演算結果

●型と精度

・整数リテラルは int 型であり、32 ビットである

・小数点数リテラルは double 型であり、64 ビットである

・精度が落ちる方向に代入するとコンパイルエラーとなるがキャストで回避できる

・boolean 型にキャストは適用できない

・byte 型は演算のときに int 型に拡張される

・String と int の連結は String になる

●ビット操作

・>> は算術ビットシフトで、>>> は論理ビットシフトである

・| はビットごとの論理和、& はビッドごとの論理積、^ はビットごとの排他的論理和である

●演算子

・&& 演算子は左オペランドが false の時、右オペランドを評価しない

・|| 演算子は左オペランドが true の時、右オペランドを評価しない

・条件式 ? 値１ : 値２　は、３項演算子で条件式が true の時は値１をとり、false の時は値２をとる

・--a プレデクリメント　　a-- ポストデクリメント

　++a プレインクリメント　a++ ポストインクリメント

●配列

・１次元配列の宣言　int[] A = new int[10];

・２次元配列の宣言　String[][] B = new String[10][3];

・A.length は配列 A の要素数を、B.length は配列 B の１次元目の要素数を表す

●enum 型

・enum で宣言した要素以外をその型に代入することはできない

・switch 文の case に enum 型を記述するときは、要素名だけを記述する

●制御文

・if 文の条件判断式は boolean 型にならなければコンパイルエラーとなる

　例）　if(i = 0) {

・boolean flag = false;

　if(flag = true) は flag に true が代入されて true と評価され、コンパイルエラーとならない

・拡張 for 文はすべての要素が処理される

```
for( 型　変数名 : 配列名 ) {
        ・・・変数名・・・;
}
```

　配列名の型を変更することはできない

オブジェクト指向

●インスタンス化

・インスタンスメソッドは、インスタンス化しないで呼び出すとコンパイルエラーとなる
・インスタンス変数はオブジェクトごとに領域が割り当てられる
・メンバ変数を初期化しないとき、それぞれの型によるデフォルト値が設定されるが、
ローカル変数を初期化しないときはコンパイルエラーとなる
・static 変数はオブジェクトを複数生成しても領域は共通となる
クラス変数とも呼ばれる
・private 指定したメンバ変数は自分のクラス内からしか参照できない
・スタティックメソッドからインスタンス変数は参照できない

```
class Ex {
    int i = 1;
    public static void main(String[] args) {
        System.out.println(i);
    }
}
```

●パッケージ

・package 文は import 文の前に記述しなければならない
・class に public がないと package 外からアクセスすることはできない

●オブジェクトの性質

・== はオブジェクトが同じかどうかをチェックする演算子であり、オブジェクトの内容をチェックする時は equals()メソッドを用いる
例） if(str.equals("ABC")) {

●総称型（ジェネリクス）

・クラス名の引数に型を指定することができる

```
class Ex<T> {
    private T val;
    void setValue(T val) {
        this.val = val;
    }
    T getValue() {
        return val;
    }
}
```

●ラッパークラス

・基本型をオブジェクト化したラッパークラスがある

```
Integer two = new Integer(2);
```

・ ラッパークラスには文字列や数値を互いに変換するメソッドが用意されている

●オーバーロード（多重定義）

・戻り値と引数は自由に定義できる

```
class Ex {
    int i, j;
    String s;
    void methodA() { }
    int methodA(int i) { return 1; }
    double methodA(int i, int j) { return 1.0; }
    String methodA(String s) { return ""; }
}
```

コレクション

●ArrayList

・要素の型は指定しなくてもよいが、指定することが推奨されている。
・宣言と異なる要素を追加するとコンパイルエラーとなる。
・配列と異なり、要素の追加と削除ができる。

```
import java.util.*;

class Ex {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<String>();

        list.add("Hello");
        list.add("Hello");
        list.add("Hello");
        for(String str : list) {
            System.out.println(str);
        }
    }
}
```

・要素の型を異なって宣言するとコンパイルエラーとなる。

```
import java.util.*;

class Ex2 {
    public static void main(String[] args) {
        ArrayList<Object> list = new ArrayList<String>();

        list.add("Hello");
        list.add("Hello");
        list.add("Hello");
        for(String str : list) {
            System.out.println(str);
        }
    }
}
```

・オブジェクトの時はキャストが必要となる。

```
import java.util.*;

class A { }
class Ex3 {
    public static void main(String[] args) {
        ArrayList list = new ArrayList();

        list.add(new A());
        A a = list.get(0);
    }
}
```

●HashSet

・同じ要素は追加しない。

```
import java.util.*;

class Ex {
    public static void main(String[] args) {
        HashSet<String> hs = new HashSet<String>();

        hs.add("Hello");
        hs.add("Hello");
        hs.add("Hello");
        System.out.println(hs.size());
    }
}
```

●HashMap

・キーと要素の２つを指定する。

・数値データなどもラッパークラスとして指定する。

```
import java.util.*;

class Ex {
    public static void main(String[] args) {
        HashMap<Integer, String> hm = new HashMap<Integer, String>();

        hm.put(1, "Tokyo");
        hm.put(2, "Shinagawa");
        hm.put(3, "Gotanda");
        for(int num : hm.keySet())
            System.out.println("キー" + num + "は" + hm.get(num));
    }
}
```

継承

●オーバーライド（再定義）

・継承して拡張したクラスをサブクラス、もとのクラスをスーパークラスと呼ぶ

・継承の階層に制限はない

・多重継承は出来ない

```
class A { }
class B { }
class Ex extends A, B { }
```

・同じメソッド呼び出しでも、受け取るオブジェクトによって振舞いが異なることをポリモフィズムという

・どのクラスも java.lang.Object クラスを継承する

・スーパークラスのオブジェクトは super キーワードを使う

・オブジェクト内でそのオブジェクトは this キーワードを使う

・クラスの実装関係は instanceof 演算子でチェックできる

例）

```
class A { }
class B extends A { }
class Ex {
    public static void main(String[] args) {
        A a = new A();
        B b = new B();
        System.out.println(a instanceof B);
        System.out.println(b instanceof A);
    }
}
```

・アクセス特権を狭くするとコンパイルエラーとなる

・methodC()はオーバーライドではなく新たなメソッド定義とみなされる

```
class A {
    void methodA() { }
    public void methodB() { }
    private void methodC() { }
}
class Ex extends A {
    public void methodA() { }
    void methodB() { }
    public void methodC() { }
}
```

・final 指定のメソッドはオーバーライドできない

```
class A {
    final void methodA() { }
}
class Ex extends A {
    void methodA() { }
}
```

・オーバーライドするときの戻り値と引数は同じでなければコンパイルエラーとなる

```
class A {
    void methodA() { }
}
class Ex extends A {
    void methodA() { }
}
```

・メンバ変数もオーバーライドでき、型を変えられる
```
class A { int i; }
class Ex extends A { double i; }
```
・private メソッドはサブクラスからでも呼び出せない
```
class A {
    private void methodA() { }
}
class Ex extends A {
    public static void main(String[] args) {
        A objA = new A();
        objA.methodA();
    }
}
```
・サブクラスで必ずオーバーライドさせたいメソッドには、abstract 修飾子を付加する
・サブクラスのオブジェクトはスーパークラス型の参照変数に代入できる
　ただし、実行するとサブクラスのメソッドが呼び出される
```
class A {
    void methodA() { }
}
class Ex extends A {
    public static void main(String[] args) {
        Ex objE = new Ex();
        A objA = new Ex();
    }
}
```
・スーパークラスのオブジェクトはキャストすればサブクラス型の参照変数に代入できる
ただし、実行時に ClassCastException の例外が発生する
```
class A {
    void methodA() { }
}
class Ex extends A {
    public static void main(String[] args) {
        A objA = new A();
        Ex objEx = (Ex)new A();
    }
}
```
●コンストラクタ
・コンストラクタはオーバーライドできない
・コンストラクタは継承されない
・引数つきのコンストラクタを定義するとデフォルトコンストラクタはなくなる
```
class A {
    int i;
    A(int i) {
        this.i = i;
    }
}
class Ex extends A {
    public static void main(String[] args) {
        A objA = new A();
    }
}
```

・サブクラスでコンストラクタを宣言しないと、コンパイラによってデフォルトコンストラクタが挿入され、super()が暗黙的に呼び出される
・コンストラクタはスーパークラスから実行される

```
class A {
    A() {
        System.out.println("class A");
    }
}
class B extends A {
    B() {
        System.out.println("class B");
    }
}
class Ex {
    public static void main(String[] args) {
        B obj = new B();
    }
}
```

・スーパークラスのコンストラクタは先頭で呼ぶ

```
class A {
    int i;
    A(int i) {
        this.i = i;
    }
}
class B extends A {
    B() {
        super(1);
        System.out.println("class B");
    }
}
class Ex {
    public static void main(String[] args) {
        B obj = new B();
    }
}
```

インタフェース

●修飾子

・インタフェース内の定数は、自動的に final, public, static 修飾子が付加される

・インタフェース内のメソッドは、自動的に public 修飾子と abstract 修飾子が付加される

●インタフェースの実装

・メソッドをすべてオーバーライドしなければコンパイルエラーとなる

・すべてをオーバーライドしないとき、abstract クラスとすればコンパイルエラーではなくなる

・実装したメソッドに public 修飾子がないとコンパイルエラーとなる

```
interface I {
    void methodA();
    void methodB();
    void methodC();
}
abstract class Ex implements I {
    public void methodA() { }
    public void methodB() { }
}
```

●インタフェースの継承

```
interface I {
    void methodA();
    void methodB();
}
interface Ex extends I {
    void methodC();
}
```

例外

●catch ブロック

・すべての例外は catch ブロックでキャッチできる
・try ブロックに対して catch ブロックか finally ブロックのどちらかがあればよい
・catch ブロックは複数記述できるが、finally ブロックの記述はひとつに限られる
・catch ブロックに return 文があっても finally ブロックは実行される
・例外が一度キャッチされるとその後に catch ブロックがあってもキャッチされない

●例外メッセージ

・getMessage()メソッドで例外のメッセージを取得できる

例）
```
} catch(Exception e) {
    System.out.println(e.getMessage());
}
```

・printStackTrace()メソッドで例外発生の過程を追跡できる

例）
```
} catch(Exception e) {
    e.printStackTrace();
}
```

●例外のスロー

・メソッド全体で例外を投げる時は、メソッド名の後に throws と例外クラス名を記述する

例）
```
public void methodA() throws IOException {
```

・throws 句には , で並べて複数の例外クラスを記述できる
・メソッド内で例外を投げる時は、例外オブジェクトを throw で投げる

例）
```
throw new IOException();
```

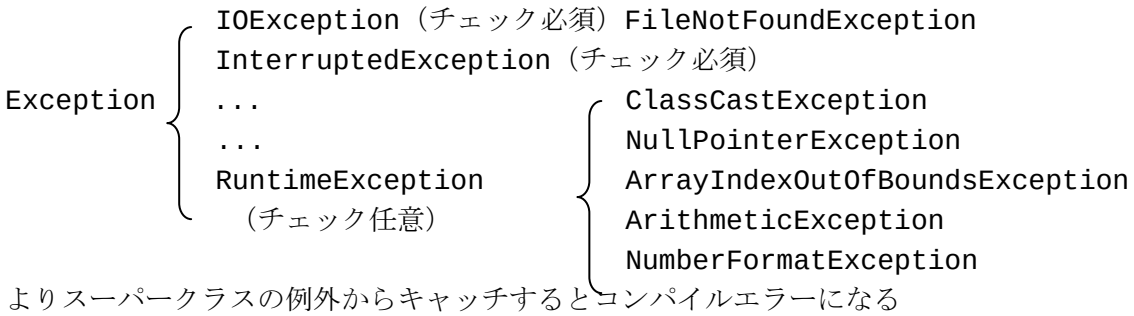
・メソッドで例外を投げた時、呼び出し側では try-catch で処理するか、さらに例外を投げる
・メインメソッド全体で例外を投げることも許される

例）
```
public static void main(String[] args) throws IOException {
```

●例外の階層

・例外クラスを継承して独自の例外クラスを作ることができる
・Exception クラスのサブクラスのうち、RuntimeException クラスとそのサブクラス以外の例外を発生する可能性のあるメソッドで、例外処理を記述しないとコンパイルエラーになる

例外のクラス階層



・よりスーパークラスの例外からキャッチするとコンパイルエラーになる

スレッド

●OS 依存性

・Java は OS がスレッドに対応していなくても、スレッド処理ができる

●Thread クラスの継承によるスレッドの作成

```
public class Ex extends Thread {
    public void run() {
        for(int i=0; i<10; i++) {
            System.out.println(Thread.currentThread().getName() + "実行中" + i);
        }
    }
    public static void main(String[] args) {
        Ex th = new Ex();
        th.start();
        for(int i=0; i<10; i++) {
            System.out.println(Thread.currentThread().getName() + "実行中" + i);
        }
    }
}
```

・メインメソッドの実行経過もひとつのスレッドである

・新たにスレッドを作成するには Thread クラスを継承して run()メソッドをオーバーライドする

・スレッド名は Thread.currentThread().getName()で取得できる

・メインのスレッド名は main である

・スレッドの開始は run()メソッドではなく start()メソッドを実行する

・run()メソッドを実行するとメインのスレッドの通常メソッドとして実行される

●Runnable インタフェースの実装によるスレッドの作成

```
public class Ex implements Runnable {
    public void run() {
        for(int i=0; i<10; i++) {
            System.out.println(Thread.currentThread().getName() + "実行中" + i);
        }
    }
    public static void main(String[] args) {
        Ex thr = new Ex();
        Thread th = new Thread(thr);
        th.start();
        for(int i=0; i<10; i++) {
            System.out.println(Thread.currentThread().getName() + "実行中" + i);
        }
    }
}
```

・Runnable インタフェースの run()メソッドを実装して新たにスレッドを作成できる

・インスタンスを作成後そのインスタンス参照変数を Thread コンストラクタの引数とする

●run メソッドのオーバーライドでスレッド処理を記述する

```
public class Ex extends Thread {
    public void run(int x) {
        for(int i=0; i<10; i++) {
            System.out.println(Thread.currentThread().getName() + "実行中" + i);
        }
    }
    public static void main(String[] args) {
        Ex th = new Ex();
        th.start();
        for(int i=0; i<10; i++) {
            System.out.println(Thread.currentThread().getName() + "実行中" + i);
        }
    }
}
```

・シグネチャの異なる run()メソッドはオーバーライドとはならず、スレッド開始はその効果がない

●スレッドを２つ作成する（失敗例）

```
public class Ex extends Thread {
    public void run() {
        for(int i=0; i<10; i++) {
            System.out.println(Thread.currentThread().getName() + "実行中" + i);
        }
    }
    public static void main(String[] args) {
        Ex th = new Ex();
        for(int i=0; i<2; i++) {
            th.start();
        }
        for(int i=0; i<10; i++) {
            System.out.println(Thread.currentThread().getName() + "実行中" + i);
        }
    }
}
```

・ひとつのスレッドを何回も起動できない

・コンパイルできるが、実行時に IllegalThreadStateException 例外が発生する

●スレッドを２つ作成する（成功例）

```
public class Ex extends Thread {
    public void run() {
        for(int i=0; i<10; i++) {
            System.out.println(Thread.currentThread().getName() + "実行中" + i);
        }
    }
    public static void main(String[] args) {
        for(int i=0; i<2; i++) {
            Ex th = new Ex();
            th.start();
        }
        for(int i=0; i<10; i++) {
            System.out.println(Thread.currentThread().getName() + "実行中" + i);
        }
    }
}
```

・オブジェクトを２つ作ればエラーなく実行できる

・上のプログラム例では、メインのスレッドと合わせて３つのスレッドが実行中となる

●スレッドの休止

```
public class Ex extends Thread {
    public void run() {
        try {
            for(int i=0; i<10; i++) {
                System.out.println(Thread.currentThread().getName() + "実行中" + i);
                Thread.sleep(1000);
            }
        } catch(InterruptedException e) { }
    }
    public static void main(String[] args) {
        Ex th = new Ex();
        th.start();
        for(int i=0; i<10; i++) {
            System.out.println(Thread.currentThread().getName() + "実行中" + i);
        }
    }
}
```

・Thread.sleep(...); でスレッドを休止させることができる

・sleep()メソッドの引数の数値の単位はミリ秒である

・sleep()メソッドは例外をキャッチしないとコンパイルエラーとなる

●スレッド終了の待機

```
public class Ex extends Thread {
    public void run() {
        try {
            for(int i=0; i<10; i++) {
                System.out.println(Thread.currentThread().getName() + "実行中" + i);
                Thread.sleep(1000);
            }
        } catch(InterruptedException e) { }
    }
    public static void main(String[] args) {
        Ex th = new Ex();
        try {
            th.start();
            th.join();
        } catch(InterruptedException e) { }
        for(int i=0; i<10; i++) {
            System.out.println(Thread.currentThread().getName() + "実行中" + i);
        }
    }
}
```

・join()メソッドの実行でそのスレッドの終了を待機させることができる
・join()メソッドは例外をキャッチしないとコンパイルエラーとなる

●スレッドの同期制御

・複数のスレッドの同期制御を行うには synchronized 指定されたメソッドあるいはブロック内で wait()、notify()、notifyAll()のメソッドを利用する
・wait()、notify()、notifyAll()のメソッドを synchronized 指定と関係なく利用してもコンパイルエラーとはならないが実行時に例外が発生する
・wait()、notify()、notifyAll()のメソッドは Object クラスのメソッドである
・同期制御を利用して排他制御を行うことができる

入出力

●キーボード入力

```
import java.io.*;
class Ex {
    public static void main(String[] args) {
        String line;
        try {
            BufferedReader br = new BufferedReader(
                                        new InputStreamReader(System.in));
            line = br.readLine();
            System.out.println(line);
        } catch(IOException e) {
            e.printStackTrace();
        }
    }
}
```

・BufferedReader クラスの readLine()メソッドはチェック例外の IOException をスローする

・try-catch の代わりにメインメソッド全体で例外をスローしてもよい

例） `public static void main(String[] args) throws IOException {`

●ファイル入出力

```
import java.io.*;
class Ex {
    public static void main(String[] args) throws IOException {
        String line;
        BufferedReader br = new BufferedReader(
                                    new FileReader("FileInOut.java"));
        BufferedWriter bw = new BufferedWriter(
                                    new FileWriter("ModifyFileInOut.java"));
        while((line = br.readLine()) != null) {
            bw.write("==== " + line);
            bw.newLine();
        }
        br.close();
        bw.close();
    }
}
```

アプレット

●applet タグ

・code の部分にはプログラムをコンパイルして作成される　クラス名.class を指定する

例）　<applet code=”Abc.class” width=”300” height=”300”> ...
　　public class Ex extends Applet ...

●param タグ

・プログラムにパラメータを渡す時は param タグを用いてパラメータ名とその値を指定する

例）　<param name=”FontType” value=”Serif”>

・プログラム中でのその値を取得するには getParameter(“パラメータ名”)とする。

例）　String fontType = getParameter(“FontType”);

●import 文

・applet パッケージ（Applet クラス）

例）　import java.applet.*; （import java.applet.Applet;）

・awt パッケージ

例）　import java.awt.*;

・event パッケージ

例）　import java.awt.event.*;

●プログラムの構造

```
import 文;
public class クラス名 extends Applet implements インタフェース名, ... {
    フィールド変数の宣言：複数のメソッドで使用する変数はここで宣言する
    public void init() {
      アプレットの初期設定：画面レイアウトの構成など
    }
    public void start() {
      アプレットの開始動作：スレッドの開始など
    }
    public void paint(Graphics g) {
      画面への描画
    }
    public void actionPerformed(ActionEvent e) {
      アクションイベント発生時の処理
    }
    public void mouseClicked(MouseEvent e) {
      マウスイベント発生時の処理
    }
}
```

●実装するインタフェース

・スレッドがある時　　Runnable
・アクションイベントがある時　ActionListener
・マウスイベントがある時　　MouseListener

●アプレットの初期設定　init()メソッドに記述

・画像の取得

init()メソッド外でフィールド変数を宣言する　Image img;

init()メソッド内で　img = getImage(getDocumentBase(), “ファイル名.gif”);

・レイアウト

自由配置　setLayout(null);

位置はsetBoundsメソッドで指定する

例）　new Label(“ラベル”).setBounds(x0, y0, width, height);

フローレイアウト　setLayout(new FlowLayout());

自動的に流し込まれてセンタリングされるので位置は指定しない

ボーダーレイアウト　setLayout(new BorderLayout());

位置は以下から選ぶ

BoderLayout.NORTH　または　“North”

BoderLayout.SOUTH　または　“South”

BoderLayout.WEST　または　“West”

BoderLayout.EAST　または　“East”

BoderLayout.CENTER　または　“Center”

例）　add(new Label(“ラベル”), BoderLayout.NORTH);

・リスナーの登録

アクションリスナー　new Button(“ボタン”).addActionListener(this);

マウスリスナー　this.addMouseListener(this);

または　addMouseListener(this);

●スレッドがある時

・アプレットの開始動作でスレッドをスタートさせる（init()メソッドに記述することもできる）

例）
```
public void start() {
    Thread th = new Thread(this);
    th.start();
}
```

・スレッドの動作はrun()メソッドを加えてそこに記述する

一定の時間間隔で動作させる時はスレッドのsleep()メソッドを使う（数値の単位はミリ秒）

例外をキャッチする（InterruptedExceptionクラスまたはExceptionクラス）

グラフィックスの描画はrepaint()を使う

例）
```
public void run() {
    while(true) {
        repaint();
        try {
            Thread.sleep(1000);
        } catch(InterruptedException e) { }
        描画条件の更新
    }
}
```

●アクションイベント発生時の処理
・どのボタンがクリックされたかは getActionCommand()メソッドを使う
・グラフィックスの描画は repaint()を使う

例）
```
public void actionPerformed(ActionEvent e) {
    if(e.getActionCommand().equals(“ボタン１”)) {
        ボタン１がクリックされた時の処理
    } else if(e.getActionCommand().equals(“ボタン２”)) {
        ボタン２がクリックされた時の処理
    } else {
        それら以外がクリックされた時の処理
    }
    repaint();
}
```

●マウスイベント発生時の処理
・マウスイベントの処理メソッド（５つ）はすべて実装する
・実際に使用するイベント処理だけを記述する
・グラフィックスの描画は repaint()を使う

例）
```
public void mouseClicked(MouseEvent e) {
    マウスがクリックされた時の処理
    repaint();
}
public void mouseEntered(MouseEvent e) { }
public void mouseExited(MouseEvent e) { }
public void mousePressed(MouseEvent e) { }
public void mouseReleased(MouseEvent e) { }
```