Java의 특징

- JVM을 통한 Platform independent
- 타입 안정성
- 객체지향 설계
- 다양한 컬렉션 프레임워크 제공
- GC에 의한 메모리 관리

JVM을 통한 Platform independent

- Write once, run everywhere
- 작성하는 코드가 실행되는 환경(운영체제)이 아니라 JVM에 종속된다.
- 이러한 이유때문에 JVM위에서 구동되는 Java프로젝트에 점진적으로 Kotlin,Scala를 함께 사용 할 수 있다.

타입 안정성

- 강타입 언어이다. 변수 선언과 동시에 Data Type이 결정되고 변하지 않는다.
- 제너릭을 통해 컴파일시 엄격한 타입체크도 가능하다.
- runtime error를 줄일 수 있지만, 상황에 따라 자료형 변환이 복잡하기도 하다.

객체지향 설계

- 인터페이스 or 상속을 통한 객체지향의 다형성 구현가능하다.
- But, Java를 사용한다고 객체지향 코드가 만들어지는것이 아니다. 그저 객체지향 기법을 이용가능하게 해준다.

다양한 컬렉션 프레임워크 제공

- Stack,Queue,LinkedList,HashMap등 자주 사용가능한 여러가지 자료구조를 컬렉션 프레임워크로 제공한다.
- 개발자는 적절한 사용처에 잘 사용하면 된다.

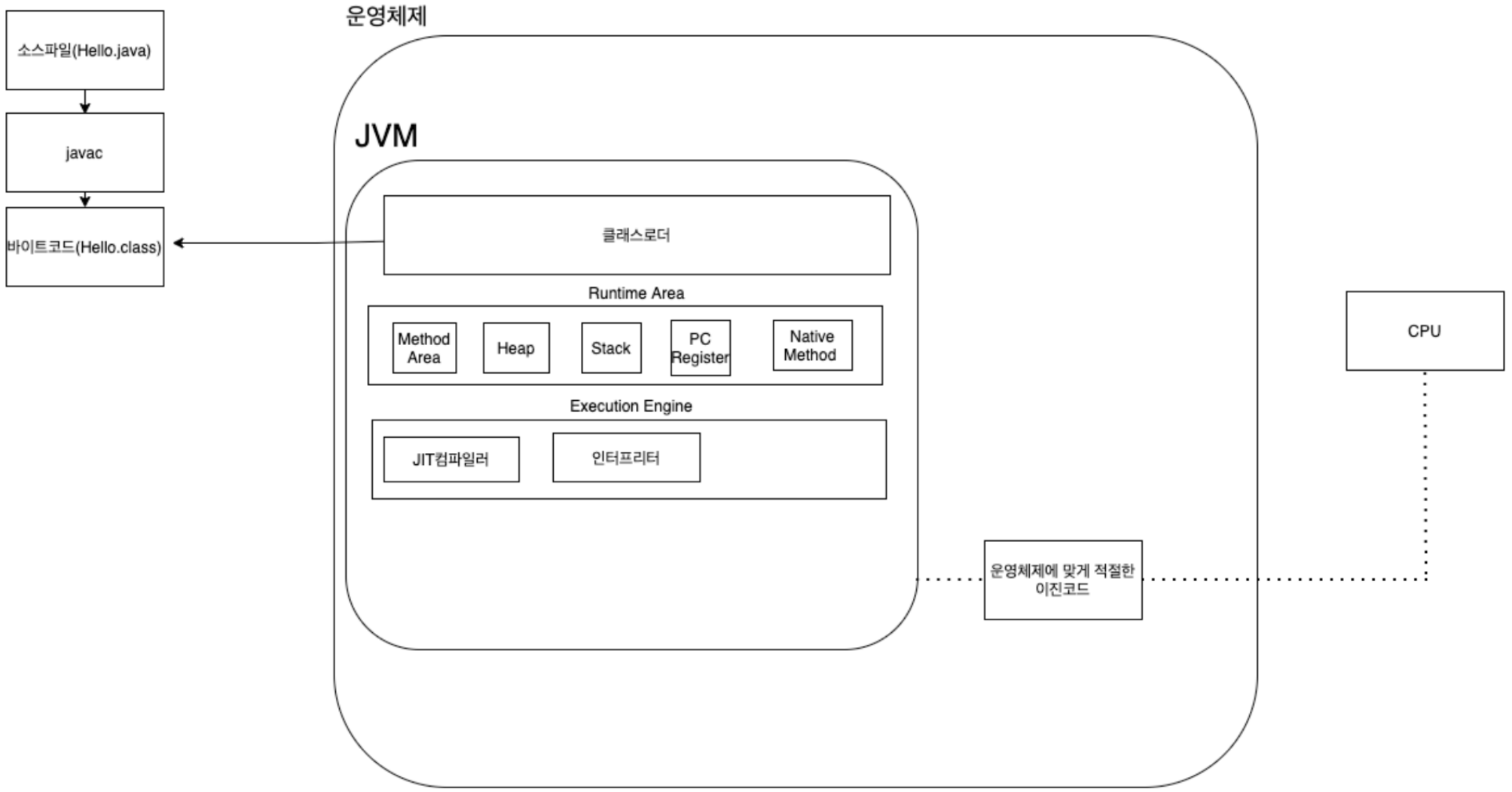
GC에 의한 메모리 관리

- 개발자가 메모리를 직접 할당하거나 해제할 필요가 없다.
- JVM의 GC가 미사용 메모리를 자동으로 수거해서 메모리 관리로 부터 자유롭다.

Java 구성요소 분류

- JVM
  - 물리적 cpu하드웨어나 레지스터는 없지만 일부 결과를 실행 스택에 보관하면서 스택위 가장 위에 쌓인 값들을 가져와서 계산하는 머신
- JRE
  - 자바 실행환경 (JVM과 자바 클래스 라이브러리, 자바 명령등 Java프로그램을 실행하는데 필요한 패키지)
- JDK
  - JRE + 개발에 필요한 프로그램들 (javac컴팡일러, javadoc같은 도구 포함)

Java의 동작 과정


소스파일(Hello.java)
javac
바이트코드(Hello.class)
운영체제
JVM
클래스로더
Runtime Area
Method Area
Heap
Stack
PC Register
Native Method
Execution Engine
JIT컴파일러
인터프리터
운영체제에 맞게 적절한 이진코드
CPU

Java는 정말 느린가?

- JIT Compiler
  - 프로그램의 성능을 최대한 내려면 네이티브 기능을 활용해 cpu에서 직접 프로그램을 실행해야 한다.
  - 인터피리티드 바이트코드에서 네이티브 코드로 컴파일 시키면 좋지 않을까?
  - 핫스팟에서는 인터프리티드 모드로 실행되는 동안 실시간 모니터링하면서 가장 자주 실행되는 코드파트(메서드,루프)를 발견해 JIT컴파일을 수행한다.

그럼 미리 컴파일 해두면 무조건 좋은거 아닌가?

- 비정상적으로 트래픽이 높은날에 JVM에 의해 JIT에 의해 최적화되는 내용과 트래픽이 별로 없을때의 JIT에 의한 최적화된 기계어 코드는 서로 다르다.
- VM이 내려가면 핫스팟은 컴파일 정보를 보관하지 않고 실행할때마다 새롭게 최적화해서 컴파일 한다.

JIT컴파일러에 의해 최적화 되는 시점

- invocation counter(메서드 시작할때마다 증가), backedge counter(메서드에 루프가 존재하는지 확인하는 카운터)를 이용
- 카운터값들이 인터프리터에 의해 증가될 때마다 한계치(CompileThreadhold)와 비교해서 한계치에 도달한경우 컴파일을 요청한다

JIT 컴파일 시점

- JIT컴파일러는 각각의 메서드를 컴파일할 만큼 시간적 여유가 없다.
- 컴파일 요청하면 컴파일 대기규에 쌓이고 하나 이상의 컴파일러 스레드가 큐를 모니터링하다가 차례차례 꺼내서 컴파일을 시작한다.
- 컴파일이 완료되면 컴파일된 코드와 메서드가 연결되어 메서드가 호출되면 컴파일된 코드를 사용

JIT On Stack Replacement

- JVM은 OSR (On Stack Replacement)라는 신기한 컴파일도 수행
- 처음 실행했지만, 루프가 길다고 판단되는 경우 중간에 컴파일 시켜버린다.
- 루프 실행을 하다가 임계치 초과하면 컴파일 시키고 스택 최상단에 넣어버린다
- 즉, OSR은 컴파일된 버전을 바로 실행시키는 기술이다.

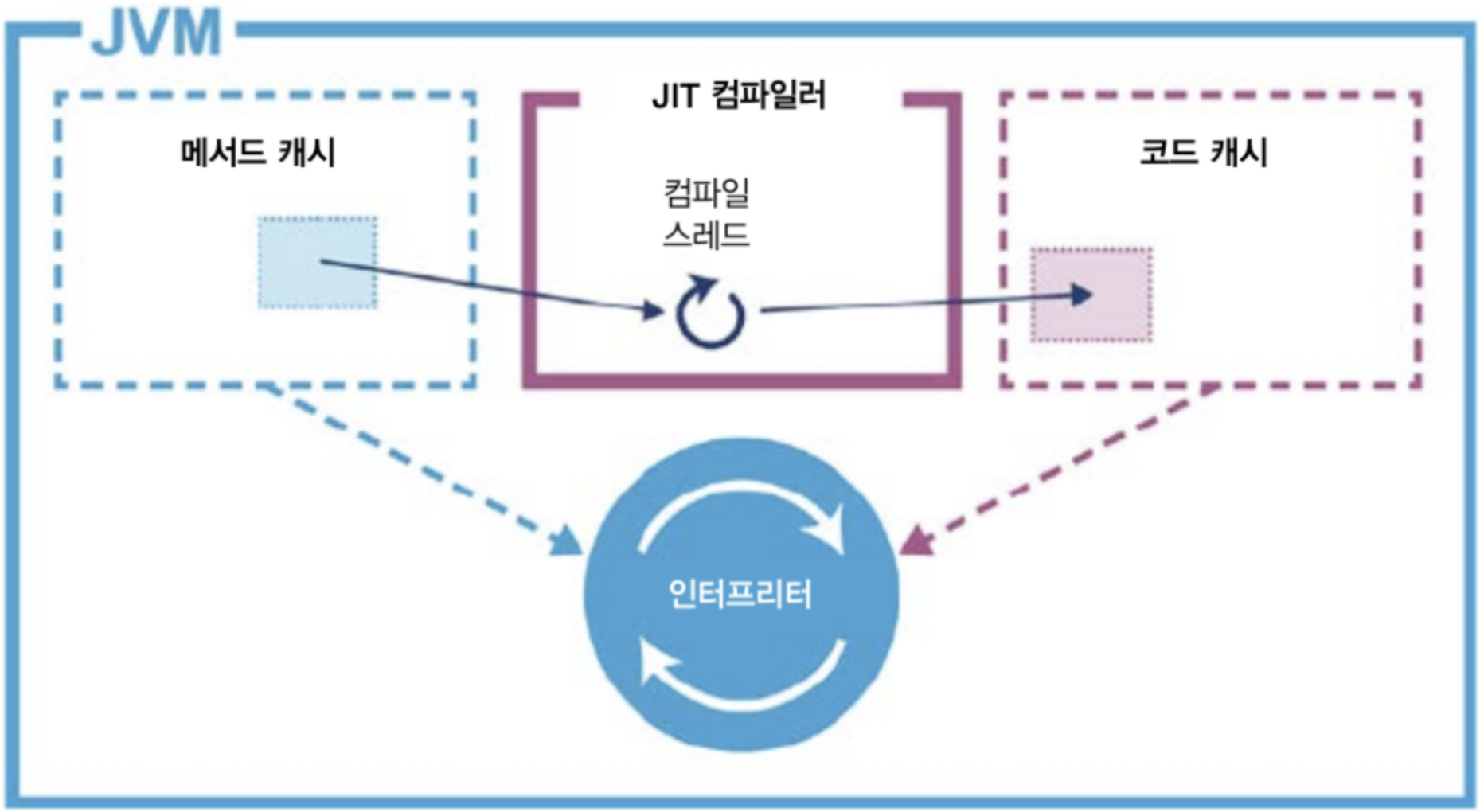
JIT컴파일러 종류

- 같은 코드여도 c1컴파일러,c2컴파일러에 의해 생성되는 표현이 완전 다르다.
- 기본적으로 모든 변수를 재할당하지 못하게 최적화 시킨다.
- c1은 c2보다 컴파일 시간이 짧고 단순
- c2는 full optimization을 한다. 즉, c1보다 컴파일 시간이 길지만 더 많이 최적화 시킨다.

JIT에 의해 컴파일된 코드는 어디에 저장?

- 클래스 로더에 의해 로딩된 바이트코드는 메서드 캐시라는 곳에 적재된다.
- 인터프리터는 메서드 캐시를 읽으면서 프로그램을 실행
- 코드캐시 라는 메모리 영역에 컴파일시킨 코드를 적재한다
- 코드캐시 영역은 vm시작시 설정되는 값으로 구동중에 확장이 불가능
- 즉, 코드캐시가 꽉차면 JIT컴파일이 불가능하고 코드캐시에 올라가지 못함

JIT



JIT

```
public class JitTest {
    //인터프리팅
    //c1 컴파일
    //c2 컴파일
    public static void main(String [] args) {
        for (int i = 0; i < 500; ++i) {
            long startTime = System.nanoTime();

            execute();

            long endTime = System.nanoTime();
            System.out.printf("%d\t%d\n", i, endTime - startTime);
        }
    }
    public static void execute() {
        for (int j = 0; j < 1000; ++j) {
            new Object();
        }
    }
}
```

JIT

- JVM옵션을 넣어서 실행해보면 `-XX:+PrintCompilation`

```
99       24806
.
.생략
.
1045  262 %     4       jit.JitTest::execute @ 2 (24 bytes) // jit에 의해 컴파일
123     141 //컴파일 이후 실행시간이 줄어듬
```

Java Version별 주요 업데이트

- Java 8
  - Lambda Expression
  - Interfcae Default Method
  - PermGen영역 삭제
    - java8 이전에 heap area의 perm영역에 static 정보들이 저장.
    - 다수의 static으로 perm size를 초과하면 out of memory 발생가능성이 크다.
    - java8 부터는 native memory영역의 metaspace에 static정보를 저장
    - metaspace영역은 운영체제에 의해 동적으로 조정된다.
  - Stream API
  - Optional

Java Version별 주요 업데이트

- Java 9
  - 런타임이 모듈화
    - awt,swing같은 불필요 라이브러리 없이 최상위 모듈인 Base만 사용가능
    - 특정 프로그램에 필요한 최소 런타임만 제작할 수 있게되서 패키징 간편
  - 인터프리터 언어처럼 쉘에서 사용가능한 JShell추가

Java Version별 주요 업데이트

- Java 10
  - var키워드를 통한 지역변수 타입추론
  - `var list = new ArrayList<String>();`

Java Version별 주요 업데이트

Java는 계속 발전하고 있습니다.

- Java 11
  - ZGC라고 불리를 고성능 가비지 컬렉터 등장
  - 람다 파라미터에 var 사용
- Java 12
- Java 13
- Java 14
- Java 15