

JVM YAPISI

Class Loader

Kodun derlenmesi sonucu oluşan .class uzantılı bytecode'u JVM'e yüklemeyi sağlar. 3 kısımdan oluşur.

- Loading

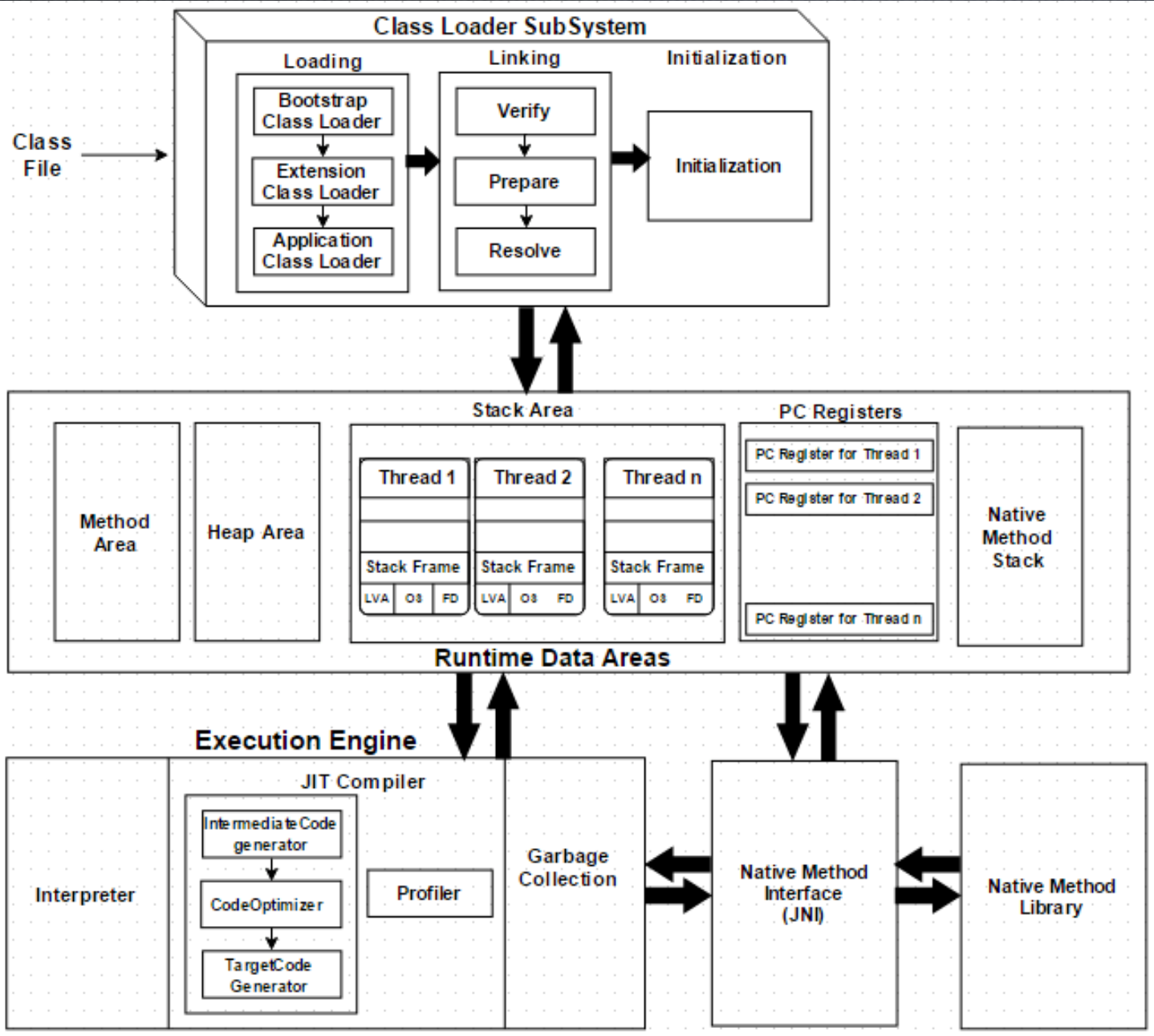
Bootstrap CL Java'nın temel paketlerini yükler. Extension CL programa eklenen ek kütüphaneleri yükler. Application CL ise uygulamanın class dosyalarını yükler.

- Linking

Verification kısmında bytecode'un doğruluğu kontrol edilir. Preparation aşamasında sınıflardaki statik alanlar için method area'da yer ayrılır ve default değerler atanır. Resolve de ise bir sınıfın diğer sınıflara ait referanslar içeriyorsa o sınıfları da yükler ve bağlar.

- Initialization

Sınıflara ait nesneler yaratılır. Statik alanların kendi değerlerinin atanması yapılır. Constructor başlatılır.



JVM YAPISI

Runtime Data Areas

- Method Area

Yüklenen sınıfların meta bilgilerinin, metot ve statik değişkenlerin bilgilerinin tutulduğu alandır.

- Heap Area

Sınıflardan türetilen tüm nesneler burada saklanır.

- Stack Area

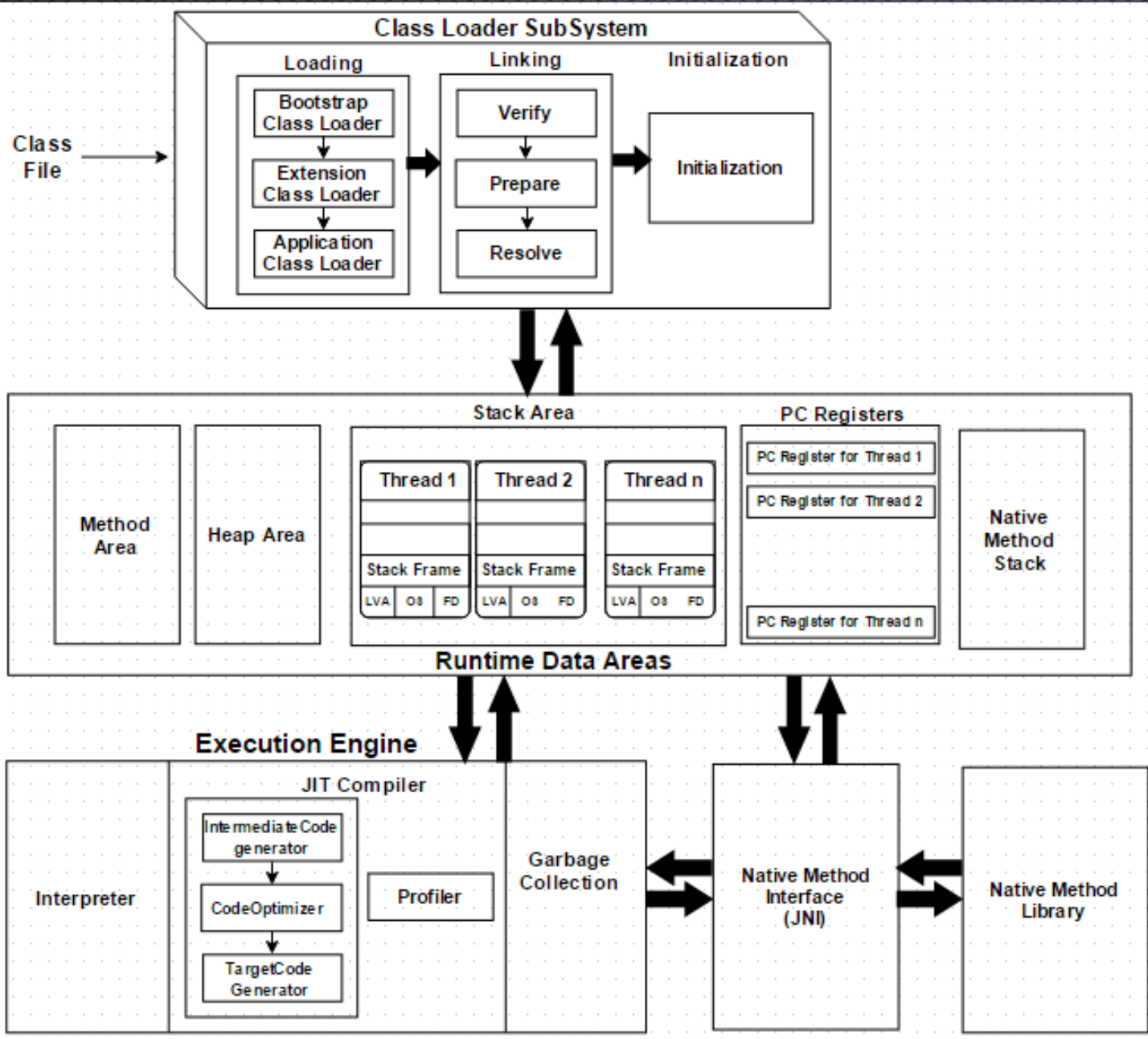
Her metot çağırımında bir frame oluşur. İçinde metot değişkenleri bulunur. Metot tamamlanınca frame silinir.

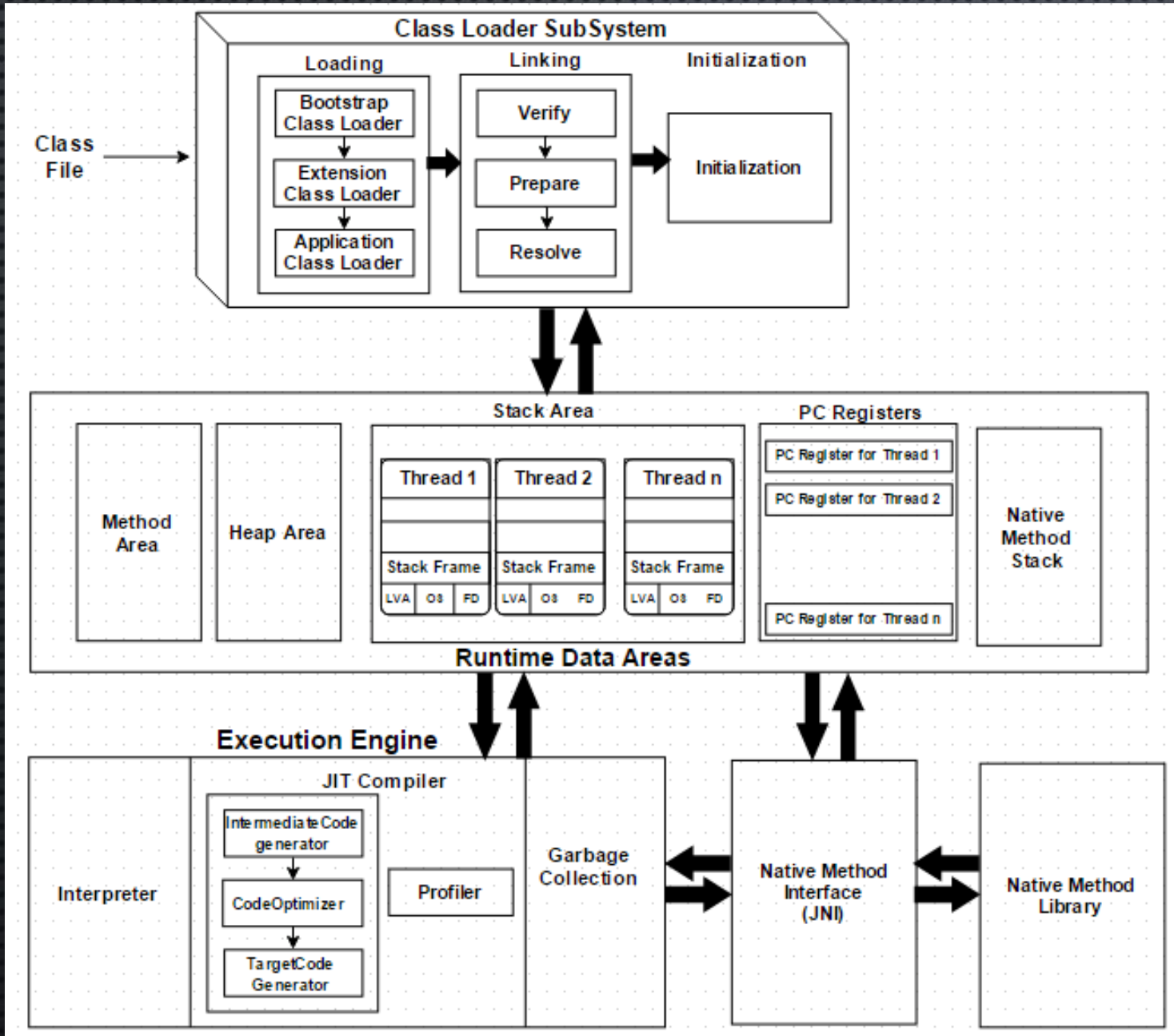
- PC Registers

İşletilen komutların sırasını kontrol eder. Her thread için ayrı bir PC vardır.

- Native Method Stack

C/C++ gibi dillerde yazılmış yerel metotlara ait verileri saklayan stack alanıdır.





JVM YAPISI

Execution Engine

Yüklenen bytecode'u satır satır okuyarak işletir.

- Interpreter

Bytecode'u makine koduna çevirmeyi sağlar. Performansı artırmak için JIT ile paralel çalışır.

- JIT Compiler

Tekrar eden kod kısımlarını önbelleğe alır ve her çağırmda tekrar yorumlanmadan kullanarak verimi artırır.

- Garbage Collector

Heap'te referanssız kullanılmayan verileri kontrol eder ve temizler.

Native Method Interface

Java dışı metod kütüphanelerini çağırmaya yarar.

Native Method Library

Java dışı metod kütüphanelerini bulunur.

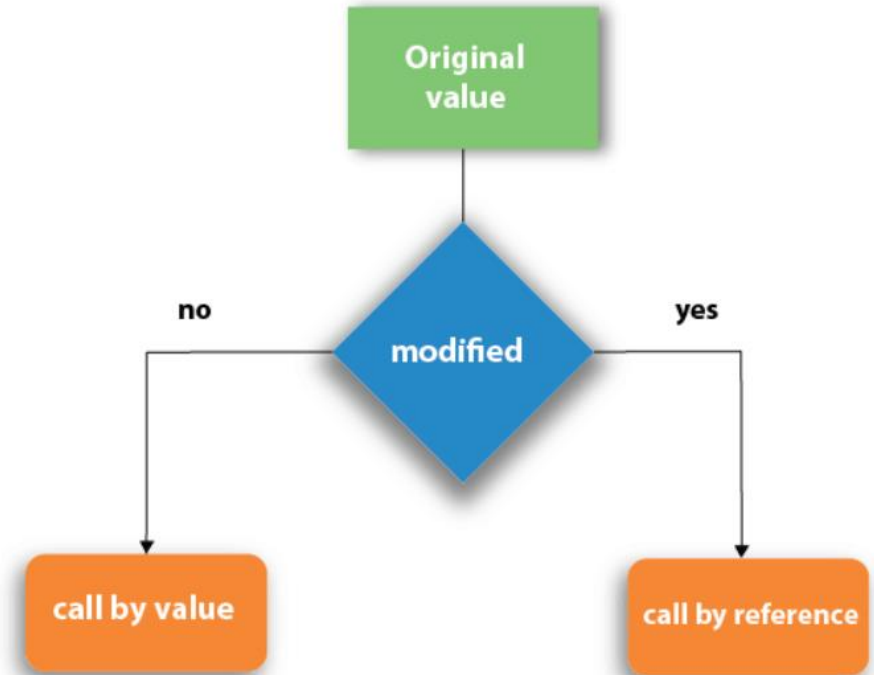
Call by Value and Call by Reference in Java

Giriş

- Java programlama dili türleri iki kategoriye ayrılır: ilkel türler ve referans türleri. İlkel türler , boole türü ve sayısal türlerdir. Sayısal türler, byte, short, int, long ve char integral türleri ve kayan nokta türleri float ve double'dır. Referans türleri sınıf türleri, arabirim türleri ve dizi türleridir. Ayrıca özel bir boş türü vardır. Bir nesne , bir sınıf türünün veya dinamik olarak oluşturulmuş bir dizinin dinamik olarak oluşturulmuş bir örneğidir. Bir başvuru türünün değerleri, nesnelere yapılan başvurulardır. Diziler dahil tüm nesneler, Object sınıfının yöntemlerini destekler. String değişmezleri, String nesneleri ile temsil edilir

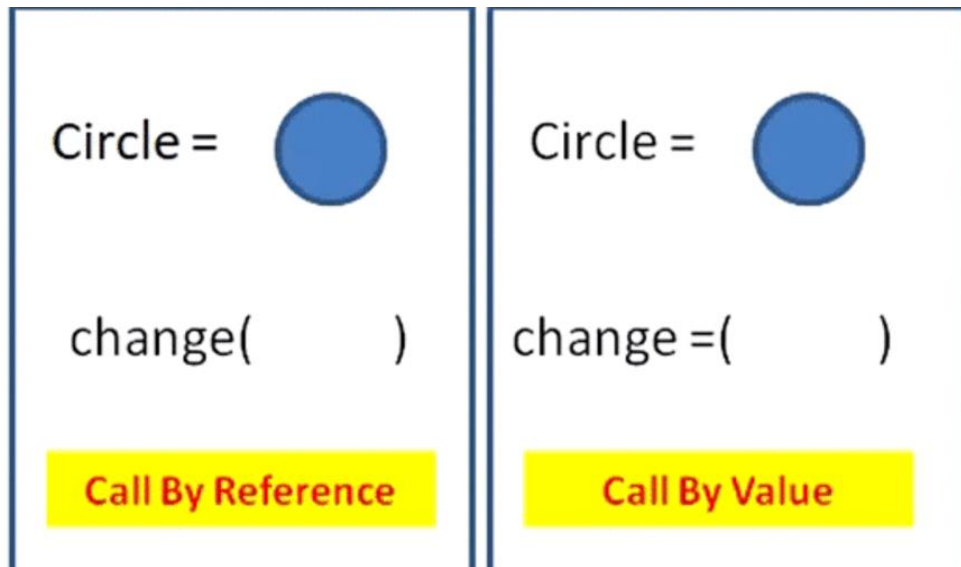
Call by Value Nedir ?

- Bu parametre geiş yönteminde, gerçek parametrelerin değeri fonksiyonun formal parametrelerine kopyalanır ve iki tip parametre farklı bellek konumlarında saklanır. Bu nedenle, fonksiyonlar içinde yapılan herhangi bir değışiklik, arayanın gerçek parametrelerine yansıtılmaz.



Call by Reference Nedir ?

- Hem gerçek hem de biçimsel parametreler aynı konumlara atıfta bulunur, bu nedenle işlem içinde yapılan herhangi bir değişiklik aslında arayanın gerçek parametrelerine yansıtılır.



- Java resmi olarak her zaman değere göre geçer. O halde soru, “değere göre iletilen nedir?” Sınıfta söylediğimiz gibi, yığındaki herhangi bir değişkenin gerçek değeri, ilkel türler (int, float, double, vb.) için gerçek değer veya referans türleri için referanstır. Diğer bir deyişle, bir referans değişkeni için yığındaki değer, gerçek nesnenin bulunduğu öbek üzerindeki adrestir. Java'da bir yönteme herhangi bir değişken iletilindiğinde, yığındaki değişkenin değeri yeni yöntemin içindeki yeni bir değişkene kopyalanır.
- Java, referans değişkenleri geçerken yalnızca değere göre çağrıyı kullanır. Referansların bir kopyasını oluşturur ve onları yöntemlere değerli olarak iletir. Referans, nesnenin aynı adresine işaret ettiğinden, bir referans kopyasının oluşturulmasının zararı yoktur. Ancak referansa yeni nesne atanırsa, yansıtılmaz.

Call by Value ve Call By Reference Örnekleri

Live Demo

```
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);
    }
}
```

Before swapping, a = 30 and b = 45
Before swapping(Inside), a = 30 b = 45
After swapping(Inside), a = 45 b = 30
Now, Before and After swapping values will be same here:
After swapping, a = 30 and b is 45

Live Demo

```
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 different here**");
        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;}
}
```

Before swapping, a = 30 and b = 45
Before swapping(Inside), a = 30 b = 45
After swapping(Inside), a = 45 b = 30
Now, Before and After swapping values will be different here:
After swapping, a = 45 and b is 30

```

public static void main(String[] args) {
    ...
    int y = 5;
    System.out.println(y); // prints "5"
    myMethod(y);
    System.out.println(y); // prints "5"
}

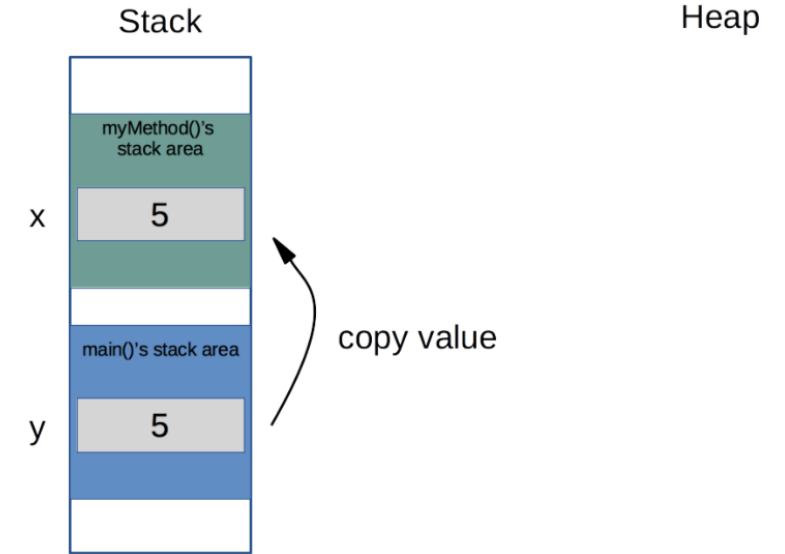
```

```

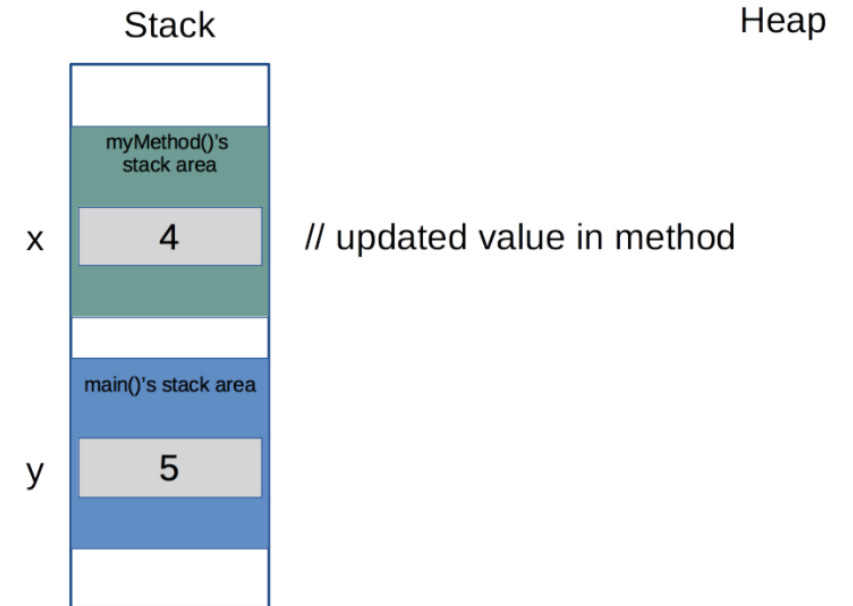
public static void myMethod(int x) {
    ...
    x = 4; // myMethod has a copy of x, so it doesn't
           // overwrite the value of variable y used
           // in main() that called myMethod
}

```

- Call `myMethod` which creates a copy of `y`'s value inside variable `x` on the stack.



- `myMethod` set's `x = 4`, changing `x`'s value while leaving `y`'s untouched.



Static/Non-Static Variables

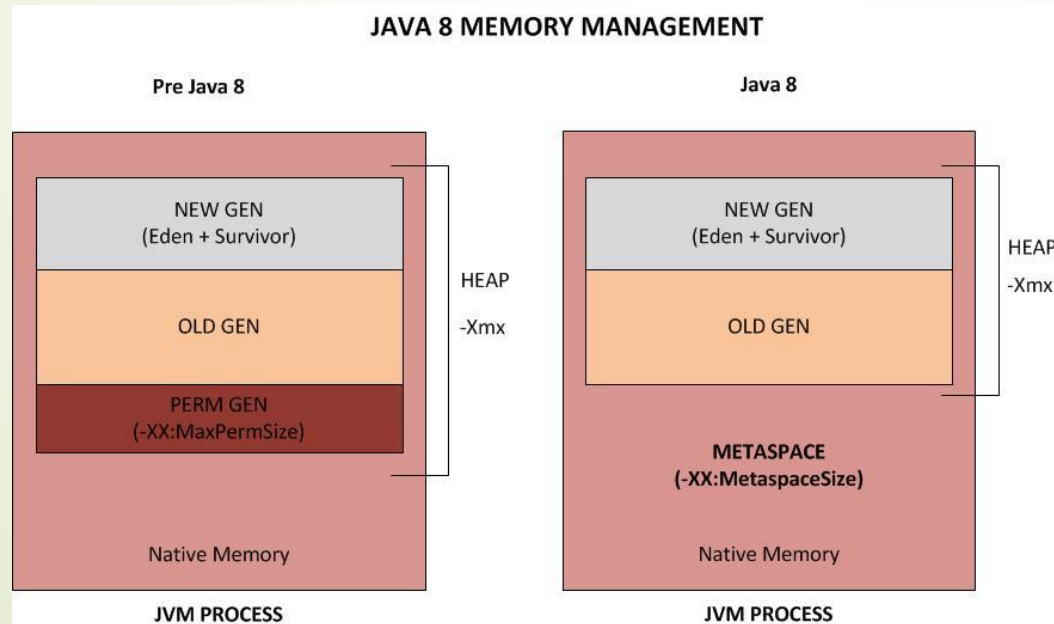
- Static Variables:
- Static keyword'ü ile tanımlanırlar.
- Bir sınıftaki tüm instanceların aynı datayı paylaşması isteniyorsa static variable kullanılır.
- Hem static hem de non static metodlardan erişilebilir.
- Non-Static Variables:
- Sınıfın bir instance'ı kullanılarak erişilebilir
- Static metodlardan erişilemezler.
- Tuttuğu değerler nesneye özeldir

```
1  ▶ class Counter{
    2 usages
2      static int count=0;//Bir kez oluşturulur
    3 usages
3      Counter(){
4          count++;//Static değişkenin değerini artıyor
5          System.out.println(count);
6      }
7  ▶   public static void main(String args[]){
8      Counter c1=new Counter();
9      Counter c2=new Counter();
10     Counter c3=new Counter();
11     }
12 }
13 /*OUTPUT
14 1
15 2
16 3
17 Seklinde olur.
18 */
```

```
1  ▶ class Counter{  
    2 usages  
2  int count=0; //non-static variable  
    3 usages  
3  Counter(){  
4      count++;  
5      System.out.println(count);  
6  }  
7  ▶ public static void main(String args[]){  
8      Counter c1=new Counter();  
9      Counter c2=new Counter();  
10     Counter c3=new Counter();  
11 }  
12 }  
13 /*OUTPUT|  
14 1  
15 1  
16 1  
17 Seklinde olur.  
18 */
```

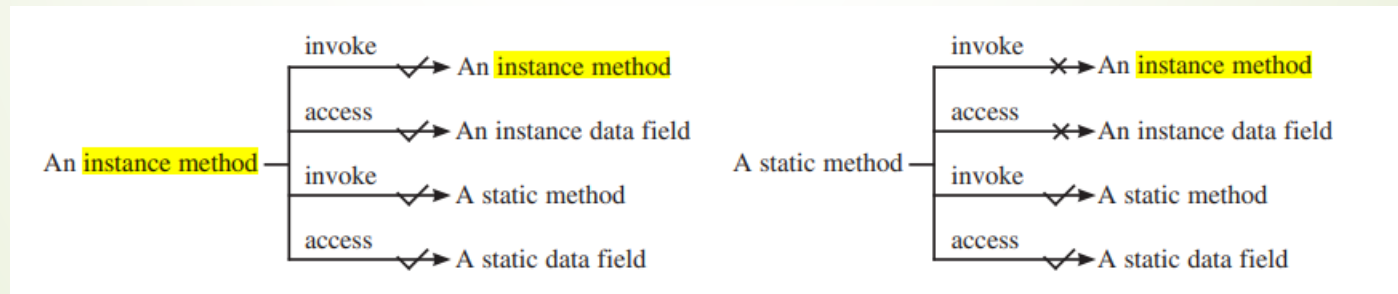

Static Methods

- Class'a ait bir nesnenin var olmasına ihtiyaç duymaz.
- Non static metodları çağırılmaz.
- Java8 öncesi Heap'ta permgen'de tutulurken java8 den sonra metaspase'de tutulmaktadır.



Non Static Methods

- Non-static metodlar nesne kullanılarak çağırılır ve bu nedenle nesnenin instance değişkenlerine erişebilirler.
- An instance method can invoke an instance or static method and access an instance or static data field.
- Statik metodların aksine override edilebilir.



STRING CLASS IN JAVA

String Sınıfı

- String sınıfı, **java.lang** kütüphanesinin altında bulunan ve bizlere karakter dizeleri üzerinde işlemler gerçekleştirmemizi kolaylaştıran bir imkan sağlar. String sınıfı metotları, **immutable** metotlardır. Yani bu sınıfın içerisinde bulunan metotlar **değiştirilemez**.
- Java'da birçok sınıfın iterable özelliği vardır fakat string sınıfında bu özellik yoktur.

Foreach string

```
char n[] = {'a', 'b', 'c', 'd', 'e'};  
for (char n2: n)  
{  
    System.out.println(n2);  
}
```

```
String n = "abcde";  
for (char n2: n)  
{  
    System.out.println(n2);  
}
```

Burada char değişkeninde bir dizi oluşturarak bunu foreach'te kullanabiliriz.

Burada bu şekilde kullanım yaptığımızda ise şöyle bir hata ile karşılaşırız;

java: for-each not applicable to expression type

Required(gerekli): array or java.lang.Iterable

String'i neden foreach'te kullanamayız ?

- Foreach döngüsü arrays ve iterable kabul eder.
- String iterable ve array değildir.
- String arrayın karakterini tutar ama arrayın kendisi değildir.
- String iterable interface'i desteklemez.

String'i foreach'te nasıl kullanabiliriz ?

- String'i foreach'te kullanabilmek için bazı metotlar vardır.

```
for (char ch: "xyz".toCharArray()) {  
}
```

toCharArray()

```
String s = "xyz";  
for(int i = 0; i < s.length(); i++)  
{  
    char c = s.charAt(i);  
}
```

charAt()



'Final' Keyword in JAVA

- **Final sınıf değişkenleri**
- **Final metot parametreleri**
- **Final metotlar**
- **Final sınıflar**

Final Sınıf Değişkenleri

- Final olan bir sınıf değişkenine sadece bir kere değer ataması yapılabilir.
- ilk değer ataması **ya tanımlandığı satırda ya da metot içerisinde.**

Final Metot Parametreleri

- Final olarak tanımlanmış bir metot parametresine sadece bir kere değer atanabilir.
- Metot parametrelerinin tamamen final olarak tanımlanmış olmalarında büyük fayda vardır.
- Bu şekilde parametrenin metot bünyesinde değişikliğe uğrama tehlikesi ortadan kaldırılmış olur.

The final local variable b cannot be assigned. It must be blank and not using a compound assignment

```
1 public final class FinalDeneme {  
2  
3     public void metod1() {  
4  
5         int a = 4;  
6         System.out.println("a'nin degeri: " + a);  
7         metod2(a);  
8         System.out.println("a'nin degeri: " + a);  
9     }  
10  
11     public void metod2(final int b) {  
12         b = b*2;  
13     }  
14 }
```

Final Metotlar

- Final olan bir metot ne alt sınıflarca yeniden yüklenebilir (method overloading) ne de saklı (hidden) tutulabilir.
- Cannot override the final method from FinalDeneme override.

```
public class FinalDeneme {  
1  
2     public final void finalMetod() {  
3         System.out.println("Ata sınıftaki finalMetod");  
4     }  
5 }  
6  
7 public class FinalDenemeSubClass extends  
8 FinalDeneme {  
9  
10    public void finalMetod() {  
11        System.out.println("Alt sınıftaki finalMetod");  
12    }  
13 }
```

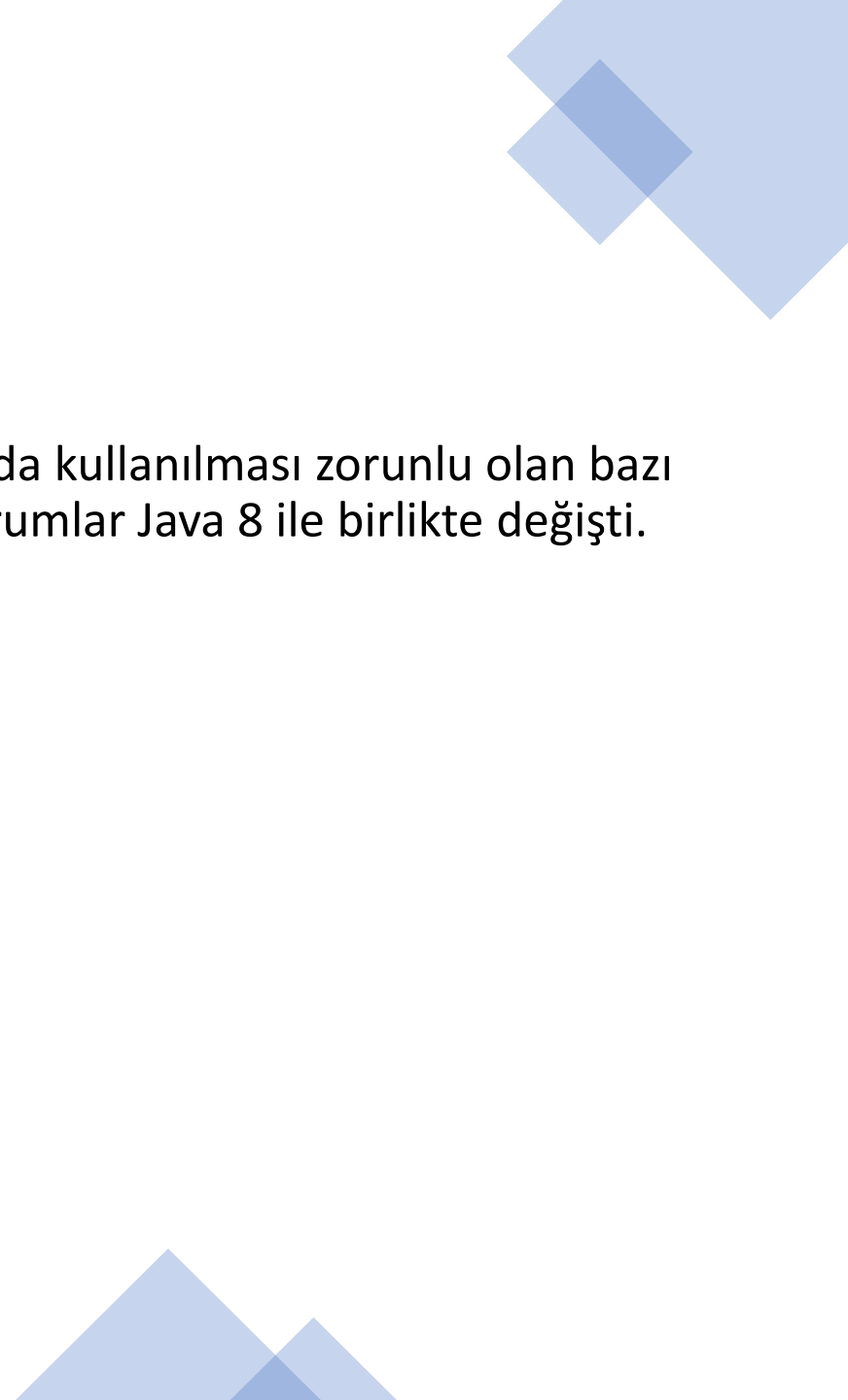
Final Sınıflar

- Final olan bir sınıf genişletilerek bir alt sınıf oluşturulamaz.
- The type FinalDenemeSubClass cannot subclass the final class FinalDeneme override.

```
1 public final class FinalDeneme {  
2  
3     public void metod() {  
4         System.out.println("Ata sınıftaki finalMetod");  
5     }  
6 }  
7  
8 public class FinalDenemeSubClass extends FinalDeneme {  
9  
10    public void metod() {  
11        System.out.println("Alt sınıftaki finalMetod");  
12    }  
13 }
```



"with Java 8"

- Java 8 öncesindeki kodlarda kullanılması zorunlu olan bazı case'ler vardı fakat bu durumlar Java 8 ile birlikte değişti.
- 

Effectively Final

- If we try to change variable we'll get a compilation error...
- `final int variable = 123;`
- If we create a variable like this, we can change its value...
- `int variable = 123;`
- `variable = 456;`
- But in Java 8, all variables are final by default. But the existence of the 2nd line in the code makes it non-final. So if we remove the 2nd line from the above code, our variable is now "effectively final"...
- `int variable = 123;`
- So.. Any variable that is assigned once and only once, is "effectively final".
- Bu sebeple final kullanımana, işaretlenmesine gerek kalmadı.