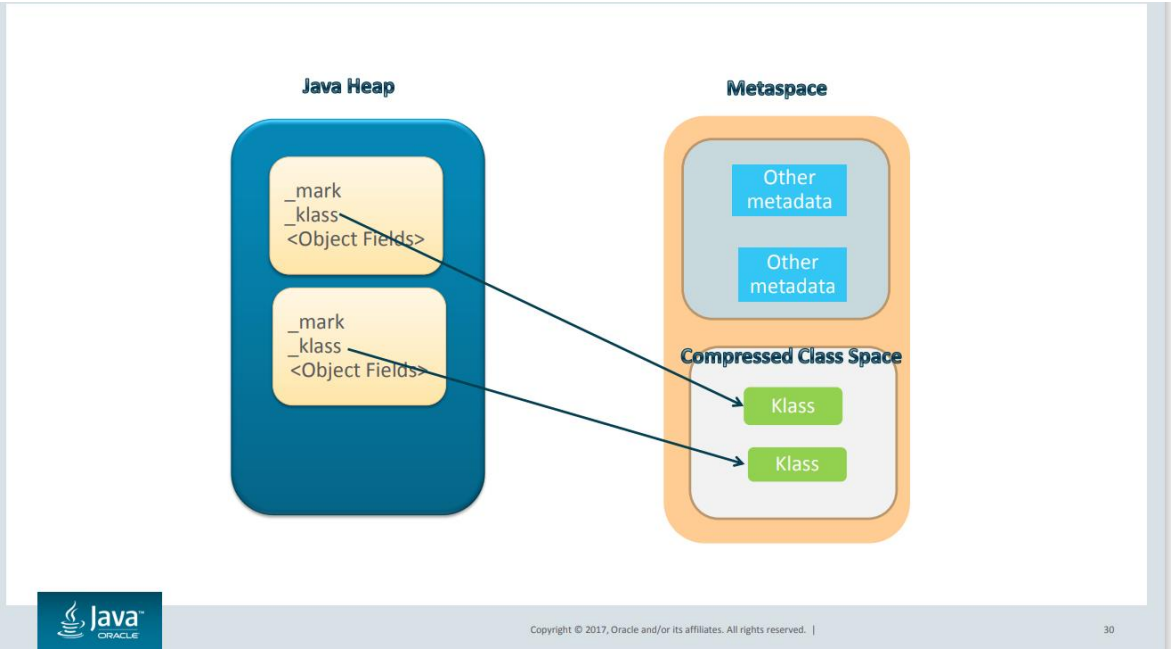
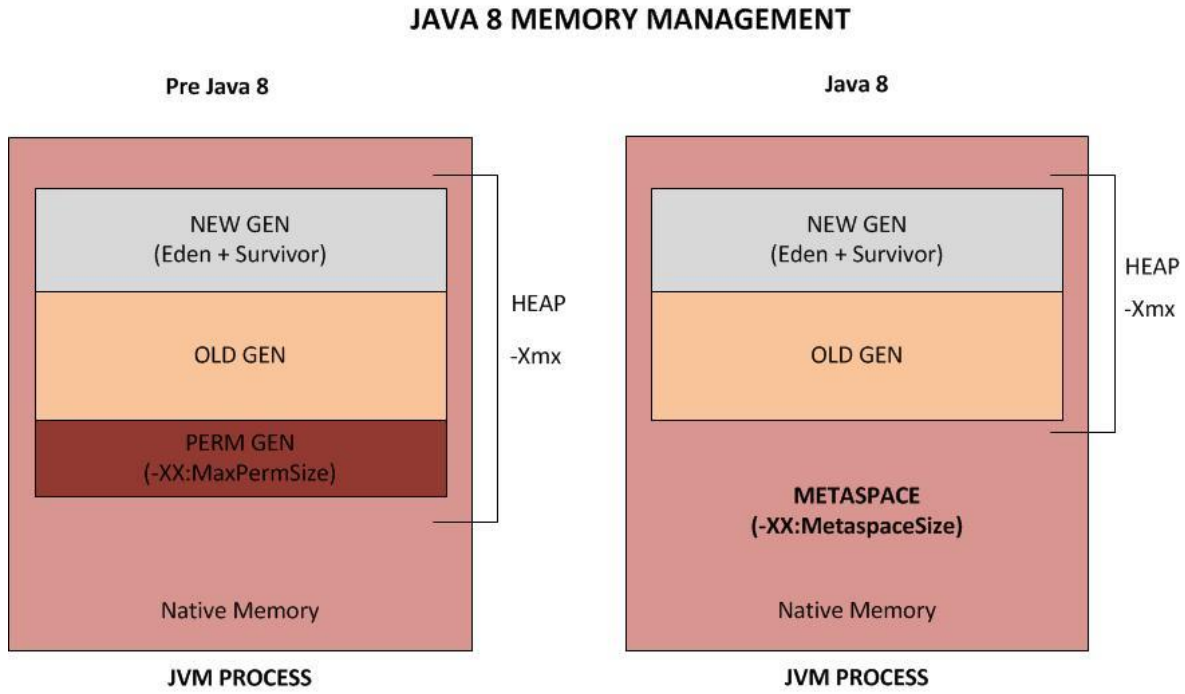
Java Statik ve Örnek(Instance) Değişkenler

- Statik değişkenler, bir sınıf değişkenleridir. Bu nedenle bir sınıfın içerisinde «statik» anahtar kelimesiyle bildirilir.
- Statik değişkenler program çalıştırıldığında bir kez oluşturulur ve program sonladığında kaybolur.
- Statik değişkenlerde kaç tane nesne oluşturulursa oluşturulsun sınıf başına her sınıftaki değişkenin tek bir kopyası oluşturulur ve sınıfın tüm örnekleri paylaşabilir.
- Statik değişkenler için bellek ayırma, sınıf belleğe yüklendiği anda sadece bir kez gerçekleşir. Bu da bellekten tasarruf sağlar.
- Statik değişkenlere sınıf ismiyle çağırarak doğrudan erişebiliriz, nesneye ihtiyaç duymaz.
- Statik değişkenleri tüm nesnelerin ortak bir özelliği için kullanabiliriz. Örneğin aynı okuldaki öğrencilerin okul adları gibi.
- Instance değişkenler de bir sınıfın içerisinde bildirilir. Instance değişkenler, «new» anahtar sözcüğüyle bir nesne oluşturulduğu zaman oluşturulur ve nesne yok edildiğinde yok edilir.
- Instance değişkenlere nesnenin referansı kullanarak erişmek daha doğrudur. Bir sınıfın belirli bir örneğine bağımlıdır.
- Instance ve statik değişkenlerin başlatılması zorunlu değildir. Default olarak 0'dır.
- Statik değişkenler gibi sınıfın diğer örnekleri arasında paylaşılamaz. Nesneye özgüdür.
- Yukarıda bahsettiğim iki değişken bellek alanında belleğin birden fazla parçaya bölündüğü ve bunlardan biri olan sınıflar ve meta veri alanındaki Metaspace olarak adlandırılan ve Heap alanından ayrı özel bir bellek alanındadır. Heap içindeki tüm statik değişkenler burada tutulur. Metaspace, Permgen e göre çöp toplama konusunda daha başarılıdır.

Değişkenlerin tutulduğu bellek alanı





```java
5 usages
class student{
    3 usages
    int schoolno;
    2 usages
    String name;
    2 usages
    static String scholl="Bartın Üniversitesi";

    2 usages
    student(int s, String n){
        schoolno=s;
        name=n;

    }
    4 usages
    void display(){
        System.out.println(schoolno+ " "+name+" "+scholl);

    }
}
public class Main {
    public static void main(String[] args) {
student s1=new student( s: 190, n: "Busra");
student s2=new student( s: 180, n: "Sema");
s1.display();
s2.display();
student.scholl="barü";
s2.schoolno=170;
s2.display();
s1.display();
```

Instance variable

Static variable

Constructor

```
Run:  Main
"C:\Program Files\Java\jdk-17.0.4\bin\
190 Busra Bartın Üniversitesi
180 Sema Bartın Üniversitesi
170 Sema barü
190 Busra barü

Process finished with exit code 0
```

Nesne oluşturma

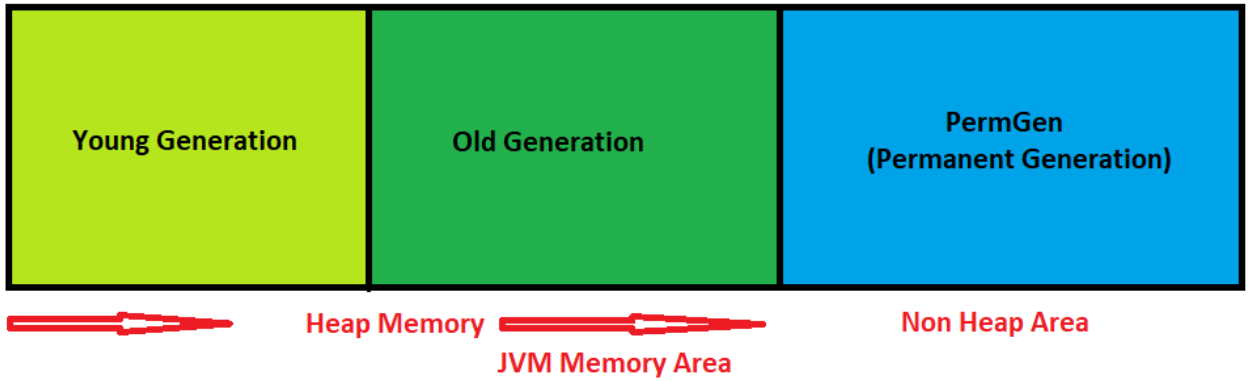
Static ve instance değişkenin değerini değiştirme

Static Metotlar

- Static değişken ve metotlar sınıfa aidiyeti gösterir.
- Sınıftan hiçbir nesne oluşturulmamış olsa bile bellekte yer kaplarlar.
- Sınıf metotlarını çağırmak için o sınıftan bir nesne oluşturmamız gerekmez.
- Static metotlar içinden static olmayan bir sınıf öğesine erişemeyiz.
- Sınıfın static elemanları bir kez yaratılır ve programın sonuna kadar yaşarlar.

JVM Static Metotları Nerede Saklar?

- JDK 8'den önce HotSpot JVM, static değişken ve metotları heap alanına bitişik olan Permgen adında bir alanda tutardı.
- JDK 8'den sonra static değişken ve metotlar metaspace denilen bir alanda tutulmaya başlanmıştır.
- Metaspace Permgenden farklı olarak bu alan Java Heap ile bitişik değildir.

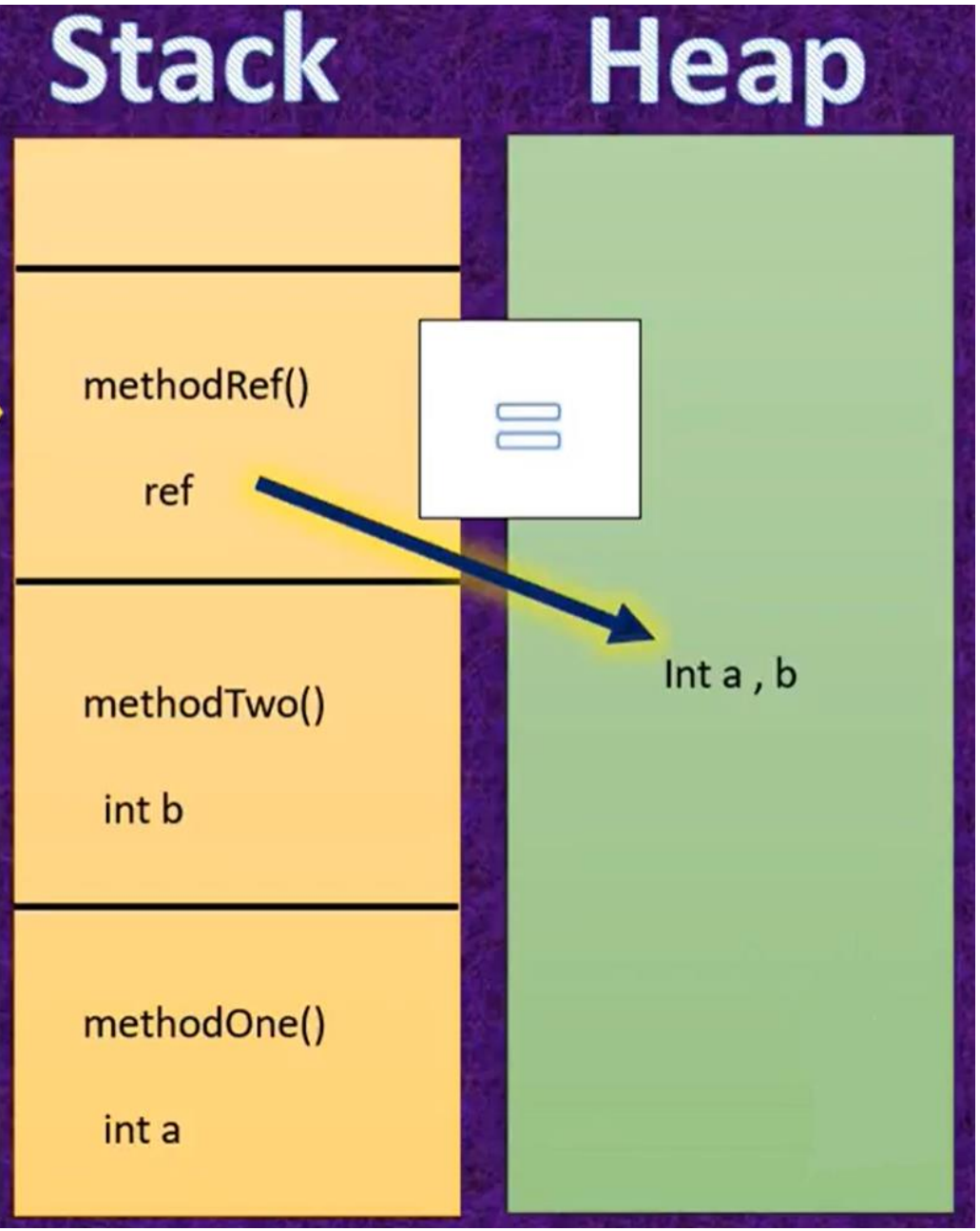


Non-static Metotlar

- Non-static değişken ve metotlar, sınıfa ait değildir. Sınıftan yaratılan objelere aittir.
- Nesne oluşturulmadıysa sınıfın non-static elemanlarına erişemeyiz.
- Sınıfın non-static bir metodu içerisinde sınıfın static bir metodu doğrudan çağrılabilir ancak sınıfın static bir metodu içerisinde sınıfın non-static elemanlarına doğrudan erişilemez.
- Non-static metotlar ve parametreleri stack içerisinde tutulmaktadır.

```
class MyClass
{
        methodOne(){
            int a;
        }
        methodTwo(){
            int b;
        }
        methodRef(){
            NextClass ref=new NextClass();
        }
}
```

```
class NextClass
{
    int a,b;
}
```



FİNAL KEYWORD

Değiştirilmemesi gereken bir uygulama gerekliyse ve bu uygulama nesnenin tutarlı durumu için kritik öneme sahipse, bir yöntemi final yapabilirsiniz.

Bir sınıfın methodlarının bir kısmını veya tamamını final olarak atayabilirsiniz. Yöntemin alt sınıflar tarafından geçersiz kılınamayacağını belirtmek için bir method bildiriminde final keyword kullanabilirsiniz.

Faaliyet alanı boyunca tek atama olması koşuluyla istenildiği yerde atama yapılabilir.

Bir metodun parametre değişkeni final olarak bildirilebilir. Bu durumda metodun final olan parametre değişkenine metot içerisinde atama yapılamaz

Final ve non-static olarak bildirilen veri elemanlarına default değer atanmaz

non-static final veri elemanlarına bildirim sırasında değer atanabilir

EFFECTİVALY FİNAL

- Yandaki bu değişken final olduğundan, başlatıldıktan sonra değerini değiştiremeyiz. Eğer denersek bir derleme hatası alırız .

```
final int variable = 123;
```

- Ama böyle bir değişken oluşturursak, değerini değiştirebiliriz

```
int variable = 123;
variable = 456;
```

- Fakat Java 8 'de tüm değişkenler varsayılan olarak final şeklindedir. Yandaki kodda 2. satırın varlığı onu son olmayan yapar. Bu yüzden eğer 2. satırı yandaki koddan çıkarırsak değişkenimiz şimdi "etkin olarak son" olur.

- Öyleyse sadece bir kez atanan herhangi bir değişken "etkin bir şekilde sonlanır" .

```
int variable = 123;
```

```java
public class Tester{
   public static void main(String[] args){
      int a = 30;
      int b = 45;
      System.out.println("Before swapping, a = " + a + " and b = " + b);
      // Invoke the swap method
      swapFunction(a, b);
      System.out.println("\n**Now, Before and After swapping values will be same here
      System.out.println("After swapping, a = " + a + " and b is " + b);
   }
   public static void swapFunction(int a, int b) {
      System.out.println("Before swapping(Inside), a = " + a + " b = " + b);
      // Swap n1 with n2
      int c = a;
      a = b;
      b = c;
      System.out.println("After swapping(Inside), a = " + a + " b = " + b);
   }
}
```

CALL BY VALUE

Değere Göre Çağrı, değer olarak parametreli bir yöntemi çağırmak anlamına gelir. Bu sayede argüman değeri parametreye iletilir. Referansla Çağırma, referans olarak parametreli bir yöntemi çağırmak anlamına gelir. Bu sayede argüman referansı parametreye iletilir.

Değere göre çağrıda, iletilen parametrede yapılan değişiklik çağıranın kapsamına yansımaz . Referansa göre çağrıda parametrede yapılan değişiklik kalıcıdır ve değişiklikler çağıranın kapsamına yansıtılır.

```java
public class JavaTester {
   public static void main(String[] args) {
      IntWrapper a = new IntWrapper(30);
      IntWrapper b = new IntWrapper(45);
      System.out.println("Before swapping, a = " + a.a + " and b = " + b.a);
      // Invoke the swap method
      swapFunction(a, b);
      System.out.println("\n**Now, Before and After swapping values will be differen
      System.out.println("After swapping, a = " + a.a + " and b is " + b.a);
   }
   public static void swapFunction(IntWrapper a, IntWrapper b) {
      System.out.println("Before swapping(Inside), a = " + a.a + " b = " + b.a);
      // Swap n1 with n2
      IntWrapper c = new IntWrapper(a.a);
      a.a = b.a;
      b.a = c.a;
      System.out.println("After swapping(Inside), a = " + a.a + " b = " + b.a);
   }
}
class IntWrapper {
   public int a;
   public IntWrapper(int a){ this.a = a;}
}
```

CALL BY REFERENCE

Referans ile Çağırma tekniğinde, çağıran program argüman olarak bir değer yerine bu değere ait bellek adresini gönderir. Böylece veri paylaşımı, değerler yerine adreslerle gerçekleştirilir. Bu durumda argüman ile buna karşı gelen fonksiyon parametresi aynı bellek alanını kullanır. Bunun sonucunda çağrılan fonksiyonda bu şekilde kullanılan bir parametredeki değişiklik buna karşı gelen çağıran program argümanına aynen aktarılacaktır.

Java Virtual Machine
JVM

NEDEN İHTİYAÇ DUYULMUŞTUR ?

- Birçok programlama dilinde derleyiciler çıktı olarak bulundukları işletim sistemlerinin çalıştırabileceği makine kodunu üretirler. Ancak Java çıktı olarak makine kodu değil byte code denilen ara bir kod üretir. Bu nedenle JVM gibi bir mekanizmaya ihtiyaç duyulmuştur.

NE SAĞLAR ?

- JVM her işletim sistemi için özel olarak tasarlanmıştır. Yani JVM'in kendisi platform bağımlıdır. Bu sayede Java programlarının platform bağımsız olmasını sağlar. Özetle JVM , bir java uygulaması herhangi bir platformda düzgün şekilde çalışıyorsa diğer platformlarda da düzgün çalışmasını sağlar.
- C , C++ gibi programlama dillerinde kullanılan nesnelerin veya işletim sistemleri kaynaklarının işleri tamamlandığında ortadan kaldırılması tamamen yazılımcının sorumluluğundadır. Ancak Java'da garbage collector mekanizması bu durumu kendisi idare eder. Böylece programcıların yapabileceği hataların önüne geçer.

NASIL ÇALIŞIR ?

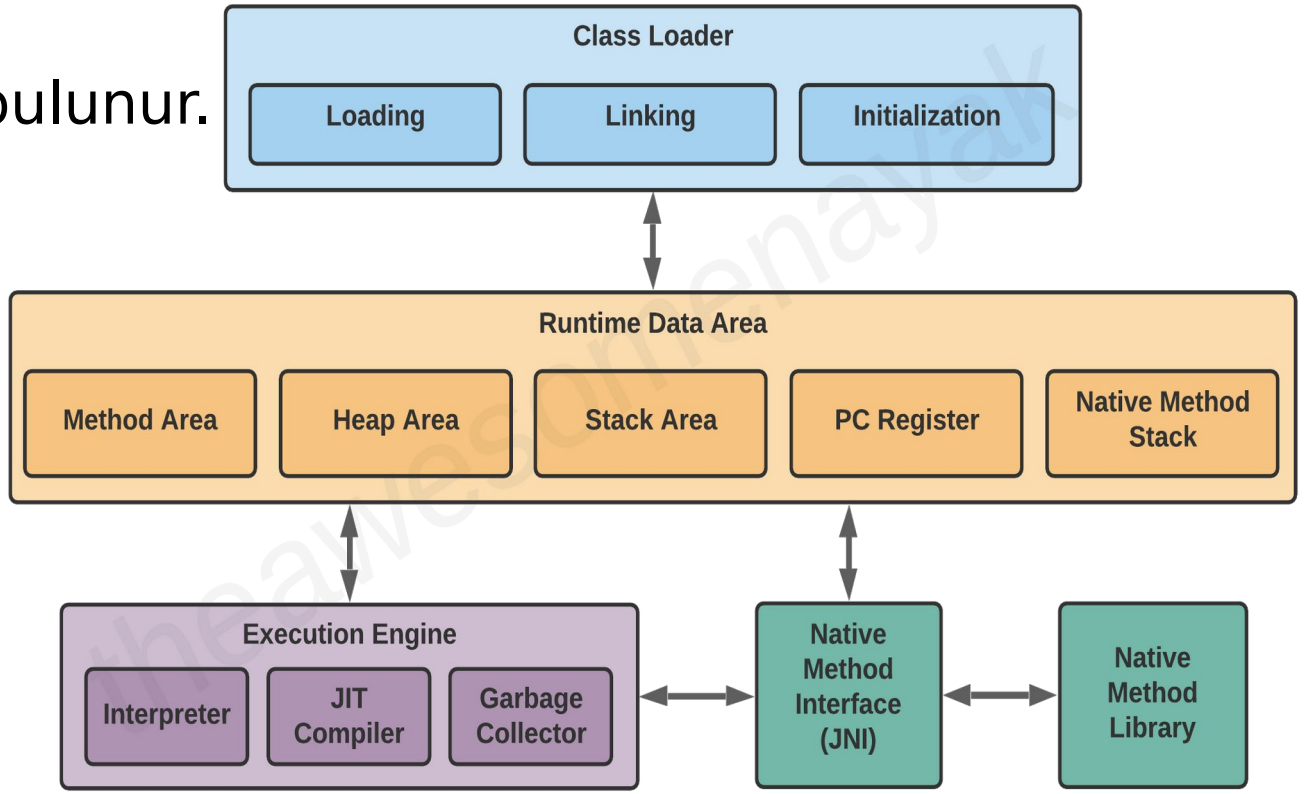
- JVM’in 3 ana bileşeni bulunur.

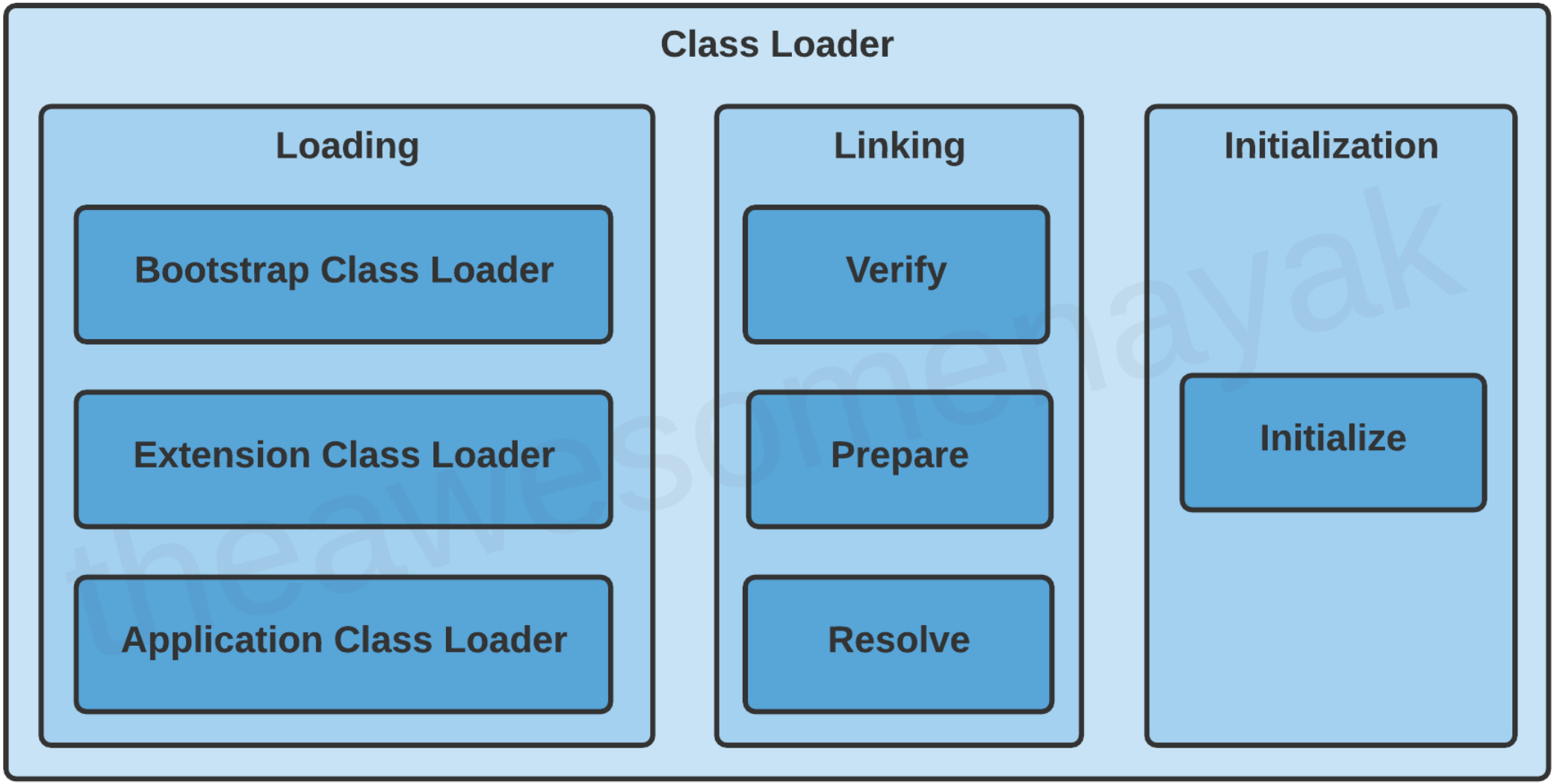
  Bunlar;

  -Class Loader

  -Runtime Data Area

  -Execution Engine




Class Loader
Loading
Bootstrap Class Loader
Extension Class Loader
Application Class Loader
Linking
Verify
Prepare
Resolve
Initialization
Initialize

Class Loader (Loading)

- .java uzantılı dosyalar Java derleyicisi tarafından derlendiğinde .class uzantılı dosyalar oluşur. Class loader .class uzantılı dosyaların belleğe yüklenmesinden sorumludur.
- 3 tipi vardır;

  -Bootstrap Class Loader : JRE için gerekli olan temel sınıfların yüklenmesinden sorumludur.

  -Application Class Loader : Oluşturduğumuz .java uzantılı dosyanın derlenmesiyle oluşan .class uzantılı dosyanın belleğe yüklenmesinden sorumludur.

  -Extension Class Loader : Ek sınıf dosyalarının yüklenmesinden sorumludur.

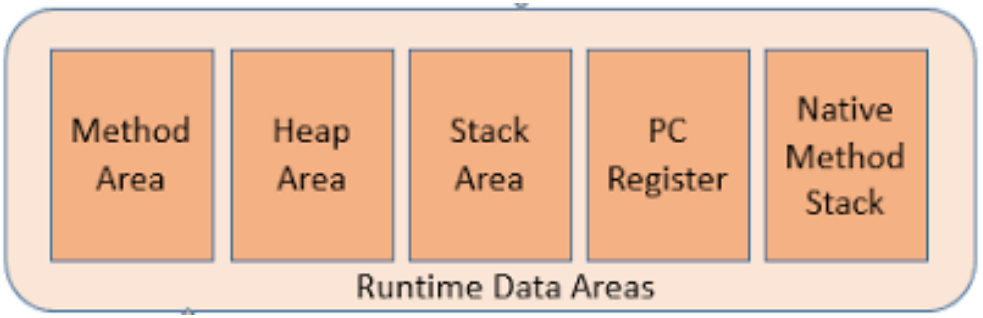
(Linking)

- Verify : Bu aşamada , yaratılan bytecode (.class) dosyaları doğrulanır. Doğrulanamazsa linking aşaması durur.
- Prepare : Bu aşamada , bellekte sınıfların referansları ve statik değişkenleri için yer ayrılır ve bu değişkenlere default değerler atanır.
- Resolve : Bu aşamada , önceki aşamada sınıf referanslarına atanan sembolik referanslar gerçek referanslarla değiştirilir.

(Initialization)

- Initialization : Class loader aşamasının son kısmıdır. Bu kısımda constructor çağrımları ,statik blokların yürütülmesi ve tüm statik değişkenlere değer atamayı içerir.

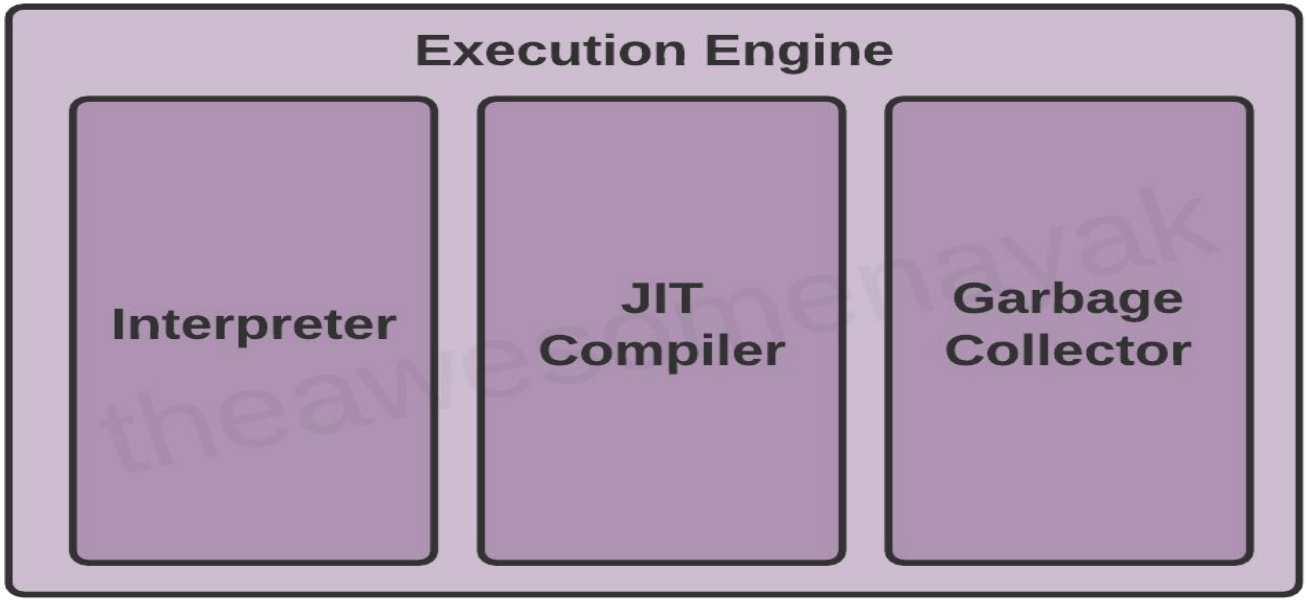
Runtime Data Area



- Method Area : Sınıf düzeyindeki tüm veriler burada saklanır.
- Heap Area : Nesnelere ait değerler ve bilgiler burada saklanır.
- Stack Area : Tüm global değişkenler , metot çağrıları , obje referansları burada saklanır.
- PC Register : JVM yürütülen talimatların adresini anlık olarak burada saklar.
- Native Method Stack : Burada tutulan metotlar harici bir programlama diliyle yazılmış metotlardır. Bu metotlara performans açısından ihtiyaç duyulabilir.

Execution Engine

- Bu bileşen class loader tarafından belleğe yüklenen bytecode halindeki dosyaları çalıştırır.



Interpreter

- Bytecode halindeki dosyayı yukarıdan aşağıya olacak şekilde çalıştırır.

  Avantaj :

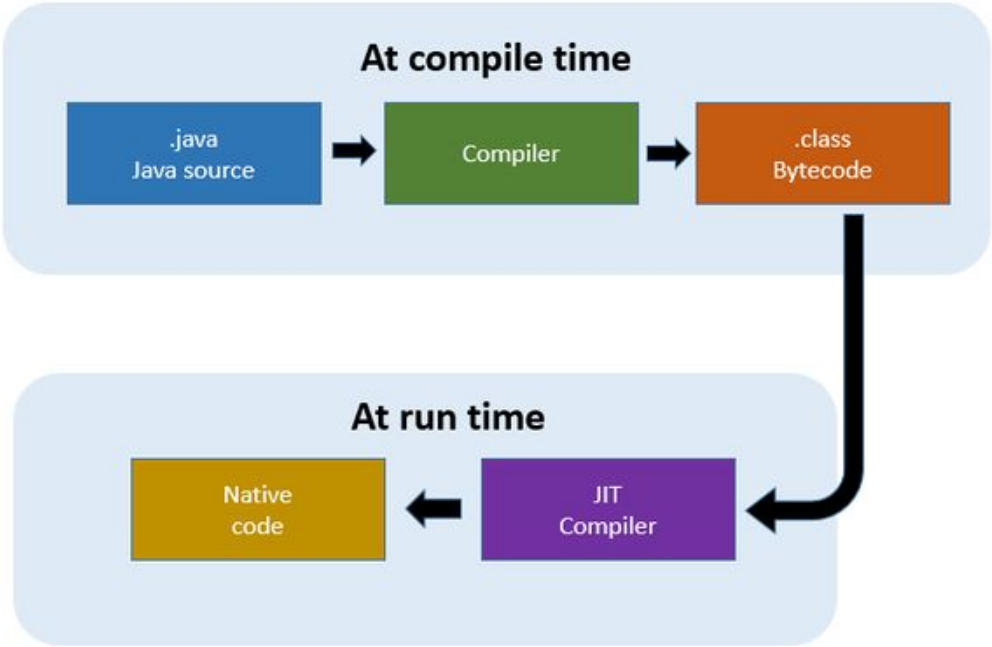
- Interpreter değişikliklerin test edilmesini ve programın debug edilmesini kolaylaştırır.

  Dezavantaj :

- Compiler’a göre yavaştır. Ancak JVM bu dezavantajı JIT(Just in time) compiler ile dengeler.

JIT(Just in time)Compiler

- JIT Compiler, tekrarlı metot çağrımlarında bytecode'u native code'a derler. Native code tekrarlı metot çağrımlarında kullanılır. Bu şekilde JIT Compiler interpreter'ın performans dezavantajını dengeler.



Garbage Collector

- Heap bellek üzerinde kullanılan nesnelerin tespit edilmesi ve referansı bulunmayanların silinmesi üzerine kuruludur. Bu sayede bellekte boş yer açılır.

Dinlediğiniz için teşekkürler.