11 変数とメモリ管理

11.1　　プリミティブ型と参照型

11.1.1 プリミティブ型と参照型の違い

Java の変数は大きく分けて二つに分けることができます。**プリミティブ型**と**参照型**です。分け方はすこぶる簡単です。

プリミティブ型は int や double といったクラス以外の値を格納する変数のことであり，参照型とはクラスのインスタンスを格納する変数です。また、クラスインスタンス以外にも、配列も参照型ですし文字列も参照型となります。あ、文字列は String クラスのインスタンスですので、参照型なのもあたりまえですが。

以上まとめると，以下のようになります.

プリミティブ型	short, int, long, float, double, boolean, char, byte
参照型	クラス・インターフェースの変数，配列

プリミティブ型は数が少ないので，全部覚えてしまった方が良いかもしれませんね.

このプリミティブ型と参照型では，データの扱い方に異なる部分があります．特に，ここではその違いについて説明して行きたいと思います.

11.1.2 参照渡しと値渡し

Java に限らず，プログラムを書いていると値の代入という操作が非常に頻繁に出てきます.

```
int a = 10;
```

というのも値の代入ですが，メソッドに渡す引数なども値の代入として扱われます.

つまり，

```
public static void main(String[] args){
          int from = 5;
          method(from);
}

static void method(int to){
          System.out.println(to);
}
```

そすると，method の引数 to には from が代入されることになります.

```
to = from;
```

と書かれるようなものです.

このような代入があったとき，プリミティブ型と参照型では大きく動作が異なります.

プリミティブ型では，代入によって**値がコピーされる**のですが，参照型の場合は**参照が渡される**だけなのです.

では，それぞれの代入についてみていきましょう．

11.1.2.1 プリミティブ型の場合

プリミティブ型の場合，代入は全て値のコピーになります．つまり，ある変数 A が持っていた値をコピーして別の変数 B に代入するわけです．

したがって，B を変化させても A は変化しません．

```
int basePrice = 100;
int taxedPrice = basePrice;

taxedPrice *= 1.05;

System.out.println(basePrice);
System.out.println(taxedPrice);
```

たとえば，こんな場合です．basePrice という基本的な値段に対して，消費税を掛けた税込価格 taxedPrice を作成しています．この結果は，

```
100
105
```

となり，basePrice を代入した taxedPrice を変更しても basePrice には影響を与えません．

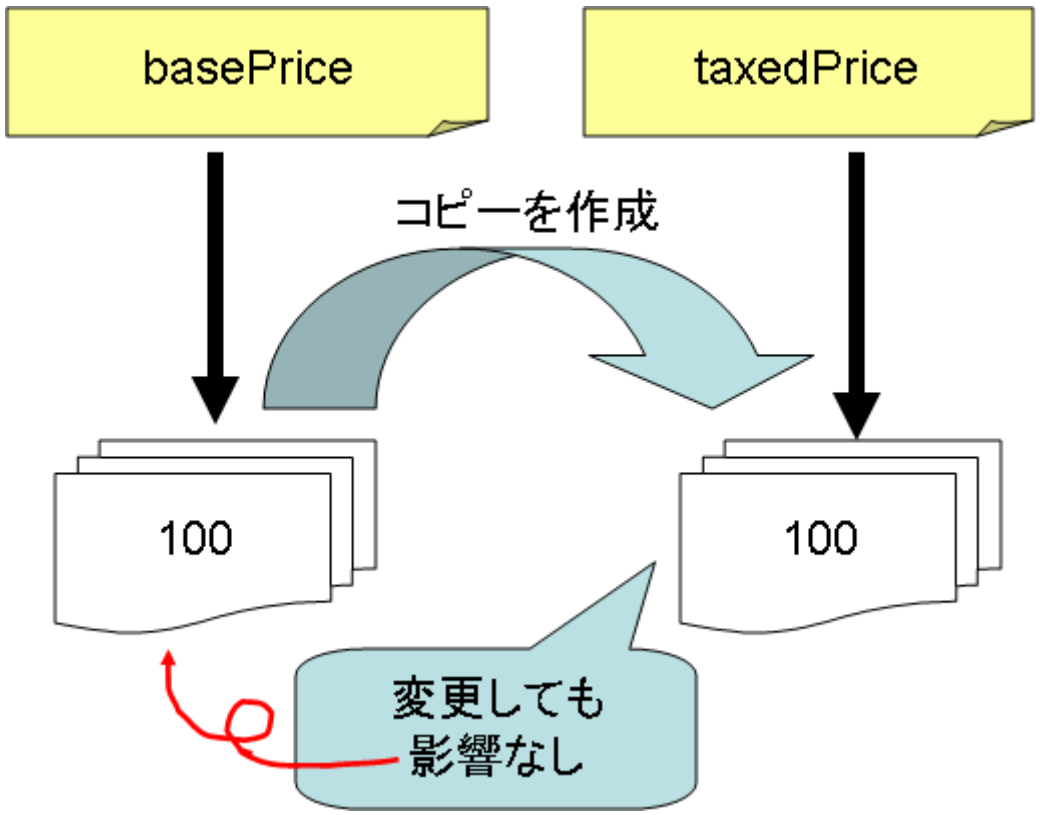


図 1 プリミティブ型の代入

11.1.2.2 参照型の場合

プリミティブ型では，値をコピーしていたため代入先でデータを変更してももとのデータには影響を与えませんでした．一方参照型の場合は，代入は参照先の代入になるため，代入先でデータを変更すると元のデータも変更されてしまいます.

ここで，House クラスというものを考えてみましょう。これは、家を示すクラスです。きっと、ドアを開けたりチャイムを鳴らしたりするメソッドがあるんでしょうけど、とりあえず、ここではそれは気にしないで表札だけつけておきます。

```
public class House{
	String namePlate;

	public String getNamePlate(){
		return namePlate;
	}

	public void setNamePlate(String name){
		namePlate = name;
	}
}
```

さて、この家クラスを使って、家を二件作る計画を立ててみましょう。面倒なので設計は全部同じ建売住宅にしちゃいましょう。ということで、一つ家を作ったら、その家をコピーして表札だけ変えて、別の家として保存しておきます。

```
House myHouse = new House();
myHouse.setNamePlate("俺の家"); //表札をつけておくよ
House hisHouse = myHouse; //とりあえずコピー
hisHouse.setNamePlate("彼の家"); //彼の家にも表札をつけてあげよう
System.out.println(myHouse.getNamePlate()); //自分の家の表札を確認
System.out.println(hisHouse.getNamePlate()); //彼の家の表札を確認
```

ここでは、まず自分の家を作って表札をつけてから、今度は友人の家を作って表札をかけてあげています。

さあ、二件の家ができて安心だ・・・と思いきや！

```
彼の家
彼の家
```

なぜか自分の家のはずの myHouse の表札まで「彼の家」になってしまいました。なんてことでしょう。自分の家が彼に取られてしまいました！これはショック。

なぜこんなことになったのでしょうか？この謎を解くには，Java がどのようにデータを管理しているかを知る必要があります.

実は、参照型の場合代入先の変数は同じオブジェクトを指すという決まりがあります。

つまり、myHouse のコピーを hisHouse にしていたつもりが、hisHouse が意味するものも myHouse そのものになっていたのです。

参照型を代入する場合は同じものに別の名前が付けられると覚えて置いてください。

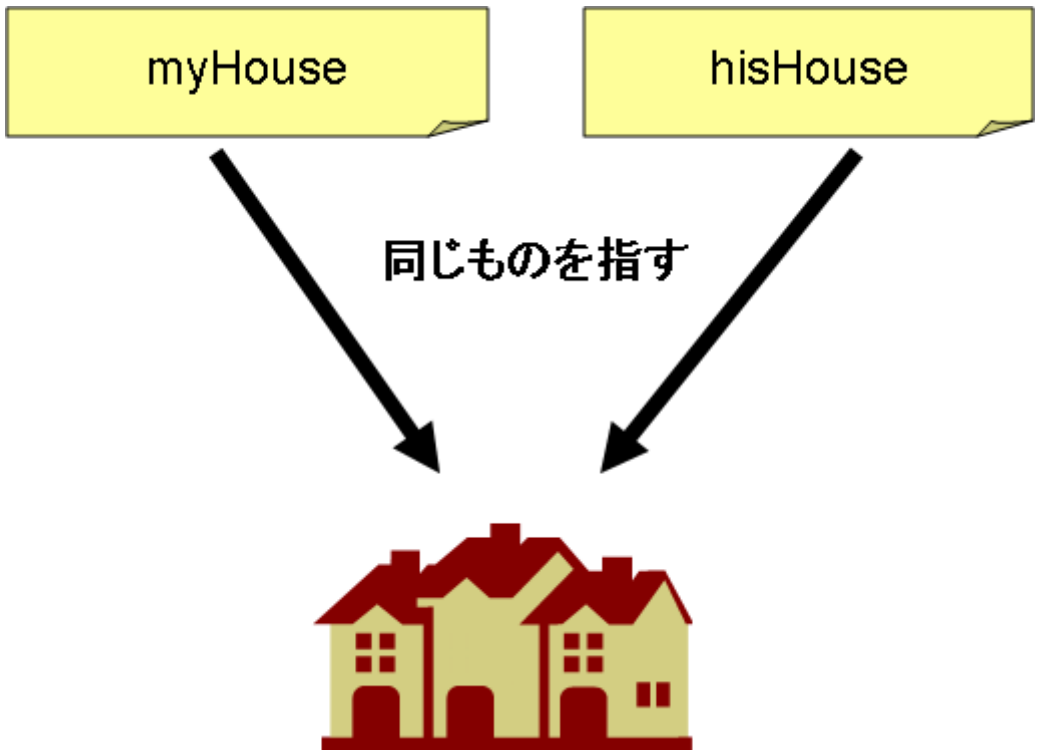


図 2 同じ物をさしています

つまり、もし二人分の家を作りたいのであれば、まったく新しくオブジェクトを作らなければいけません。

```
House myHouse = new House();
House hisHouse = new House();
```

これで、二つの家が作られて、myHouse と hisHouse が別のものをさすことになります。

11.1.3 参照型の詳しい話

参照型の変数というのは、変数と変数が指す実態そのものが異なっているということに注意が必要です。

初心者のうちはどうしても変数=実態と思い込んでしまいがちですが、実際には実態は別に存在し、変数名はただその実態に名前を付けて便宜上呼んでいるだけのものなのです。

これについて理解しておかないと、Java でプログラミングをしているときにどうしても行き詰まってしまう可能性がありますので、さらに詳しく説明したいと思います。

まず、Java で変数を使うときは、実態(オブジェクト)と変数と二つのものがあると考えてください。

このとき、オブジェクトというのは house クラスならば家そのものだし、String クラスなら文字列そのものだとします。**実際のデータそのもの**、それがオブジェクトです。ただし，このオブジェクトはプログラム上から直接扱うことは出来ません.

それに対して変数は**オブジェクトを指し示すもの**になります。つまり、変数というのは

名目上のお飾りであって、オブジェクトは別のところにあるのです．ただし，プログラム上から変数を利用することは出来るので，我々は変数を介してオブジェクトに触れることができるのです。

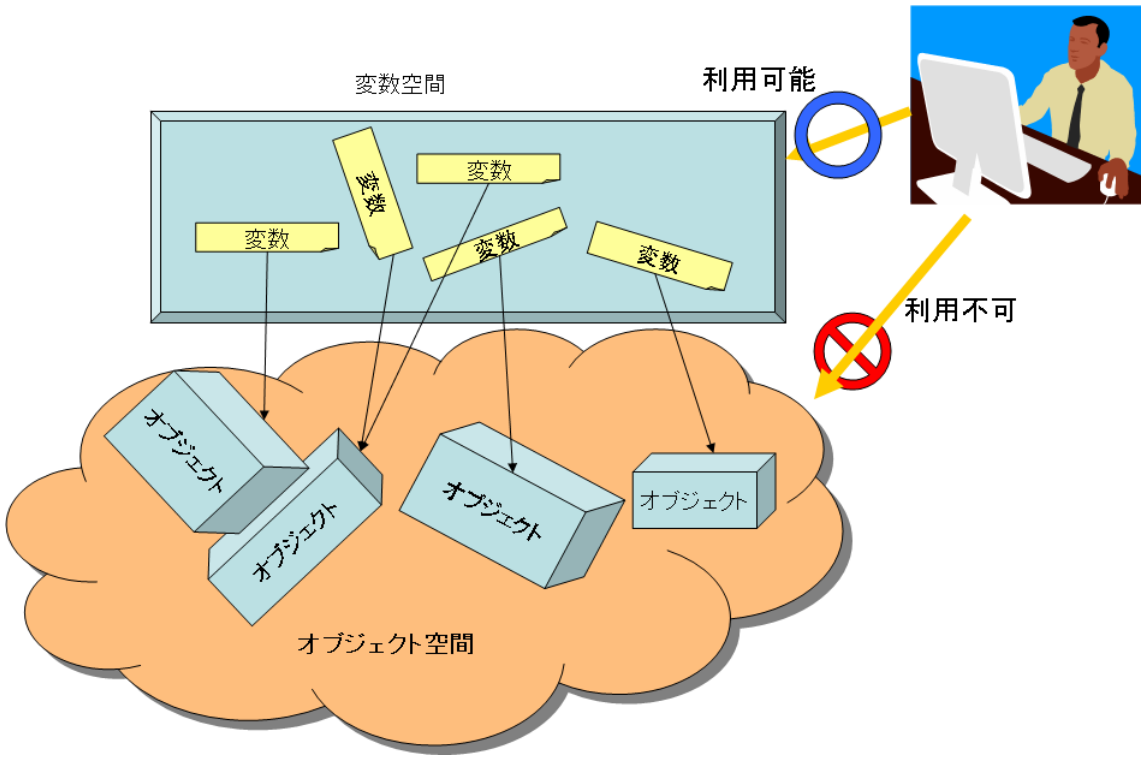


図 3 利用できるのは変数だけ

ここで、参照型変数を作成したときのことを考えてみましょう。今回は分かりやすくするために、変数作成部分とオブジェクト作成部分を分けておきます。

```
House myHouse;
myHouse = new House();
```

こんな書き方をしましたよね。これ、最初の段階では myHouse という変数を確保したものの、そこに値は入っていません。後ほど説明しますが、このとき myHouse 変数には何も入っていないことを意味する「null」が保存されています。つまり、変数だけ作られて実態はどこにもないわけです。

その次に、myHouse 変数に new で作った House オブジェクトを代入しています。つまり、このとき初めて House オブジェクトが作成されるのです。これまでは House はどこにも存在しないのです。

そして、myHouse 変数に代入することによって、House オブジェクトを利用する手段が我々にもたらされるのです。

ちなみに、もしここで、こんな風にしてみたらどうなるでしょうか？

```
House myHouse;
new House();
```

先ほどと違って、new によって House オブジェクトを作成した後、myHouse 変数に代

入していません。こうするとどうなるのか。実は House オブジェクトはちゃんと作成されます。皆さんのパソコンのどこかに House オブジェクトはちゃんと作られています。ただし、それがどこに作られているのか分からないため、我々は二度と利用することができなくなってしまいます。オブジェクト空間には誰も触れることが出来ないのですから，変数として記録されていない限り二度と探し出すことは出来ないのです.

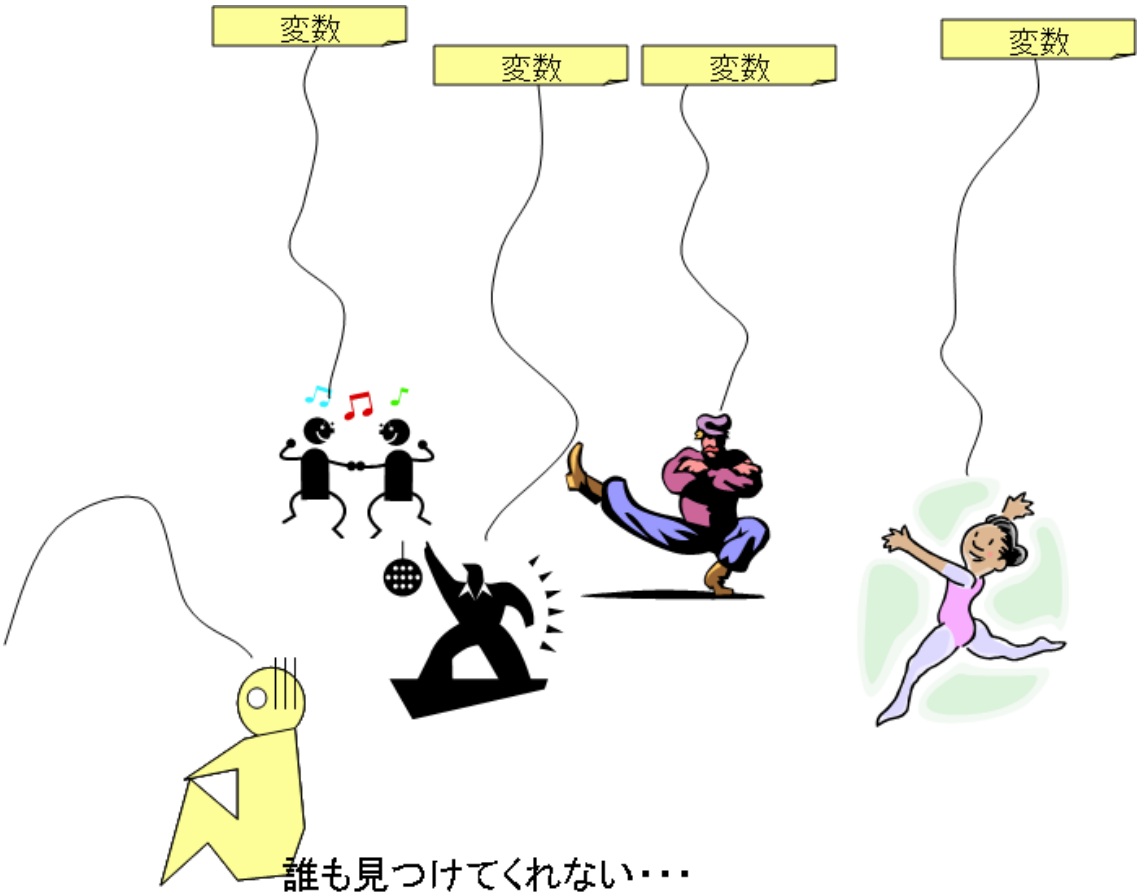


図 4 変数につながっていなければ二度と探せない

オブジェクトというのが House そのものですが、そのものはほうっておくとすぐに行方不明になってしまうため、myHouse 変数に代入することで場所を忘れずに済むのです。変数というのは、すぐにどこかに消えてしまうオブジェクトを忘れないように場所を記録しているものだと思ったほうが良いかもしれません。

そう考えれば、myHouse を hisHouse に代入したときにオブジェクトはコピーされずに、同じオブジェクトを指したものができたかも納得いきますよね。myHouse 変数はあくまでも House のオブジェクトを保存してある場所を記録しているだけですので、その場所の記録をコピーすれば、同じ物を指すようになるわけです。

参照型のイメージを持つことは，Java でプログラミングを行ううえで結構重要な話になってきますので，是非明確なイメージを持つようにしてください.

コラム

ここでお見せした例のように，どの変数からも指されていないオブジェクトはどうなってしまうのでしょうか？

実は，このようなオブジェクトは利用できない割にはコンピュータのメモリ中には存在してしまい，将来的にメモリ不足を引き起こす可能性があります．これでは困ってしまいますよね．

そのため，Java では，このような誰からも利用できないようなオブジェクトを探し出して自動的に削除するという機能を持っています．それがガベージコレクタと呼ばれる機能です．簡単に言えば使われる可能性のなくなったオブジェクトを勝手に探し出して削除してくれるわけですね．これは結構便利な機能です．

Java の前身ともいえる C++ではガベージコレクタがなく，誰からも指されなくなったオブジェクトもプログラム中で削除しない限り永久にメモリに残ってしまっていました．そのため，メモリリークと呼ばれる，無駄なオブジェクトによるメモリ領域の圧迫がたびたび生じていたのです. C++プログラマにとってメモリリークは悪夢のようについて回る嫌な問題でした. しかしながら, Java ではそのメモリ関連を, プログラマには気にさせずに Java 自身が管理してくれることになり，非常に使いやすくなりました．

皆さんは Java の勉強をしているので特に楽になったなあ，という感覚はないかもしれませんが，実は Java 君が裏ですごい頑張っていることを忘れないでおいてあげてください．

11.1.4 null

参照型、つまりクラスの変数の場合、実態であるオブジェクトはオブジェクト空間というどこか良く分からないところにあって、我々の手元にある変数はその場所を示したものだけということを書きました。では、もし手元にある変数が場所を指し示していなかったらどうなるでしょうか？

こんなことをやっていましょう。

```
House myHouse =new House();

String namePlate = myHouse.getNamePlate(); //まだネームプレートを作成していないのに・・・
System.out.println(namePlate);
```

まだオブジェクトを作成していないのに、myHouse 変数を使おうとしています。この場合どうなってしまうでしょうか？

```
null
```

というわけで、null という文字列が表示されました。

ここでポイントは「null」と表示されたからといって null という文字列が入っているわ

けではないという点です.

その証拠に，この文字列の長さを調べてみましょう．null という文字列が入っていれば４文字のはずですが・・・

```
System.out.println("ネームプレートの長さは"+namePlate.length());
```

さあ，試してみましょう.

```
Exception in thread "main" java.lang.NullPointerException
        at NullPointerMain.main(NullPointerMain.java:13)
```

NullPointerException というエラーが発生しました。というわけで，null という文字列が入っているわけではなさそうです.

実は，null というのは、この変数はオブジェクトを指していないよ！ということを意味しています。

ここで House クラスを思い出してみましょう.

```
public class House{
          String namePlate;

          public String getNamePlate(){
                    return namePlate;
          }

          public void setNamePlate(String name){
                    namePlate = name;
          }
}
```

変数 namePlate は，setNamePlate が呼び出されれば初期化されますが，それ以外の場合は初期化されていません．この初期化されていない状態というのは，変数だけはあるものの，どこも指していない状態となります．つまり、本来ならどこかの倉庫にしまってある String インスタンス，すなわち文字列を指したいのですが、どこを指せばいいのか決まっていない状態なのです。

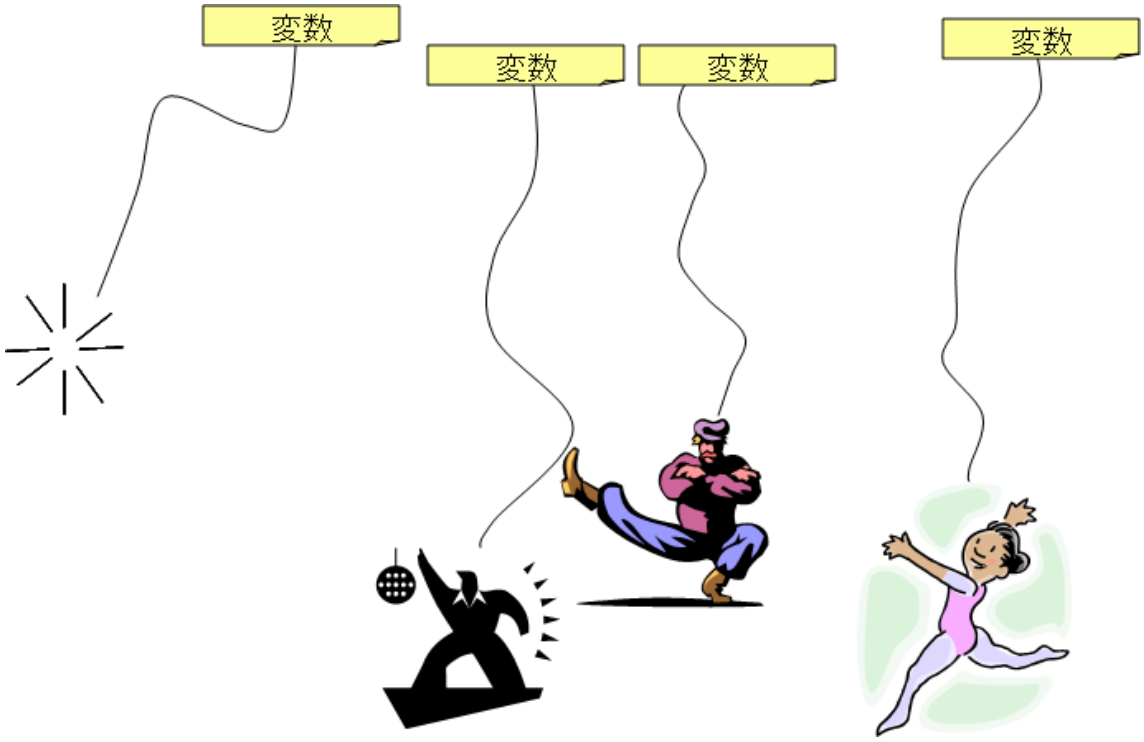


図 5 変数が何も指していない状態

というわけで、どこも指していない文字列変数の長さを調べろと言われたら，そんな無茶な命令を受けた Java 君は怒って「何もないところには何もないんじゃ！」と逆切れしたわけです。

この NullPointerException は Java を使っているの頻出するエラーですので、注意してください。

ある参照型のポインタが null かどうかをチェックするには，以下のようにすれば簡単に調べられます.

```
if(namePlate == null){
        System.out.println("ネームプレートはまだ指定していません");
}else{
        System.out.println("ネームプレートの長さは"+namePlate.length());
}
```

null というのは Java の予約語で，ある参照型変数と==や！＝で比較することが可能です.

11.1.5 参照型の比較 equals

さて、参照型の変数はオブジェクトを指しているだけのものです．そのため，二つの異なる変数が同じオブジェクトを指していることが良くあります.

```
House myHouse = new House();
myHouse.setNamePlate("俺の家");
House hisHouse = myHouse;
```

たとえば，こうすることによって myHouse と hisHouse が同じオブジェクトを指すことになります.

このとき，ある二つの参照型の変数が同じオブジェクトを指しているかどうかを調べるためには，＝＝記号を使います.

```
if(myHouse == hisHouse){
          System.out.println(“同居中”);
}
else{
          System.out.println(“別居中”);
}
```

というわけで、プリミティブ型を比較するように==で比較すると、同じオブジェクトかどうかを判断します。

ここで注意したいのが「同じオブジェクトかどうか」を比較しているのであって、「同じ値であるかどうか」をチェックしているわけではないという点です。

たとえば、

```
House myHouse = new House();
House hisHouse = new House();
myHouse.setNamePlate(“俺の家”);
hisHouse.setNamePlate(“俺の家”);
if(myHouse == hisHouse){
          System.out.println(“同じ家！”);
}
```

ということはできないのです。なぜか？そりゃ簡単で、同じオブジェクトじゃないからです。ということは、表札を同じにしても同じものだとは認めてくれないわけですね。

でも、これではちょっと困ってしまうことがあります。その例を示すために、こんなクラスを作ってみたいと思います。

```
class Fruit{
          String name;
          Fruit(String type){
                    name = type;
          }
}
```

この Fruit クラスは、ある果物を示すものだと思ってください。こんな風に使います。

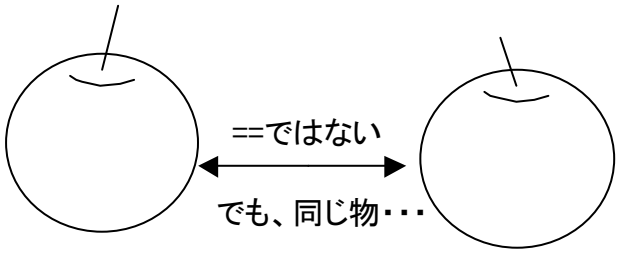
```
Fruit apple = new Fruit(“リンゴ”);
Fruit orange = new Fruit(“ミカン”);
```

このとき、もう一つリンゴを持ってきたとしましょう。

```
Fruit apple2 = new Fruit(“リンゴ”);
```

さて、このとき apple と apple2 は同じ物でしょうか？

厳密に言えば、二つのリンゴは違うものですよね。でも、apple と apple2 を持ってきて「これの二つの果物は同じ物だよね？」と聞かれたら 100 人中 200 人くらいが「そうだよ」と答えてしまうでしょう。



そんな風に、あるオブジェクトとあるオブジェクトが「同じ物体」を指しているわけではないけれど、「意味的には同じ」ものであることを判断したい場合があります。

そんな比較をしたい場合、参照型の場合 equals メソッドを使うという決まりがあります。

```
class Fruit{
          String name;
          Fruit(String type){
                    name = type;
          }
          public boolean equals(Fruit fruit){
                    return fruit.name.equals(name);
          }
}
```

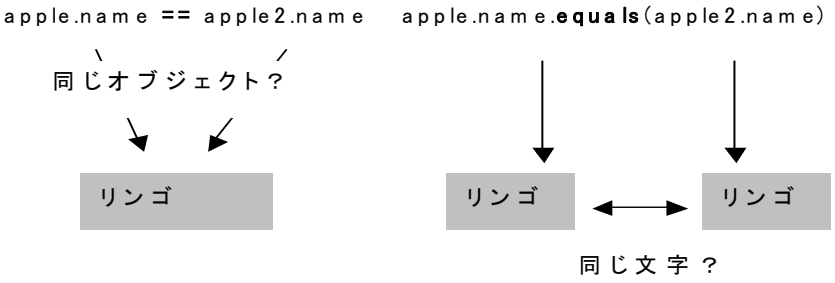
Fruit クラスにこのようなメソッドを追加することで、クラス内のフィールド name が等しければ、true を返すメソッドの完成です。

使い方はこんな感じ。

```
if(apple.equals(apple2)){
          System.out.println(“同じ種類の果物だ”);
}
```

簡単ですね。

というより、equals の中で String クラスの変数である name の equals を使っています。そういえば、String 型も参照型でしたよね。だから、==で結ぶと、まったく同じオブジェクトを指しているか同化の判断になってしまって、「同じ文字列かどうか」を比較しなくなってしまうんです。したがって、equals を使わないといけないのです。これは、以前 String クラスで一回説明しましたが、参照型とプリミティブ型の違いが分かってからだと、より理解が深まったのではないでしょうか？



特に、文字列でハマりやすい罠ですので、お気をつけください。

ちなみに, equals メソッドは Object クラスのメソッドです. 通常の Object の equals は，単に同じオブジェクトかどうかを比較しているだけですので，異なるオブジェクトが equals で結ばれるべき可能性がある場合は，equals をオーバーライドして条件を記述するようにしましょう.

11.1.6 参照型のコピー

さて,=では参照型をコピーできませんでした. 本当にコピーをしたい場合は **clone** というメソッドを利用します．clone メソッドは，equals と同じく Object のメソッドです．その Object のメソッドをオーバーライドして独自のメソッドを作成します.

ただし，clone メソッドは Object クラスでは protected として宣言されています．そのため，他のクラスで利用する場合は public としてオーバーライドしてあげなければいけません．また，clone メソッドを使う場合は，そのクラスは **Cloneable インターフェースを実装していなければいけない**，というよく分からないルールがあります.

さらに，Object クラスの clone を使う場合はちょっとしたおまじないが必要です．なぜこんなおまじないが必要化は次章の Exception で学ぶのでここでは割愛しますが，clone メソッドを使えるようなクラスを作るには，以下のようにメソッドを実装しなければいけません.

```
public class House implements Cloneable{

	String namePlate;

	public String getNamePlate(){
		return namePlate;
	}

	public void setNamePlate(String name){
		namePlate = name;
	}
```

```
	public Object clone(){
		try {
			return super.clone();
		} catch (CloneNotSupportedException e) {
			return this;
		}
	}
}
```

かなり面倒ですね.

しかし，これはもうおまじないだと思って覚えてください.

また，clone メソッドを使って実際にコピーを行う場合は，このようにします.

```
House myHouse = new House();
myHouse.setNamePlate("俺の家"); //表札をつけておくよ
House herHouse = (House)myHouse.clone();
herHouse.setNamePlate("彼女の家");

System.out.println(myHouse.getNamePlate()); //自分の家の表札を確認
System.out.println(herHouse.getNamePlate()); //彼の家の表札を確認
```

ここでも注意しなければいけない点があります．それは，clone メソッドが返すのは Object 型であるという点です．つまり，どんなクラスが返ってくるのか分からないわけですね. まあ, 通常は clone を使ったコピー元インスタンスと同じ型が返ってきます. ですが, Java プログラム的には Object 型しか返ってくると判断しないので, 明示的に型変換をしてあげなければいけません.

ここまで作業をして，ようやくコピーが完成します.

```
俺の家
彼女の家
```

実行結果を見ると，ちゃんとコピーがされていることが分かりました.

ふ～，疲れた.

作るのも使うのも面倒くさい clone メソッドですが, インスタンスをコピーする場合には便利なので活用してください.

11.2　変数の有効範囲と初期化ルール

11.2.1 変数の種類

Java の変数は大きく 4 つに分けることができます。え？さっき 2 つに分けることができるって書いたばかりだって？まあ、そのとおりなんですけどね。見方を変えればまた別の分け方があるんですよ。言うなれば、ラーメンが「味噌」「塩」「醤油」に分けられるけど、

「インスタント」と「本格派」に分けられるようなものです。

というわけで、ここで紹介する変数の種類は、

- ローカル変数
- フィールド
- 定数

の 3 つです。

それぞれの変数で、その**有効範囲と初期化ルールが異なります**。

ローカル変数は一つのメソッド（または、ブロック）の中でのみ有効であり、フィールドは一つのインスタンスの中で有効です。また、クラス変数は同じクラスであればインスタンスが異なっても利用することができます。また、定数は通常プログラム中のどこからでも使うことができます。また，それぞれ初期化を行う際にルールがあります.

有効範囲については、慣れるまでは意外と間違えがちですのでご注意ください。ここでは，この有効範囲について詳しく説明して行きたいと思います.

11.2.2 有効範囲の基本ルール

本節では 4 種類も変数を紹介するということで，覚えることが多くて大変そうな気がしますが，実は基本ルールはあまり難しくありません.

変数の有効範囲を決めるルールは以下の 4 点にまとめられます.

- 変数は変数宣言したブロック及び下位ブロックでのみ有効
- ローカル変数は宣言してから有効になる
- ローカル変数以外は宣言の場所にかかわらず利用可能
- 引数及び for 文には特殊ルールがある

以上です．割と簡単ですね.

さて，ここで**ブロック**という言葉が出てきました．Java においてブロックとは，{}内に囲まれた範囲を指します.

例えば,

```
public class BlockMain {

        public static void main(String[] args) {
                int price = 0;
                while(price < 1000){
                        price += 100;
                        if(price > 500){
                                System.out.println("1000 円まで残り"+(1000-price));
                        }
                        else{
                                System.out.println("まだ余裕");
                        }
                }
                System.out.println("1000 円突破！");
        }
```

```
}
```

このプログラムの場合，全部で 5 つのブロックがあります.

- BlockMain クラスをあらわすブロック
- main メソッドをあらわすブロック
- while 文をあらわすブロック
- if 文をあらわすブロック
- else 文をあらわすブロック

です.

このように，クラスを記述するための範囲を示していた{}もメソッドの記述のための{}もすべてブロックと呼びます.

また, このとき **main メソッドのブロックはクラスブロックの上位ブロックである**ということができ，**while ブロックの下位ブロックは if ブロックと else のブロックである**ということができます．また，**else ブロックは if ブロックの外にある**という言い方をします.

この，上位，下位，外の関係が変数の有効範囲を考える上で重要になるのです．つまり，ルールの下位ブロックで有効というのは，while ブロック内で宣言した変数は，if ブロックと else ブロックで持つかえるという意味になるわけですね.

ちょっと，フィールドについて思い出してみましょう.

フィールドは，メソッドの外に記述しましたので，クラスブロックの中に書かれているということが出来ます．メソッドはクラスブロックの下位ブロックとなりますので，クラスブロックに書いてあるフィールドはすべてメソッド内で利用できるというわけです.

この辺を考慮に入れて，ローカル変数から順番に変数の有効範囲についてみていきましょう.

11.2.3 ローカル変数

11.2.3.1 ローカル変数の有効範囲

ローカル変数とは、メソッド内で宣言される変数です.

ローカル変数の有効範囲は,

- 変数宣言してから
- 変数宣言したブロックが終了するまで

が有効範囲となります.

例を見てみましょうか。

```
public class Variable {
          static public void main(String[] args){
                    String onlyYou = “I love you”;
                    localTest();
          }

          static public void localTest(){
                    System.out.println(onlyYou);
          }
```

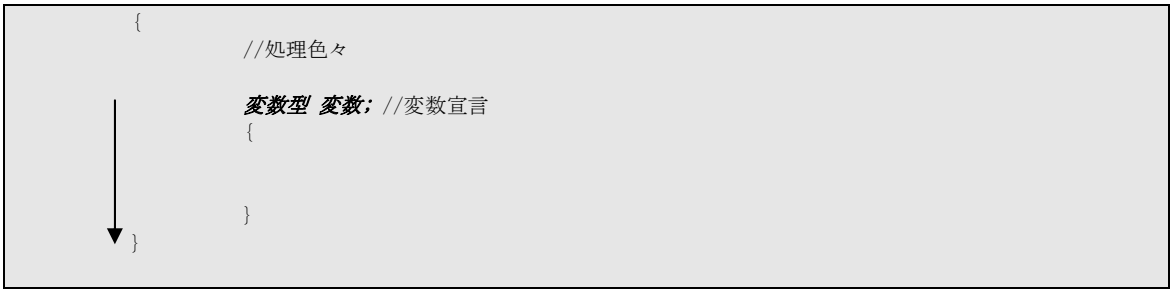
```
}
```

このようなクラスを作って、コンパイルをしてみると・・・

```
Variable.java:10: シンボルを見つけられません。
シンボル: 変数 onlyYou
```

というわけで、変数 onlyYou は残念ながら localTest メソッド内では使うことができません。つまり、変数 onlyYou は main メソッドでしか使えないわけです。localTest メソッドに到達する前に main メソッドのブロックが終了しているので，当たり前ですよね.

ローカル変数の有効範囲は以下の矢印で示された範囲のみという事になります.



ここ以外の場所ではまったく使うことが出来ませんのでご注意を.

ところで，ローカル変数の中でちょっと特殊なものとして，引数があります.

そういえば、メソッドの引数はどこで宣言されているでしょうか？

```
static 返値型 average(引数 1, 引数 2){//ここで引数を宣言
        処理
        return 返値
}
```

こう書きましたよね。引数は｛｝の外で宣言されているので、メソッドの外でも使える変数っぽいきがします．しかしながら，引数に関しては例外で，**その引数が宣言されたメソッドの中でしか使えません**．この点をご注意ください.

また、メソッドの引数以外にも，実はもう一つだけ例外があります。その例外とは、**for 文**です。

これまでにも for 文を使っていろいろプログラムを書いていましたが、こんなことを良くやっていたと思います。

```
for(int i = 0; i < 10; i++){
        //処理
}
```

さて、ここで宣言している変数 i はどこまで使うことができるでしょうか？

当然ながら、for 文の中は使うことができます。さらに、宣言しているのは for 文ブロックの外ですよね。ということは、その後も使うことができる気がします。どうでしょうか・・・？

```
for(int i = 0; i < 10; i++){
        //処理
}
i = 0;
```

こんなプログラムを確かめて見ましょう.

```
BlockVariable.java:17: シンボルを見つけられません。
シンボル: 変数 i
場所    : BlockVariable の クラス
                i=0;
```

まあ、ここでわざわざ言い出したところからも予想はついていたと思いますけれど、この場合は変数 i を for 文の外で使うことはできません。

これは、非常に例外的なことなのですが、Java では for 文で処理回数をカウントするためだけに変数を宣言することが非常に良くあります。

そんなときに、いちいち

```
int i;
for(i = 0; i < 10; i++){
        //処理
}
```

と書くのが大変なため、特別処置としてこのような仕組みができています。

参考までに、while 文などでは使えない、for 文特有の機能です. ご注意ください.

最後にローカル変数の有効範囲ををまとめておきましょうか。

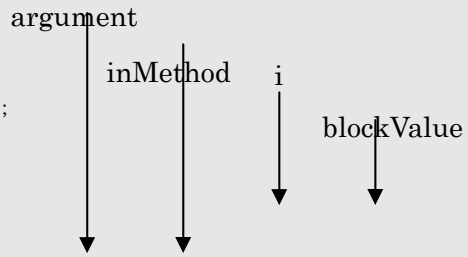
```
void method(int argument){

        int inMethod = 0;
        for(int i = 0; i < 10; i++){
                String blockValue = “ブロックの中”;

                //処理
        }
//処理
}

void otherMethod(){

}
```



11.2.3.2 初期化ルール

さて，次にローカル変数の初期化ルールについて説明します.

実はさらにローカル変数の場合，単に宣言すれば使えるようになるわけではないという特徴があります.

たとえば、こんな使い方はどうでしょうか？

```
        String onlyYou;
        System.out.println(onlyYou);
```

ここでのポイントは、onlyYou 変数を作成したものの、そこには何の値も入れていないという点です。

じつは、これをコンパイルしようとすると

```
Variable.java:6: 変数 onlyYou は初期化されていない可能性があります。
                System.out.println(onlyYou);
                                   ^
エラー 1 個
```

というわけで、エラーが発生してコンパイルできません。**ローカル変数は必ず初期化しなければいけない**という決まりがあるのです。ここは間違えやすいポイントなので気をつけてください。

11.2.4 フィールド

11.2.4.1 フィールドの有効範囲

フィールドは，クラス内のメソッドでならどこでも利用することが出来る変数です．変数宣言の前に書かれているメソッドでも利用することが可能です．

しかしながら，注意して欲しいのがメソッド内では宣言順番を問わずに使えるのですが，メソッドの外では宣言した後でしか利用することが出来ません．

メソッドの外で変数を使うことなんてあるのか？と思われるかもしれませんが，あとで説明する，初期化において別メソッドを初期化に使うことが出来るのです．そのときは，宣言した後でしか利用することが出来ません．

```
public class FieldMain {

          void showPrice(){
                    System.out.println("価格："+price);
                    System.out.println("税込み価格："+taxedPrice);
                    System.out.println("割引価格："+salePrice);
          }


          double salePrice = price*0.8;
          int price = 100;
          double taxedPrice = price*1.05;


}
```

この例の場合，フィールド salePrice を宣言して初期化を行うときに，まだ宣言していないフィールド price を使おうとしています．このようなプログラムは，

```
FieldMain.java:16: 順方向参照が不正です。
        double salePrice = price*0.8;
                           ^
エラー 1 個
```

となり，コンパイルすることが出来ません．ただし，これはフィールドをメソッド外で初期化するときにのみ気をつけるべきことですので，あまり気にすることはないかもしれ

ません. 基本的に，フィールドの初期化はコンストラクタで行うことのほうが多いのです.

もちろん，コンストラクタはメソッドの一種なので，どこで宣言したフィールドでも使うことが出来ます.

また，protected 以上のアクセス修飾子をつけていれば，派生クラス全体が有効範囲となります.

フィールドに関してはあまり難しいルールはありませんので，簡単に有効範囲を示した図を提示して終了したいと思います.

以上，矢印で示されている範囲がフィールド value の利用可能な範囲となります.

11.2.4.2 フィールドの初期化

フィールドの初期化は，コンストラクタで行われることが推奨されています. しかしながら，フィールドの宣言時に同時に初期化することも可能です.

```
public class FieldMain {
        int price = 100;
}
```

このように初期化したフィールドは，コンストラクタが呼ばれる前に初期化されるので，すべてのコンストラクタで初期化しなくても，値を決めておくことが出来て便利です.

もちろん，参照型もこんな形で初期化することが出来ます.

```
ublic class FieldMain {
        PersonalData personalData = new PersonalData("マリオ");
p }
```

覚えておいて損はないテクニックです.

このような初期化は，static なフィールドを作成するときに威力を発揮します.

static なフィールドはクラスのインスタンスを使わなくても利用することが可能です. しかしながら，初期化をコンストラクタで行ったのでは少なくとも一つはインスタンスを作らなければ初期化されないことになってしまいます.

そんなときは，フィールドの宣言と同時に初期化を行うことで，インスタンスを作らなくても static フィールドを初期化することが可能となります.

```
public class StaticFieldMain {
	public static String initializedString = "初期済みフィールド";
	public static String constractorInitString;

	StaticFieldMain(){
		constractorInitString = "コンストラクタで初期化";
	}
}
```

このクラスに対して，

```
	public static void main(String[] args) {
		System.out.println(StaticFieldMain.initializedString);
		System.out.println(StaticFieldMain.constractorInitString);

		StaticFieldMain sfm = new StaticFieldMain();
		System.out.println(StaticFieldMain.constractorInitString);
	}
```

このプログラムを実行すると，

```
初期済みフィールド
null
コンストラクタで初期化
```

となります.

このように，StaticFieldMain.constractorInitField はインスタンスを作る前は初期化されていない状態(中身が null)で，インスタンス作成後は文字列がちゃんと表示されるようになります.

一方，initializedString は最初から初期化が終わっています．いちいちコンストラクタを作る必要はありません.

このように，特に static なフィールドを使う場合は初期化の場所に気をつけましょう.

さて，次に特殊な例として final なフィールドの初期化についてです.

final 修飾子がついたフィールドは**値を変更することが出来ない**という特徴があります．ということは，作成したらすぐに値を入れないと，値を与えることが出来なくなってしまいそうですよね．でも，そうするとすべてのインスタンスで同じ値をもつフィールドになってしまい， static なフィールドのようになってしまいます.

それを回避するために，Java では **final フィールドもコンストラクタでなら初期化することができる**ようになっています．そのため，このようなクラスを作成することが可能となります.

```
public class Millionaire {

	public Millionaire(String answer) {
		finalAnswser = answer;
	}

	final String finalAnswser;
```

```
}
```

これで，finalAnswer をインスタンスごとに作成することが可能となります.

```
        public static void main(String[] args) {
                Millionaire ffm = new Millionaire("4");
                System.out.println("ファイナルアンサー？"+ffm.finalAnswser);
        }
```

このように，コンストラクタで値を与えることが可能であり，かつ二度とその値は変更できないようなフィールドが完成しました．答えが 4 番じゃなければ賞金は没収です.

11.2.4.3 フィールドの初期値

さて，フィールドは宣言時に初期化できることが分かったと思います.

では，フィールドで初期化しなかった場合はどんな値になるでしょう．正解は，配列の初期値と同じ値です. 例えば, int 型だったら 0 に初期化されますし, boolean だったら false に初期化されることが決まっています.

その一覧を以下に示します.

表 フィールドの初期値

変数	タイプ	初期値
int 型	整数	0
short 型	整数	0
long 型	整数	0
double 型	実数	0.0
float 型	実数	0.0
boolean 型	論理値	false
char 型	文字	0
byte 型	バイト	0
参照型変数	参照型	null
配列	配列	null

11.2.5 定数

さて，フィールドの中でも static 且つ final なものを**定数**と呼びます．定数も基本的にはフィールドの一種なのですが，static なので，すべてのインスタンスで共通の値であり，また final なので変更不可能です.

つまり，いつでもどこでも不変な値を持っているフィールドとなります.

このようなフィールドは，宣言と同時に値を初期化しなければいけないというルールが

あります.

したがって，こんな感じになります.

```
public class ConstantFieldMain {
          static final int DAYS_OF_WEEK = 7;
          static final int HOURS_OF_DAY = 24;
}
```

なお，慣習的に static で final なフィールドの名前は大文字とアンダーバーを使って書くことになっています．皆さんも定数を使う場合は，大文字とアンダーバーを使って変数名を記述するようにしましょう.

ここで,定数を使った例を一つご紹介しましょう.

前章でご紹介した Calendar クラスを覚えているでしょうか？Calendar クラスのインスタンスから年月日を取得する場合，こんな書き方をしましたよね.

```
                    int year = calendar.get(Calendar.YEAR); // 年を取得
                    int mon = calendar.get(Calendar.MONTH) + 1; // 月を取得
                    int date = calendar.get(Calendar.DATE); // 日を取得
```

ここで使われている,

```
Calendar.YEAR
```

などが Calendar クラスで定義されている int 型の定数となります．この場合カレンダーのどの値を取得したいを指定しているわけですが，Calendar.YEAR は 1，Calendar.MONTH は 2 という値になっています.

直接,

```
                    int year = calendar.get(1); // 年を取得
```

としても年を取得可能ですが，人間の目に見て分かりやすいように YEAR という定数を作って年を取得することを分かりやすくしているわけです.

このような変数は途中で値が変わってしまったら大変ですよね．また，インスタンスによって値が変化してしまっても困ります.

そんなわけで，YEAR は常に 1 だ！という保障を与えるために，static かつ final な定数を使って 1 という数字を YEAR という変数に対応させているわけです.

このように，値を変化させたくない普遍的な変数を作成したい場合は定数を利用しましょう.

11.3 間違えやすいメモリ

11.3.1 null の発生にご注意

ある変数が null になる時には以下のような場合があります．このような操作をするときは注意してください.

11.3.1.1 フィールドの初期化前

あるクラスのフィールドは基本的に初期値は null です. 先ほどの House 型の namePlate フィールドは最初は null 状態になります.

```
public class House{
          String namePlate; //このとき, null
}
```

null 状態では困ってしまうときは，フィールドの初期値を設定するか，コンストラクタで任意の値に変更します.

```
public class House{
          String namePlate =  ""; //初期値を指定

          public House(){
                    namePlate =  ""; /コンストラクタ内で初期化
          }
}
```

どちらもやらないでおくと，null のままになってしまいますので注意しましょう.

特に，ArrayList などは，new し忘れることが多いのでコンストラクタなどで忘れずに new する癖をつけましょう.

11.3.1.2 参照型の配列を作成したとき

参照型でも配列を作成することが出来ますが, その初期値はすべて null になっています.

```
String[] strArray = new String[100];
strArray[0] //null
strArray[1] //null
strArray[2] //null
strArray[3] //null
strArray[4] //null
strArray[5] //null
strArray[6] //null
・
・
・
```

とくに大きな配列を作成した場合は，全部中身が null ですので，データの入れ忘れがあった場合にすぐに NullPointerException が発生してしまいます.

気をつけてください.

11.3.1.3 null を代入したとき

参照型には null を代入することが出来ます．そのため，ある参照型変数に null を代入すると，値は null となりオブジェクトを指していない状態になります.

なお，メソッドの返り値などが null の場合気づかずに null を代入したのに，気づかずにオブジェクトがあると思って使ってしまうことが良くあるので，注意しましょう.

```
          String getString(){
                    return null;
          }

          public static void main(String[] ars){
                    String data = getString(); //data の値は null
```

```
        }
```

また，HashMap で保存されていないキーを使ってデータを取ろうとした場合，null が返ってくるという仕様になっています.

このことを忘れて，つい存在しないキーを使ってデータを取ろうとしてそのまま NullPointerException を発生させてしまうことがよくありますので，これまた注意してください.

```
HashMap<String, String> map = new HashMap();
String data = map.get("存在しないキー");
int length = data.length(); //NullPointerException 発生！
```

11.3.2 ディープコピーとシャロウコピー

参照型のコピーをする場合は，clone メソッドを使いますが，実は clone メソッドには大きな欠点があります．それは，clone メソッドが行えるのはシャロウコピーだけ，という点です.

ん？シャロウコピー？

実は，コピーにはシャロウコピーとディープコピーの二種類があるのです．簡単に言えば，シャロウコピーは表面的なコピーで，ディープコピーは深くまでコピーする物を言います.

具体的に見た方が早いでしょう.

```
public class ShallowCopy implements Cloneable{

        public String[] data =new String[1];


        public Object clone(){
                try {
                        return super.clone();
                } catch (CloneNotSupportedException e) {
                        e.printStackTrace();
                        return this;
                }
        }

}
```

ここでは単純化に話を進めるために clone しかない ShallowCopy クラスを作成します．このクラスはデータとして String の配列を持っています．面倒なので，一つしかデータがない配列ですけど.

さて，このクラスのインスタンスを作って，clone でコピーしてみましょう．そして，コピー先のインスタンスの data 配列の最初の要素を別の文字列に書き換えてしまいます．clone でコピーしたのですから，当然コピー先を変更してもコピー元には影響がないはずですよね.

```
public class CopyTypeMain {

	public static void main(String[] args) {
		ShallowCopy shallow = new ShallowCopy();
		shallow.data[0] = "シャロウコピー";

		ShallowCopy deep = (ShallowCopy)shallow.clone(); //コピー
		deep.data[0] = "ディープコピー"; //コピー先のデータの書き換え

		System.out.println(shallow.data[0]); //念のためコピー元データの確認
	}
}
```

さあ，この結果はというと・・・

```
ディープコピー
```

なんと，コピー先の値を変更したらコピー元のデータまで変更されてしまいました．Clone を使ったので同じオブジェクトを指してはいないはずなのに・・・

念のため確かめてみましょう．

```
		if(shallow != deep){
			System.out.println("異なるオブジェクト");
		}
```

shallow と deep が違うオブジェクトを指していれば異なるオブジェクトと表示されるはずですが・・・

```
異なるオブジェクト
```

う～ん，確かに異なるオブジェクトです．なのに，なぜデータを共有しているのでしょうか？

実は，これがシャロウコピーなのです．シャロウコピーでは，オブジェクトはコピーしますが，**オブジェクトの中身はコピーされない**のです．

正確に言うと，オブジェクト内のフィールドは，すべて=で代入されるだけなのです．

したがって，shallow.data と deep.data は同じオブジェクトになっているのです．確認してみましょうか．

```
		if(shallow.data == deep.data){
			System.out.println("同じオブジェクト");
		}
```

これを実行してみると，

```
同じオブジェクト
```

う～ん，確かに同じオブジェクトです．つまり，確かに shallow と deep は異なるオブジェクトとなりましたが，その中にあったフィールドが同じオブジェクトを指してしまっていたわけですね．

このように，単に clone メソッドを使うだけだとコピーが不十分な場合というのが良くあります．

そのため，clone メソッドを使う場合は自分で clone メソッドの中身まで書いた方が良い場合が多いでしょう．

今回の場合だと，このようなメソッドを実装すると望みの動作となります．

```
public class ShallowCopy implements Cloneable{

          public String[] data =new String[1];

          public Object clone(){
                    ShallowCopy copy = new ShallowCopy();
                    copy.data = (String[])data.clone();
                    return copy;
          }
}
```

この clone メソッドは，Object の clone とは違い，配列 data をまで clone しています．このように，フィールドまで clone を使って正確にコピーしていくことを**ディープコピー**といいます．clone を使う場合は，シャロウコピーなのかディープコピーなのかを常に意識して使うようにしましょう．

また，自分で clone を実装するときは，シャロウコピー，ディープコピーどちらを実装するべきか考えてから作成するようにしましょう．

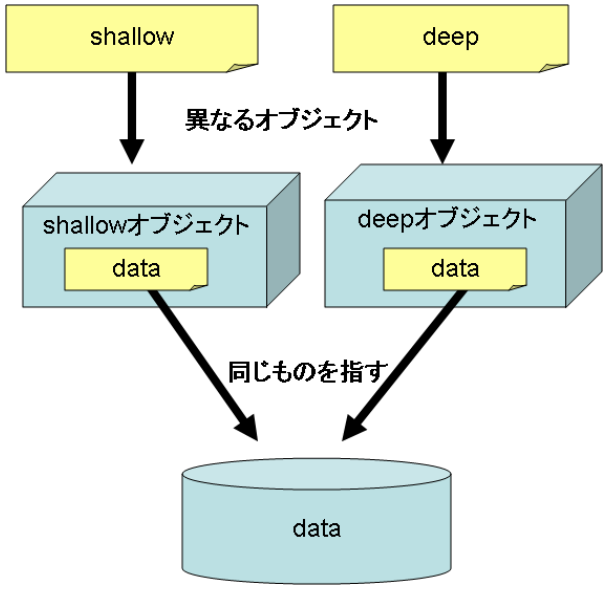


図 6 異なるオブジェクトのフィールドが同じものを指す

11.3.3 ローカル変数の初期化

ローカル変数は，初期化しなければ使えません．

そのため、少しでも初期化されていない可能性があるとコンパイルできません。たとえば、

```
int i = 15;
String onlyYou;
if(i > 10){
          onlyYou = "I love you";
}
```

```
System.out.println(onlyYou);
```

これを見ると、i が 10 以上の時は onlyYou は愛の言葉でちゃんと初期化されるように見えるので、大丈夫そうですが、残念ながらこれは認められません。

```
NotInitializedMain.java:11: 変数 onlyYou は初期化されていない可能性があります。
                System.out.println(onlyYou);
                                   ^
エラー 1 個
```

なぜかといえば、i の値が 10 未満の**可能性がある**からです。いや、直前で i の値を 15 と決めているので大丈夫だろう、と思うかもしれませんが、この辺は Java 君も意外と頑固で、万が一愛の言葉をもらえなかったときのことを考えて初期化されているとは認めてくれないのです。

では、どうすればいいのでしょうか？

じつは、こうすることで回避できます。

```
int i = 15;
String onlyYou;
if(i > 10){
        onlyYou = "I love you";
}
else{
        onlyYou = "I don't like you";
}
System.out.println(onlyYou);
```

これならば、もし万が一 i が 10 未満になったときは「あなたのことは好きじゃない」とちゃんと初期化してくれることが保証されるので、Java 君も安心してコンパイルしてくれるのです。if と else 両方があれば，どちらかが必ず呼ばれることが保障されているので，Java 君も確実に初期化されるな，と分かるわけです．そのため，それらの条件すべてで初期化されていれば絶対に初期化されることが保証されるので、安心だ、と判断できます。

となると、こう書いてもいいのかなと思ってしまいますよね。

```
int i = 15;
String onlyYou;
if(i > 10){
        onlyYou = "I love you";
}
else if(i <= 10){
        onlyYou = "I don't like you";
}
System.out.println(onlyYou);
```

が、これは認められません。i が 10 より大きいまたは 10 以下の場合の条件が指定されているのですが、そこまで Java 君は理解してくれないのです。

実は、Java 君は if 文の条件までは見ていないのです。ちょいとややこしいですね。が、そういうものなのだと覚えて置いてください。

基本的には変数を作ったときについでに適当な値で初期化しておくのが無難でしょう。

できるだけ、いつでも、

```
String onlyYou = "";
```

のように初期化しておくクセをつけるようにしましょう。