Java研修

研修予定一覧

HelloWorld

プログラミングファイル作成

まずはファイルを作成しよう。 コーディング用に用意したフォルダをエクスプローラで開いて、ファイルの新規作成で拡張子.javaファイルを作成する。

`HelloWorld.java`

作成したファイルをVSC等のエディタで開いて、プログラムを記述。

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello world");
    }
}
```

コマンド実行手順

- コマンドプロンプトを起動する。

【起動方法例】

1. windowsキー＋rキーで「ファイル名を指定して実行」を起動する。
2. `cmd`と入力してOKボタン または、 Enterキーを押下する。

`cd` コマンドでカレントディレクトリを移動する。

```
cd c:¥java
```

javacコマンドで実行したいjavaファイルをコンパイルする。

```
javac HelloWorld.java
```

コンパイルが完了すると、拡張子`.class`ファイルが作成される。 `.class`を省いたファイル名をjavaコマンドで実行する。

```
java HelloWorld
```

Question01

型と変数

型と変数名の宣言

型を書いて変数を書きます。

```
int `hanakoAge`;
```

これで、int型の`hanakoAge`変数が宣言できました。 `int`は、整数を扱う型なので`hanakoAge`は、整数を扱う変数となります。

いい変数名　わるい変数名

変数名は、自由に付けることができます。ただし、以下のルールがあります。

- 予約後は使用できない。（`public, private, int, String...`）
- 先頭文字に数値は使えない。
- camelCase記法を使う
    - 英字小文字を使う
    - 言葉の区切りだけ大文字にする。

camelCase記法は、単語と単語を組み合わせて変数名に意味を持たせるようにします。

例 camelCase記法

- hanakoAge
- addStr
- sendFile
- isDaikichi

他にも単語と単語の間をアンダーバー"_"でつなげるsnake_case記法もあります。

例 snake_cace

- hanako_age
- add_str
- send_file
- is_daikichi

C++やHTMLではsnake_case記法で書かれます。
JavaではcamelCase記法が推奨されているので、Javaでプログラムをするときはは camelCace記法で書いてください。

変数名の命名

変数名は自由に付けれますが、**変数の目的に合った意味をもつように**名称を工夫しましょう。
例えば、同じシステムの中で人、犬、猫を扱う事があるとして、変数名`hanakoAge`は、何を表す変数名でし

ょうか。はなこさんの年齢？はなこさんって人ですか？人だと思いこんで変数hanakoAgeを使っていたら、実は犬用や猫用の変数かもしれません。
hanakoAgeと変数名を付けた私の意識としては、hanakoAgeは犬です。我が家の愛犬の名前です。
もう少し明確な変数名にするなら、kumanoHouceDogHanakoAgeで熊野家の犬である花子の年齢って読めますかね。変数を使う範囲がもう少し幅広く犬なら使える変数であれば、dogAgeとすれば犬の年齢です。
hanakoAgeは研修内だけで使う変数名であり、皆さんもhanakoAgeは犬の年齢だと認識したと思いますので、このまま使います。

変数名の長さ

変数名の長さについてですが、変数名は長くなるとタイピング数が増え、スペルミスの可能性が高くなります。しかしIDEの補完機能もありますし、コピペを使ってもいいので、変数名を短くすることにこだわるよりは、変数名で意味が通じる事に重点をおきましょう。

初期化

初期化は以下のように記述します。

```
int `hanakoAge`;

`hanakoAge` = 3;
```

変数の宣言と初期化は、同時に行うことができます。 基本的には、初期化を忘れないためにも変数の宣言と初期化は同時にやりましょう。
見た目もすっきりします。見た目は大事です。

```
int `hanakoAge` = 3;
```

花子は、３歳なのでhanakoAgeの初期値として3を代入しました。 変数は更新が可能なので、hanakoAgeを4で更新してみましょう。

```
int `hanakoAge`;

`hanakoAge` = 3;

`hanakoAge` = 4;
```

これで初期化処理時は、hanakoAgeに3を入れていましたが、同じhanakoAgeに4が入り更新されました。
hanakoAgeはint型なので、int型の数値であれば、再代入することが可能となっています。

確認する場合は、以下のコードを実行してみてください。

```
public void JavaTraining {
    public static void main(String[] args) {
```

```
        // 変数の選言と初期化
        int `hanakoAge` = 3;
        System.out.println("初期化直後　花子の年齢 : " + `hanakoAge`);

        // 変数の更新
        `hanakoAge` = 4;
        System.out.println("再代入　花子の年齢 : " + `hanakoAge`);
    }
}
```

初期化しないと

まず、変数を作ります。

```
int `hanakoAge`;
```

今hanakoAgeは宣言しただけで、中身がありません。この状態でコンパイルするとエラーが出力されます。

嘘です。初期化について調べてみると変数を宣言時に値を指定しなかった場合は、初期値が設定されます。

型	初期値
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
boolean	false
参照型	null

実際に初期値が設定されているか確認してみましょう。

```
public void JavaTrainning {
    public static void main(String[] args) {
        int `hanakoAge`;
        System.out.println("変数宣言直後　花子の年齢 : " + `hanakoAge`);
    }
}
```

ってコンパイル通らんのかい！

調べました。 初期化処理を行わない初期値については、こちらを参照
https://nagise.hatenablog.jp/entry/20160422/1461335629

今説明するべき事の本筋から外れるうえに時間もかかるので、興味があれば読んでください。
ざっくり説明すると、変数の宣言時に初期化しない場合には初期値が設定されて、後々に値を代入して変数として利用できるけど、初期値のまま変数を利用しようとするとコンパイルエラーとなる。です。
変数を宣言したら、初期化をしましょう(´▽｀)

プリミティブ型

型付きで変数を宣言することで、型に合わせてメモリ上に領域が確保されます。確保された領域を利用して様々な値や文字あるいは参照アドレスを保持する事ができるようになります。

基本データ型 一覧表

種類	データ型の名前	説明	値の範囲
整数型	byte	8ビット整数	-128 〜 127
整数型	short	16ビット整数	-2,768 〜 32,767
整数型	int	32ビット整数	-2,147,483,648 〜 2,147,483,647（約±21億）
整数型	long	64ビット整数	-9,223,372,036,854,775,808 〜 9,223,372,036,854,775,807(約±900京）
浮動小数点型	float	32ビット単精度浮動小数点数	約(-)3.40282347E+38（約6〜7桁の精度）
浮動小数点型	double	64ビット倍精度浮動小数点数	(-).79769313486231570E+388（15桁の精度）
文字型	char	16ビットUnicode文字	\u0000 〜 \uFFFF
論理型	boolean	真偽値	「true」 or 「false」
文字列型	String	文字列	扱えるデータの長さはint型と同じ（約21億）

```
int i;
```

int型で宣言すると、32bit分の領域が確保されます。

```
int i = 5;
```

`i`に5を代入していますがコンピュータは２進数で認識するため、メモリ上で`i`には5が`101`で代入されます。
1bitで0か1を表現できるので、101は3bitの領域を使って5となります。

突然ですが！ドキドキ5秒クイズ！5を代入した変数`i`は、何ビットの領域を使っているでしょうか。

5

4

3

2

1

たった今5は2進数で101の3bitって言ったやん！って心の中で突っ込みをくれた人は、残念ながら不正解です！

```
int i = 5;
```

int型で変数を宣言した時点で32bit分の領域が確保されたあとは、ずっと32bit分の領域を使って変数を取り扱います。 なので変数`i`に`5`を代入すると前に0をたくさんつけて表示します。

`00000000000000000000000000000101`が`int i = 5;`の5を32bitで表す情報です。 intの変数は、変数が使われている間は32bitの領域を確保しているので、32bit分の整数を代入可能な状態となっているわけですね。 イメージわきましたでしょうか？

```
public class VariableTest {
    public static void main(String[] args) {
        int hanakoAge = 4;
        System.out.print("hanakoAge:");
        System.out.print(hanakoAge);
    }
}
```

演算子

演算子には、種類があります。

- 計算をするための演算子は算術演算子
- 左辺に右辺の値を代入する代入演算子
- 値を比較する比較演算子
- インクリメント・デクリメント
- 論理評価をする論理演算子

算術演算子

演算子	概要	例
+	加算	1 + 2 // 3
-	減算	5 ? 2 // 3
*	乗算	3 * 4 // 12

演算子	概要	例
/	除算	5.0 / 2 // 2.5
%	剰余	5 % 2 // 1
+	文字列の連結	“A” + “B” // AB
++	インクリメント	x = 1; x++; // 2
--	デクリメント	x = 1; x--; // 0

question2

演算子の優先順位と評価

プログラミングで演算子は、評価の考えがあります。

```
4 + 3 * 2
```

この例であれば、まずは3*2が評価され、6になります。
次に4+6が評価され、10になります。

小学校の授業でならうような当たり前のことやんって思うかもしれませんが、評価はとても大事です。 算術演算子に限らず、どこが評価されてどのような結果が帰るかを意識してください。

インクリメント、デクリメント

演算の中でも１つだけ増加させたり、１つだけ減少させる事がよくあります。 １つだけ増加させる演算としてインクリメント演算子、１つだけ減少さえる演算をデクリメント演算子が用意されています。 インクリメントは、以下のように書きます。

```
i++;
```

i++は、

```
i = i + 1;
i += 1;
i++;
```

これらと同じです。

デクリメントは、以下のように書きます。

```
i--;
```

i--は、

```
i = i - 1;
i -= 1;
i--;
```

これらと同じです。

インクリメント演算子とデクリメント演算子は、前置きと後置きの２種類があります。

```
int i = 5;

System.out.println("----------後置き----------");
System.out.println(i++); // 5・・・①
System.out.println(i); // 6・・・②

i = 5;
System.out.println("----------前置き----------");
System.out.println(++i); // 6・・・③
System.out.println(i); // 6・・・④
```

①では、変数iの値である、**5を出力してから1が加算**されています。
②では①でインクリメントされているので、6が出力されています。

③では、変数iの値に**1が加算されてから、iの値が出力**されています。 ④はiの値を出力。

デクリメント演算子でも同様の動きになります。前置きか後起きで結果が変わることを意識して、インクリメント演算子とデクリメント演算子を使ってください。

代入演算子

演算子	概要	例
=	右辺の値を左辺に代入	1 + 2 // 3
+=	右辺の値を加算した結果を代入	i += 5 // 15
-=	右辺の値を減算した結果を代入	i -= 5 // 5
*=	右辺の値を乗算した結果を代入	i *= 5 // 50
/=	右辺の値で除算した結果を代入	i /= 5 // 2
%=	右辺の値で除算した余りを代入	i %= 5 // 0
&=	右辺の値で論理積演算した結果を代入	i &= 5 // 0
=	右辺の値で論理和演算した結果を代入	i = 5 // 15
^=	右辺の値で排他的論理和演算した結果を代入	i ^= 5 // 15

演算子	概要	例
<<=	右辺の値だけ左シフトした結果を代入	i <<= 5 // 320
>>=	右辺の値だけ右シフトした結果を代入	i >>= 5 // 0
>>>=	右辺の値だけ右シフトした結果を代入	i >>>= 5 // 0

比較演算子

演算子	概要	例
==	左辺と右辺が等しければtrue	5 == 5 // true
!=	左辺と右辺が等しくなければtrue	5 != 5 // false
<	左辺が右辺より小さければtrue	5 < 7 // true
<=	左辺が右辺以下であればtrue	5 <= 3 // false
>	左辺が右辺より大きければtrue	7 > 5 // true
>=	左辺が右辺以上であればtrue	5 >= 7 // false
?:	「条件式? 式1 : 式2」。条件式がtrueなら式1、falseなら式2	i >= 1 ? “真” : “偽”

結果は、`boolean`だよ。

論理演算子

演算子	概要	例
&&	左辺右辺がともにtrueの場合はtrue	true && false // false
||	左辺右辺どちらかがtrueの場合はtrue	true || false // true
!	否定	!false // true
^	左辺右辺いずれかがtrueで、かつ、ともにtrueでない場合にtrue	^ false // true

if

もしも君がひとりなら、迷わず飛んでいくさ （俺の行く末密かに案じる人Honey!） もしも誰かといた時は解けるのかな魔法は 張り裂けそう胸の痛みは...

```
 Boolean isAlone = true;
if (isAlone) {
     System.out.println("迷わず飛んでいくさ");
     System.out.println("俺の行く末密かに暗示する人Honey!")
}
```

if文は処理の分岐を作る事ができます。if()の()の中に条件を書いて、条件の結果をbooleanで判定します。 つまりtrueかfalseかを判定することで、if(){}の{}内に書かれた処理を実行するかしないかを条件判定します。 例として書いているif文だと、if文の前にboolean型の変数isAloneとして宣言して、同時にtrueを代入しています。 if (isAlone) で、isAloneがもっているtrueかfalseを評価しています。
isAloneを評価した結果、trueを受け取った場合は、{}内に書いた処理である文字出力を実行します。
isAloneを評価した結果、falseを受け取った場合は、{}内の処理は行わず、次の処理へ移っていきます。

if文にはセットで使われる、elseという条件文があるのであわせて学びましょう。

```
 Boolean isAlone = true;
if (isAlone) {
    System.out.println("迷わず飛んでいくさ");
    System.out.println("俺の行く末密かに暗示する人Honey!")
} else {
    System.out.println("解けるのかな魔法は");
    System.out.println("張り裂けそうな胸の痛みは...");
}
```

if文のelseは、ifの条件式を評価した結果がfalseだった場合に、else{}へ処理が移りelse{}の{}ブロックの中に書かれた処理を実行します。

もう一つif文の書き方として学んでもらいたいのが、else ifです。 ifの条件も少し変えてみましょう。

```
int areYouAlone = 3 // 1:一人  2:恋人おるし  3:その他      ①
if (areYouAlone == 1) {                 // ②
    System.out.println("迷わず飛んでいくさ");
    System.out.println("俺の行く末密かに暗示する人Honey!")
} else if (areYouAlone == 2) {          // ③
    System.out.println("解けるのかな魔法は");
    System.out.println("張り裂けそうな胸の痛みは...");
} else {                                // ④
    System.out.println("花子がおそばにおりますゆえ！");
}
```

①の部分は、int areYouAlone変数を作成して、何やら値を代入しています。
今回はこの値に意味を持っている事とします。 1なら一人、2なら恋人おるし、3その他と、値によって意味を決めて、よりプログラムらしい処理感がでています。

値	状態
1	一人
2	恋人おるし
3	その他

if文についてですが、`if()`の`()`の中で`int`の値を比較演算子で評価できます。 比較演算子は`boolean`を返しますよね！比較演算子の結果を`true`や`false`で受けて条件分岐しています。

②の条件判定`if`は、`areYouAlone`の値が`1`と等しいかを評価して、`true`であれば`{}`の中を実行します。処理を実行後は、if文を抜けます。 ③の条件判定`else if`は、②が`fale`だった場合に条件判定を行って、`areYouAlone`の値が`2`と等しいかを評価して、`true`であれば`{}`の中を実行します。処理を実行後は、if文を抜けます。 ④の条件判定`else`は、全てのif条件判定を評価したけど全て`false`だった時に、`else{}`の中を実行します。処理を実行後は、if文を抜けます。

配列

テストでAさん、Bさん、Cさん、Dさん、Eさんの５人の点数を管理するプログラムを作りたいとします。

まずは、AさんからEさんの点数を入れるための変数を用意します。

```
int scoreA;
int scoreB;
int scoreC;
int scoreD;
int scoreE;
```

同じような変数型を何度も書くのは面倒くさいです。下記のようにまとめて書くことも可能です。

```
int scoreA, scoreB, scoreC, scoreD, scoreE;
```

でもやっぱり、同じような目的の変数をいくつも用意するのは面倒です。 そんな時は配列を使うと便利です。

配列の宣言と初期化

`scoreA`から`scoreE`までのintの宣言文をやめて、配列を使うために配列の宣言を書きます。 変数名は、`score`が複数あるので`scores`とします。

```
int[] scores;
```

型の宣言の後ろに`[]`を書くことで、型の配列ができます。今回は`int[]`でint型の配列を作成しました。

次にint型の配列変数を初期化します。

```
int[] scores;
scores = new int[5];
```

`new`についてですが、クラスやらオブジェクトやらインスタンスやら大事な要素なのですが、今はとりあえずの認識として、プログラムで便利な機能を使う事ができる道具を作っていると思ってください。 `new`

`int[5]`で、５つの要素を持ったint型の配列の機能を持った道具を作って、道具を`scores`変数に代入しています。 初期化を行った時点で、各要素には全て`0`の値が入っています。

他の変数と同様に変数型の宣言と初期化を同時にできます。

```
int[] scores = new int[5];
```

配列の各要素に値を代入

配列の各要素に今回の点数を代入してみましょう。

```
int[] scores = new int[5];

scores[0] = 90;
scores[1] = 45;
scores[2] = 72;
scores[3] = 85;
scores[4] = 64;
```

`scores[]`の`[]`に、何番目の配列かをして、`=`で値を代入しています。 注意事項として、配列の一番最初の要素番号は、`0`から始まる事に注意してください。 今回は配列を５つ分用意していますが、配列の最後の要素番号を指定したい場合は、`scores[5]`にするとコンパイルエラーとなります。 配列の要素番号は0から始まっているので、最後の要素番号をする場合は、`scores[4]`となります。

これで配列を使わなかった場合は、int型の変数を５つ用意していたのが、int型の配列変数`scores`一つに全ての点数を保持できるようになりました。 5つぐらいなら頑張って変数を作ってもいいかなってなるかもしれませんが、配列であれば100でも1000でも一つの配列変数に保持する事が可能です。

配列を出力する場合は、

```
int[] scores = new int[5];

scores[0] = 90;
scores[1] = 45;
scores[2] = 72;
scores[3] = 85;
scores[4] = 64;

System.out.println(scores[3]);
```

これで、配列変数`scores`の要素番号が`3`の値を出力できます。

配列は繰り返し文と組み合せての使用がよくあります。 次の繰り返し文もバッチリ修得しましょう。

繰り返し for

for文は、繰り返しの処理です。 for文無しで書くと、

```
int result = 1;

result = result * 2;   //1回
result = result * 2;   //2回
result = result * 2;   //3回
result = result * 2;   //4回
result = result * 2;   //5回

System.out.print("2の5乗 : ");
System.out.println(result);
```

面倒くさい！（雑ですみません。）
for文で書くと同様の処理を何百何千何万回でも処理してくれます。

```
int result = 1;

for(int i = 0; i < 5; i++) {
    result = result * 2;
}
System.out.println("2の5乗 : " + result);
}
```

これぞコンピューターの真骨頂！人間だと同じ事を何回もしていたら飽きてしまうし、間違えます。プルグラムによる処理なら飽きもせずに同じ処理を何万回でもやってくれるし、間違えないし、高速です。 コンピューターは現代における合法的な奴隷だという話を聞いたこともありますが、この奴隷は我々のような一般人が使役可能なのです！ 奴隷っていうのは基本的に面倒でやりたくないことを安い報酬で、人権を踏みにじって使役する糞な制度です。 が、コンピューターは文句を言わず、人間がやれば糞つまらん奴隷的な計算処理をいくらでもやってくれます。素晴らしい。

閑話休題。

for文は、for(初期設定; 条件; 継続処理){処理}と書きます。
例でみると初期設定では、for文の中だけで使うint型の変数iを変数宣言して、同時に初期化で`0`を代入しています。
条件は比較演算子で`i < 5`なので、`i`が`5`未満の間は`true`を返して、`{}`ブロック内の処理を実行します。`i`が`5`以上になると`false`を返して、`for`文を抜けて次の処理にいきます。
継続処理についてですが、ブロックの処理が一番最後まで実行されたら、継続処理が実行されます。`i++`と書いているので、ブロックの処理が終わるごとで`i`に`1`が加算されます。
今回は、`i`が`0`から始まって、`i`が`5`未満の間は処理実行と`i`への加算を繰り返して、`i`が`5`になったらfor文の処理を終了します。 処理の中は、何をやっているか考えてみてください。 `result`の初期値は何で、どんな処理が何回実行されて、最後は何が出力されますか？

少しだけ改造して下記のように作ると、より汎用的です。

```
int result = 1;
int n = 5;

for(int i = 0; i < n; i++) {
    result = result * 2;
}
System.out.println("2の" + n + "乗 : " + result);
}
```

改造前と比べると、変数nの初期化の値を変えるだけでn乗を自由に変更できます。

配列とfor文

配列とfor文の組み合わせはよく使うのでバッチリ修得しましょう。

```
int[] scores = new int[5];

scores[0] = 90;
scores[1] = 45;
scores[2] = 72;
scores[3] = 85;
scores[4] = 64;

for(int i = 0; i < scores.length; i++) {
    System.out.println(scores[i]);
}
```

配列の説明の時は、配列を出力するときに

```
System.out.println(scores[3]);
```

と書いて、scoresの3番目の要素番号を指定していました。 今回は、for文でiを使って配列の要素番号を指定しています。 scores.lengthは、長さを返してくれるメソッドです。今は詳細な説明をせずに使っていきますが、今回のように配列.lengthと使うことで、配列の要素数を教えてくれます。 scores.lengthで教えてくれる値は、5になります。 if文の中の処理実行条件は、iが5未満であれば実行するとなります。

実行されていく順番を追いかけてみましょう。

```
for(int i = 0; i < scores.length; i++) {
    System.out.println(scores[i]);
}
```

iが0で初期化されているので、最初の処理は、

```
System.out.println(scores[0]);
```

が実行されます。今回は出力処理(println)だけなので、出力処理が終わるとブロックの最後にきて、

```
for(int i = 0; i < scores.length; i++)
```

()内で三つ目のi++が実行されて、iの値が0から1になります。 iが1なので、1 < 5(scores.length)の条件を満たすため、2回目の処理を実行します。

```
System.out.println(scores[1]);
```

この要領でiの値をインクリメントで加算していき、scores配列の各要素を順番に出力します。 iが4の時、最後の配列要素であるscores[4]の出力が実行され、i++でiが5になると、5 < 5 は条件を満たさずfalseとなり、for文の処理を終了します。 これで全ての配列要素に対して処理を実行して、次の処理へ移る処理ができるようになりました。

参照型の変数

変数を作るとき、基本型とは別に参照型の変数があります。基本型との違いは何でしょう？ 基本型で変数を作った場合は、値そのもねが変数に保持されていますが、参照型は値そのものをもちません。参照型が保持するのは、メモリ上の値がある箇所を示すアドレスコードである参照値です。

参照型の例として、配列変数で学んでいきましょう。int型の変数は基本型ですが、int型でも配列の変数は参照型となります。他のbooleanやらdoubleなどの基本型でも、配列変数であれば参照型になります。

```
int[] hanakoStep; //①
hanakoStep = new int[] {1, 2, 3}; //②
```

①int型の配列変数aが宣言されています。 その後に②で３つぶんの領域を確保した配列のインスタンスを生成し、値を配列の各要素に設定しています。

hanakoStepに代入されている値は、{1, 2, 3}そのものではありません。{1, 2, 3}はメモリのどこかに作られます。作られたどこかには必ずアドレスコードが割り当てられ、例えば{1, 2, 3}が87878787というアドレスコードを持つ領域に作成されます。hanakoStepに代入されるのは87878787というアドレスコードの*値*が代入されます。 この代入されたアドレスコードの値を**参照値**と呼びます。

メモリに主眼をおいて流れを説明すると、

1. int[] hanakoStep;でint型の配列変数を格納している、参照値を収めるためのメモリ領域が確保されます。
2. new int[] {1, 2, 3};メモリのどこかにint型の配列で要素が3つぶんの領域が確保されて、それぞれ1、２、３という値を保持します。

3. `hanakoStep = new int[] {1, 2, 3};`は、`hanakoStep`にメモリのどこかを示す、**参照値を代入**します。