

文字を格納できる char 型!?

あれ!? char 型って整数を格納するデータ型だったんじゃないの?

その通り!だけど char 型は、Unicode(ユニコード)という文字コード規格での一文字を、0~65535 の範囲の整数で表すこともできるんだ!

分類	型名	容量 (ビット)	表現できる範囲	変数宣言の例
文字	char	16ビット	0~65535 (1文字 (Unicode))	<code>char initial = 'A' ;</code>

Java のデータは「**文字としてではなく Unicode の文字コードに基づいて**」を格納しているんだ。例えば、「A」であれば Unicode の文字コードでは、「65」(16 進数表記だと 0x41)といった風に、それぞれの文字に 0~65535 までの一意 (onlyone) の番号が割り当てられているんだ。

文字と数字の紐付け方には、たくさんのルールがあるんだ。Java ではその一つである Unicode が最初から使用されていて、その中の UTF-16 というルールが使われているんだ!この UTF-16 の“16”は、16 ビットが処理単位だよということで、だから char は 16 ビットなんだ!

同じようなルールには、日本語環境ではいわゆる**シフト JIS** や **EUC-JP**、**JIS** などがあるんだ。Unicode の中でも、UTF-8 や UTF-32 というものもある。Java でもこれらのルールはもちろん扱えることができることを知っておこう!

```
public class UnicodeExample {

    public static void main(String[] args) {
        // 文字列を定義
        String myString = "A";

        // 文字列から Unicode のコードポイントを取得
        int unicodeCodePoint = myString.codePointAt(0);

        // Unicode のコードポイントを 16 進数で表示
        String hexRepresentation = Integer.toHexString(unicodeCodePoint);

        // 結果を出力
        System.out.println("文字列: " + myString);
        // 文字列: A
        System.out.println("Unicode のコードポイント: " + unicodeCodePoint);
        // Unicode のコードポイント: 65
        System.out.println("16 進数表記: 0x" + hexRepresentation.toUpperCase());
        // 16 進数表記: 0x41
    }
}
```

String と char の違い

あれ？先ほどの char でも文字を格納することができたけど、String と char って何が違うんですか？

char 型と String 型の大きな違いは、「**扱える文字の範囲（文字数）**」なんだ！

String では、**複数の文字列**を表すことができる

でも char 型は、**1 文字だけ**しか表すことができないんだ。

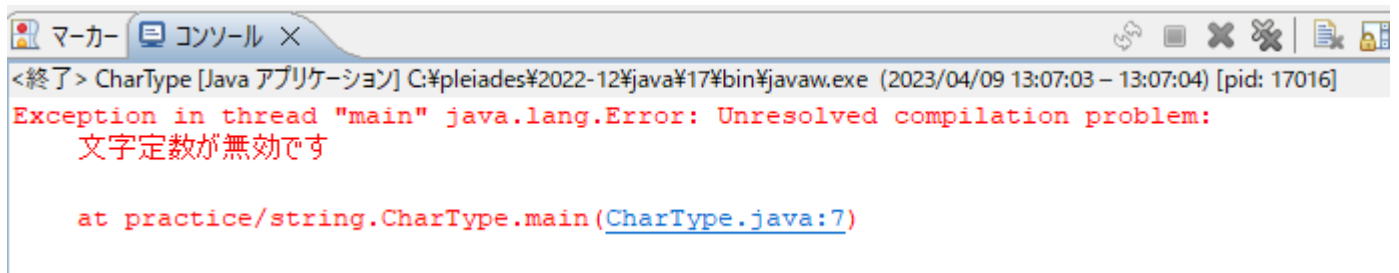
下の例を見てみよう

char 型で複数の文字列を入れた状態で実行するとエラーになっているのがわかるよね！

これが char 型と String 型の違いになるから覚えてこう！

```
1 public class CharType {
2
3     public static void main(String[] args) {
4         //char型に文字列を入れるとどうなるか見てみよう！
5         char name = 'akemi';
6     }
7
8 }
```

実行結果



The screenshot shows a Java IDE window with a tab labeled 'コンソール' (Console). The console output displays a compilation error. At the top, it says '<終了> CharType [Java アプリケーション] C:\pleiades\2022-12\java\17\bin\javaw.exe (2023/04/09 13:07:03 - 13:07:04) [pid: 17016]'. Below this, the error message is: 'Exception in thread "main" java.lang.Error: Unresolved compilation problem: 文字定数が無効です' (Exception in thread "main" java.lang.Error: Unresolved compilation problem: String literal is invalid). The error points to the line 'at practice/string.CharType.main(CharType.java:7)'.

エラーの意味を説明するよ。

「**java.lang.Error: Unresolved compilation problem**」

で「未解決のコンパイル問題があります」。

何が原因かというところ「**文字定数が無効**」つまり文字数に問題がありそうだね！

問題のある部分というのが、

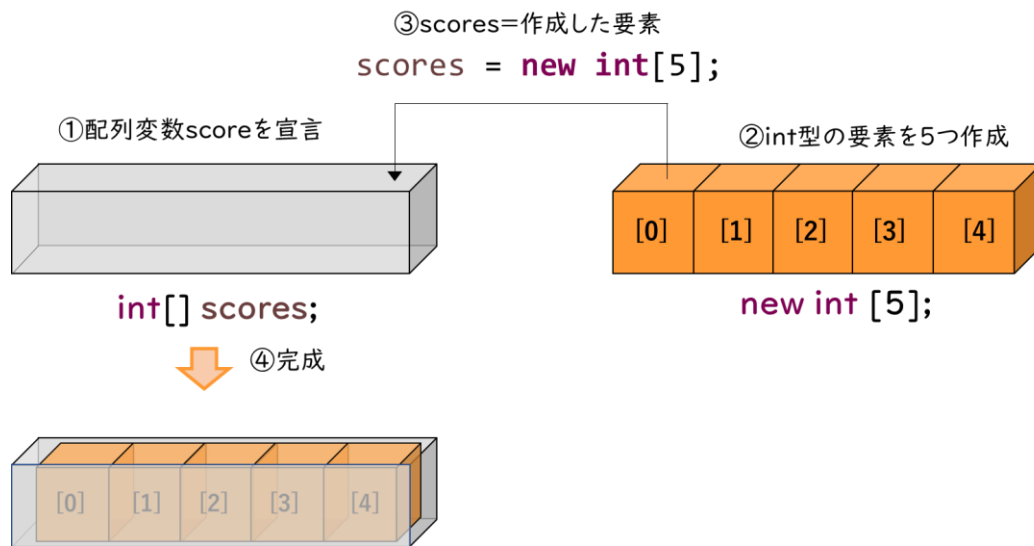
「**at practice/string.CharType.main(CharType.java:7)**」

で CharType.java ファイルの7行目に問題があることを教えてくれてるよ。

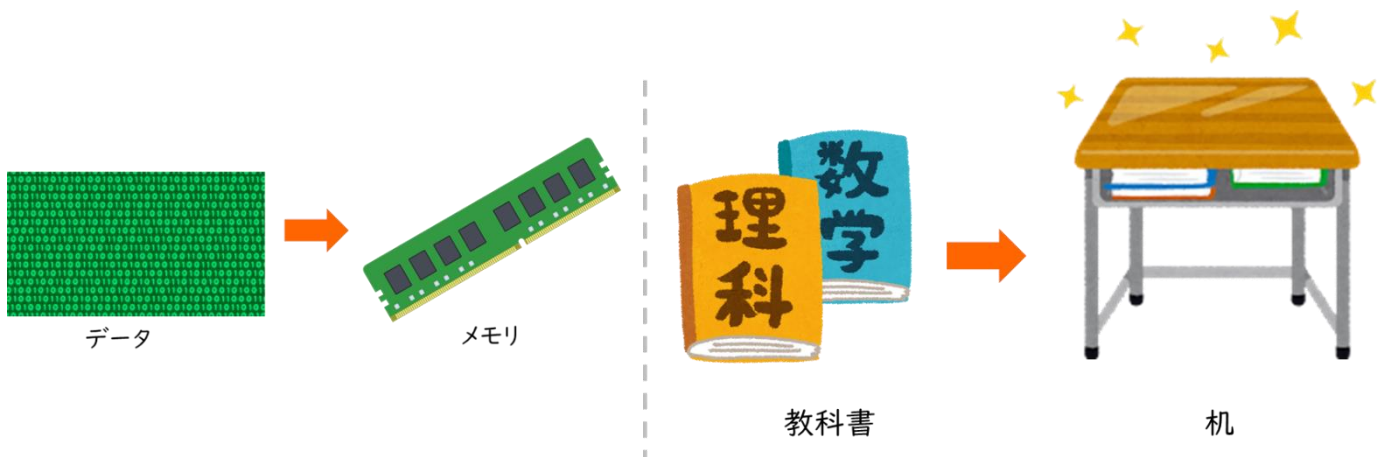
メモリと変数

まず、イメージ話なんだけど、私たちが配列を作成した時って下の図を使って理解を進めてきたよね。

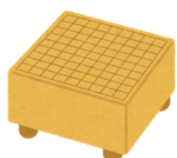
だけど、これは「私たち人間が理解しやすいイメージ図」であって「コンピュータの中では図のような構造になっていない」のが現実なんだ。



そもそも、コンピュータは使用するデータを「メモリ」に記録するんだ。みんなも学校で使うものは、ロッカーや机の中にしまっていていつでも使えるようにしておく感覚と一緒かな



そして、そのメモリの中は、囲碁や将棋盤見たく区画がきちんと整理されていて、その区画には住所（アドレス）が振られているんだ。



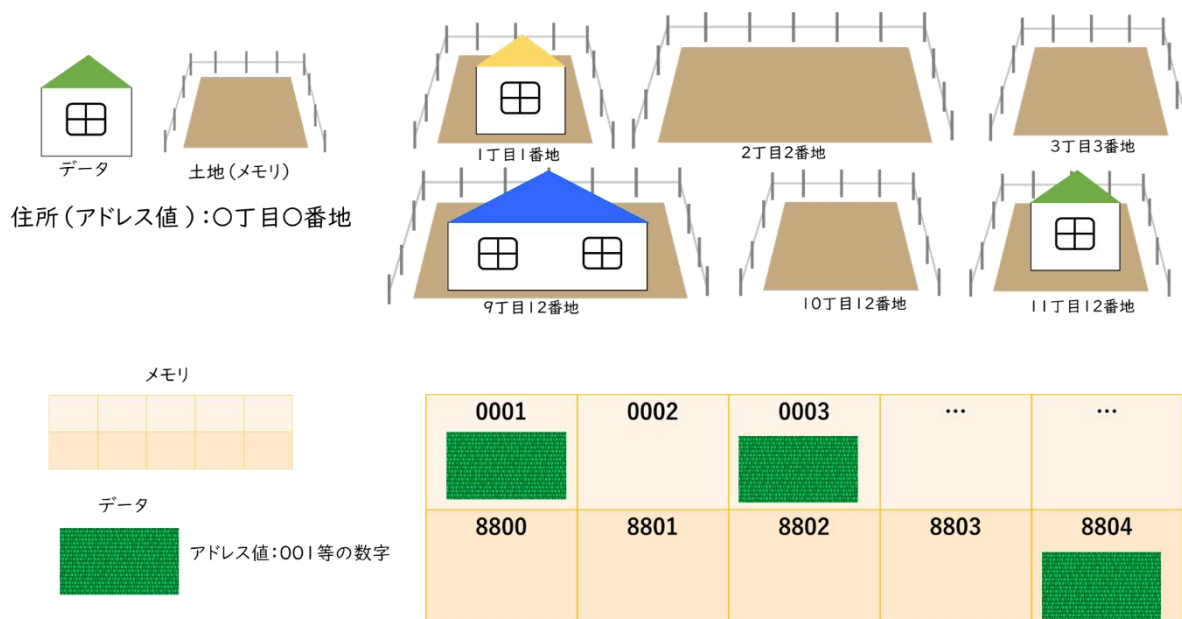
将棋盤

0001	0002	0003
0101
...	7700
8800

区画整理されたメモリ

数字は、住所で、
値が入った時に、どこにあるかを住所の数字から探していく。

土地に例えると、わかりやすいかもしれないね。土地がいくつかあって、そこには住所がちゃんとある。
 そして、その土地に、家が建てられていたら、もうその土地に新たに家を建てることはできないから、空いている土地を
 探す必要がある。この土地がメモリというイメージで家がデータというイメージがつけると良いかもしれない



まず、変数を宣言すると、空いている区画 (空いている土地) を変数 (データ (新たな家を作る)) のために、確保する
 んだ。で、その時に気を付けるのがデータ型 (家の大きさ) !

使用するデータ型によって容量が異なったよね!

家も住む人数によって家の大きさを調整するのと一緒で、使用するデータ型によってどのくらい区画 (土地) を使うの
 か異なるんだ。

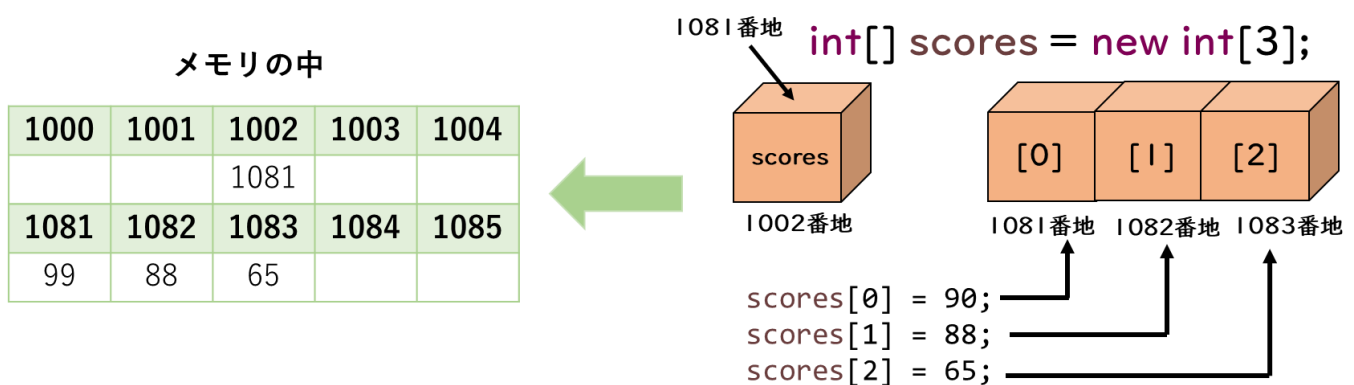
そして、使うデータ型 (家の大きさ) がきまったら、確保していた区画 (土地) に、値 (家) を、記録する (建てる) こと
 になるんだ。

では、早速なんだけど、例えば、以下の配列を作成したとしよう

```
// 配列の宣言と要素の作成
int[] scores = new int[3];
// 要素に値を代入
scores[0] = 90;
scores[1] = 88;
scores[2] = 65;
```

まず、メモリ上でどうなっているかを見ていくよ

まず例として、int 型の3つの要素を格納する配列を作成したとき、メモリの中ではどのようになっているかを考えていくね。



配列変数の宣言によって int[] 型変数が、new 演算子によって配列の実態（要素の集まり）がメモリ上の区画に作成されるんだ。

そして、配列変数には、3つの要素まるごとじゃなくて「最初の要素のアドレス」が代入されるんだ。

int scores[] = new int[3]が実行されたら!?

1: int型の要素を3つ持つ配列がメモリに作成される

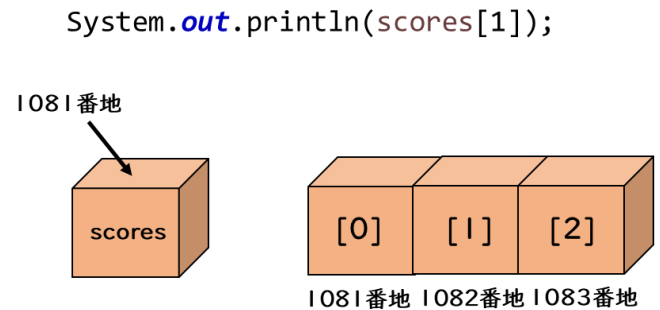
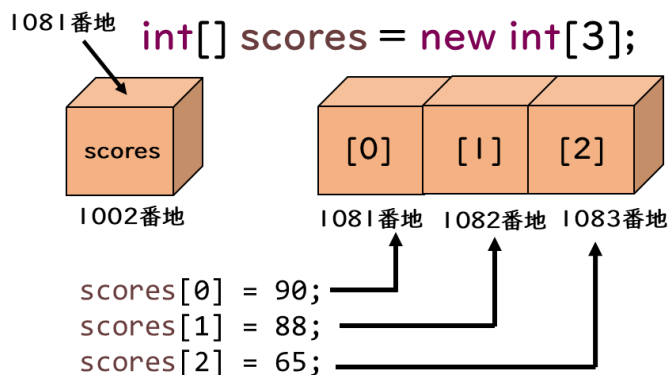
2: int[] 型の配列変数 scores がメモリ上に作成される

3: 配列変数 scores に配列の先頭要素のアドレスが代入される

配列変数名[n]と指定されたら

図を見ながら考えていこう! `scores[n]`と指定されたら、要素の先頭の番地を見つけて、見つけたら、配列の先頭の要素から `n` 個後ろの要素の区画を読み書きするんだ。

Index 番号 (添え字の番号) が 0 から始まるのは、定規の考え方と似ていて、定規のスタートは 1cm からではなく、0 からスタートだよ。それと一緒に添え字も 0 からスタートして、自分自身つまり、0 から何個後ろにあるかというのを考える必要があるんだ。下の例では、1081 から 1 個後ろを見たいから 1 を指定しているよね。



1: `scores`から1081番地を取り出して、配列(先頭要素)を見つける

2: 見つけたら配列の先頭要素から`n`個後ろの要素の区画を読み書きする

配列の後始末

ガベージコレクション

諒君どう?配列の裏側は理解できそうかな?

はい!大丈夫そうです。

そういえば疑問なんですけど、例えば以下のソースを見てほしいのですが、

前回の if 文の時に、if 文で宣言した変数は if 文の中でしか使えないといったと思うのですが、この場合、メモリの中ってどうなっているのでしょうか。

```
1 public class Main {
2
3     public static void main(String[] args) {
4         int coin = 200;
5
6         if ( coin > 100) {
7             int[] getCoins = {100,300,600};
8         }
9     }
10 }
```

なるほど!いい質問だね!確かに if 文の中に入れない限りつかわれないのでからずーっとメモリの中にあったらじゃまだよね。

じゃあソースを見ていこう

```
1 public class Main {
2
3     public static void main(String[] args) {
4         int coin = 200;
5
6         if ( coin > 100) {
7             int[] getCoins = {100,300,600};
8         }
9     }
10 }
```

getCoinsが
存在できる
範囲

1081番地

get
Coins

1002番地

100

300

600

[0]

[1]

[2]

1081番地

1082番地

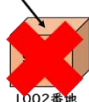
1083番地

1000	1001	1002
		1081
1081	1082	1083
100	300	600

範囲外の部分
つまり8行目以降からは
getCoinsは寿命が切れて存
在できない

メモリの中

1081番地



1002番地

100

300

600

[0]

[1]

[2]

1081番地

1082番地

1083番地

1000	1001	1002
1081	1082	1083
100	300	600

寿命が切れているからメモリから値が
不要領域 (ゴミ置き場) に移動する

値が残ってしまっている部分はどうす
るの?

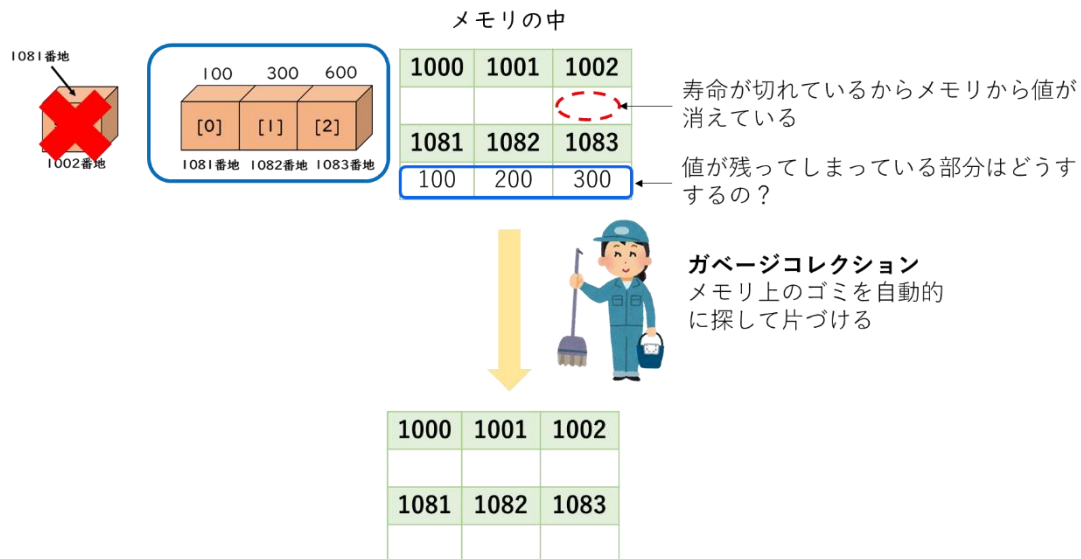
この if 文が通った場合だけ配列の初期化をしているのがこのソースだと思うんだけど、

諒君のいった通り、変数には寿命があって、自分が宣言されたブロックが終了するまでがその寿命だったよね。この考え方をスコープというのだけど詳しくは第 4 章でお話をするね。

つまり、このソースは、8 行目の時点で配列変数 getCoins は消滅する。なぜかといえブロックの範囲外 (寿命が切れる) から。だけど、ここで紛らわしいのが、初期化された要素 3 つ分。


```
if ( coin > 100) {
    int[] getCoins = {100, 300, 600};
}
```

つまりソースのこの赤で囲んだ部分だね。この 3 つの要素は普通の変数じゃないからブロックが終了してもメモリに残ってしまうんだ。その結果、何が起こるのかというと、配列はどの配列変数からも参照されない状態になるからメモリに残ってしまうんだ。



残ってしまった配列は、JAVA のプログラムからどのような方法を使ったとしても、読み書きすることはできないから、メモリの中ではごみになるよね。

ゴミとなった配列を放置し続けると当たり前だけどメモリを圧迫しちゃうよね。

皆も部屋のごみをすてるも忘れて部屋が汚くなるということになったことはない？

それと一緒にんだ。

本来であれば、「使用できなくなった配列 (ゴミ) は、使わないのだから、破棄してメモリの領域を確保する」というメモリの後片付けを指示しないとイケない、私たちの日常に例えると、ごみ収集業者にゴミを持って行ってもらうことをしないとイケないんだ。

だけど、JAVA にはガベージコレクションという仕組みが常に動いていて、実行中のプログラムが生み出したメモリ上

のゴミを自動的に探して片づけてくれるんだ。

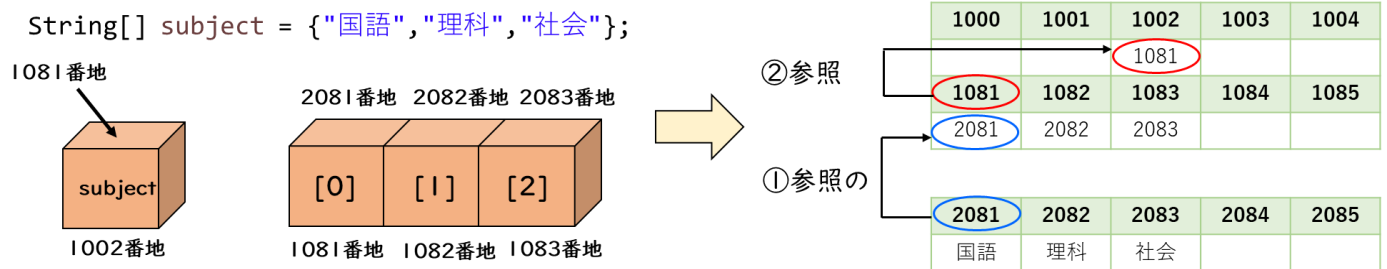
へー!すごいですね。自動掃除機のルンバみたいですね。

String の配列

そういえば、String 型の配列ってメモリはどうなっているのでしょうか？

というのも、配列は参照型だったからメモリの番地を格納するということだったけど、String も同じだったじゃないですか？この場合、メモリ中ってどうなっているのですか？

確かに、そうだね。答えを言うなら、「参照の参照」をしているんだけど、言葉だけだと、意味不明だから、図を見ながら一緒に確認していくね！



なるほど、String 型の配列の場合は、要素の中に値ではなく、「アドレス値」は入るのですね！

だから参照の参照なんですね！

確かに、言葉だけだとわかりづらいけど、図で見ると、イメージが付きやすいです！

良かった！物事の本質を知らないと、エラーやプログラムを書く時に、困ってしまうから、知識の一環として理解しておこう！

Null

先生、たまにソースを書いていると null と出てきたりするじゃないですか

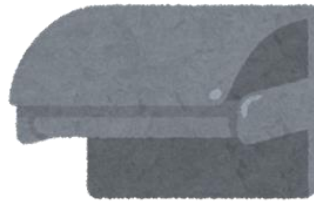
その null ってなんなのですか？

確かにそうだね、Null というのは、「何もない」という意味なんだけど、それだけだとイメージが付きにくいからトイレト
ペーパーでイメージをしてもらうよ。

値が0の状態



値がnullの状態



値が 0 の状態の時は、トイレトペーパーの芯は残っているけど紙が残ってないから、お尻がふけない状態。

値が null の状態は、もはやトイレトペーパーの芯すら残ってない状態。

じゃあ、いまの考えを踏まえたうえで、ソースコードをみながら null について詳しく見ていこう！

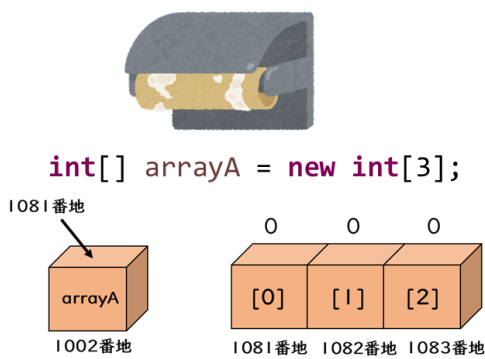
```
1 public class Main {  
2  
3     public static void main(String[] args) {  
4         int[] arrayA = new int[3];  
5         for (int i = 0; i < arrayA.length; i++) {  
6             System.out.println("arrayA[" + i + "]の値は" + arrayA[i]);  
7         }  
8         System.out.println("-----");  
9         int[] arrayB = new int[3];  
10        arrayB = null;  
11        System.out.println(arrayB);  
12    }  
13  
14 }
```



```
Console ×  
<terminated> Main (30) [Java Appli  
arrayA[0]の値は0  
arrayA[1]の値は0  
arrayA[2]の値は0  
-----  
null
```

このソースを理解するためにまずは状況を図で確認していこう！

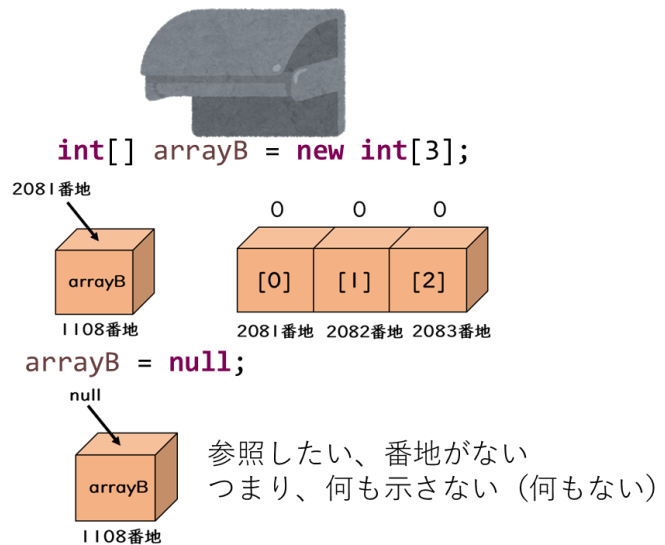
値が0の状態



初期値として0が入っている
つまり何かしら箱には入っている状態

1000	1001	1002	1003	1004
		1081		
1081	1082	1083	1084	1085
0	0	0		

値がnullの状態



1108	1109
null				
2081	2082	2083	2084	2085

なるほど、nullを入れた場合は、index 番号 0 の先頭番地を見ることができないから、何もない状態と一緒になるということですね！

その通り、さすがだね！

じゃあ、繰り返し for 文で要素の中身を確認した場合、どうなるのですか？

とても良い質問だね！では実行して内容を確認してみよう

```
1 public class Main {
2
3     public static void main(String[] args) {
4         System.out.println("-----");
5         int[] arrayB = new int[3];
6         arrayB = null;
7         System.out.println(arrayB);
8
9         for (int i = 0; i < arrayB.length; i++) {
10             System.out.println("arrayB[" + i + "]の値は" + arrayB[i]);
11         }
12     }
13 }
```



Console ×

<terminated> Main (32) [Java Application] C:\pleiades\2023-03\java\17\bin\javaw.exe (2023/09/15 13:25:38 - 13:25:39) [pid: 17636]

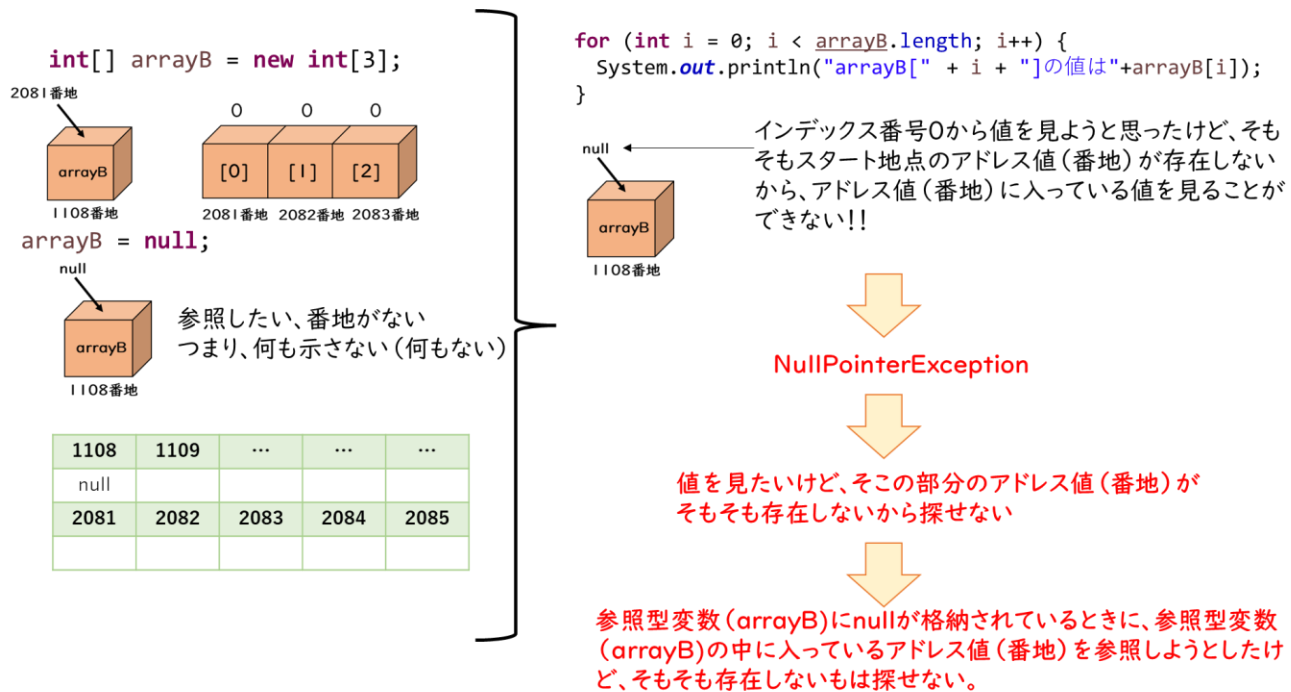
null

Exception in thread "main" java.lang.NullPointerException: Cannot read the array length because "arrayB" is null
at null_ex.ex2.Main.main(Main.java:11)

なんかとんでもないエラーがでてますね！

これはどういう意味のエラーなのですか？

「**NullPointerException**」というエラーで、「参照型変数に **null** が格納されているときに、参照型変数を参照しようとした場合」に、発生する例外エラーというものになるだ。



なるほど、何もないのに、探そうとしたって、そりゃあ探せないのだから、エラーの内容については、納得です。

もう一つの疑問なのですが、配列を初期化させた場合に、その値の中に **null** が入っている場合は、どうなるのでしょうか？

確かに、当然の疑問だね。

まずは、ソースを実行した後、図と一緒に内容を確認していこう！

お願いいたします！

```

1 public class Main {
2
3     public static void main(String[] args) {
4         String[] menu = { "ハンバーグ", null, "からあげ" };
5         for (int i = 0; i < menu.length; i++) {
6             System.out.println("menu[" + i + "]の値は" + menu[i]);
7         }
8     }
9
10 }

```



```

Console x
<terminated> Main (33) [Java
menu[0]の値はハンバーグ
menu[1]の値はnull
menu[2]の値はからあげ

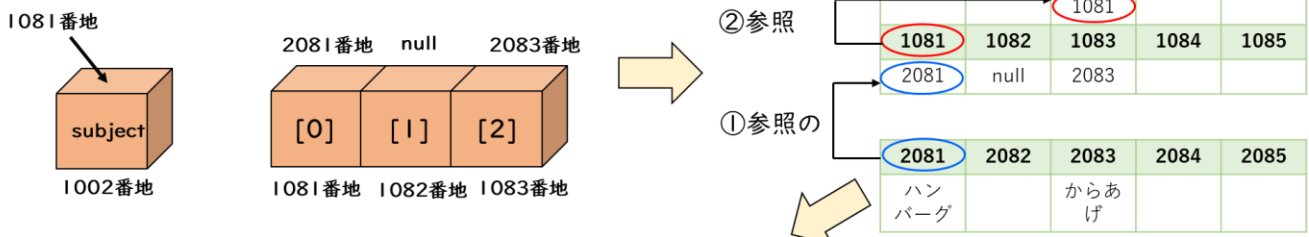
```

今度は、nullが入っているにも関わらず、「**NullPointerException**」がでていないのですね。

色々と、頭の中がこんがらがっています。どうしてエラーがでてこないのでしょうか？

確かにそう思うのは当然だと思うから、次は現状を図で確認して理解を深めていくね！

```
String[] menu = {"ハンバーグ", null, "からあげ"};
```



```

for (int i = 0; i < menu.length; i++) {
    System.out.println("menu[" + i + "]の値は" + menu[i]);
}

```

iが0の時、1081番の箱の中の2081番地を見て、2081番の中の「ハンバーグ」を表示する

iが1の時、1082番の箱の中は、「null」を表示する

iが2の時、1083番の箱の中の2083番地を見て、2083番の中の「からあげ」を表示する



1082番地の要素を参照しているわけであって、nullそのものを参照しているわけではないので、エラーが発生しない。

なるほど、先ほどの例では、そもそも参照先がない。つまり1081番地そのものがないからエラーになってしまったものに対して、今回は、箱そのものは存在しているから、エラーがおきていないのですね！

その通り！

なるほど、どこに null を入れるかによってとても意味が違うのですね！非常に勉強になります！

それは良かった！

ここまでが、配列の基本なのだけど、理解の方は大丈夫そう？

何となくです。ただ、まだ頭の中の整理が完璧ではないです。

そうだね。この部分は概念として本当に大事なんだ。どうしてかという、他の言語でも同じ考え方をするから。

例えば C 言語とか、この部分の理解を深めておくと、非常に言語の取得はスムーズなんだ。だからもう一度自分で図を書いて頭の中を整理整頓しておこう！

多次元配列

多次元配列とは？

先生、「多次元配列」と聞いただけでちょっと嫌なんですけど。

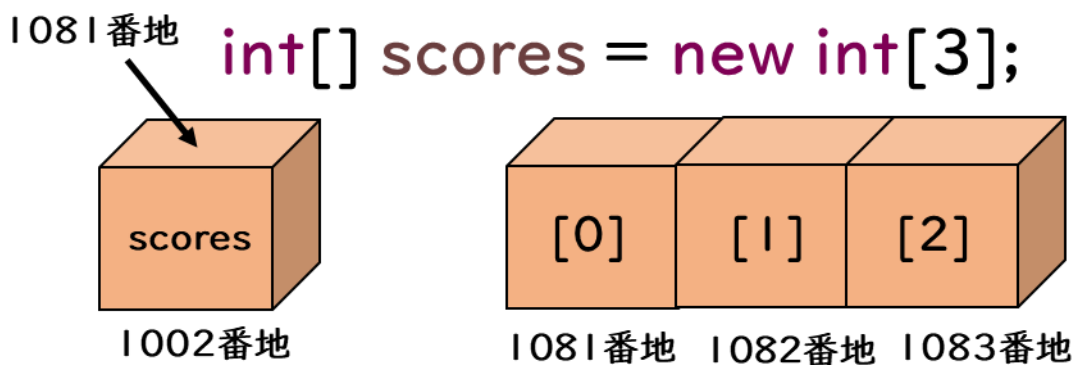
だって言葉通りにとらえてたら、配列がたくさんあるみたいな意味ですよ。

まあ、諒君の言っていることは間違えてはいないね。

多次元配列というのは、簡単に言えば、「配列の配列を作れる」という意味なんだ。

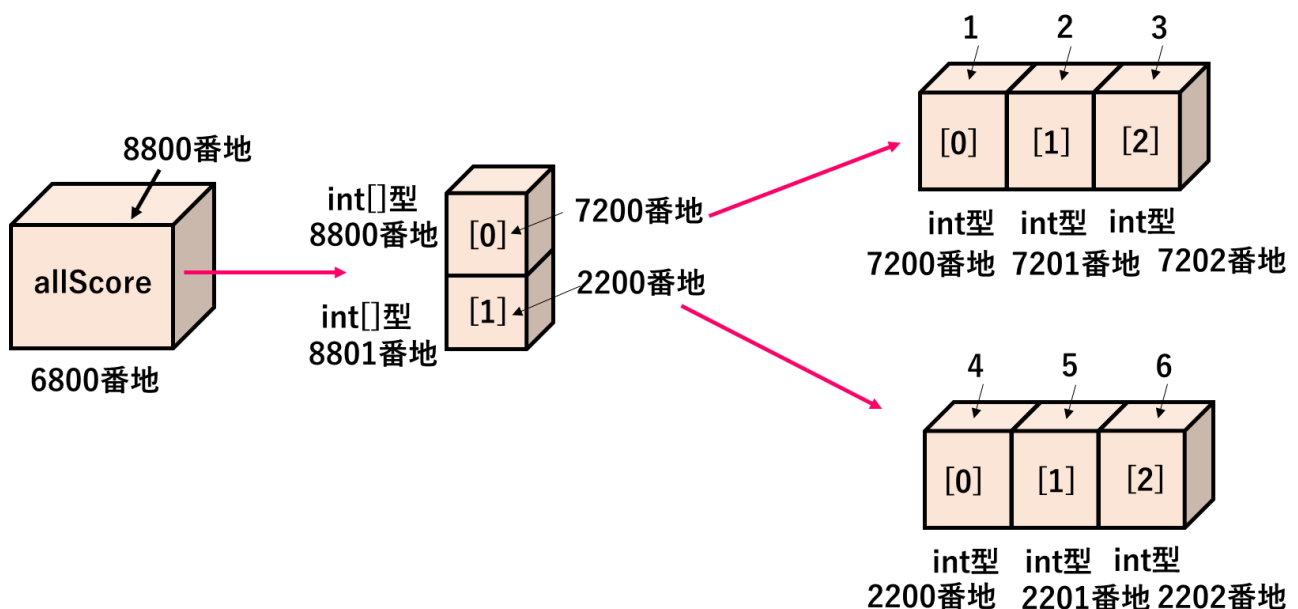
今回は多次元配列の中の 2 次元配列を例に話をしていくよ

まずは、復習なのだけど、今までの配列って下の図のように考えてきたよね。



これを「1 次元配列」という言い方をするんだ。

じゃあ 2 次元配列はどうかというと下の図のように、横の並びに縦の並びが追加されるんだ。



多次元配列というのは、「2 次元以上の配列」のことを多次元配列と呼ぶことも覚えておいてほしいかな！

で、現場でこれが使われるかという、あまり使う機会は正直少ないと思う。だけど科学計算技術等では多く利用されているんだ。だから皆がお目にかかる機会は少ないと思うけど、考え方は非常に大事だから覚えておいてね！

では、早速構文を見ていこう

宣言

要素の型[][] 配列変数名; int[][] scores;

代入

配列変数名 = new 要素の型[縦の要素数][横の要素数]; scores = new int[2][3];

要素への値の代入

配列変数名[縦の添え字(インデックス番号)][横の添え字(インデックス番号)] = 値; scores[0][0] = 100;

参照

配列変数名[縦の添え字(インデックス番号)][横の添え字(インデックス番号)] を書くだけ

例) System.out.println(配列変数名[縦の添え字][横の添え字]); System.out.println(scores[0]);

省略した書き方

要素の型[][] 配列変数名 = new 要素の型[縦の要素数][横の要素数]; int[][] scores = new int[2][3];

要素の型[][] 配列変数名 = new 要素の型[][]{{値1, 値2},{値3, 値4}...}; int[] scores = new int[] {{60, 70}, {80, 90}};

要素の型[][] 配列変数名 = {{値1, 値2},{値3, 値4}...}; int[][] scores = {{60, 70}, {80, 90}};

縦の要素の個数を調べる

配列変数名.length; scores.length

横の要素の個数を調べる

配列変数名[縦の添え字(インデックス番号)].length; scores[0].length

構文を見たところで、早速ソースコードを書いてみよう

```
1 public class Main {
2     public static void main(String[] args) {
3         // 2行3列の配列を宣言
4         int[][] allScore = new int[2][3];
5         // 配列に値を代入
6         allScore[0][0] = 1;
7         allScore[0][1] = 2;
8         allScore[0][2] = 3;
9         allScore[1][0] = 4;
10        allScore[1][1] = 5;
11        allScore[1][2] = 6;
12
13        // コンソールに値を表示
14        System.out.println(allScore[0][0]);
15        System.out.println(allScore[0][1]);
16        System.out.println(allScore[0][2]);
17        System.out.println(allScore[1][0]);
18        System.out.println(allScore[1][1]);
19        System.out.println(allScore[1][2]);
20    }
21 }
```

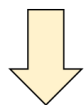
インデックス番号指定の仕方が複雑すぎて意味不明ですね。

確かにそうだね。ではどうやって指定するのかの部分を図で確認していこう

```
int[][] allScore = new int[2][3];
```

2行3列

縦列
横列



配列に値を代入

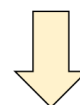
```
allScore[0][0] = 1;
allScore[0][1] = 2;
allScore[0][2] = 3;
allScore[1][0] = 4;
allScore[1][1] = 5;
allScore[1][2] = 6;
```



	横列		
	0	1	2
縦列 0	1	2	3
1	4	5	6



	横列		
	0	1	2
縦列 0	1	2	3
1	4	5	6



実行すると5が表示される

なるほど、このように表で考えるとわかりやすいですね！

これであれば、僕も指定できそうです。例えば 3 を表示させたい場合は、縦の index 番号が 0 横の index 番号が 2 だから、[0][2]と指定すればよいということですよ！

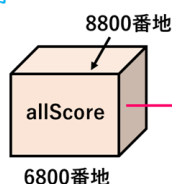
その通り!さすが諒君、理解が早いね!ではメモリ上はどうなっているのかも一緒に確認していくよ!

```
int[][] allScore = new int[2][3];
```

2行3列

縦列
横列

```
allScore[0][0] = 1;
allScore[0][1] = 2;
allScore[0][2] = 3;
allScore[1][0] = 4;
allScore[1][1] = 5;
allScore[1][2] = 6;
```



メモリの中



6800	6801
8800	

縦列	8800	8801
	7200	2200

横列		
7200	7201	7202
1	2	3

横列		
2200	2201	2202
4	5	6

なるほど、メモリの中はこんな感じになっているのですね、
図があると、どういう状態なのかが非常にわかりやすいです。

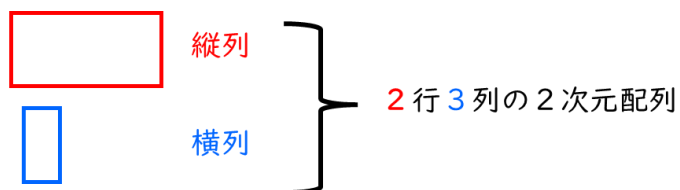
それは、良かった、それでは下のソースは、何行何列の配列かわかる？

```
1 public class Main {  
2  
3     public static void main(String[] args) {  
4         int[][] allScore = {{1,2,3},{4,5,6}};  
5  
6     }  
7 }
```

いやわからないです。

だよね。どうやって何行何列の配列なのかを判断するのか一緒に見ていくよ！

`int[][] allScore = {{1,2,3},{4,5,6}};`



なるほど! {}の中にある{}の個数が、縦列の数になって
{}の中にある値が横列の個数になるのですね!

その通り!これで、初期化された2次元配列を見てもすぐに何行何列か判断することができそうだね!