日本電子専門学校ゲーム制作科 2 年次

Java I 基礎

言語仕様と用例

Norihisa Ishida

2015/04/20

このドキュメントは言語仕様の説明とその用例が記されています。

1. 基本データとリテラル

型	種類	範囲(最小値)	範囲(最大値)	リテラル(例)
byte	整数（1 バイト）	-128	127	3
short	整数（2 バイト）	-32768	32767	3
int	整数（4 バイト）	-2147483648	2147483647	3
long	整数（8 バイト）	-9223372036854775808	9223372036854775807	3L
char	文字（2 バイト）	¥u0000	¥uFFFF	‘A’
float	実数（4 バイト）	-3.4028235E+38 ～ -1.401298E-45	1.401298E-45 ～ 3.4028235E+38	3.14F
double	実数（8 バイト）	-1.79769313486231570 E+308 ～ -4.94065645841246544 E-324	4.94065645841246544E -324 ～ 1.79769313486231570E +308	3.14
boolean	論理（1 バイト）	false	true	true

整数はサイズ(バイト数)の違いにより 4 種類ある。文字は 2 バイトでかつ負値がない(符号なし)。実数は単精度と倍精度の浮動小数点数により決められている。論理型は比較・論理演算の演算結果の値となる。

リテラルは long 型では数字の後ろに「L」、float 型で「F」をつける。大文字でも小文字でも良い。文字型は値の前後をシングルクォーテーション「’」で挟む。論理型は true, false で 2 つの値を表記する。

なおこれらの基本型に対して、その機能を補完するプリミティブ・ラッパークラスがある。

リテラルの詳細

整数　　0123（8 進数）、123（10 進数）、0xA7（16 進数）
浮動小数点　　12.3F、12.3f、12.3e2F、（単精度(float)）
　　12.3、12.3D、12.3d、12.3e2、12.3e2D、12.3e2d（倍精度(double)）
文字　　‘A’、’あ’、’¥u0123’
　　‘¥t’、’¥n’、’¥r’、’¥b’、’¥¥’、’¥”、’¥””（エスケープシーケンス）
文字列　　“abc”（基本型では扱えないので String クラスで扱う）

２．コメント

//　　（単一行コメント）

/*
*/　　（複数行コメント）

/**
*/　　（複数行コメント、JavaDoc 対応）

３．演算

３－１．算術演算

演算子	種類	オペランドと評価値の型	用例	評価値
+	加算	byte, short, int, long, char, float, double, (String)	2+3	5
-	減算	byte, short, int, long, char, float, double	2-3	-1
*	乗算	byte, short, int, long, char, float, double	2*3	6
/	除算	byte, short, int, long, char, float, double	2/3	0
%	剰余	byte, short, int, long, char, float, double	2%3	2

３－２．ビット演算

演算子	種類	オペランドと評価値の型	用例	評価値
&	AND	byte, short, int, long, char	5&3	1
|	OR	byte, short, int, long, char	5|3	7
^	XOR	byte, short, int, long, char	5^3	6
<<	左シフト	byte, short, int, long, char	5<<3	40
>>	右シフト	byte, short, int, long, char	5>>3	0
>>>	右シフト(ゼロ埋め)	byte, short, int, long, char	-1>>>30	3
~	補数	byte, short, int, long, char	~1	-2

３－３．符号

記号	種類	オペランドと評価値の型	用例	評価値
+	正	byte, short, int, long, char, float, double	+1	1
-	負	byte, short, int, long, char, float, double	-1	-1

３－４．代入演算

演算子	種類	用例	評価値
+=	加算と代入	x += 2	x=3 のとき x は 5
-=	減算と代入	x -= 2	x=3 のとき x は 1
*=	乗算と代入	x *= 2	x=3 のとき x は 6
/=	除算と代入	x /= 2	x=3 のとき x は 1
%=	剰余と代入	x %= 2	x=3 のとき x は 1
&=	AND と代入	x &= 2	x=3 のとき x は 2
|=	OR と代入	x |= 2	x=3 のとき x は 3
^=	XOR と代入	x ^= 2	x=3 のとき x は 1
<<=	左シフトと代入	x <<= 2	x=3 のとき x は 12
>>=	右シフトと代入	x >>= 2	x=3 のとき x は 0
>>>=	右シフトと代入	x >>>= 30	x=-1 のとき x は 3
++	1 を加算して代入	x ++または++x	x=3 のとき x は 4
--	1 を減算して代入	x--または--x	x=3 のとき x は 2

３－５．関係演算(評価値の型はすべて boolean)

演算子	種類	オペランドの型	用例	評価値
>	大きい	byte, short, int, long, char, float, double	3 > 5	false
<	小さい	byte, short, int, long, char, float, double	3 < 5	true
>=	大きいまたは等しい	byte, short, int, long, char, float, double	3 >= 5	false
<=	小さいまたは等しい	byte, short, int, long, char, float, double	3 <= 5	true
==	等しい	byte, short, int, long, char, float, double	3 == 5	false
!=	等しくない	byte, short, int, long, char, float, double	3 != 5	true

３－６．論理演算

演算子	種類	オペランドと評価値の型	用例	評価値
||	または	boolean	3 > 5 || 2 == 2	true
&&	かつ	boolean	3 > 5 && 2 == 2	false
!	否定	boolean	!(3 > 5)	true

３－７．3 項演算

論理値?値 1:値 2　　←論理値が true なら値 1、false なら値 2 が評価値となる

用例

int x = 3<2 ? 4 : 5 ;　　←x には 5 が代入される

４．制御構文

４－１．条件分岐（if 文）

```
if(論理値){
        処理;
}
```

論理値が true のとき処理を行う。

```
if(論理値){
        処理 1;
}else{
        処理 2;
}
```

論理値が true のときは処理１、false のときは処理２を行う。

```
if(論理値１){
        処理 1;
}else if(論理値 2){
        処理 2;
} else{
        処理 3;
}
```

論理値１が true のときは処理１、論理値１が false で論理値２が true のときは処理２、論理値１も論理値２も false のときは処理３を行う。

ちなみに論理値は関係演算や論理演算などを使って計算する。以下は用例。

```
int age = 25;
if(age < 18) System.out.println(“未成年”);
else if(age >= 75){
    System.out.println(“後期高齢者”);
}else{
    System.out.println(”成年”);
}
```

４－２．条件分岐(switch 文)

```
switch(値){
case ラベル１：
        処理１；
        break;
case ラベル２：
        処理２；
        break;
default:
    処理３；
}
```

「値」と「ラベル」は byte, short, int, long, char, String の型を用いることができる。なおスコープの先頭での変数宣言はできない。

用例

```
char rank = 'A';
String result;
switch(rank){
case 'A':
    result = "優";
    break;
case 'B':
    result = "良";
    break;
case 'C':
    result = "可";
    break;
default:
    result = "不可";
}
System.out.println("結果は" + result + "です。");
```

４－３．繰り返し(while 文)

```
while(論理値){
```

```
        処理;
}
用例
int cnt= 0;
while( (cnt++) < 3 ){
        System.out.println(cnt+"回");
}

繰り返し(do-while 文)
do{
        処理;
}while(論理値);

用例
Scanner sc;
do{
        System.out.print("yes/no?");
        sc = new Scanner(System.in);
}while(sc.next().equals("no"));
```

４－４．繰り返し(for 文)

```
for( 初期式 ; 論理値 ; 式 ){
        処理;
}

用例
for( int i=0 ; i<3 ; i++ ){
        System.out.println(i+1+"回");
}
```

４－５．繰り返し(foreach 文)

```
for(変数:データ){
        処理;
}
```

データは配列やコレクタ、変数はそのデータを扱う型を用いる。

用例

```
int[] data = {1,2,3};
for(int x:data){
        System.out.println(x+“回”);
}
```

５．配列型変数と配列オブジェクト

５－１．配列型変数の宣言

型名[] 変数名;
または
型名 変数名[];

上は配列型として変数を宣言しているので以下の２つの例はｂの意味が異なる。

```
int[] a, b;                 //配列型変数 a,b を宣言
int a[], b;                 //配列型変数 a と整数型変数 b を宣言
```

５－２．配列オブジェクトの生成（配列インスタンス生成）

new 型名[要素数];

配列オブジェクトは生成時に単体で使用することも可能だが、配列型変数へ参照しないとデータが自動的に破棄(ガベージコレクション)されてしまうので、一般には以下のように使用する。

```
int[] a;
a = new int[10];
```

５－３．配列型変数の宣言と配列オブジェクトへの参照

型名[] 変数名 = new 型名[要素数];
または

型名 変数名[] = new 型名[要素数];

用例
int[] a = new int[10];

５－４．配列の初期化

new 型名[]{値 1,値 2 ,値 3 , ...};

この場合も配列型変数へ参照させないとオブジェクトが破棄されてしまうので、以下のように使用する。

int[] a;
a = new int[]{1,2,3};

この例では変数宣言と初期化を別々で行っているが、同時に行うこともできる。

５－５．配列型変数宣言と初期化

型名[] 変数名 ={ 値 1, 値 2, 値 3, ... };
または
型名 変数名[] ={ 値 1, 値 2, 値 3, ... };

この初期化の方法は配列型変数の宣言と同時に行うときのみ可能。

５－６．配列の用法

値の代入または参照
配列型変数[要素番号] = 値;

配列が基本データ型なら代入、それ以外は参照となる。要素番号は必ず 0 から始まる。

用例
int[] a = new int[2];
a[0] = 1;
a[1] = 2;

５－７．配列オブジェクトのフィールドとメソッド

詳細は API ドキュメントおよび JLS によるとして､代表的なものを以下に示しておく。

length	配列の要素数
clone()	配列の複写データを生成

用例

```
int[] a,b;
a = new int[]{1,2,3};
System.out.println(a.length);
b = a.clone();
System.out.println(b[1]);
```

出力結果

```
3
2
```

５－８．多次元配列

５－８－１．配列参照型変数の宣言（2 次元）

```
型名[][]　変数名;
```

または

```
型名　変数名[][];
```

次元数を増やすには[]を追加する。

５－８－２．配列のオブジェクト生成（インスタンス生成, 2 次元）

```
new 型名[要素数 1][要素数 2];
```

次元を増やすには[要素数 N]を追加する。一般には配列参照型変数へ参照させて使う。

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

５－８－３．配列オブジェクトの初期化(2 次元)

new 型名[][]{{値 1, 値 2}, {値 3, 値 4}, …};

次元数は{}内の{}の入れ子の深さで決まる。

用例

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

５－８－４．配列参照型変数の宣言と配列オブジェクトの初期化（2 次元）

型名[][] 変数名 = {{値 1, 値 2}, {値 3, 値 4}, …};
または
型名 変数名[][] = {{値 1, 値 2}, {値 3, 値 4}, …};

用例

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

なお初期化時に内側の{}の要素が異なるデータを用意することもできる。以下。

```
int[][] a = {{1,2,3},{4,5},{6, 7, 8, 9}};
for(int i = 0; i<a.length;i++){
for(int j = 0; j<a[i].length;j++){
        System.out.print(a[i][j]);
}
System.out.println();
}
```

出力結果

```
123
45
6789
```

６．ガベージコレクション

参照型変数で扱うオブジェクトのすべて（配列、クラスなど）は参照を失った際に自動的に破棄される。たとえば以下のような場合、

```
int[] a;
a = new int[5];    //A
a = new int[3];    //B
```

コメント A で生成されたオブジェクト(要素数 5)は変数 a に参照されているが、コメント B で生成されたオブジェクトが変数 a で参照されると A のオブジェクトへのアクセスができなくなる。このようなオブジェクトは破棄対象となる。以下の例の実行結果を確認されたい。

```
int[] a;
a = new int[]{1, 2, 3};
a = new int[]{4, 5, 6, 7};
System.out.println(a.length);
for(int i = 0; i < a.length; i++){
        System.out.print(a[i]);
}
```

出力結果：

```
4
4567
```

なお、破棄を明示する方法は用意されていないが、null 値を参照させることで結果的にガベージコレクションを行わせることができる。たとえば以下。

```
String s = new String(“HELLO”);
s = null;
System.out.println(s);
```

出力結果

```
null
```

なお、null 値は参照型であり、参照型変数が何も参照していないことを表す場合に用いる。

基本データに用いることはできない。

７．クラス参照型変数とクラスオブジェクト

７－１．クラス参照型変数の宣言

型名　変数名;

用例
String str;

なお、配列宣言は上記配列の項目に準ずる。

７－２．クラスオブジェクトの生成（クラスインスタンス生成）

new　クラス名(引数);

このままでは参照型変数への参照がおこなわれていないので、ガベージコレクションが発生し、継続して使用できない。したがって一般には以下のように参照型変数を用意して使用する。

String s;
s = new String(“HELLO”);

なお引数についてはクラスのコンストラクタの仕様によって決まる。

８．文字列オブジェクト

“文字列”は文字列オブジェクトとしてふるまう。文字列オブジェクトは String クラスのインスタンスと同じである。

文字列の連結

“文字列１” + “文字列２”

これは”文字列１文字列２”と同じになる。

文字列の集約

String s1 = “HELLO”;
String s2 = “HELLO”;

上記の s1 と s2 は同じ文字列のため集約され、１つのオブジェクトとして認識される。すなわち s1 == s2 の演算結果は true である。ちなみに以下の場合は２つのオブジェクトとなる。すなわち s1 == s2 の演算結果は false である。

String s1 = “HELLO”;
String s2 = “hello”;

なお、同じ文字列を異なるオブジェクトとして使うには以下のような方法をとる。

String s1 = new String(“HELLO”);
String s2 = new String(“HELLO”);

文字列の比較

上記の事例からわかるように、文字列が同一であるということと文字列オブジェクトが同じであることとは異なる。文字列が同一であることを確認するには String クラスのメンバメソッドである equals を使用する。

“HELLO”.equals(“HELLO”)

このメソッドの戻り値は boolean で、この場合は true となる。

９．クラス定義

クラスは内部にフィールド、メソッド、コンストラクタの３つを含む構造データの一つである。以下のような書式である。

```
class クラス名{
        型名 フィールド名;
        クラス名(引数の型 引数名){
                処理;
        }
        戻り値の型 メソッド名(引数の型 引数名){
                処理;
        }
}
```

クラスの内部構造はメンバと呼ばれ、クラスに属するという意味であえてメンバフィールド、メンバメソッドと呼ぶこともある（C++ではメンバ変数、メンバ関数と呼ぶ)。

この定義により生成されるオブジェクトは内部にメンバも生成され利用可能となる。オブジェクトごとに生成されるメンバのことをインスタンスメンバと呼ぶ。

用例

```
class Enemy{
        int hp;
        String name;
        Enemy(String name, int hp){
                this.name = name;
                this.hp = hp;
        }
        void showProperties(){
                System.out.println(“名前：”+name+” , HP：”+hp);
        }
}
class Game{
        public static void main(String[] args){
                Enemy e = new Enemy(“Pikachuu”, 128);
                e.showProperties();
        }
}
```

実行結果

名前：Pikachuu, HP：128

１０．コンストラクタ

クラス名と同名でクラス内に配置された処理区分をコンストラクタという。コンストラクタはオブジェクト生成時に呼び出されて実行する。以下のとおり。

```
class Enemy{
	Enemy(){
		System.out.println(“敵が出現します”);
	}
}
class Game{
	public static void main(String[] args){
		new Enemy();
	}
}
```

出力結果

敵が出現します

１０－１．コンストラクタのオーバーロード

コンストラクタは引数の異なるものを複数用意することができる(オーバーロード)。またコンストラクタから別のコンストラクタを呼び出すには this を使う。以下の通り。

```
class Enemy{
	Enemy(){
		System.out.println("敵");
	}
	Enemy(String s){
		this();
		System.out.println(s);
	}
}
class Game5{
	public static void main(String[] args){
		new Enemy("ボス");
	}
}
```

出力結果

敵

ボス

なお、this キーワードはコンストラクタの先頭でなければならない。

１１．スタティック・メンバ

クラス内部で static キーワードのついたメンバは実体を伴うメンバであり、インスタンス化される前述のものと異なりクラスメンバとして唯一のデータとなる。以下に例を示す。

```
class Enemy{
	int hp;
	String name;
	static String title = “Pokemon”;
	Enemy(String name, int hp){
		this.name = name;
		this.hp = hp;
	}
	void showProperties(){
		System.out.println(“ゲーム名：”+title+“ ,名前：”+name+” , HP：”+hp);
	}
}
class Game{
	public static void main(String[] args){
		Enemy e1 = new Enemy(“Pikachuu”, 128);
		e1.showProperties();
		Enemy e2 = new Enemy(“Pokachuu”, 255);
		e2.showProperties();

	}
}
```

出力結果
ゲーム名：Pokemon, 名前：Pikachuu, HP：128
ゲーム名：Pokemon, 名前：Pokachuu, HP：255

ここでメンバフィールドの title はスタティックメンバであり、e1 と e2 の２つのインスタンスに共通した値である。たとえば e1 から title を書き換えると e2 の title も同じになって

いることが確認できる。以下。

```
class Game{
	public static void main(String[] args){
		Enemy e1 = new Enemy(“Pikachuu”, 128);
		e1.title = “Youkai Watch”;
		e1.showProperties();
		Enemy e2 = new Enemy(“Pocachuu”, 255);
		e2.showProperties();

	}
}
```

出力結果
ゲーム名：Youkai Watch, 名前：Pikachuu, HP：128
ゲーム名：Youkai Watch, 名前：Pocachuu, HP：255

したがって通常は混乱を避けるため「クラス名.メンバ」でアクセスするのが普通である。上の例では「e1.title」ではなく「Enemy.title」と書く。

１２．パッケージとアクセス制限

パッケージの実体はクラスを保存したディレクトリ（フォルダ）またはアーカイブファイル（JAR）のことであり、パッケージをアクセス制限の範囲(名前空間)と考える。これにより、同名のクラスをパッケージごとに配置して名前の衝突を避けたり、アクセス単位のグループ化を行ったりすることができる。

パッケージはEclipseやIDEAなどの統合開発環境を使用する場合はプロジェクト設定で行うが、コードでパッケージを指定してその中にコンパイル済みのクラスファイルを置くこともできる。以下。

```
package test;
class Test0{
	...
}
```

この場合はtestフォルダにTest0.classが配置される（Test0.javaとは別の場所になる）。

１２－１．パッケージのインポート

パッケージはディレクトリの階層またはアーカイブファイルの内部階層構造によって位置が決まるので、パッケージ内のクラスにアクセスするには以下の方法をとる。

① 参照位置が現在のクラスの位置と同階層のディレクトリ（ここでは TEST とする）なら
TEST.クラス名
のようにアクセスする。

② パッケージをインポートする。特に jdk の提供する API を使用するなら
import java.パッケージ名.クラス名
標準 API なら
import com.パッケージ名.クラス名
などとする。

１２－２．デフォルトパッケージ

import で何も指定していない場合でも、java.lang.*がインポートされているということになっているので、java.lang に属するクラスはパッケージ指定なしで利用が可能である。例えば System クラスなどがそれに該当する。

パッケージはディレクトリ階層を含むので場合によって参照位置がかなり深くなることもある。

用例

```
import java.util.Random;
class P04031{
	public static void main(String[] args){
		Random r = new Random();		//import でクラスの階層を指定
		System.out.println(r.nextInt());
		System.out.println(Math.PI);		//デフォルトパッケージを使用
		java.util.Date d = new java.util.Date();	//パッケージの階層で指定
		System.out.println(d.getTime());
	}
}
```

１３．アクセス制限

修飾子	範囲
private	クラス内
(なし)	同一パッケージ内のすべて
protected	同一パッケージ内のすべておよびパッケージ外のサブクラス内
public	すべて
final	継承させないクラス、値を変えない変数（定数）

アクセス制限はクラス、インターフェース、コンストラクタ、フィールド、メソッドの前につけることでその使用範囲を限定する。ただし修飾子の付け方にはルールがあり、種類や組み合わせによって決まっている。詳しくは JLS を参照のこと。

１４．継承

別のクラスの定義を引き継いで新たなクラスを定義する方法を継承という。以下のとおり。

```
class スーパークラス名{
        …
}
class サブクラス名 extends スーパークラス名{
        …
}
```

継承元のクラスを「スーパークラス」、継承先のクラスを「サブクラス」と呼ぶ。サブクラスはスーパークラスのメンバを引き継いで使用することができる。以下の通り。

```
class Enemy{
        void message(){
                System.out.println("敵が現れた");
        }
}
class Monster extends Enemy{
}
class Game{
        public static void main(String[] args){
                new Monster().message();
```

```
        }
}
```

出力結果
敵が現れた

クラス Monster は Enemy を継承しているので、Monster クラスに何の記述がなくても Enemy のメンバメソッド message を呼び出すことができる。

１５．オーバーライド

サブクラスのメソッドの名前と引数のシグニチャ（引数の型と数）がスーパークラスのメソッドの名前と引数のシグニチャが同一ならサブクラスはスーパークラスの同一メソッドを呼び出せず、サブクラスでの定義に従った動作となる(上書きされる)。以下の通り。

```
class Enemy{
        void message(){
                System.out.println("敵が現れた");
        }
}
class Monster extends Enemy{
        void message(){                                 //←オーバーライドメソッド
                System.out.println("モンスターが現れた");
        }
}
class Game{
        public static void main(String[] args){
                new Monster().message();
        }
}
```

出力結果
モンスターが現れた

この例では message メソッドがスーパークラスとサブクラスで同一シグニチャなのでサブクラスのインスタンスから呼び出される message はサブクラスで定義された message となる。なお、シグニチャが異なる場合はそれぞれ個別のメソッドとして使用可能（オーバーロード)。

オーバーライドされるスーパークラス側のメソッドのアクセス制限と、オーバーライドするサブクラス側のメソッドのアクセス制限は同じかまたはサブクラス側が弱いアクセス制限である必要がある。以下の例はエラーで動かない。

```
class Enemy{
	public void message(){
		System.out.println("敵が現れた");
	}
}
class Monster extends Enemy{
	void message(){                                    //←アクセス制限が強いのでエラー
		System.out.println("モンスターが現れた");
	}
}
```

１６．継承におけるコンストラクタの動作

サブクラスのコンストラクタはその呼び出しの最初にスーパークラスのコンストラクタが呼ばれる。コンストラクタを明示すると以下の通り。

```
class Enemy{
	Enemy(){}
}
class Daemon extends Enemy{
	Daemon(){
		super();
	}
}
```

サブクラスのコンストラクタからスーパークラスのコンストラクタを呼び出すには super を使う。ただし super の呼び出しはコンストラクタの先頭で必ず行わなければならない。またスーパークラスのコンストラクタに引数がある場合は、super は必ずサブクラスに記述が必要になる。

１７．ポリモーフィズム

サブクラスのインスタンスをスーパークラスの変数で扱うことをポリモーフィズムという。複数のサブクラスが共通のスーパークラスに属する異なるクラスならばスーパークラスの変数は多様なオブジェクトに代わりうることに由来する言葉である。

以下はポリモーフィズムの例である。

```
class Enemy{
	void message(){
		System.out.println("敵が現れた");
	}
}
class Monster extends Enemy{
	void message(){
		System.out.println("モンスターが現れた");
	}
}
class Satan extends Enemy{
	void message(){
		System.out.println("サタンが現れた");
	}
}
class Game{
	public static void main(String[] args){
		Enemy e;
		e = new Monster();
		e.message();
		e = new Satan();
		e.message();
	}
}
```

ここで main の変数 e はモンスターのオブジェクトを扱うと同時にサタンのオブジェクトも扱っている。

１８．オブジェクトクラス

明示的には記されていないが、Object クラスはすべてのクラスのスーパークラスとして継承されているということになっている。したがってすべてのクラスインスタンスは Object クラスの変数を使って扱うことができる。

１９．サブクラス内でのスーパークラスメンバの扱い

スーパークラスのメンバはサブクラスで super キーワードにより呼び出すことができる。オーバーライドメソッドを含むサブクラス内でスーパークラスのオーバーライドメソッド

を呼び出すような以下の用例では必須である。

```
class Enemy{
	void message(){
		System.out.println("敵が現れた");
	}
}
class Monster extends Enemy{
	void message(){
		System.out.print("巨大な");
		super.message();			//ここ
	}
}
class Game{
	public static void main(String[] args){
		new Monster().message();
	}
}
```

２０．コンストラクタの明示

コンストラクタの記述は明示しなくとも動作は存在する。記述のないコンストラクタをデフォルトコンストラクタと呼ぶ。引数のないコンストラクタがスーパークラスで動作を記述しているときはサブクラスにコンストラクタの記述がなくてもスーパークラスのコンストラクタが呼び出される。以下。

```
class Enemy{
	Enemy(){
		System.out.println("敵が登場");
	}
}
class Monster extends Enemy{
}
class Game3{
	public static void main(String[] args){
		new Monster();
	}
}
```

出力結果

敵が登場

しかしスーパークラスのコンストラクタが引数つきの場合はサブクラスのコンストラクタを明示し、かつスーパークラスのコンストラクタの呼び出しを明示しなければならない。

このときのスーパークラスのコンストラクタの呼び出しには super を使う。以下。

```
class Enemy{
	Enemy(String s){
		System.out.println(s+"が登場");
	}
}
class Monster extends Enemy{
	Monster(){
		super("モンスター");
	}
}
class Game{
	public static void main(String[] args){
		new Monster();
	}
}
```

出力結果

モンスターが登場

なお、super は必ずコンストラクタの先頭に記述しなければならない。

２１．抽象メソッド、抽象クラス

処理内容のない以下のような表記のメソッドを抽象メソッドといい、抽象メソッドを含むクラスを一般に抽象クラスという。抽象クラスはインスタンスを生成できないので、継承させて使うことになる。

```
abstract class  クラス名{
	abstract  戻り値の型  メソッド名(引数);
}
```

抽象クラスを継承したサブクラスは抽象クラスに定義された抽象メソッドの処理内容を記述しなければならない。

```
abstract class Enemy{
	abstract void message();
}
class Monster extends Enemy{
	void message(){                    //←この行以下の処理がないとエラー
		System.out.println("モンスターが現れた");
```

```
	}
}
```

なお、抽象メソッドを含まないクラスでも abstract キーワードをクラス定義の前につけることで抽象クラスになる。

２２．オブジェクトの属性

オブジェクトが同一かどうかを判断する場合は==を使えばいいが、オブジェクトがどのクラスに属するかを比較したい場合には instanceof を使う。以下の通り。

```
class Enemy{
}
class Evil extends Enemy{
}
class Game13{
	public static void main(String[] args){
		Enemy e1 = new Enemy();
		Enemy e2 = new Evil();
		if(e1==e2) System.out.println("オブジェクトが同一");
		if(e1 instanceof Enemy){
			System.out.println("e1 は Enemy のオブジェクト");
		}
		if(e2 instanceof Enemy){
			System.out.println("e2 は Enemy のオブジェクト");
		}
		if(e1 instanceof Evil){
			System.out.println("e1 は Evil のオブジェクト");
		}
		if(e2 instanceof Evil){
			System.out.println("e2 は Evil のオブジェクト");
		}
	}
}
```

出力結果

e1 は Enemy のオブジェクト

e2 は Enemy のオブジェクト

e2 は Evil のオブジェクト

結果からわかるように継承関係にあるサブクラスは上位であるスーパークラスにも属する

２３．インターフェース

処理内容の記述のないメソッド（抽象メソッド）と定数のみを扱う構造データをインターフェースと呼ぶ。インターフェースの書式は以下。

```
interface インターフェース名{
        型名 定数名;
        戻り値の型 メソッド名(引数の型 引数名);
}
```

インターフェースの書式には明示されないが、フィールドは定数として扱われ(クラスならば final が付いているのと同じ)、メソッドは処理を記述できない(クラスならば abstract がついているのと同じ)。またアクセス制限は public である。

２４．実装

インターフェースはクラスに実装(implements)して使用する。以下の通り。

```
interface インターフェース名{
        …
}
class クラス名 implements インターフェース名{
        …
}
```

用例

```
interface Enemy{
        String s = "敵です";
        void message();
}
class Monster implements Enemy{                    //ここ
        public void message(){
                System.out.println(s);
        }
}
class Game{
        public static void main(String[] args){
                new Monster().message();
```

```
        }
}
```

出力結果
敵です

クラスは複数のクラスを継承（多重継承）ができないが、クラスは複数のインターフェースを実装できる。

class クラス名 implements インターフェース名 1, インターフェース名 2 {
        …
}

インターフェースはインターフェースを継承できる。

```
interface Enemy{
        int hp=128;
        void showProp();
}
interface Mover{
        int speed = 100;
        void run();
}
interface MyEnemy extends Enemy, Mover{          //ここでインターフェースを継承
        int damage = 10;
        void attack();
}
class Boss implements MyEnemy{
        int x, gain, point;
        public void showProp(){
                System.out.println(hp+","+speed+","+damage);
                System.out.println(x+","+gain+","+point);
        }
        public void run(){
                x += 10*speed;
                point +=10;
        }
        public void attack(){
                gain += damage;
                point += hp;
        }
}
class Game{
        public static void main(String[] args){
                Boss b=new Boss();
                b.run();
```

```
                b.attack();
                b.showProp();
        }
}
```

出力結果

128 100 10

10 1000 138

２５．インターフェースによるポリモーフィズム

インターフェースはインスタンスを生成できないが、参照型変数として宣言することはできる。インターフェースを実装したクラスインスタンスは実装元のインターフェースの変数で扱うことができる。以下の通り。

```
interface Enemy{
        String s = "敵です";
        void message();
}
class Monster implements Enemy{
        public void message(){
                System.out.println(s);
        }
}
class Game{
        public static void main(String[] args){
                Enemy e = new Monster();            //ここ
                e.message();
        }
}
```

２６．匿名クラス

クラス名のないクラスを匿名クラス(Anonymous class)という。継承関係となるスーパークラスを使って以下のように記述する。

```
new  スーパークラス名(引数の型  引数名){
        クラスメンバ;
}
```

スーパークラスに相当する部分がインターフェースであってもかまわない。以下の通り。

```
interface Enemy{
	String s = "敵です";
	void message();
}
class Game{
	public static void main(String[] args){
		Enemy e = new Enemy(){
			public void message(){
				System.out.println(s);
			}
		};
		e.message();
	}
}
```

再利用の可能性が小さい、またはクラスの規模が極めて小さい場合は有効な表現である。

２７．内部クラス

クラスの定義の中に別のクラスの定義があるとき、そのクラスを内部クラスと呼ぶ。クラスメンバーを内部クラスで利用できるので便利である反面、クラスの独立性は低くなり、依存度は高まってしまう。簡易にクラス構造が必要な時には便利である。以下の通り。

```
class Enemy{
	String s = "敵";
	class Attacker{                                          //ここから
		String kind = "パンチ";
		void attack(){
			System.out.println(s+"が"+kind+"しました");
		}
	}                                                        //ここまで
	void showProp(){
		new Attacker().attack();
	}
}
class Game8{
	public static void main(String[] args){
		Enemy e = new Enemy();
		e.showProp();
	}
}
```

実行結果

敵がパンチしました

２８．ラップクラスと Auto Boxing

あるクラスの機能を拡張したいとき、引数にクラスのインスタンスを渡し、新たなインスタンスを作るように設計されたクラスをラップクラスという。継承とは異なる機能拡張の方法なので、継承に関わる制約に縛られない。以下の通り。

```
class Wrapping{
	public static void main(String[] args){
		Enemy e= new Enemy();
		new Boss(e).message();
	}
}
class Enemy{
	void message(){
		System.out.println("敵");
	}
}
class Boss{                                    //ラップクラス
	Enemy e;
	Boss(Enemy e){
		this.e = e;
	}
	void message(){
		System.out.println("素");
		e.message();
	}
}
```

また基本データはクラスではないので機能を持たないが、自動的に機能拡張クラスにラップされる。この機能は Auto Boxing と呼ばれる。たとえば Integer クラスは基本データの整数(int)を拡張したクラスで、通常は以下のようにラップする。

```
Integer in = new Integer(123);
```

ところが AutoBoxing では

```
Integer in = 123;
```

と書けてしまう（実際には new Integer(123)と同等ではなく、Integer.valueOf(123)と同じ）。これにより Integer クラスのもつ機能(例えば toString()や hashCode()など)が直接使える。ただし、Integer クラスの持つ機能のほとんどは型変換などが多く、大抵の場合は他の方法で代用でき、この目的だけだとあまり AutoBoxing の恩恵はない。一般には以下の項目にあるコレクションで最も恩恵があるといわれている。なお、基本データ拡張ラッパークラスから基本データへ戻す処理は Auto Unboxing といわれる。以下の通り。

```
Integer in = new Integer(123);
int i = in;          //Auto Unboxing
```

２９．例外処理

文法上の誤りとは異なり、実行時にプログラムの動作上の問題や不正な処理を捕捉して代替処理を行うことを例外処理という。一般には以下のように記す。

```
try{
        問題のある処理;
}catch(例外クラスの型  変数名){
        代替処理;
}finally{
        例外が生じても生じなくても行う処理;
}
```

なお finally は不要なら省略できる。

```
class Game{
        public static void main(String[] args){
                try{
                        int x = 3/0;
                }catch(ArithmeticException ae){
                        System.out.println("0 で割れません");
                        System.out.println(ae.toString());
                }finally{
                        System.out.println("例外処理を通過");
                }
        }
}
```

実行結果
0 で割れません
java.lang.ArithmeticException: / by zero
例外処理を通過

３０．例外クラスと例外の発生

例外にはその種類によって多くのクラスが用意されている。例えば算術計算の誤りで発生する ArithmeticException や配列要素の数を超えてインデックスを指定したときに発生する ArrayIndexOutOfBoundsException など様々であり、詳細は API リファレンスを参照してほしい。なお例外はそのすべてが Exception クラスを継承しており、例外を自身でデザインしたり、例外を意図的に発生させることもできる。以下の通り。

```
class MyException extends Exception{
	MyException(){
		System.out.println("俺は例外野郎");
	}
}
class Game{
	public static void main(String[] args){
		try{
			new MyException();			//ここで例外を発生
		}catch(Exception me){				//Exception で捕捉
			System.out.println(ae.toString());
		}
	}
}
```

例外を発生させるとそこで処理が中断するので、ここでは try～catch で挟み例外を捕捉しているが、この例は明示的に例外を投げていないので MyException での捕捉ができない。以下を参照のこと。

３１．throw と throwable

一般に意図的な例外を発生させるには、Exception クラスのスーパークラスでもある Throwable クラスを継承したクラスインスタンスを throw する（もちろん Exception でもよい)。以下の通り。

```
class MyException extends Throwable{
	MyException(){
		System.out.println("俺は例外野郎！");
	}
}
class Game{
	public static void main(String[] args){
		try{
			throw new MyException();
		}catch(MyException me){
			System.out.println(me.toString());
		}
	}
}
```

３１．throws

例外が発生するとそこで処理がとまるが、例外での処理停止位置を発生したメソッド内ではなく呼び出し側で停止するように仕向けるのが throws である。以下の通り。

```
class Enemy{
	void proc() throws ArithmeticException{
			int x = 3/0;			//ここで例外が発生
	}
}
class Game{
	public static void main(String[] args){
		try{
			new Enemy().proc();	//ここで例外を捕捉
		}catch(ArithmeticException ae){
			System.out.println(ae.toString());
		}
	}
}
```

こうすれば例外をメソッド内ではなくメソッドの呼び出し側で捕捉できる。

３２．アノテーション

ある処理やデータに対して関連する情報（メタデータ）を付加する方法を提供するのがアノテーションである。単なる付加情報なので具体的な機能ではないが、エディタの表示や XML を使った処理の手続き、例外の発生などでこれが機能的に役目を果たす場合がある。

一般には以下のようなものがよく使用される

@Deprecated
@Override
@Surpresswarning
@Target

たとえば以下の例では Enemy を継承した Boss クラスで attack メソッドをオーバーライドするつもりで記述しているが、スペルミスで attach になっておりオーバーライドされていない。もしアノテーションがなければ単に Enemy クラスのメンバとしての認識でしかないので実行結果は Enemy クラスのメンバである attack メソッドが実行され、最悪の場合気がつかずに終わってしまう。アノテーションを付けるとオーバーライドがされていない旨のエラーメッセージを確認できる。

```
class Enemy{
	void attack(){
		System.out.println("攻撃！");
	}
}
class Boss extends Enemy{
	@Override
	void attach(){
		System.out.println("ボス攻撃！");
	}
}
class Annotation01{
	public static void main(String[] args){
		Enemy e = new Boss();
		e.attack();
	}
}
```

以下はエラーメッセージ：

```
Annotation01.java:7: エラー: メソッドはスーパータイプのメソッドをオーバーライドまたは実装しません
	@Override
	^
エラー1 個
```

アノテーションの詳細は JLS を参照のこと。

３２．コレクション

配列とは異なるデータの集合体のこと。List や Vector などデータの格納方法や使い方などを目的に応じて使用する。コレクションの種類は多く、コレクション・フレームワークと呼ぶこともある。スーパーインターフェースは Iterable で、すべてのコレクションはイテレータ(反復子)として用いることができる（foreach 文でも扱える）。以下は Vector の用例。

```
import java.util.*;
class VecUsage{
	public static void main(String[] args){
		Vector v = new Vector();
		v.add(1);			//追加するだけ（Autoboxing の適用例）
		v.addElement("HELLO"); //追加してサイズを増やす
		v.insertElementAt(5,1);	//5 を 1 番目に追加
		System.out.println(v);
	}
}
```

出力結果：[1, 5, HELLO]

他に ArrayList, Stack, Queue, Map, Set などがある。コレクションはジェネリクス(次項を参照)を併用することが推奨されており、上記のような使用法は非推奨の警告がビルド時に出力される。

３３．ジェネリクス

コレクションで異なるデータを扱うと、コレクションの種類を特定できず、コレクションとしての意味合いが曖昧になってしまう。以下の表記を記すと、コレクションで扱うデータの種類を限定でき、使用時の余分なキャストが不要になる。

```
import java.util.*;
class GenUsage{
	public static void main(String[] args){
		Vector<Integer> v = new Vector<Integer>();
		v.add(1);			//追加するだけ（Autoboxing 適用）
		v.addElement(0);		//追加してサイズを増やす
		v.insertElementAt(5,1);	//5 を 1 番目に追加
		System.out.println(v);
	}
}
```

ジェネリクスは Java 5（jdk1.5 以降）に導入された機能である。

３４．ラムダ式

ラムダ式とは匿名メソッドのことで、引数と処理内容（戻り値）を以下のフォーマットで記述する。

(クラスのキャスト)()->{処理;}

例えば匿名クラスを使用する際、オーバーライドメソッドはメソッド名を明記する必要があるがラムダ式ならメソッド名を省略できる。以下のインターフェースの場合は次のようになる。

```
interface Enemy{
	String s = "敵です";
	void message();
}
```

ラムダ式を使用しない場合：

```
class Game{
	public static void main(String[] args){
		Enemy e = new Enemy(){
			public void message(){
				System.out.println(s);
			}
		};
		e.message();
	}
}
```

ラムダ式を使用した場合：

```
class Game{
	public static void main(String[] args){
		Enemy e = ()->System.out.println("素"+Enemy.s);    //ラムダ式
		e.message();
	}
}
```

ラムダ式は機能クラスまたは機能インターフェースという単一メソッドのみのものに使える（単一だからメソッド名を省略できる）。継承・実装元が複数のメソッドを含む場合は当然使用できない。

３４－１．引数を含む例

```
interface Enemy{
	String s = "敵です";
	void message(String s);
}
class Game{
	public static void main(String[] args){
		Enemy e = (String s)->System.out.println(s+Enemy.s);//ラムダ式
		e.message("素");
	}
}
```

この例ではメソッドの引数が１つだけなので以下のように省略して表記できる。

```
Enemy e = s->System.out.println(s+Enemy.s);
```

引数が２つ以上ならカッコは省略できないが、データ型は省略できる。例えば（int a, int b）なら（a,b）とも書ける。

３４－２．戻り値を含む例

```
interface Enemy{
	String s = "敵です";
	String message();
}
class Game{
	public static void main(String[] args){
		Enemy e = ()->{
			return Enemy.s;
		};
		System.out.println("素"+e.message());
	}
}
```

３４－３．スコープ

ラムダ式ではインターフェース内のスコープが無効になる。そのためインターフェース内の定数を使う場合はインターフェース名を冠する必要がある。上記の例でインターフェース内の定数に対して Enemy.s と表記しているのはそのためである。以下の２例の出力結果を比較してほしい。インターフェースは両者とも３２章の最初に挙げたものを使用している。

ラムダ式を使わないとき：

```
class Game{
```

```
	public static void main(String[] args){
		String s = "素";
		Enemy e = new Enemy(){
			public void message(){
				System.out.println(s+Enemy.s);
			}
		};
		e.message();
	}
}
```
出力結果：敵です。敵です。

ラムダ式を適用した例：
```
class Game{
	public static void main(String[] args){
		String s = "素";
		Enemy e = ()->System.out.println(s+Enemy.s);
		e.message();
	}
}
```
出力結果：素敵です。

基本的にラムダ式内で扱う外部変数は定数扱い(final)となる。たとえ変数で宣言してもラムダ式内部で一旦使用した変数は値を変えることができなくなる。

またラムダ式の引数で使用する変数名と同じ名前の変数が外部変数で宣言されていれば、同一スコープとみなされエラーになる。

３４－４．キャスト

クラス変数の型を Object にした場合などはキャストが必要になる。

```
class Lambda004{
	public static void main(String[] args){
		Object e = (Enemy)()->System.out.println("素"+Enemy.s);
		((Enemy)e).message();
	}
}
```

また複数のインターフェースを実装したい場合はクロス・キャストが可能である。書式は以下。

（インターフェース１ ＆ インターフェース２）変数

以降で示すシリアライズなどで使用する。

なおラムダ式は Java8（JDK1.8 以降）に導入された概念である。

３５．シリアライズ

直列化ともいう。オブジェクトを丸ごとデータとしてファイルやネットワークリソースとして活用する方法を提供する。まず、以下のコードを実行してオブジェクトをファイルに書き込んでみる（可読性を考慮して例外を投げてしまっています)。

```
import java.io.*;
class Enemy implements Serializable{
	String s = "敵です";
	void message(){
		System.out.println(s);
	}
}
class Serial000w{
	public static void main(String[] args) throws IOException{
		ObjectOutputStream oos;
		oos= new ObjectOutputStream(
				new FileOutputStream("enemy.obj"));
		oos.writeObject(new Enemy());
		oos.close();
	}
}
```

次にこのコードで書き出された enemy.obj を読み込んで Enemy クラスのメンバを呼び出してみる。

```
import java.io.*;
class Serial000r{
  public static void main(String[] args) throws IOException, ClassNotFoundException{
	ObjectInputStream ois;
	Object o;
	ois= new ObjectInputStream(new FileInputStream("enemy.obj"));
	o=ois.readObject();
	ois.close();
	((Enemy)o).message();
  }
}
```

この例では２つのコードを同一パッケージにして実行しているので２つ目のコードでEnemy クラスが使えているが、この部分はインターフェースをあらかじめ用意しておく必要がある。

このようにオブジェクトをそのままファイルなどのリソースとして利用することでメモリに一時的に確保される揮発性データから永続的に使用可能なデータとして活用可能になる。これをさらにフォーマットを含め厳密化したのが JSON であり、Web アプリケーションでの活用が広く行われている。

３６．スレッドと同期

スレッドとはプロセス内で処理する動作区分で、複数の処理を同時実行するとき、これをマルチスレッドと呼ぶ。これにより、例えばネットワークからのデータ受信の処理をしながら同時に送信処理などを行うことができる。
スレッドはThread クラスを継承するかRunnableインターフェースを実装したクラスを使う。以下。

```
//Thread クラスを継承した例
import java.lang.Thread;
class ThreadTest extends Thread{
	public void run(){
		System.out.println("hello");
	}
}
class Thread01{
	public static void main(String[] args){
		Thread t = new ThreadTest();
		t.start();
	}
}
//Runnable インターフェースを実装した例
import java.lang.Runnable;
import java.lang.Thread;
class ThreadTest implements Runnable{
	public void run(){
		System.out.println("hello");
	}
}
class Thread02{
	public static void main(String[] args){
		Thread t = new Thread(new ThreadTest());
		t.start();
	}
}
```

スレッドは実行区分を run メソッド内の動作として実行する。start メソッドでスレッドの実行を開始するが、実行の順番は必ずしも start の呼び出し順にはならない。上記の例も、プロセスがスレッド実行の前に終了した場合には出力結果は表示されない。

次の例は

スレッドプログラミングにおいて、同時実行されるスレッド同士での競合を避けるために同期を行うことがある。Synchronized キーワードを使い、以下のように記述する。