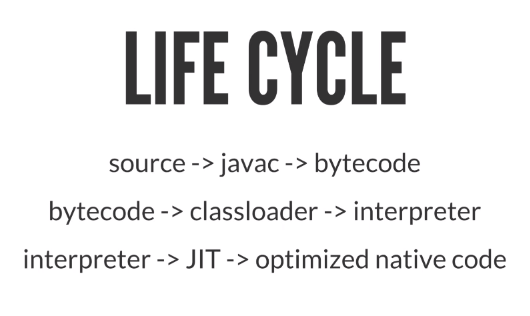
# JVM Internals

Java Virtual Machine jest zwykłą aplikacją napisaną w glównej mierze w C++.

Cykl życia aplikacji javovej:



## JIT – Just In Time Compiler – optymalizuje zinterpretowany kod

JVM odpali wszystko co możemy skompilować do kodu bajtowego, czyli nie tylko kod Javo’wy.

Przykładowo scala i java mogą zrobić to samo bo oba są kompilowane do bytecodu, ew. Kompilatory mogą pozwalać na różne instrukcje.

### JAVAC

* Konwertuje kod źródłowy na kod bajtowy.
* Sprawdza np. typy zmiennych.
* Może wygenerować część kodu np. lambdy.
* Proste optymalizacje kodu typu sklejanie string’ów ‘+’ zamienia na StringBuilder (ale bardzo uproszczone).

### BYTECODE

To kod bajtowy, czyli kod, z którego każda operacja w tym kodzie ma długość jednego bajta. Taki zapis pozwala na utworzenie 256 operacji, przy czym JVM wykorzystuje obecnie 203 operacje.

### PLIKI CLASS

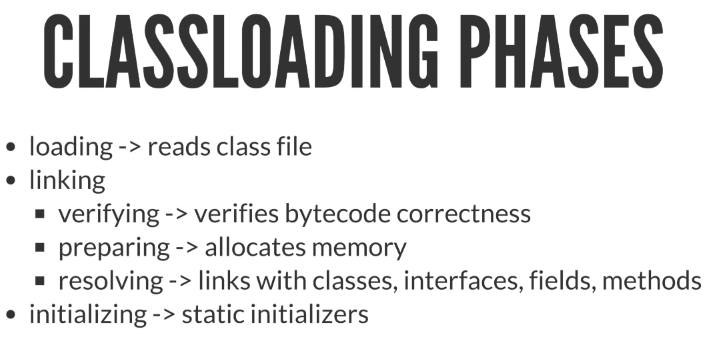
W nich mamy zapisaną całą informację taką jak:

* Meta info: wersja javy, format pliku,
* Informacja o klasie: flagi dostępu, nazwa, klasa nadrzędna, interfejsy,
* Zawartość: pola, sygnatury metod, kod bajtowy

### CLASSLOADER

W javie nie ma jednego classloadera, a jest ich cała hierarchia.

Odpowiada za dynamiczne ładowanie klas. Classloader ładuje klasy tworząc obiekty danej klasy.



* Ładuje plik z klasą.
* Linking
  + Weryfikacja – sprawdza poprawność bytecode’u,
  + Przygotowanie – alokacja pamięci,
  + Resolving – połączenia między klasami, interfejsami, polami, metodami,
* Inicjalizacja – statyczna inicjalizacja

### Interpreter

Interpretuje kod na podstawie szablonów.

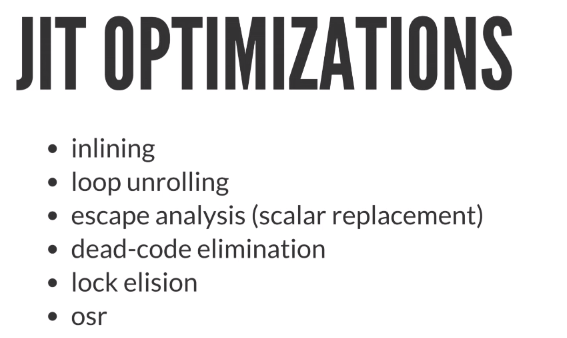
Wykrywa gorące punkty „hot spot’y” w programi. Np. „Ta metoda została wywołana wiele razy, więc muszę ją poprawić.

### JIT

* Just-In-Time
* Kompiluje metody do kodu natywnego.
* Taki kod natywny potrafi się wykonywać do 20 razy szybciej od zwykłego bajtowego.

Jest w dwóch wersjach w wersji klienckiej i serwerowej.

Poza zwykła kompilacją wykonuje wiele optymalizacji kodu. Obserwuje nasz kod i np. w przypadku czestego wywoływania getter’ów .



Inlining – kopiuje ciało metody w miejsce wywołania, przez co niepotrzebny jest dodatkowy skok.

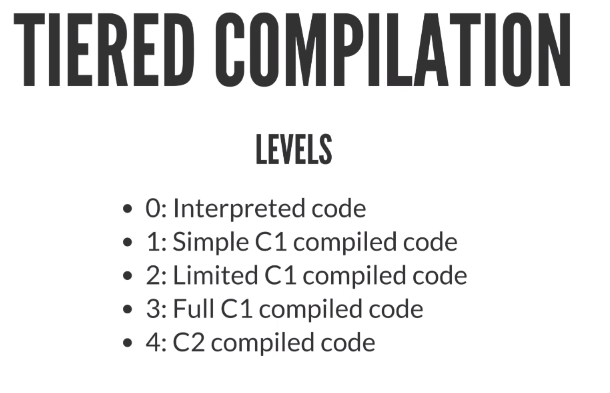
Escape analysis – zastępowanie skalarów, w przypadku gdy np. wykrozystujemy tylko pola obiektu, które są skalarami do jakichś obliczeń

Dead-code elimination – usunięcie kodu, który nie ma efektu żadnego

Lock-elision – optymalizuje locki jak np. jakaś synchronizacja kodu jest zbędna

To wszystko razem to tzw. **adaptive tuning**.

### Kompilacja etapowa (tiered compilation)



-client –server (C1, C2) to wersje JIT’a.

### Wątki

Wątki w javie są mapowane 1 do 1 do wątków natywnych procesora. Scheduling i dispatching jest wydelegowane do OS.

Mamy dwa rodzaje wątków w JVM:

* Wątki aplikacyjne (nasze),
* Wątki JVM’owe (garbage collectoa, JIT’a itd.)

### Synchronizacja

### Memory Blocks

* Rejestr komputera,
* Ramki,
* Stos,

Na stos odkładane są ramki.

Ramka to kontekst wywoływanego kodu.

Stack trace to zbiór ramek odłożonych na stosie.

### Memory Layout – 1:04.11