

# 第6章

## クラス定義とオブジェクトの 生成・使用

# 補足資料

# クラスの定義

```
public final class String
```

修飾子

クラス名

ファイル名はクラス名と一致させる。

```
implements java.io.Serializable, Comparable<String>, CharSequence {
    /* The value is used for... storage. */
    private final char value;

    /** use serialVersionUID from JDK 1.0.2 for interoperability */
    private static final long serialVersionUID = 1213141516171819;

    /**
     * Class String is special cased within the Serialization Stream Protocol.
     *
     * A String instance is written into an ObjectOutputStream according to
     * <a href="{@docRoot}/../platform/serialization/spec/output.html">
     * Object Serialization Specification, Section 6.2, "Stream Elements"</a>
     */
    private static final ObjectStreamField[] serialPersistentFields =
        new ObjectStreamField[0];

    /**
     * Initializes a newly created {@code String} object so that it represents
     * an empty character sequence. Note that use of this constructor is
     * unnecessary since Strings are immutable.
     */
    public String() {
        this.value = "";
    }
}
```

# 変数の宣言

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {
    /** The value is used for character storage. */
    private final char value[];

    /** Cache the hash code for this String. */
    private int hash; // Default: 0

    /** use serialVersionUID from JDK 1.0.2 for interoperability */
    private static final long serialVersionUID = -6849794470754667710L;

    /**
     * Class String implements Serializable within the Serialization Stream Protocol.
     *
     * A String instance is written into an ObjectOutputStream according to
     * <a href="{@docRoot}/../platform/serialization/spec/output.html">
     * Object Serialization Specification, Section 6.2, "Stream Elements"</a>
     */
    private static final ObjectOutputStreamField[] serialPersistentFields =
        new ObjectOutputStreamField[0];

    /**
     * Initializes a newly created {@code String} object so that it represents
     * an empty character sequence. Note that use of this constructor is
     * unnecessary since Strings are immutable.
     */
    public String() {
        this.value = "";
    }
}
```

メンバ変数

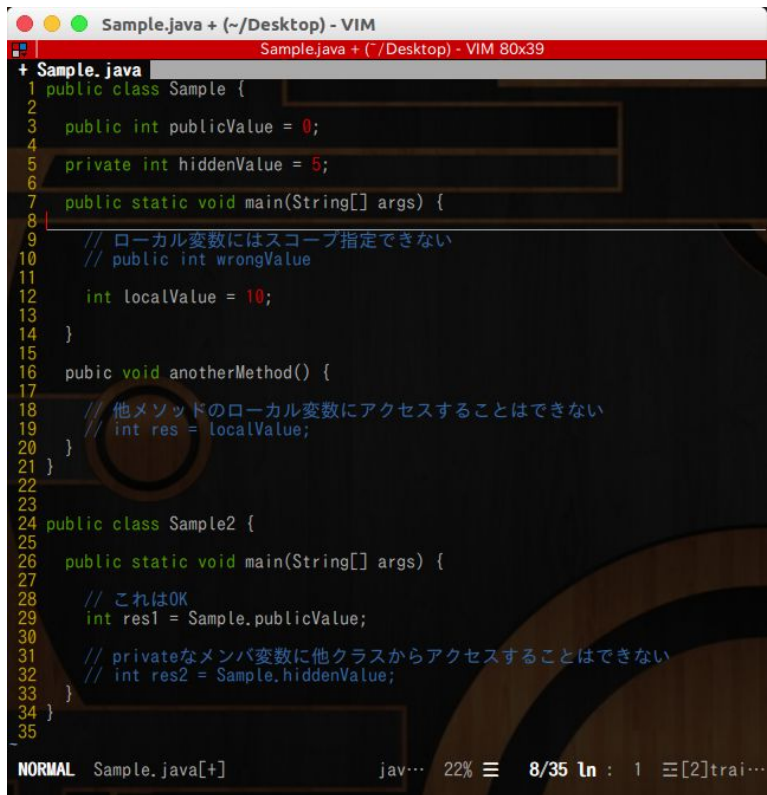
メンバ変数

# 変数の宣言

```
public int compareTo(String anotherString) {  
    int len1 = value.length;  
    int len2 = anotherString.value.length;  
    int lim = Math.min(len1, len2);  
    char v1[] = value;  
    char v2[] = anotherString.value;  
  
    int k = 0;  
    while (k < lim) {  
        char c1 = v1[k];  
        char c2 = v2[k];  
        if (c1 != c2) {  
            return c1 - c2;  
        }  
        k++;  
    }  
    return len1 - len2;  
}
```

メソッド(この場合はcompareTo)で  
宣言する変数は「ローカル変数」という。

# 変数の宣言



```
Sample.java + (~/Desktop) - VIM
Sample.java + (~/Desktop) - VIM 80x39
+ Sample.java
1 public class Sample {
2
3     public int publicValue = 0;
4
5     private int hiddenValue = 5;
6
7     public static void main(String[] args) {
8
9         // ローカル変数にはスコープ指定できない
10        // public int wrongValue
11
12        int localValue = 10;
13
14    }
15
16    public void anotherMethod() {
17
18        // 他メソッドのローカル変数にアクセスすることはできない
19        // int res = localValue;
20    }
21 }
22
23
24 public class Sample2 {
25
26     public static void main(String[] args) {
27
28        // これはOK
29        int res1 = Sample.publicValue;
30
31        // privateなメンバ変数に他クラスからアクセスすることはできない
32        // int res2 = Sample.hiddenValue;
33    }
34 }
35
NORMAL Sample.java[+] jav... 22% 8/35 ln : 1 [2]trai...
```

# メソッドの定義

```
@Override
@SuppressWarnings("unchecked")
public void sort(Comparator<? super E> c) {
    final int expectedModCount = modCount;
    Arrays.sort((E[]) elementData, 0, size, c);
    if (modCount != expectedModCount) {
        throw new ConcurrentModificationException();
    }
    modCount++;
}
```

ArrayListクラスより

public: 修飾子

void: 戻り値の型

※voidは「戻り値が無い」ことを示す

メソッド名: sort

引数リスト: Comparator<? super E> c

<...>とかsuperについては  
別テーマの話になるので割愛。  
詳細はSilver以降で。

# メソッドの定義

```
/**
 * Appends the specified element to the end of this list.
 *
 * @param e element to be appended to this list
 * @return <tt>true</tt> (as specified by {@link Collection#add})
 */
public boolean add(E e) {
    ensureCapacityInternal(size + 1); // Increments modCount!!
    elementData[size++] = e;
    return true;
}
```

ArrayListクラスより

public: 修飾子

boolean: 戻り値の型

メソッド名: add

引数リスト: E e

# メソッドの定義

```
// Add remaining segment  
if (!limited || list.size() < limit)  
    list.add(substring(off, value.length));
```

ArrayListクラスより

```
/**  
 * Inserts the specified element at the specified position in this  
 * list. Shifts the element currently at that position (if any) and  
 * any subsequent elements to the right (adds one to their indices).  
 *  
 * @param index index at which the specified element is to be inserted  
 * @param element element to be inserted  
 * @throws IndexOutOfBoundsException {@inheritDoc}  
 */  
public void add(int index, E element) {  
    rangeCheckForAdd(index);  
  
    ensureCapacityInternal(size + 1); // Increment  
    System.arraycopy(elementData, index, elementData,  
                      size - index, size - index);  
    elementData[index] = element;  
    size++;  
}
```

メンバ変数の配列である  
elementDataに要素を追加している。  
→ArrayListに要素が追加される。

メンバ変数の配列である  
sizeをインクリメント。  
→ArrayListの数が1つ増えたことを認識させている



# メソッドの定義

```
// Add remaining segment  
if (!limited || list.size() < limit)  
    list.add(substring(off, value.length));
```

ArrayListクラスより

```
/**  
 * Inserts the specified element at the specified position in this  
 * list. Shifts the element currently at that position (if any) and  
 * any subsequent elements to the right (adds one to their indices).  
 *  
 * @param index index at which the specified element is to be inserted  
 * @param element element to be inserted  
 * @throws IndexOutOfBoundsException {@inheritDoc}  
 */  
public void add(int index, E element) {  
    rangeCheckForAdd(index);  
  
    ensureCapacityInternal(size + 1); // Increment  
    System.arraycopy(elementData, index, elementData,  
                      size - index);  
    elementData[index] = element;  
    size++;  
}
```

メソッドはvoid型

呼び出しているメソッドがvoid型なので、  
変数 = list.add(...)で戻り値を受けることはできない。

# インスタンス化

# 割愛

155ページの「参考」について

余裕のある人は、初回に話をしたスタック領域 / ヒープ領域の話を思い出してみてください。

Silver/Goldにすら出てこないテーマですが、これを理解しているか否かで、バグを埋め込む可能性が減ると思います。

変数のスコープ

割愛

# コンストラクタの定義 & 呼び出し

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {
    /** The value is used for character storage. */
    private final char value[];
```

引数ナシのパターン

String str = new String();

strには""が入っている。

```
/**
 * Initializes a newly created {@code String} object so that it represents
 * an empty character sequence. No
 * unnecessary since Strings are immutable.
 */
public String() {
    this.value = "".value;
```

```
/**
 * Initializes a newly created {@code String} object so that it represents
 * the same sequence of characters as the argument;
 * newly created string is a copy of the argument's
 * explicit copy of {@code original} is needed, use
 * unnecessary since Strings are immutable.
 */
```

```
* @param original
 *   A {@code String}
 */
```

```
public String(String original) {
    this.value = original.value;
    this.hash = original.hash;
```

引数アリのパターン

String str = new String("java");

strには"java"が入っている。

# デフォルトコンストラクタ

## 割愛

160ページの図が全て。

- ①コンストラクタが明示的にコーディングされていない場合  
→「引数ナシ」のコンストラクタがコンパイル時に追加される
- ②コンストラクタが1つでも明示的にコーディングされている場合  
→自動追加されない

# オーバーロード

```
public int lastIndexOf(int ch) {  
    return lastIndexOf(ch, value.length - 1);  
}
```

```
public int lastIndexOf(int ch, int fromIndex) {  
    if (ch < Character.MIN_SUPPLEMENTARY_CODE_POINT) {  
        // handle most cases here (ch is a BMP code point or a  
        // negative value (invalid code point))  
        final char[] value = this.value;  
        int i = Math.min(fromIndex, value.length - 1);  
        for (; i >= 0; i--) {  
            if (value[i] == ch) {  
                return i;  
            }  
        }  
        return -1;  
    } else {  
        return lastIndexOfSupplementary(ch, fromIndex);  
    }  
}
```

Stringクラスより

以下が同じ

- ・メソッド名
- ・戻り値の型

違うのは引数だけ

# オーバーロード

```
public int lastIndexOf(int ch, int fromIndex) {  
    if (ch < Character.MIN_SUPPLEMENTARY_CODE_POINT) {  
        // handle most cases here (ch is a BMP code point or a  
        // negative value (invalid code point))  
        // ...  
    }  
}
```

public int lastIndexOf(int ch, int fromIndex)  
について、

①

public int lastIndexOf(int fromIndex, int ch)はオーバーロードではない。コンパイルエラー。  
→コンパイラから見たらいずれのメソッド宣言も、  
「1つ目の引数がint」「2つ目の引数がint」なので違いがない。

②

public String lastIndexOf(int ch, int fromIndex)はオーバーロードではない。コンパイルエラー。  
→lastIndexOf(10, 200)と呼び出した場合に、  
どちらのメソッドを呼び出したら良いかをコンパイラが判断できない。

# コンストラクタのオーバーロード

```
public String() {  
    this.value = "".value;  
}
```

Stringクラスより

引数ナシ

```
public String(String original) {  
    this.value = original.value;  
    this.hash = original.hash;  
}
```

String(String)

```
public String(char value[]) {  
    this.value = Arrays.copyOf(value, value.length);  
}
```

String(char[])

```
public String(char value[], int offset, int count) {  
    if (offset < 0) {  
        throw new StringIndexOutOfBoundsException(offset);  
    }  
    if (count <= 0) {  
        if (count < 0) {  
            throw new StringIndexOutOfBoundsException(count);  
        }  
        if (offset <= value.length) {  
            this.value = "".value;  
            return;  
        }  
    }  
    // Note: offset or count might be near -1>>>1.  
    if (offset > value.length - count) {  
        throw new StringIndexOutOfBoundsException(offset + count);  
    }  
    this.value = Arrays.copyOfRange(value, offset, offset+count);  
}
```

String(char[], int, int)

前述のオーバーロードとの違いは、戻り値型の考慮がないことだけ。

←コンストラクタに戻り値はない。



# static変数とstaticメソッド

static変数

インスタンスごとに変数に異なる値を設定する必要がない場合

staticメソッド

インスタンスごとに挙動を変化させる必要がない場合

# static変数とstaticメソッド

```
public class ArrayList<E> extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
    private static final long serialVersionUID = 8683452581122892189L;

    /**
     * Default initial capacity.
     */
    private static final int DEFAULT_CAPACITY = 10;
```

ArrayListクラスより

DEFAULT\_CAPACITYを定数(常に同じ値)として扱う場合、インスタンスごとに変数を割り当てる必要はない。  
→staticキーワードをつける



「static final 型 変数名」は定数宣言を意味する。  
頻出なので覚えましょう。

# static変数とstaticメソッド

```
public static String valueOf(int i) {  
    return Integer.toString(i);  
}
```

Stringクラスより

valueOfは、引数で受けた値を String型に変換する変数。

intの値が7であれば、返ってくるのは必ず "7"となる。

メンバ変数の内容によって結果が変化することがない。  
→staticメソッドとして定義する。

# static変数とstaticメソッド

```
public static String valueOf(int i) {  
    return Integer.toString(i);  
}
```

Stringクラスより

「インスタンスごとに挙動を変化」するのは、メソッド内でメンバ変数を使用しているから。

Stringクラスの場合は、“7”や“ABC”といった文字列をメンバ変数として保持している。  
equals(Object anObject)は自分(メンバ変数に保持された“7”や“ABC”)と  
引数の文字列を比較しているので、staticではない。

valueは  
メンバ変数

```
public boolean equals(Object anObject) {  
    if (this == anObject) {  
        return true;  
    }  
    if (anObject instanceof String) {  
        String anotherString = (String)anObject;  
        int n = value.length;  
        if (n == anotherString.value.length) {  
            char v1[] = value;  
            char v2[] = anotherString.value;  
            int i = 0;  
            while (n-- != 0) {  
                if (v1[i] != v2[i])  
                    return false;  
                i++;  
            }  
            return true;  
        }  
    }  
    return false;  
}
```

// strの内容(メンバ変数の状態)によって挙動が変化する

String str = “ABC”;

// これはtrue  
str.equals(“ABC”);

// これはfalse  
str.equals(“7”);

# static変数とstaticメソッド

```
public static String valueOf(int i) {  
    return Integer.toString(i);  
}
```

Stringクラスより

## ★マメ知識★

2ページ前で、「intの値が7であれば、返ってくるのは必ず "7" となる。」と言ってるけど、これは「同じパラメータに対して同じ結果が返ってくる」ことを意味する。

じゃあ、パラメータがなかったら???

定数は引数ナシのメソッドから常に同じ結果が返ってくると考えることもできる。

```
public static final String fixValue() {  
    return "FIX VALUE";  
}
```

と

```
public static final String FIX_VALUE = "FIX VALUE";
```

実際、Haskellのような関数型言語では、  
このように定数を定数関数として宣言する。

<xの値によって結果が変化>	<xとyの値によって結果が変化>	<パラメータによって結果が変化しない>
$f(x) = x + 1$	$f(x, y) = x * y + 1$	$f() = 1$ つまり $f = 1$ と同義

# インスタンスメンバとstaticメンバのクラス内でのアクセス

	インスタンス変数	static変数
インスタンスメソッド	○:アクセス可	○:アクセス可
staticメソッド	×:アクセス不可	○:アクセス可



意味がわかれば覚えるまでもない明快なルールだけど、  
意味がわからない内は丸暗記でOK。

# インスタンスメンバとstaticメンバのクラス内でのアクセス

	インスタンス変数	static変数
インスタンスメソッド	○: アクセス可	○: アクセス可
staticメソッド	×: アクセス不可	○: アクセス可

インスタンス変数はインスタンスごとに挙動を変化させたい場合に使用する変数。  
一方、staticメソッドは、インスタンスごとに挙動を変化させる必要がない場合に使用するメソッド。  
よって、staticメソッドからインスタンス変数を参照することはできない。

もし参照することができたとしたら、staticメソッドの結果がインスタンス変数の内容によって変化してしまう??? それはもはや static(静的)ではない。。

# インスタンスメンバとstaticメンバのクラス内でのアクセス

	インスタンス変数	static変数
インスタンスメソッド	○:アクセス可	○:アクセス可
staticメソッド	×:アクセス不可	○:アクセス可

static変数の値はインスタンスごとに設定する必要のない変数。  
なので、どちらからもアクセス可。



# アクセス修飾子

169ページの表 6-2を暗記すればOK。

ただし、private classが「×」になっているのは正確ではない。privateなclassを作成する方法はある。

←Bronzeの範囲ではないので、詳細は割愛。

試験では「×」と答えれば良いと思う。

# アクセス修飾子

```
1 public class Sample {
2
3     public static void main(String[] args) {
4
5         Sample inner = new Sample();
6         Inner a = inner.new Inner();
7         a.inMethod();
8     }
9 }
10
11 private class Inner {
12
13     public void inMethod(){
14         System.out.println("INNER");
15     }
16 }
17 }
18
19 }
```

# アクセス修飾子

fish Desktop/aa  
Sample.java (~ /Desktop/aa) - VIM 50x61

```
1 public class Sample {  
2  
3     public static void main(String[] args) {  
4  
5         Sample inner = new Sample();  
6         Inner a = inner.new Inner();  
7         a.inMethod();  
8     }  
9  
10  
11     private class Inner {  
12  
13         public void inMethod(){  
14             System.out.println("INNER");  
15         }  
16  
17  
18  
19     }
```

```
[23:34:39] /Desktop/aa (0)  
> javac Sample.java  
[23:34:42] /Desktop/aa (0)  
> java Sample  
INNER  
[23:34:44] Desktop/aa (0)
```

inner(内部)クラスを呼び出して使用することもできる。

<privateなinnerクラスのメリット>

用意したクラスを外部(この場合、Sampleクラス以外)に意識させたくない場合に、Sampleクラス以外からInnerクラスが見えなくなる。

innerクラスを隠蔽することで、変更が強くなる。

※オブジェクト志向において、「隠蔽」はプログラムを変更に強くするためのキーワード

# アクセス制御の推奨ルール

ArrayListクラスより

```
/**
 * The size of the ArrayList (the number of elements it contains).
 *
 * @serial
 */
private int size;
```

メンバ変数はprivateにする。

←publicにすると、外部から直接参照/編集することができる。  
予期せぬバグを生むので避けること。

```
/**
 * Returns the number of elements in this list.
 *
 * @return the number of elements in this list
 */
public int size() {
    return size;
}

/**
 * Returns <tt>true</tt> if this list contains no elements.
 *
 * @return <tt>true</tt> if this list contains no elements
 */
public boolean isEmpty() {
    return size == 0;
}
```

「直接参照/編集しなければ良い」と思う  
かもしれないけど、コーディングの基本は  
「他人を信じない」こと。  
不正な処理をされる可能性は可能な限り  
排除する。

publicなメソッドを通してprivateなメンバ  
変数にアクセスする。

# ガベージコレクタとは

```
public boolean equals(Object anObject) {  
    if (this == anObject) {  
        return true;  
    }  
    if (anObject instanceof String) {  
        String anotherString = (String)anObject;  
        int n = value.length;  
        if (n == anotherString.value.length) {  
            char v1[] = value;  
            char v2[] = anotherString.value;  
            int i = 0;  
            while (n-- != 0) {  
                if (v1[i] != v2[i])  
                    return false;  
                i++;  
            }  
            return true;  
        }  
    }  
    return false;  
}
```

Stringクラスより

equalsメソッドの処理が終わったら、これらのローカル変数は不要になる。

ガベージコレクタ(GC)によって、使用しなくなった変数のゴミ削除が実行される。  
ただし、これがいつ実行されるかを制御することはできない。

# オブジェクトをガベージコレクタの対象にする

説明の意味を理解して暗記しましょう。

理屈としては覚えておいてほしいけど、実際に使うことは多くないです。

メソッド内で宣言したローカル変数であればメソッドの処理が完了したタイミングで自動的に GC対象となるため、わざわざnullを代入するまでもなくすぐに GC対象となる。

業務機能のコーディングではあまり意識するケースはないはず。