Java Programming I

Review

Exercises 1 to 6
Chapters 1 to 8

Contents

- Exercises 1 - 6: Common errors

Exercise 3, Problem 3: Classifying Toys

◆ Wrong solution

```
--result—
Value x = 9.0E-4
Value y = 0.0016
Value z = 0.0025000000000000005
It is not a right-angled triangle
```

```
public class TestPower2 {
  public static void main(String[] args) {
    double x,y,z;
    x = 0.03;
    y = 0.04;
    z = 0.05;
    Hantei(x,y,z);
  }
  public static void Hantei(double hen1, double hen2, double hen3)  {
      double x = hen1 * hen1;
      double y = hen2 * hen2;
      double z = hen3 * hen3;
      System.out.println("Value x = " + x);
      System.out.println("Value y = " + y);
      System.out.println("Value z = " + z);
      if ( (x+y) == z ) System.out.println("It is a right-angled triangle");
      else System.out.println("It is not a right-angled triangle");
  }
}
```

Exercise 3, Problem 3: Classifying Toys

- The Equal operation (==) cannot be used with variables of real or double type. Their values are not presented as exact numbers. Just compare:

  type int: 1 + 1 + 1 = 3 (this is always the true because of the integer value: They are exact values in the memory of a computer)

  numbers like these: 1/3 + 1/3 + 1/3 should be 1

  how about the numerical system with base 10 ?
  0.33(3) + 0.33(3) + 0.33(3) is it equal to 1 ?

  We cannot present these numbers in the memory of a computer as they are... As a result, the sum will be less than 1.

Exercise 3, Problem 3: Classifying Toys

- To solve this problem, you should take into account an error.
- Do not consider the *absolute error*.
- You have to work with a *relative error* to make it independent from the input data set.
- You should recall theorems on triangles you have studied at school.

Exercise 3, Problem 3: Classifying Toys

- Wrong solution

```
public class TestPower2 {
  static float EPS = 1E-5f;
 public static void main(String[] args) {
   double x,y,z;
   x = 0.03;
   y = 0.04;
   z = 0.05;
   Hantei(x,y,z);
 }
 public static void Hantei(double hen1, double hen2, double hen3)  {
     double x = hen1 * hen1;
     double y = hen2 * hen2;
     double z = hen3 * hen3;
     System.out.println("Value x = " + x);
     System.out.println("Value y = " + y);
     System.out.println("Value z = " + z);
     if ( Math.abs((x+y) – z) < EPS ) System.out.println("It is a right-angled triangle");
     else System.out.println("It is not a right-angled triangle");
 }
}
```

--result—

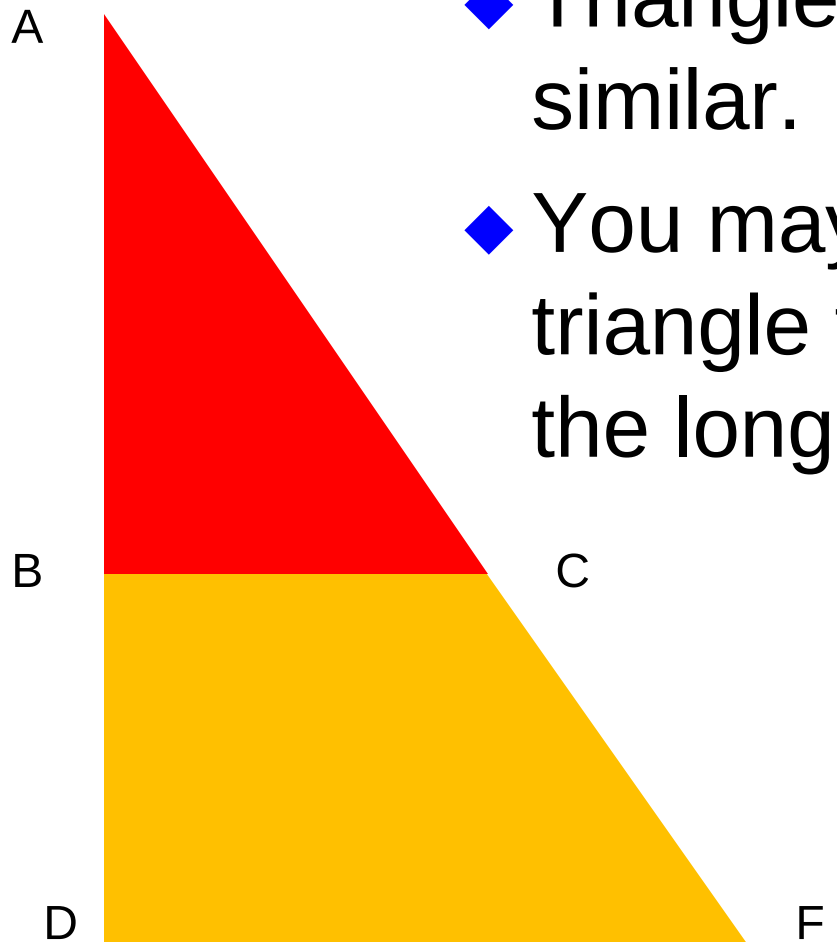
You may play with the set : 5 12 13 multiply every number by 0.1 several times and analyze the results of the calculations of this solution: It will be providing you with correct answers on some values and incorrect ones on the others.

Exercise 3, Problem 3: Classifying Toys

- In the solution on the previous slide:
  - EPS is an absolute error!
    - You should find the relative error
  - if (Math.abs((x+y) – z) < EPS)
    - Minus operation is applied to the values which are very close to each other: The error of calculations is increasing drastically!
      - You should avoid the usage of the minus operation with values which are very close to each other.

Exercise 3, Problem 3: Classifying Toys

- Use your knowledge from school
  - Triangles ABC and ADF are similar.
  - You may convert every triangle to the triangle with the longest side equal to 1.



Exercise 3, Problem 3: Classifying Toys

- The core of your solution may look like this:

```
// Relative error
static float EPS = 1E-5f;

static int testFigure(float a, float b, float c) {
//   1 - right-angled,
    a = a / c;
    b = b / c;
    c = 1;
    if (((a*a + b*b) >(1- EPS))&& ((a*a + b*b) <(1+ EPS)))
            return 1;
    else
            return 0;
}
```

- EPS is applied to the triangles with the longest side equal to 1.
- There are no the minus operation on close to each other values.

Exercise 3, Problem 3: Classifying Toys

- More information in Japanese about the absolute and relative errors:
  - http://msugai.fc2web.com/java/class/DecimalTips.html

Exercise 4, Problem 2: Passing References to a Method

- To find the right solution, you should study examples on Slides 21 to 23, Lecture 4.

Passing Primitive Data Type Arguments

- Primitive arguments, such as an int or a float, are passed into methods *by value.*
- If a parameter changes its value in the method, that changed value exists only within the scope of that method. When the method returns, the parameters are gone and any changes to them are lost.

```
public class PassPrimitiveByValue {
   public static void main(String[] args)
      int x = 3;
           // invoke passMethod() with x as
           // argument
      passMethod(x);
           // print x to see if its value has
           // changed
      System.out.println(
    "After invoking passMethod, x = " + x);
   }
   // change parameter in passMethod()
   public static void passMethod(int p) {
      p = 10;
   }
}
```

The output of this program is:

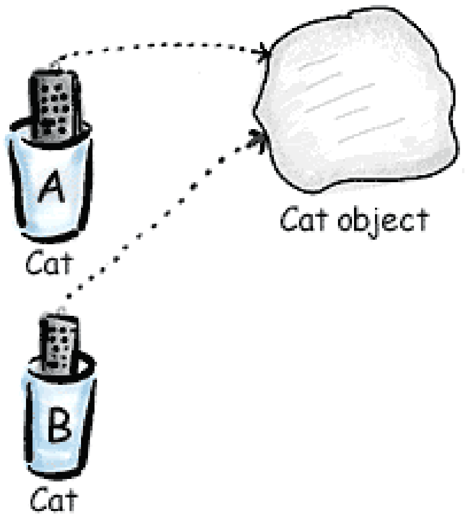
```
After invoking passMethod, x = 3
```

Passing Reference Data Type Arguments

- Java passes everything by value.
- With primitives, you get a copy of the contents.
- When you pass an object reference into a method, you are passing a COPY of the REFERENCE.
  - Example on the right: There is still just ONE Cat object. But now TWO remote controls (references) can access that same Cat object.
- You can change the Cat, using your new B reference (copied directly from A), but you can't change A.

```
Cat A = new Cat();
doStuff(A);

void doStuff(Cat B) {
   // use B in some way
}
```

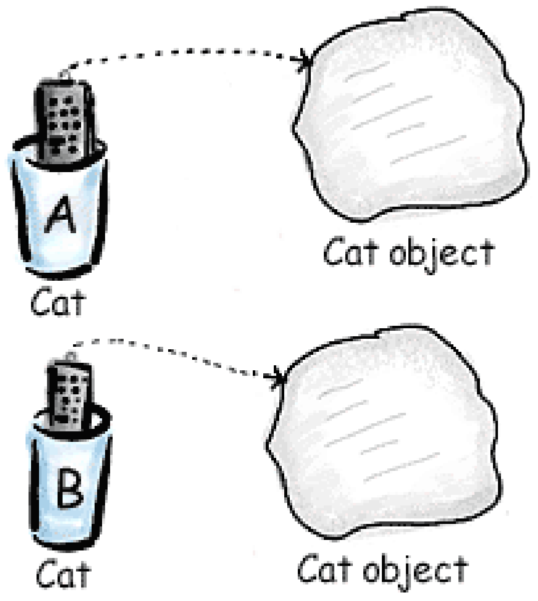


Passing Reference Data Type Arguments

- Example on the right: B = new Cat();
- Statement above simply "points" B to control a different object.

```
Cat A = new Cat();
doStuff(A);

void doStuff(Cat B) {
   B = new Cat(); // did NOT affect
                  // the A reference
}
```



Exercise 5, Problem 1: Independent Classes

- Encapsulation principal, you should study Slides 17, Lecture 2.

What is Encapsulation?

- Hiding internal state and requiring all interaction to be performed through an object's methods is known as *data encapsulation* — a fundamental principle of object-oriented programming.
  - In other words, there is not direct access to the fields of the object.

Exercise 5, Problem 1: Independent Classes

GOOD

```
class Rectangle {
   private float length;
   private float height;
   public Rectangle(float length, float height) {
                                  this.length = length;
                                  this.height = height;
   }
// omitted
}
public class TestObjects {
    public static void main(String args[]) {
        Rectangle r1 = new Rectangle(2.0f,3.0f);

        System.out.println("Rectange: length=" +
r1.getLength() + " height=" + r1.getHeight() +
" perimeter=" + r1.perimeter());
/ /omitted
   }
}
```

BAD

```
class Rectangle {
   float length;
   float height;
   public Rectangle(float length, float height) {
                                  this.length = length;
                                  this.height = height;
   }
// omitted
}
public class TestObjects {
    public static void main(String args[]) {
        Rectangle r1 = new Rectangle(2.0f,3.0f);
        r1.length = 5.0f;   // direct access to the field
        System.out.println("Rectange: length=" +
r1.getLength() + " height=" + r1.getHeight() +
" perimeter=" + r1.perimeter());
/ /omitted
    }
}
```

Exercise 5, Problem 1: Independent Classes

GOOD

```
class Rectangle {
   private int length;   // type is changed
   private int height;   // type is changed
   public Rectangle(float length, float height) {
                                   this.length = (int)length;
                                   this.height = (int)height;
   }
// omitted
}
public class TestObjects {
    public static void main(String args[]) {
        Rectangle r1 = new Rectangle(2.0f,3.0f);

        System.out.println("Rectange: length=" +
r1.getLength() + " height=" + r1.getHeight() +
" perimeter=" + r1.perimeter());
/ /omitted
   }
}
```

BAD

```
class Rectangle {
   int length; // type is changed
   int height; // type is changed
   public Rectangle(float length, float height) {
                                   this.length = (int)length;
                                   this.height = (int)height;
   }
// omitted
}
public class TestObjects {
    public static void main(String args[]) {
        Rectangle r1 = new Rectangle(2.0f,3.0f);
        r1.length = 5.0f; // error: why?
        System.out.println("Rectange: length=" +
r1.getLength() + " height=" + r1.getHeight() +
" perimeter=" + r1.perimeter());
/ /omitted
     }
}
```

Exercise 5, Problem 3: Overriding and Hiding Methods

- To find the right solution, you should study examples on Slides 17 to 22, Lecture 5.

Overriding Instance Methods

- *Overriding* means that a subclass redefines a method from a superclass when:
  - Both methods have the same signature;
  - Both methods have the same return type.
- A *covariant* return type – an overriding method can also return a subtype of the type returned by the overridden method.
- By using the keyword *super*, the overridden method can be invoked.

```
class AA {

       Object method(int i) {
             Object oo;
                   …
              return oo;
        }       // end of the method
}               // end of the AA class

class BB extends AA {
    String method(int k) {
          String os;
          Object oo = super.method(5);
                 …
           return os;
     }          // end of the method
}               // end of the BB class

String os = new BB().method(4);
```

Overriding Class Methods

- If a subclass defines a class method with the same signature as its superclass, the subclass' method *hide*s the superclass' method.
- The distinction between hiding and overriding is important when invoking:
  - The subclass version of an overridden method gets invoked.
  - The version that gets invoked depends on the namespace from which it is invoked.

Example: Overriding and Hiding Methods

```
class Animal {      //  the file name: Cat.java
    public static void testClassMethod() {
        System.out.println("The class method in Animal.");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Animal.");
    }
} // end of the Animal class
public class Cat extends Animal {
    public static void testClassMethod() {
        System.out.println("The class method in Cat.");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Cat.");
    }
    public static void main(String[] args) {
        Cat myCat = new Cat();
        Animal myAnimal = myCat;
        Animal.testClassMethod();
        myAnimal.testInstanceMethod();
    }
} // end of the Cat class
```

- Compile and run:
  - Save the text to the file: Cat.java
  - Compile the program typing: javac Cat.java
  - Run the program typing: java Cat
  - Output of the program:

```
The class method in Animal.
The instance method in Cat.
```

Comments on the Previous Slide

- The Cat class overrides the instance method in Animal and hides the class method in Animal.
- The main method in this class creates an instance of Cat and calls testClassMethod() on the class and testInstanceMethod() on the instance.
- The version of the hidden method that gets invoked is the one in the superclass, and the version of the overridden method that gets invoked is the one in the subclass.

Example: Overriding Methods

```
class AA {
    void insMethod() { … }
}

class BB extends AA {
     void insMethod() { … }
}

AA oa = new AA();
oa.insMethod();          // AA

BB ob = new BB();
ob.insMethod();          // BB

oa = ob;
oa.insMethod()           // BB
```

```
class AA {
     static void stcMethod() { … }
}

class BB extends AA {
     static void stcMethod() { … }
}

AA.stcMethod();
BB.stcMethod();

AA oa = new AA();
oa.stcMethod();          // AA

oa = new BB();
oa.stcMethod();          // AA
```

Overriding Methods: Summary

- A subclass can redefine the methods it inherits from its superclass:
  - Overriding instance methods
  - Hiding class methods
- Defining a method with the same signature:

	Superclass instance methods	Superclass static methods
Subclass instance methods	Overrides	Generates a compile-time error
Subclass static methods	Generates a compile-time error	Hides

Ex 6, Problem 3: The Case of the Confusing Constructor

- You were given:

```
class Confusing {
  private Confusing (Object o) { // DO NOT CHANGE THIS LINE
    Systemout.println("Object");
  private Confusing (double[] dArray) { // DO NOT CHANGE THIS LINE
    System.out.println"double array");
  static void main(String args) {
    o = new Confusing(null); // DO NOT CHANGE THIS LINE
}
```

- Did someone followed instructions on debugging given at Lecture 6?

Sl 30, L6: Strategy 4: jdb Debugger

- jdb is a command-line utility that enables you to debug Java programs.
- Terminology
  - A breakpoint is a line of code specified by the user using the debugger that the interpreter will halt at each time it reaches that point in the program.
  - Single-stepping is the process of executing your code one line at a time (in single steps).
- http://docs.oracle.com/javase/6/docs/technotes/tools/solaris/jdb.html

Sl 31, L5: Strategy 4: How to Use jdb

- Compile your program
  - javac -g AssertClass.java
- Run the debugger (two numbers are command line parameters for the AssertClass program)
  - jdb AssertClass 40 70
- Setup break-points in the program
  - Stop in Assertclass.main
- Run the program
  - run
- Execute the single stepping process
  - step
- Print a value
  - print sum

SI32, L5: Strategy 4: Key Commands of jdb

Command	**Comments**
help	displays the list of recognized commands with a brief description.
run	After starting **jdb**, and setting any necessary breakpoints, you can use this command to start the execution the debugged application.
cont	Continues execution of the debugged application after a breakpoint, exception, or step.
print name	Displays Java objects and primitive values.
stop at Class:line	stop at MyClass:22 *(sets a breakpoint at the first instruction for line 22 of the source file containing MyClass)*
stop in Class.method	*sets a breakpoint at the beginning of the method*
step	Execute the current line (go into the the method)
next	Execute the current line (step over calls)
clear Class.line	Clear a breakpoint at a line
exit	Finish debugging

Ex 6, Problem 3: The Case of the Confusing Constructor

- Interesting variant:

```
public class Confusing {
  private Confusing (Object o) { // DO NOT CHANGE THIS LINE
                    System.out.println("Object");
  }
          private Confusing (double[] dArray) { // DO NOT CHANGE THIS LINE
                    System.out.println("double array");
}
          public static void main(String[] args) {
                    Object o = null;
                    new Confusing(null); // DO NOT CHANGE THIS LINE
                    new Confusing(o);
          }
}
```

- The most specific constructor is selected!
- Output: double array