Java Programming I

Review

Lectures 9 to 14

Exercises 7 to 14

Quizzes 9 to 14

Contents

- Quizzes 9 to 14
- Exercises: Common errors
- Examination: Instructions

Quiz 9, question 1

- You get an error message, when you compile this program. The statement ***System.out.println(h.getID());*** is wrong.
- What is wrong with it? Please propose your variant of this statement.

```
class Person {
    private String name;
    public Person(String name) {
       this.name = name;
    }
}
class Student extends Person {
    private String id;
    public Student(String name, String id){
       super(name);
       this.id = id;
    }
   public String getID() { return id; }
}
```

```
public class PolymorphismDemo6 {
  public static void main(String[] args) {
      Student s =
        new Student("Saito","s115333");
      Person h = s;
      System.out.println(h.getID());
   }
}
```

- Answer: ***if (h instanceof Student) {***

  System.out.println(((Student)h).getID());

- Why? See Slides 6, 7, Lecture 8; Slide 25, Lecture 9

Sl 6, L 9: Example 4

```
class Person {
  private String name;
 public Person(String name) {
   this.name = name;
 }
 public String introduction() {
    return "My name is " + name +".";
  }
 public String getInfo() {
    return introduction();
  }
}
class Student extends Person {
  private String id;
  public Student(String name, String id){
    super(name);
    this.id = id;
  }
public String getID() { return id; }
public String introduction() {
    return "I am a student. "+
     super.introduction()+" My ID is " +
       getID()+".";
  }
}
public class PolymorphismDemo4 {
  public static void main(String[] args) {
       Student s =
         new Student("Saito","s115333");
       Person p = s;
       System.out.println(s.getInfo());
       System.out.println(p.getInfo());
   }
}
```

- Output of this program:
  - I am a student. My name is Saito. My ID is s115333.
  - I am a student. My name is Saito. My ID is s115333.

Sl 7, L 9: Comments on the Previous Slide

- The difference between Example 1 and this example:
  - The *Person* class has a *public String getInfo()* method.
- Why do they print the same output? (The reason is the same as for Example 1 from the previous lecture):
  - System.out.println(s.getInfo());
  - System.out.println(p.getInfo());
- The following statement is wrong. (The *getID* method is not in the set of the *Person* class methods; the compiler produces an error message):
  - System.out.println(p.getID()); // Error

Sl 25, L 9:Casting-Dot Operator Precedence

- The casting operator has lower precedence than the "." (dot) operator:
  - ((Student)p).getID()
- Without the parentheses the cast is associated with the method (*getID* in our example) and attempts to change its return type:
  - (Student)p.getID()

Quiz 9, question 2

- What is the result of the compilation and running of the following code?

```
class Bread {
  Bread() { System.out.println("Bread()"); }
} // end of the Bread class
class Cheese {
  Cheese() { System.out.println("Cheese()");
}
} // end of the Cheese class
class Meal {
  Meal() { System.out.println("Meal()"); }
} // end of the Mealclass
```

```
public class SpecialSandwich extends
Meal {
  private Bread b = new Bread();
  private Cheese c = new Cheese();
  public SpecialSandwich() {

System.out.println("SpecialSandwich()");
  }
  public static void main(String[] args) {
    new SpecialSandwich();
  }
} // end of the SpecialSandwich class
```

- Similar program see on Slides 19-21, Lecture 9

- Output of this program:
  Meal()
  Bread()
  Cheese()
  SpecialSandwich()

Sl 19, L 9: Order of Constructor Calls

```
class Meal {
  Meal() { System.out.println("Meal()"); }
}
class Bread {
  Bread() { System.out.println("Bread()"); }
}
class Cheese {
  Cheese() { System.out.println("Cheese()"); }
}
class Lettuce {
  Lettuce() { System.out.println("Lettuce()"); }
}
class Lunch extends Meal {
  Lunch() { System.out.println("Lunch()"); }
}
class PortableLunch extends Lunch {
  PortableLunch() {
      System.out.println("PortableLunch()");}
}
```

```
public class Sandwich extends
                        PortableLunch {
  private Bread b = new Bread();
  private Cheese c = new Cheese();
  private Lettuce l = new Lettuce();
  public Sandwich() {
    System.out.println("Sandwich()");
  }
  public static void main(String[] args) {
    new Sandwich();
  }
}
```

Output:

```
Meal()
Lunch()
PortableLunch()
Bread()
Cheese()
Lettuce()
Sandwich()
```

Sl 20, L 9: Comments on the Previous Example

- This example creates a complex class out of other classes, and each class has a constructor that announces itself.
- The important class is *Sandwich*, which reflects three levels of inheritance (four, if the implicit inheritance from *Object* is counted) and three member objects.
- The order of constructor calls for a complex object:
  - The base-class constructor is called. This step is repeated recursively such that the root of the hierarchy is constructed first, followed by the next-derived class, etc., until the most-derived class is reached.
  - Member initializers are called in the order of declaration.
  - The body of the derived-class constructor is called.

Sl 21, L 9: Comments on the Previous Example

- The order of the constructor calls is important
  - When you inherit, you know all about the base class and can access any **public** and **protected** members of the base class.
  - This means that you must be able to assume that all the members of the base class are valid when you're in the derived class.
  - Inside the constructor, you must be able to assume that all members that you use have been built.
  - The only way to guarantee this is for the base-class constructor to be called first.

Quiz 10, question 1

- How can you define an interface in the most common form?

- Answer is on Slide 8, Lecture 10

Sl 8, L 10: What is an Interface?

- In its most common form, an interface is a group of related methods with empty bodies. A bicycle's behavior, if specified as an interface, might appear as follows:

```
interface BicycleInterface {
        void changeCadence(int newValue);
        void changeGear(int newValue);
        void speedUp(int increment);
        void applyBrakes(int decrement);
}
```

Quiz 10, question 2

- What can a Java interface contain?
- Answer is on Slide 12, Lecture 10

Sl 12, L 10: Interfaces in Java

- In the Java programming language, an *interface* is a reference type, similar to a class, that can contain *only*
  - constants,
  - method signatures, and
  - nested types.
- There are no method bodies.
- Interfaces cannot be instantiated—they can only be *implemented* by classes or *extended* by other interfaces.

Quiz 10, question 3

- We have a declaration of two classes: Cube and CubePlus. Which one of them is correct?

```
interface Shape {
        public double area();
        public double volume();
}
class Cube implements Shape {
        int x = 10;
        public double area( ) {
            return (6 * x * x);
        }
}
abstract class CubePlus implements Shape
{       int x = 10;
        public double area( ) {
            return (6 * x * x);
        }
}
```

The possible answers:

a. Cube: correct
b. Cube: wrong
c. CubePlus: correct
d. CubePlus: wrong
e. I do not know

- Answer: b); c)
- Why? See on Slides 28, Lecture 10

Sl 28, L 10: Abstract Classes vs. Interfaces

- A class that implements an interface must implement *all* of the interface's methods. It is possible, however, to define a class that does not implement all of the interface methods, provided that the class is declared to be abstract.

```
abstract class X implements Y {
  // implements all but one method of Y
}

class XX extends X {
  // implements the remaining method in Y
}
```

- In this case, class X must be abstract because it does not fully implement Y, but class XX does, in fact, implement Y.

Quiz 11, question 1

- What is a package?
- Answer is on Slides 5, Lecture 11

Sl 5, L 11: What are Packages

- Definition
  - A package is a grouping of related types providing access protection and name space management.
- Note that types refers to classes, interfaces, enumerations, and annotation types.
- Enumerations and annotation types are special kinds of classes and interfaces, respectively, so types are often referred here as classes and interfaces.

Quiz 11, question 2

- Look at the Simple Example on Slide 21. How many classes are in the DemoPackage?
- Answer: ClassOne and ClassTwo
- Why? See Slide 11, Lecture 11

Sl 21, L 11: Simple Example

```
// file ClassOne.java in the directory
//  /home/s111111/java/Ex08/demopackage
package demopackage;
public class  ClassOne {
   public void methodClassOne() {
      System.out.println("methodClassOne");
   }
}
// file ClassTwo.java in the directory
//  /home/s111111/java/Ex08/demopackage
package demopackage;
public class  ClassTwo {
   public void methodClassTwo() {
      System.out.println("methodClassTwo");
   }
}
```

- Compilation:

  javac *.java

```
// file UsageDemoPackage.java in
// the directory
//  /home/s111111/java/Ex08/
import demopackage.*;
class UsageDemoPackage {
  public static void main(String[] args) {
   ClassOne v1 = new ClassOne();
   ClassTwo v2 = new ClassTwo();
   v1.methodClassOne();
   v2.methodClassTwo();
  }
}
```

- Compilation:

  javac UsageDemoPackage.java
- Run:

  java UsageDemoPackage

Quiz 12, question 1

- What is an Exception?
- Answer is on Slide 10, Lecture 12

Sl 10, L 12: What Is an Exception?

- An **exception** (“exceptional event) is an event, which occurs during the execution of a program that disrupts the normal flow of the program’s instructions.
- When an error occurs within a method, the method creates an object and hands it off to the runtime system. The **object (“exception object”)** contains information about the error.
- Creating an exception object and handing it to the runtime system is called “**throwing an exception**”

Quiz 12, question 2

- What is a checked exception?
- Answer is on Slide 14, Lecture 12

Sl 14, L 12: Kinds of Exceptions

- Three Kinds of Exceptions
  - Checked exception: can anticipate and recover the exception. Checked exceptions are subject to the Catch or Specify Requirement (CSR). All exceptions are checked exceptions, except for *Error*, *RuntimeException*, and their subclasses.
  - Error (unchecked exception): cannot anticipate and recover, not subject to CSR, ex) system malfunction
  - Runtime exception (unchecked exception): cannot anticipate and recover, not subject to CSR, ex) logic error or improper use of an API

Quiz 12, question 3

- What is the purpose of usage of the *finally* clause?
- Answer is on Slides 27, Lecture 12

Sl 27, L 12: try, catch, and finally

The finally clause is used to clean up internal state or to release non-object resources, such as open files stored in local variables.

```
public boolean searchFor (String file, String word) throws StreamException
{
  Stream input = null;
  try {
      input = new Stream(file);
      while (!input.eof())
        if (input.next().equals(word)) return true;
      return false;       // not found
   } finally {
        if (input != null) input.close();
   }
}
```

Quiz 13, question 1

- Look at the Example on Slide 17. What can you say about the following statement?

throw new EmptyStackException();

- This statement throws:
  a. An object
  b. An event
  c. A variable
  d. A type
  e. _____
  Your variant
- Answer : a)
- Why? see Slide 17, Lecture 13

Sl 17, L 13: How to Throw Exceptions

- Before we can catch an exception, some code somewhere must throw one. Regardless of what throws the exception, it's always thrown with the throw statement.
- The throw Statement

  throw someThrowableObject;

```
public Object pop () {
   Object obj;
   if (size == 0) {
     throw new EmptyStackException();
   }
   obj = objectAt(size – 1);
   setObjectAt (size – 1, null);
   size--;
   return obj;
}
```

If the stack is empty, pop instantiates a new EmptyStackException object and throws it.

Quiz 13, question 2

- What are advantages of Exceptions?
- Answer is on Slide 33, Lecture 13

Sl 33, L 13: Summary

- Error Handling is a safe and convenient way to deal with problems in a program.
- Exceptions are Java's tool for error handling.
- Exceptions allows us to separate the normal flow of the program from the error handling code.
- The compiler checks that all exceptions are caught.
- If the exception is not caught anywhere, the application terminates with a corresponding message.

Quiz 14, question 1

1. What is the result of the compilation and running of the following code? Files PrintBytes.java and input.txt are in the same directory. The content of the input.txt file is as follows:

one

```
import java.io.FileInputStream;
import java.io.IOException;
public class PrintBytes {
    public static void main(String[] args) throws IOException {
        FileInputStream in = null;
        try {
            in = new FileInputStream("input.txt");
            int c;
            while ((c = in.read()) != -1) {
                System.out.println((char)c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
        }
    }
}
```

Write your answer here:

Quiz 14, question 1

- Answer:
  - o
  - n
  - e
- Why? See Slides 19 - 20, Lecture 14

Sl 19, L 14: FileInputStream: Example

```
import java.io.*;
public class ReadStringsFromFile {
 public static void main(String[] args) {
    File file = new File("ReadStringsFromFile.java"); //create  a file object
    int ch; // variable to store the current byte
    try  {
      FileInputStream fin = new FileInputStream(file);     // 1
      while( (ch = fin.read()) != -1)                      // 2
              System.out.print((char) ch);                 // convert to char and display it
      fin.close();                                         // 3
    }
    catch(FileNotFoundException e) {
      System.out.println("File " + file.getAbsolutePath() +
                         " could not be found on filesystem");
    }
    catch(IOException ioe) {
      System.out.println("Exception while reading the file" + ioe);
    }
    System.out.println("\n Program is finished.");
  }
}    // Output: lines of the ReadStringsFromFile.java file
```

Sl 20, L 14: Comments on the Previous Slide

- Line marked with // 1
  - Creates new FileInputStream object. Constructor of FileInputStream throws FileNotFoundException if the agrument *file* does not exist.
- Line marked with // 2
  - To read bytes from stream use, int read() method of FileInputStream class. This method reads a byte from stream. This method returns next byte of data from file or -1 if the end of the file is reached. Read method throws IOException in case of any IO errors.
- Line marked with // 3
  - To close the FileInputStream, use void close() method of FileInputStream class. close method also throws IOException.

Quiz 14, question 2

2. What is the result of the compilation and running of the following code? Files ScanFile.java and input.txt are in the same directory. The content of the input.txt file is as follows:

one two three
four

```
import java.io.*;
import java.util.Scanner;
public class ScanFile {
    public static void main(String[] args)throws IOException {
        Scanner s = null;
        try {
            s = new Scanner(new BufferedReader(new
                                    FileReader("input.txt")));
            while (s.hasNext()) {
                System.out.println(s.next());
            }
        } finally {
            if (s != null) {
                s.close();
            }
        }
    }
}
```

Write your answer here:

Quiz 14, question 2

- Answer:

  one

  two

  three

  four

- Why? See Slides 38, Lecture 14

Sl 38, L 14: Selecting tokens in a file

```
import java.io.*;
import java.util.Scanner;
public class ScanFile {
    public static void main(String[] args)
                        throws IOException {
        Scanner s = null;
        try {
            s = new Scanner(new
                BufferedReader(new
                FileReader("test.txt")));
            while (s.hasNext()) {
                System.out.println(s.next());
            }
        } finally {
            if (s != null) {
                s.close();
            }
        }
    }
}
```

- Scanner is useful for breaking down formatted input into tokens and dealing with individual tokens according to their data types.
- ***test.txt*** is a file created by program on Slide 34.
  - java ScanFile

```
I'm
a
sentence
1
In
. . . . .
```

Exercises: Common Errors

- Many students did not pay attention to the lecture materials and lecture examples.
  - This is a wrong way to study.
- Many students did not read carefully descriptions of the problems:
  - You have to read every sentence in order to understand what to do.
- Every software engineer, every programmer MUST pay attention to every detail in the description of a problem.

Ex 10, Problem 3: Area Values of the Figure Objects

- Note: The part of your application responsible for finding the sum should not know the places of objects in the array.
- Variant for TestShape.java

```
public class TestShape {
    public static void main(String[] args) {
            Object[] s = {new Point(21, 16),
                          new Circle(5),
                          new Rectangle(10, 15),
                          new Circle(2)
                         };
            double total =0;
            for(int i =0; i < s.length; i++)
                 total+= ((Shape)s[i]).getArea(); // casting to the type of interface
            System.out.println("total area is " + total);
    }
}
```

Ex 11, Problem 2: Designing a Package to Calculate Fuel Consumption

- What is the point:
  - First, you should create the package and compile it.
  - Code of the package is unchangeable!
  - Then you create a new class BusTaxi
  - The question is: How does the package know the future (information about the methods of the BusTaxi class)?

Ex 11, Problem 2: Designing a Package to Calculate Fuel Consumption

- Three key elements:
  - fuelconsumption package

```
package fuelconsumption;
public class FuelConsumptionCalculation {
            . . .
            private SimpleCar[] cars;
 public FuelConsumptionCalculation(SimpleCar[] cars) {
   this.cars = cars;
   calculateFuelconsumption();
  }
 private void calculateFuelconsumption() {
   fuelConsumptionOldRegulation = 0;
   fuelConsumptionNewRegulation = 0;
   for (int i = 0; i < cars.length ; i++) {
      fuelConsumptionOldRegulation +=
            cars[i].calculateFuelConsumptionOldRegulations(); // HOW DOES PACKAGE KNOW
            . . .                                             // ABOUT BusTaxi code?
   }                                                          // IT (BusTaxi) WAS DESIGNED
}                                                             // AFTER THE PACKAGE
```

Ex 11, Problem 2: Designing a Package to Calculate Fuel Consumption

- Three key elements:
  - BusTaxi class:
    - Created after compilation of the fuelconsumption package.
    - fuelconsumption package was not changed.

```
import fuelconsumption.*;
class BusTaxi extends DeluxeCar {
        . . .
public float calculateFuelConsumptionOldRegulations() {
    return getDistance() / getFuelConsumption() +  (getEndWork() –
        getBeginWork()) / (getAcFuel() / miminumNumberPassengers);
 }
        . . .
```

Ex 11, Problem 2: Designing a Package to Calculate Fuel Consumption

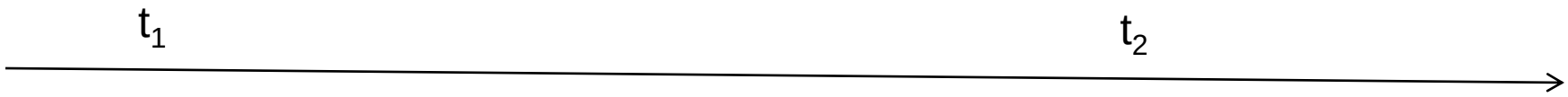
- Three key elements:
  - TestPackage.java
    - —Created after compilation of the fuelconsumption package.
    - —fuelconsumption package was not changed.

```
import fuelconsumption.*;
public class TestPackage {

  public static void main(String[] args) {
    SimpleCar[] cars;
. . .
    cars[4] = new BusTaxi("Mercedes", 6.6f, 11.5f);
    FuelConsumptionCalculation   d; // component from the package
. . .
    d = new FuelConsumptionCalculation(cars);

 }
}
```

Ex 11, Problem 2: Designing a Package to Calculate Fuel Consumption

fuelconsumption package
SimpleCar
AdvancedCar
DeluxeCar
FuelconsuptionCalculation
Byte code is created at t_1

BusTaxi
TestPackage
Byte code is created at t_2



Time line

```
public FuelConsumptionCalculation
                (SimpleCar[] cars) {
   this.cars = cars;
   calculateFuelconsumption();
}
private void calculateFuelconsumption() {
. . .
      fuelConsumptionOldRegulation +=
    cars[i].calculateFuelConsumptionOldRegulations();
. . .
```

call

call

call

```
cars[4] = new BusTaxi("Mercedes", 6.6f, 11.5f);
d = new FuelConsumptionCalculation(cars);
 . . .
class BusTaxi extends DeluxeCar {
                . . .
  public float
calculateFuelConsumptionOldRegulations() {
                . . .
```

When i =4, cars[4]: there is a call to the future!
Why?

Ex 11, Problem 2: Designing a Package to Calculate Fuel Consumption

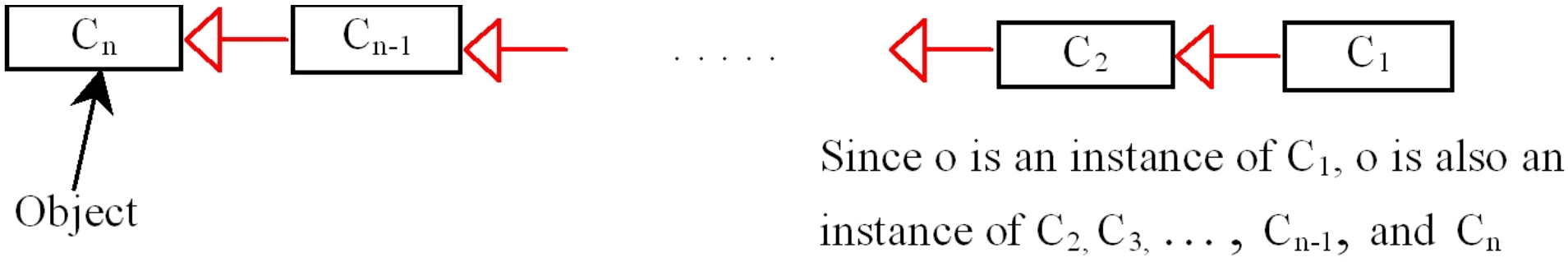
- Answer: Dynamic binding
- Why? See Slides 14, Lecture 7 and Slide 20, Lecture 7

Sl 14, L 7: Static and Dynamic Binding

- Non-polymorphic methods (static methods) are “bound”
  - at compile time
  - called *early binding* or static binding.
- Polymorphic methods are “bound”
  - at run time
  - called *late binding* or dynamic binding (also called dynamic dispatch).
- Alternate views of polymorphism:
  - One objects sends a message to another object without caring about the type of the receiving object.
  - The receiving object responds to a message appropriately for its type.
- Java methods are polymorphic by default
  - *static* or *final* (*private* methods are implicitly *final*) are bound at compile time.

Sl 20, L 7: Dynamic Binding in Java

- We can conceptually think of the dynamic binding mechanism as follows: Suppose an object *o* is an instance of classes C_1, C_2, ..., C_{n-1}, and C_n, where C_1 is a subclass of C_2, C_2 is a subclass of C_3, ..., and C_{n-1} is a subclass of C_n.
- That is, C_n is the most general class, and C_1 is the most specific class. In Java, C_n is the *Object* class.
- If *o* invokes a method *p*, the JVM searches the implementation for the method *p* in C_1, C_2, ..., C_{n-1} and C_n, in this order, until it is found. Once an implementation is found, the search stops and the first-found implementation is invoked.



Ex 14, Problem 3: How ot debug the program *Analysing.java*?

- A program generating two sets of data should be created.
- The first one is to generate a set consisting of 60 items with value of 30 for each item.
- The second one is to generate a set consisting of 60 items with the value of 31 for each item.
- For the first set the AnalyzingData should print nothing. For the second set it should print the message:
  - Treatment is necessary: average rate of breath per minute is 31.
- Recall the debugging topic: Lecture 6, Slides 22.

Sl 22, L6: Debugging: Key Definitions

- Bug: An error in a program.
- Testing: A process of analyzing, running a program, looking for bugs.
- Test case: A set of input values, together with the expected output.
- Debugging: A process of finding a bug and removing it.
- Exception: An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.
  - When an error occurs, like dividion by 0, Java throws an exception. A lot (generally too much) information is provided.

Examination: Instructions

- You are allowed to take to the exam room:
  - Electronic dictionaries,
  - Paper-based dictionaries,
  - Pens and pencils.

- Prohibited:
  - Books,
  - Lecture materials,
  - Discussion with colleagues,
  - Dictionaries on mobile devices.

- Exam questions are in English.

How to prepare yourself for the exam

1. Read lecture slides;
2. Read the corresponding chapters in the text book;
3. Study a Special Care Sets on the course Webpage;
4. Run examples from the lectures using *Jeliot*

   http://web-int.u-aizu.ac.jp/~yaguchi/courses/Java1/2016/sp_care_01.html
5. Work on exercises.

Examination day

- Date: August 4, Friday
- Time: 13:10 – 14:40 (3rd period)
- Rooms:
  - Prof. Yaguchi class: M1
  - Prof. Kitazato class: M2
  - Prof. Huang class: M3
  - Prof. Klyuev class: M4
  - Prof. Hamada class: M5
  - Prof. Shin class: M6
- Room assignment will be on the course Web site:

  http://web-int.u-aizu.ac.jp/~vkluev/courses/javaone/
- Good luck!