OPEN STREAM

210-11.JavaVMの動き

Make IT your Success

目次

- クラスの読み込まれ方
  - クラスローダー
  - クラスローダーの種類
  - 3つのクラスローダ
  - クラスパス
  - jarファイル
  - クラスが見つからないとき
- メモリについて
  - スタックとヒープ（おさらい）
  - 改めてスタックとは？
  - スタック(stack)
  - キュー(queue)
  - 変数とスタック
  - メソッド呼び出しとスタック
  - ガベージコレクション

この講義について

目的：

- JavaVMの動作に関してプログラマーとして知っておくべきことを学ぶ。
- コンパイル時・実行時のJavaVMの動きについて学ぶ。
- スタック・ヒープについておさらいし、メモリについて学ぶ。

ゴール：

JavaVMの動作・仕組みを理解する。

クラスの読み込まれ方

今まで特に意識せずJavaが提供しているクラスを利用してきたが、

コンパイル時（javacコマンド）、実行時（javaコマンド：JavaVM）に、どのように利用するクラスを読み込んでいるのか？

クラスの読み込みには、以下の２つを理解しておく必要がある。

①クラスローダー

②クラスパス

クラスの読み込まれ方

クラスローダー

Javaで作成したアプリケーションは、様々なクラスで構成される。

（自分で作製したクラスの他に、Javaで用意されているAPIなど、プログラム内で利用するクラスも）

JavaのクラスはJavaVMが読み込まないと実行されない。

自分が作成したクラスや、これまで利用してきたJavaが提供しているクラス（Stringクラス、Integerクラスなど API）はどのように読み込んでいるのか？

クラスの読み込みはクラスローダーが行う。

クラスローダーがJavaVMにクラスを読み込む。

クラスローダーはJavaVMの一部だが、オブジェクト(Javaのクラス)でもある。

クラスローダーの種類

クラスローダーは3種類ある。Javaの基本的なAPIを読み込むのは以下の2つ。

① ブートストラップ・クラスローダー

Javaが提供する基本的なクラスをロードする。

「環境変数JAVA_HOMEで指定されたフォルダ/jre/lib」にある「ファイルの拡張子がjar」であるファイルを読み込む。

※ プログラマが意識することはほぼありません。

② 拡張クラスローダー

拡張フォルダにあるクラスをロードする。

「環境変数JAVA_HOMEで指定されたフォルダ/jre/lib/ext」にある「ファイルの拡張子がjar」であるファイルを読み込む。

※ プログラマが意識することはあまりありません。

このようにクラスローダーは環境変数JAVA_HOMEの値を使ってクラスを読み込むため、

Javaコンパイラ・JavaVMを動作させるためには、環境変数JAVA_HOMEを設定しなければならない。

クラスの読み込まれ方

自らが作成したクラスや、第三者が提供するクラスはどのように読み込まれているのか？
これは以下のクラスローダーが行う。

③システムクラスローダー

クラスパス上にあるクラスをロードする。
「クラスパスで指定されたフォルダにあるクラス」「クラスパスで指定された拡張子がjarのファイル」を読み込む。

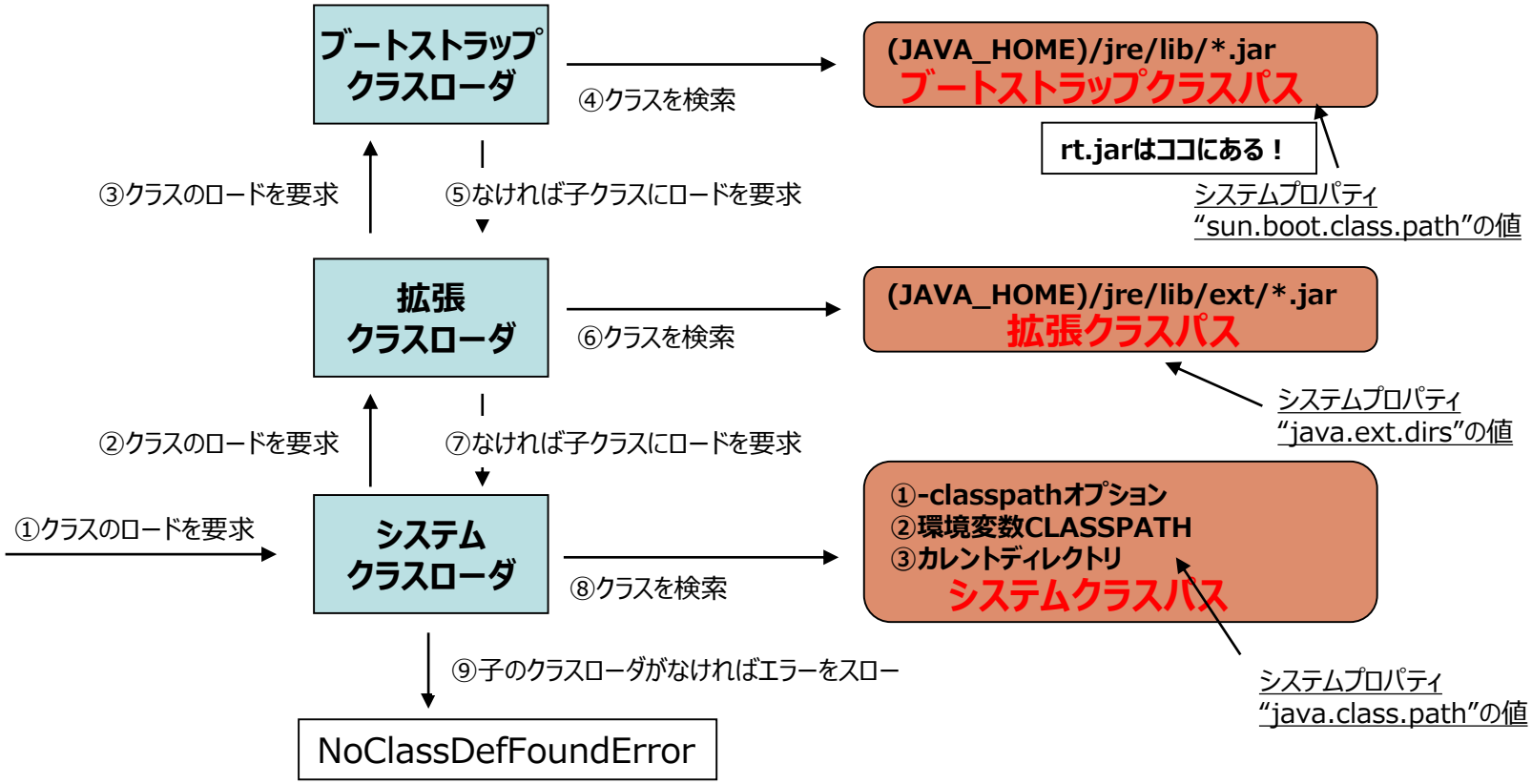
※プログラマが意識するのは主にこれになります。

今までクラスパスを指定していなかったのに動作していたのは何故か？
→ 指定しないとカレントフォルダを指定しているものとみなされるから。
→ 逆にいうとカレントフォルダ以外は指定されていない状態。

クラスの読み込まれ方

3つのクラスローダ

JavaVMが、クラスを読み込む必要があるとクラスローダーにクラス読み込みの要求を出す。

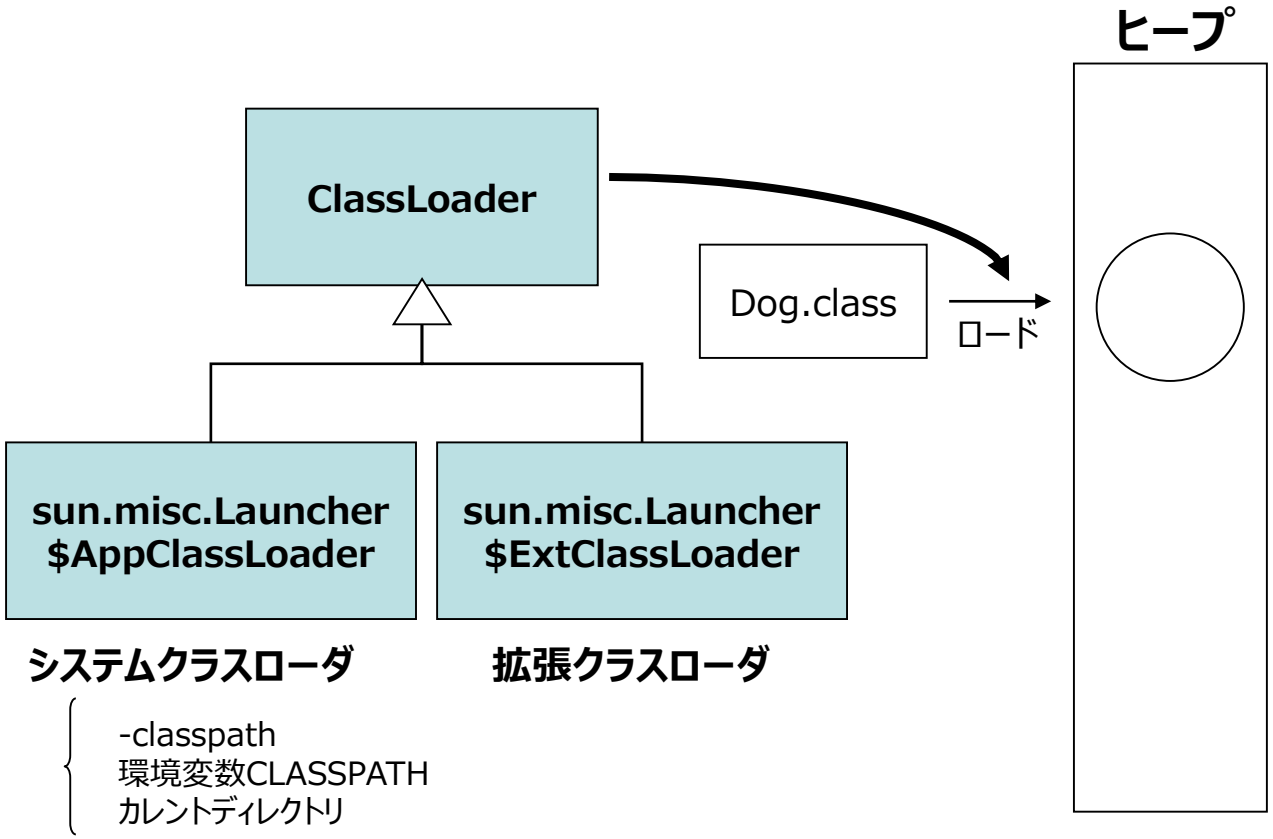


クラスの読み込まれ方

3つのクラスローダ

ブートストラップクラスローダ　・・・ JavaVMの中心部分のひとつ、ネイティブコード

システムクラスローダ、拡張クラスローダ　・・・ クラスで実装されている



クラスの読み込まれ方

クラスパス

Javaアプリケーションを実行（またはコンパイル）するときに、JavaVM（コンパイラ）がどの場所からクラスファイルを読み込めばいいかを指定するためのもの。

複数のフォルダ・拡張子がjarのファイルを指定できる。

クラスパスの指定方法は主に以下の２通りの方法をとる。

① コンパイル時（javacコマンド）・実行時（javaコマンド：JavaVM）に「-classpath」オプションで指定

② 環境変数CLASSPATHで指定

※ 両方指定されている場合①が優先される（排他的）

クラスの読み込まれ方

jarファイル

Javaプログラムの実行に必要なクラスファイルやデータファイルをひとつにまとめるためのもの

jar → Java Archive

- Jarファイル ○○○.jar

自分で作成したクラスをjarファイルにまとめることができる

- jar cvf myjar.jar ＊.class → 圧縮
- jar tvf myjar.jar → 内容の詳細
- jar xvf myjar.jar → 解凍

jarファイルはクラスパスに指定できる

- java –classpath /export/home/yourname/kadai.jar

クラスの読み込まれ方

クラスが見つからないとき

実際の開発の中でも「クラスが見つからない」というエラーに遭遇することがある。
（「NoClassDefFoundError」「ClassNotFoundException」といった例外）
その際は大抵クラスパスの指定・クラスファイルの配置に問題がある。

※ 今回の研修の後半ではWEBアプリを作成する。
様々なAPI(ライブラリやフレームワーク)を読み込むクラスローダー、ファイルパス、フォルダを意識する必要がある。
ここで「クラスが見つからない」というエラーに遭遇した時は、クラスパスの指定の問題か、WEBアプリ特有のフォルダに関することがほとんど。

※ NoClassDefFoundError …… 暗黙的なクラス読み込み、
アプリケーション起動時にクラスが見つからない。
→継承するクラスが無い、メインクラスが無い等

※ ClassNotFoundException …… 明示的なクラス読み込み、
アプリケーションが起動した後にクラスが見つからない
→プログラム実行中のクラスロードでクラスが無い等

クラスの読み込まれ方

アプリケーションごとに、どこからクラスをロードするのかを分けたい
→ Java 仮想マシンが動的にクラスをロードできるようようClassLoader のサブクラスを実装する

実装したサブクラスでは、主にロードするクラスファイルがあるディレクトリのパスを返すメソッドをオーバーライドする
つまりロードするクラスファイルがある場所を指定するためにメソッドをオーバーライドする

- ブートストラップクラスローダの場合は/jre/lib/
- 拡張クラスローダの場合は/jre/lib/ext/
- システムクラスローダの場合はクラスパスを指定する

メモリについて

スタックとヒープ（おさらい）

これまでの講義で変数が保持される場所として以下を説明してきました。

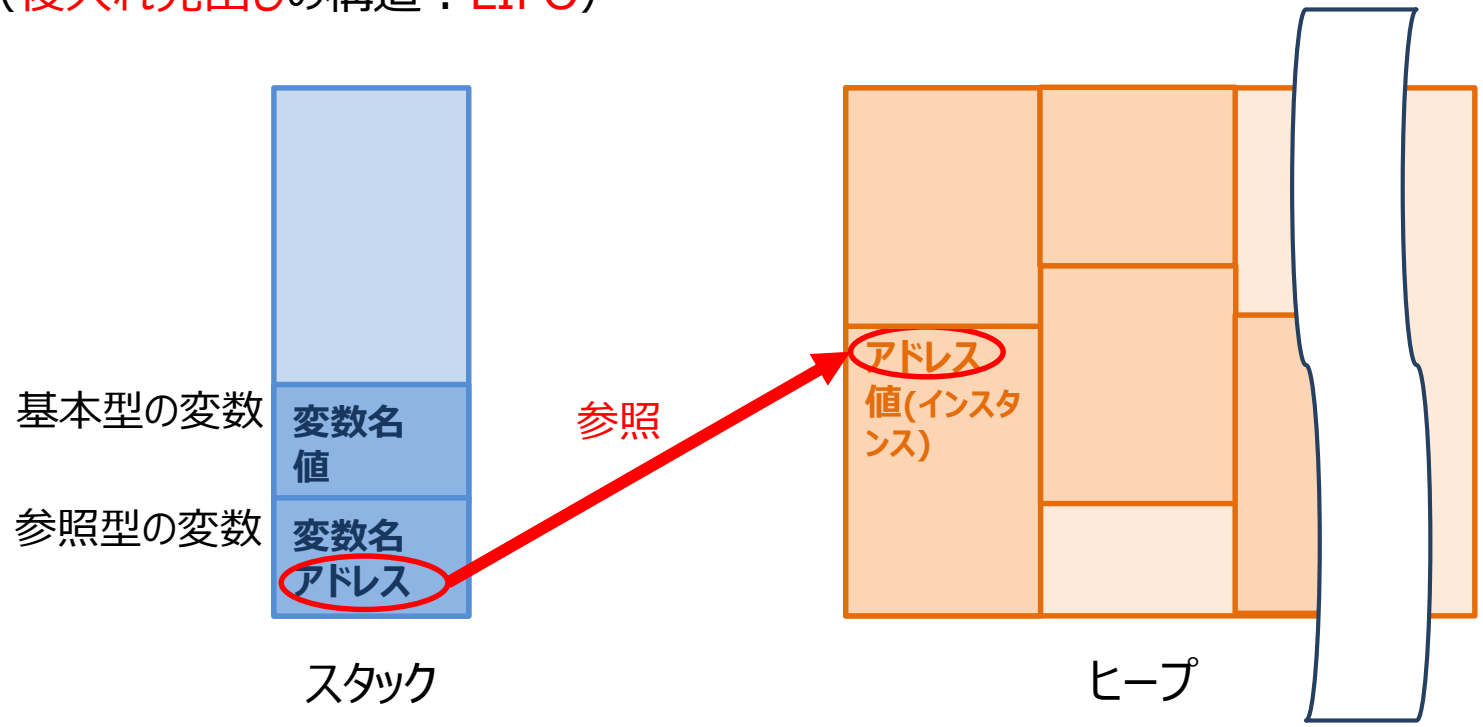
「基本型（プリミティブ型）」→スタック

「参照型（オブジェクト型）」→スタック＋ヒープ

改めてスタックとは？

スタックはデータ構造としての意味もある。

（後入れ先出しの構造：LIFO）



基本型はスタックに、変数名(識別子)と値が直接保持されます。

参照型はスタックに変数名(識別子)が保持されますが、値(インスタンス)はヒープに保持され、スタックにはヒープの参照情報(ヒープ内の場所)が保持されます。

スタック(stack)

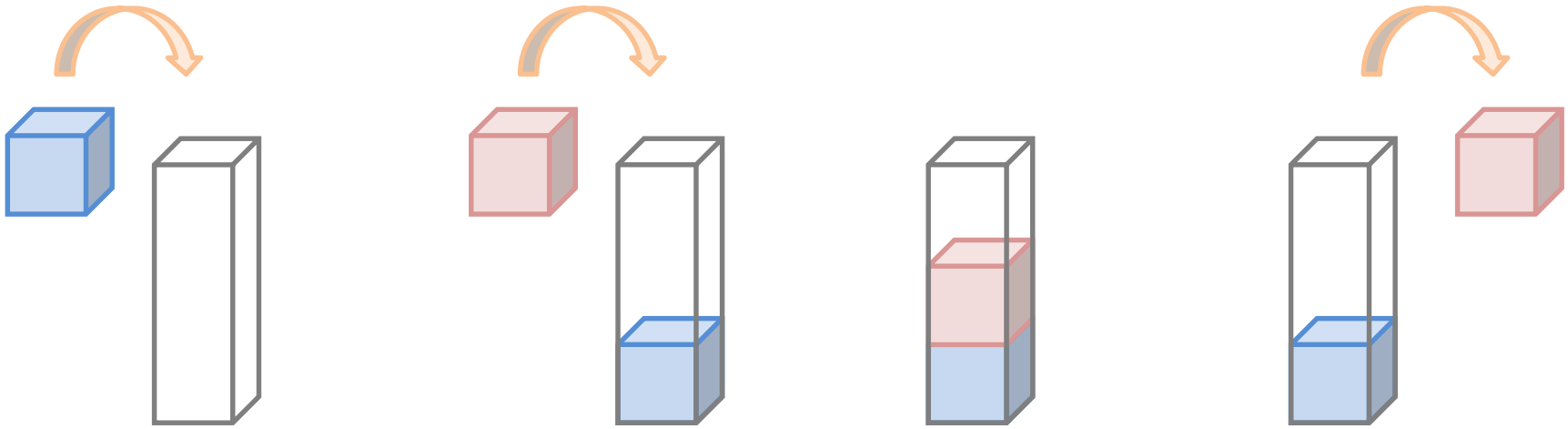
後入れ先出しの構造：LIFO(Last In First Out)

格納されたデータは最も新しい物から取り出されます。

つまり、最後に格納されたデータが最初に取り出されます。

データを格納する操作をpushを言い、

データを取り出す操作をpopと言います。



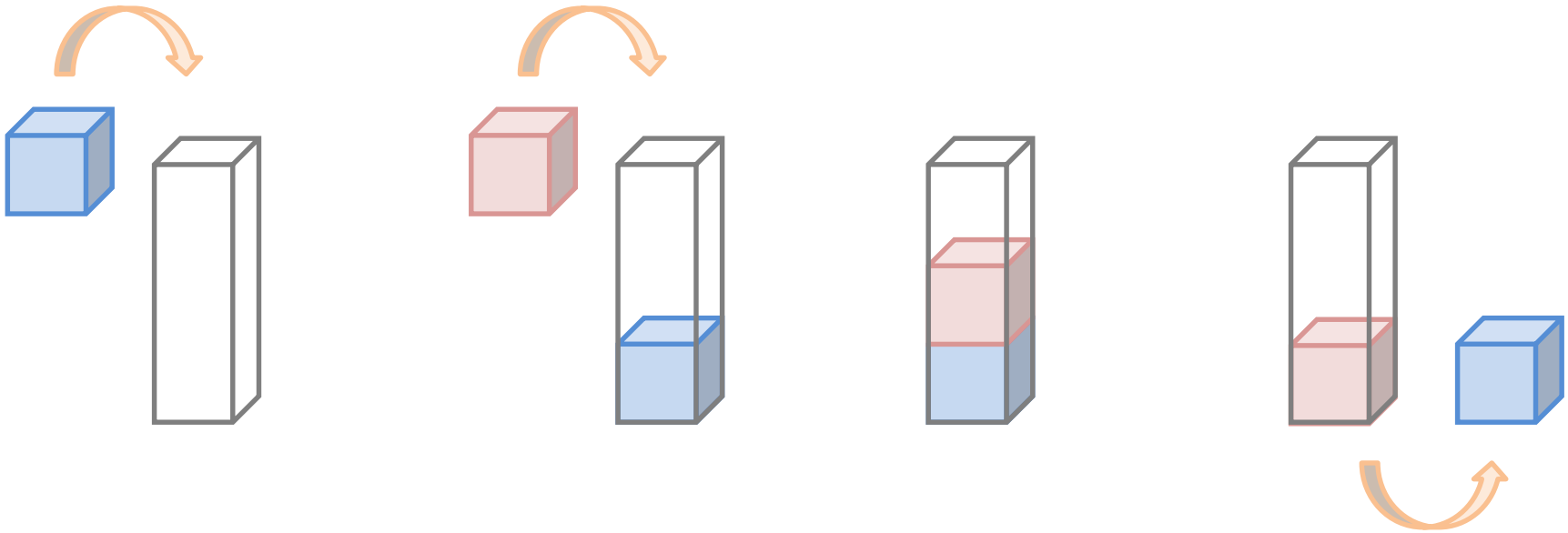
キュー(queue)

先入れ先出しの構造：FIFO(First In First Out)

最初に格納されたデータが最初に取り出されます。

データを最後に格納する操作をenqueueを言い、

先頭から順にデータを取り出す操作をdequeueと言います

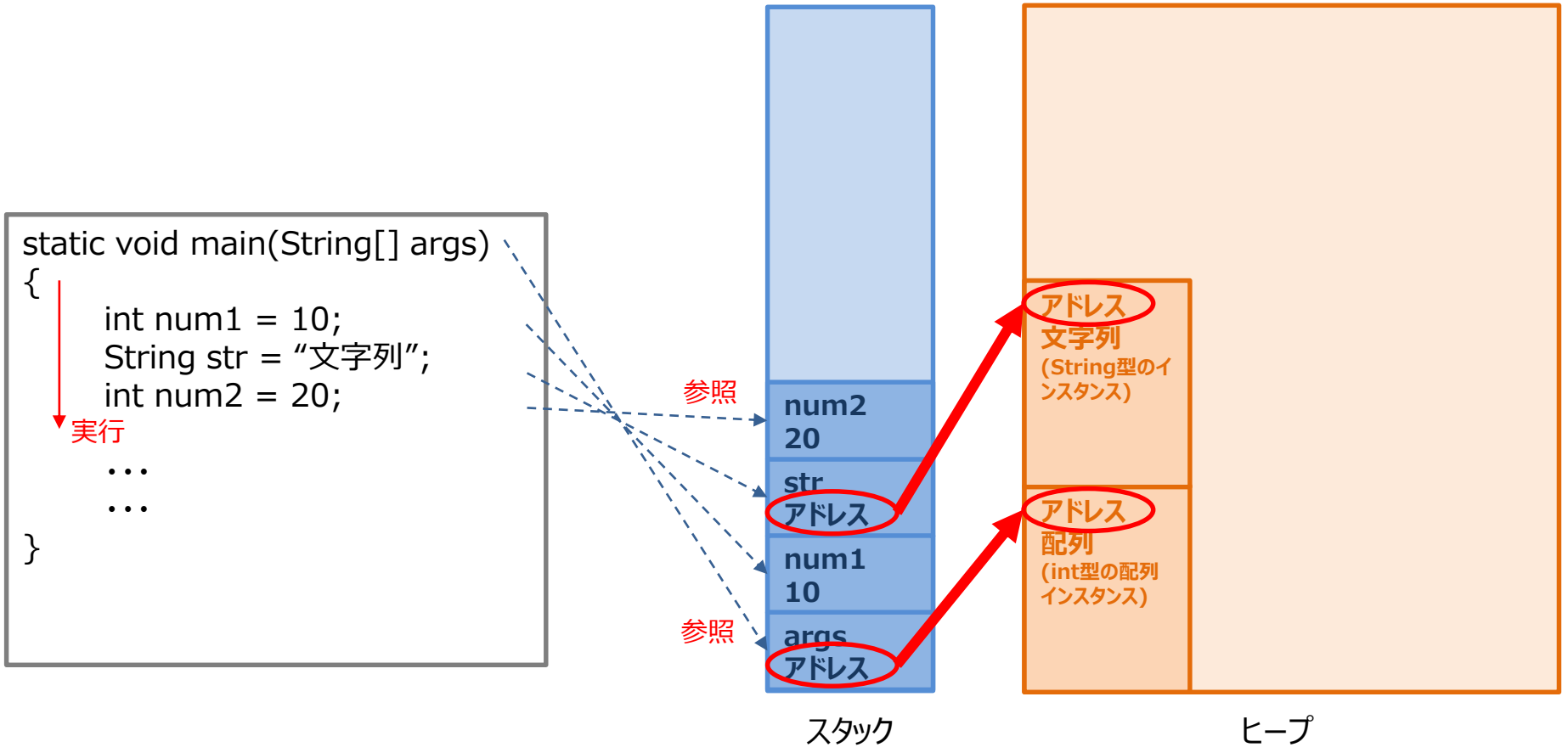


メモリについて

メソッドが実行されると、
そのメソッドのアドレスと変数がスタックに入れられます。（PUSH）
メソッドの実行が終了すると
スタックに入れたものを取り出します。（POP）
こうして元に戻ります。

変数とスタック

プログラムで変数が宣言されると、スタックに変数のデータが保持されます。
(参照型の場合、ヒープへの参照が保持されます)



メソッド呼び出しとスタック

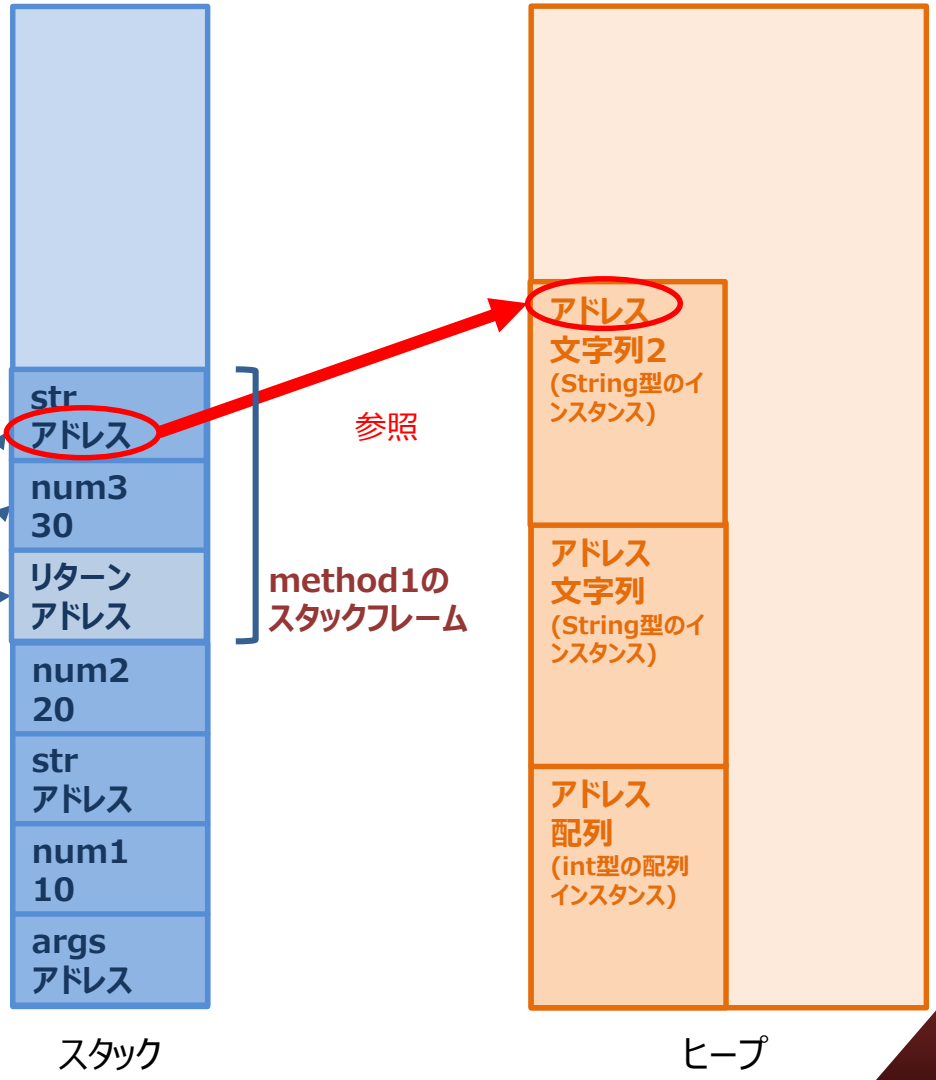
プログラムでメソッドが呼び出されると、
スタックにメソッドの呼び出し情報が保持されます。
また、メソッド内で宣言された変数が
スタックに保持されます。

```
static void main(String[] args)
{
    int num1 = 10;
    String str = “文字列”;
    int num2 = 20;
    method1();
}

static void method1()
{
    int num3 = 30;
    String str = “文字列2”;
}
```

実行

実行



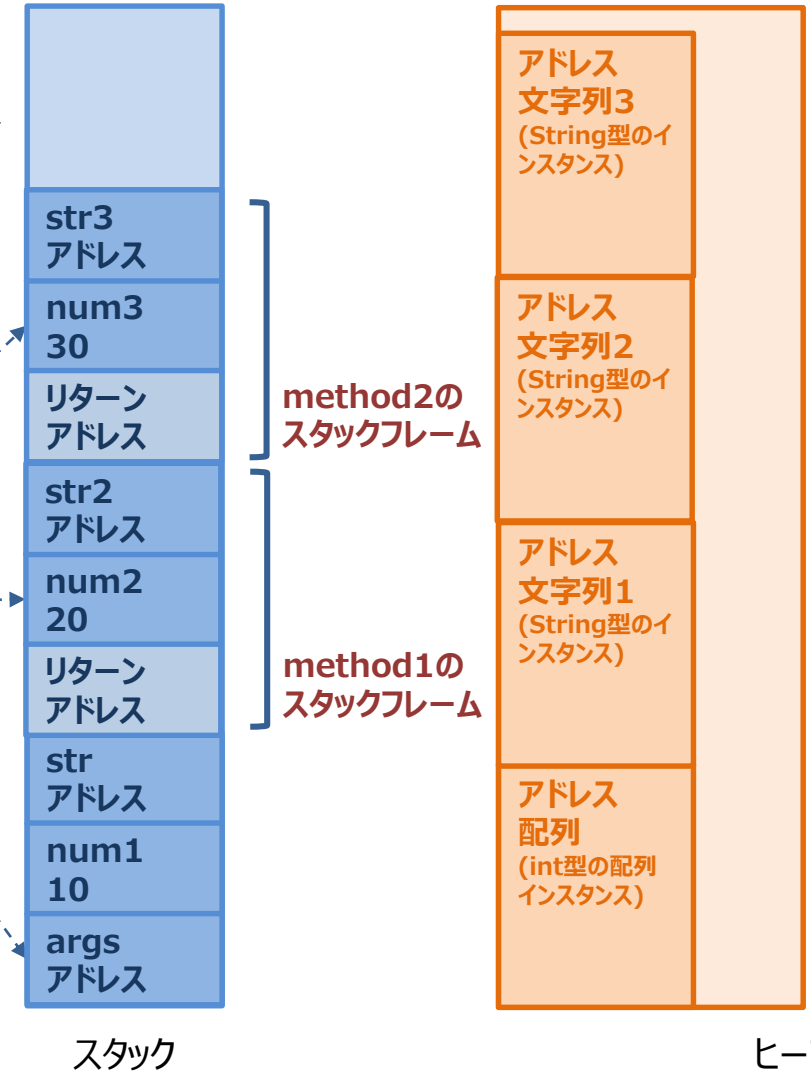
メソッド呼び出しとスタック

複雑な階層化されたメソッド呼び出しであっても、
メソッドの呼び出し情報と各メソッドで宣言された変数は
全てスタックに保持されます。

```
static void main(String[] args)
{
    int num1 = 10;
    String str = “文字列1”;
    method1();
}

static void method1()
{
    int num2 = 20;
    String str2 = “文字列2”;
    method2();
}
Static void method2()
{
    int num3 = 30;
    String str3 = “文字列3”;
}
```

実行



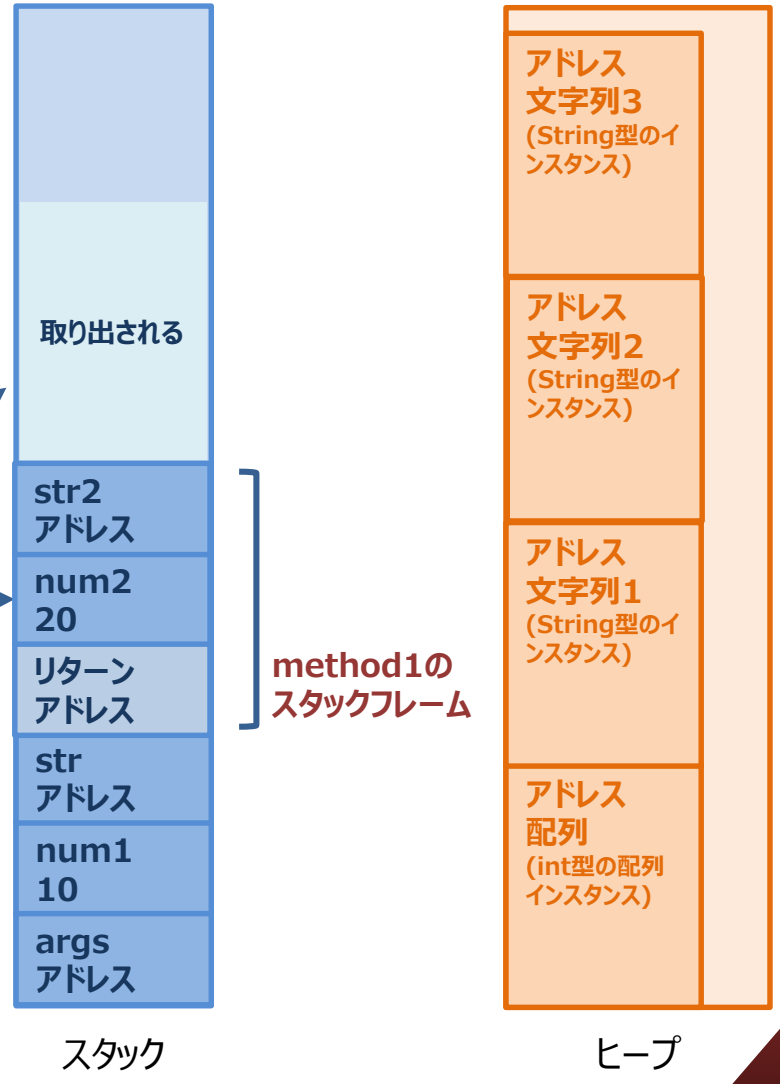
メソッド呼び出しとスタック

実行が終了したメソッドのスタックフレームは取り出されます。
終了したメソッドのスタックフレームは順次取り出されてゆき、
最後はmainメソッドのスタックフレームが取り出されます。

```
static void main(String[] args)
{
    int num1 = 10;
    String str = “文字列1”;
    method1();
}

static void method1()
{
    int num2 = 20;
    String str2 = “文字列2”;
    method2();
}
Static void method2()
{
    int num3 = 30;
    String str3 = “文字列3”;
}
```

実行
実行
実行終了



メソッド呼び出しとスタック

実行が終了したメソッドのスタックフレームは取り出されます。
終了したメソッドのスタックフレームは順次取り出されてゆき、
最後はmainメソッドのスタックフレームが取り出されます。

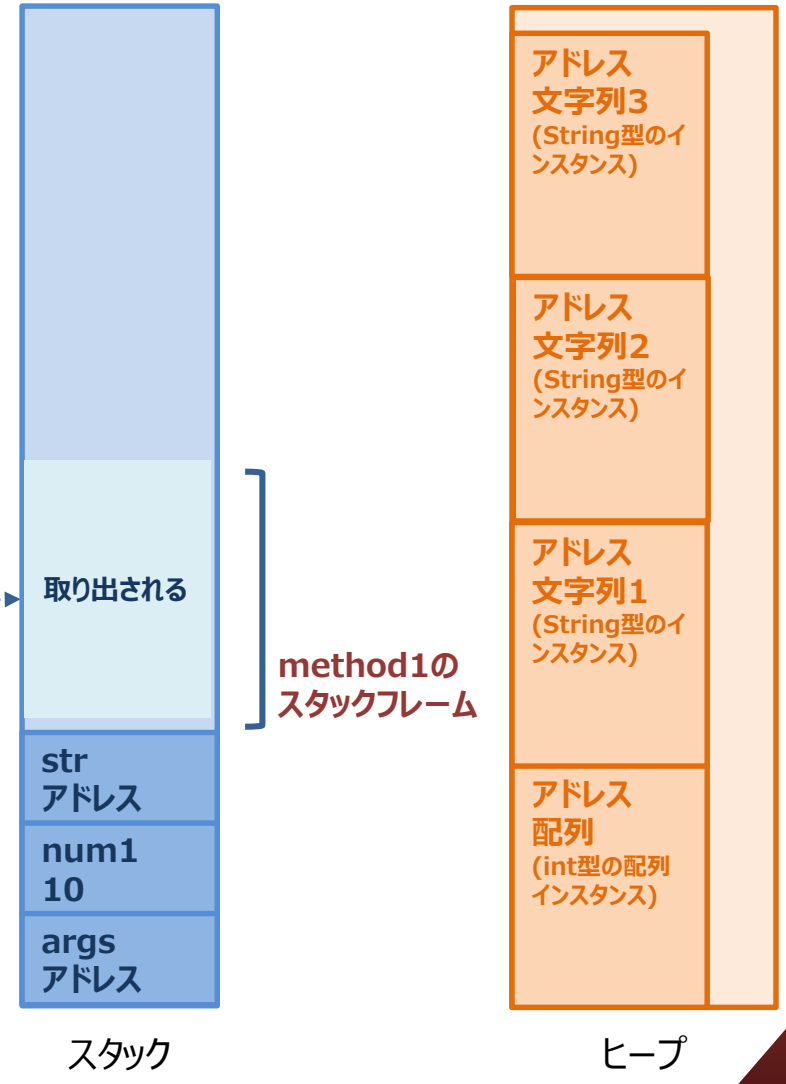
```
static void main(String[] args)
{
    int num1 = 10;
    String str = “文字列1”;
    method1();
}

static void method1()
{
    int num2 = 20;
    String str2 = “文字列2”;
    method2();
}
Static void method2()
{
    int num3 = 30;
    String str3 = “文字列3”;
}
```

実行

実行終了

実行終了



ガベージコレクション

メモリを確保しっぱなしでは、いずれ無くなってしまう。
メモリはいつ解放されるのか？

- スタック

  問題ない。

  構造上、メソッドの実行が終了すればメソッド実行前の状態に戻るため。

- ヒープ

  スタックのように動作するわけではないので確保した領域を解放する仕組みが必要。

  この仕組みをガベージコレクションという。

メモリについて

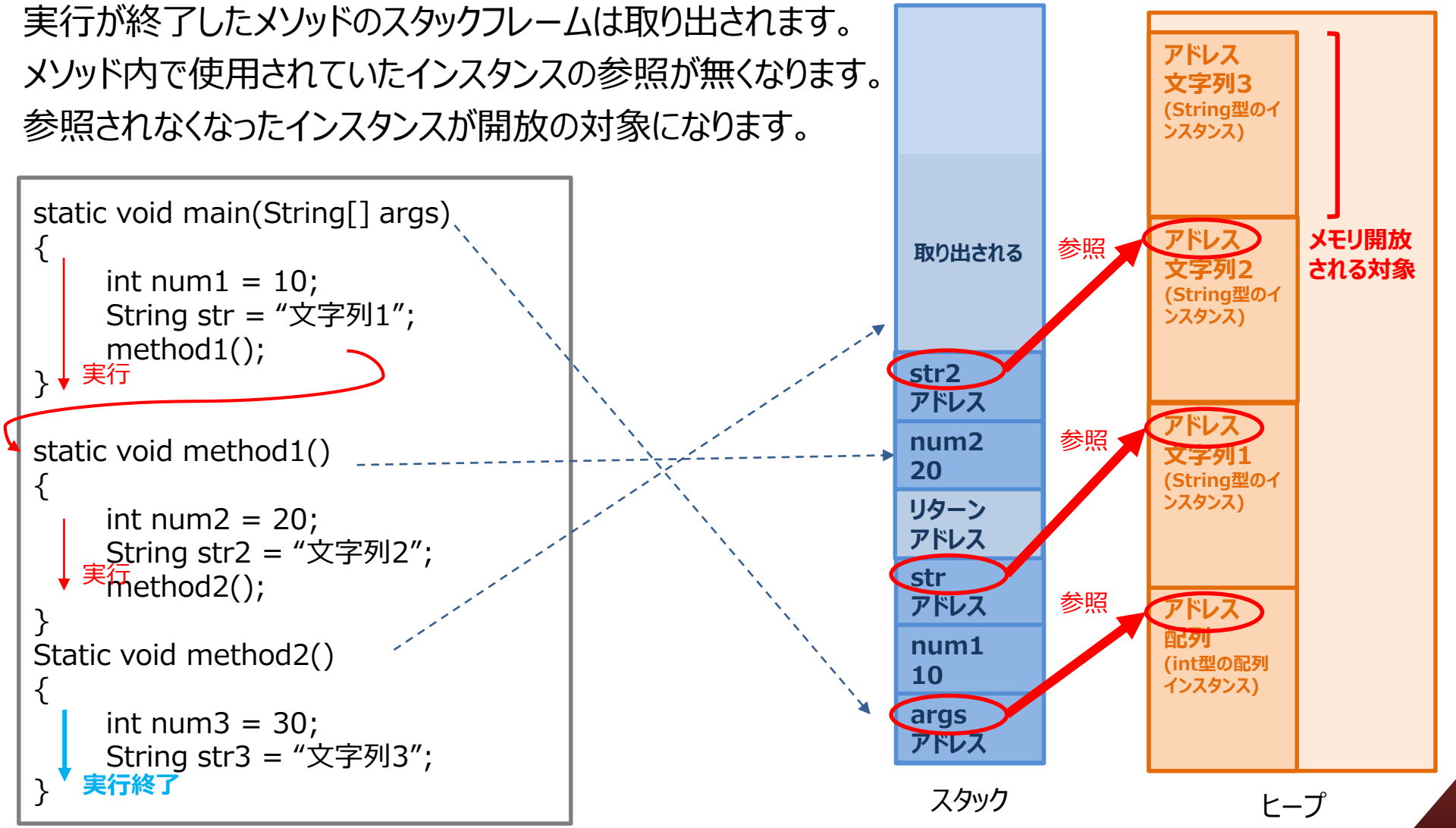
ガベージコレクション

実行が終了したメソッドのスタックフレームは取り出されます。
メソッド内で使用されていたインスタンスの参照が無くなります。
参照されなくなったインスタンスが開放の対象になります。

```
static void main(String[] args)
{
    int num1 = 10;
    String str = “文字列1”;
    method1();
}

static void method1()
{
    int num2 = 20;
    String str2 = “文字列2”;
    method2();
}
Static void method2()
{
    int num3 = 30;
    String str3 = “文字列3”;
}
```

実行
実行
実行終了



メモリについて

ガベージコレクションの対象となるのはどういった「参照型」の変数なのかというと、「ヒープのアドレスを保持するスタックが無い＝未使用であるもの」となります。

ガベージコレクションを行うのはガベージコレクタといいますが
これはJavaVMが「任意のタイミング」で動作させています。
（プログラマーがコントロールするものではない）

メモリについて

OutOfMemoryError

- ヒープ領域にメモリを確保できない場合に生じるエラー

StackOverflowError

- スタック領域にメモリを確保できない場合に生じるエラー

スレッドについて

いままで意識していませんでしたが、
Javaのプログラムはスレッドというもので動作しています。

いままで作成してきたクラスは「mainスレッド」が生成され、そのスレッド上で動作しています。
シングルスレッドのアプリケーションとなります。

対して同時に不特定多数の実行があるアプリケーションは
マルチスレッドのアプリケーションとなります。