Writing assignment 3, Java

Authors:	Rein Spanjer, Bob Schijf, Jorrit Stutterheim
Group number:	10
Student ID Jorrit Stutterheim	13957899
Student ID Rein Spanjer	13558307
Student ID Bob Schijf	14076357
Study:	Premaster software engineering
Course:	Programmeertalen

Table of Contents

Primitive types and reference types

Q1 For every primitive type there is a reference type counterpart. Name at least 3 primitive types and their reference type counterpart.

primitive	Reference
boolean	java.lang.Boolean
int	java.lang.Integer
long	java.lang.Long

Q2 The boxing conversion converts the value of a primitive type to an object of the corresponding reference type. Give a code example on which the Java compiler performs this conversion automatically and explain why this conversion is helpful to the programmer in this example.

```
Boolean bool = true;
```

The reference types contains more then only the value. It contains functions that can be used like `equals`.

```
boolean boolFalse = false;
Boolean boolTrue = true;

boolTrue.equals(boolFalse); // Will return false
```

`boolFalse.equals` is not possible because it is a primitive type and it doesn't contain the function `equals`.

It is also helpful because it is less verbose. If this is not possible

```
Boolean bool = true;
```

you would have to write it like this:

```
Boolean bool = Boolean.TRUE;
```

and when using other objects.

```
boolean boolFalse = false;
Boolean bool = boolFalse;
```

you dont have to do it like the following:

```
boolean boolFalse = false;
Boolean bool = (Boolean) boolFalse;
```

Q3 Given that primitive types have a reference type counterpart, one might argue that the primitive types are redundant. Provide at least one argument motivating the existence of primitive types in Java and at least one argument against their existence.

Primitives are way more performant. Because primitives aren't objects and when they are used a lot then it can save a lot of memory and performance.

```
true == true; //this is faster
Boolean.TRUE.equals(Boolean.FALSE); //this is slower
```

The reference types can be useful when null values can occur. Null values are possible with references but are not with primitives. Because NULL values are often a thing in Java you could argue against the usage of primitives.

Q4 Which parameter-passing strategy does Java apply? Explain your answer.

Java has an pass-by-value strategy. When a function is called with its parameters, the values will be copies and stored in stack memory of the function. The behavior is different when using primitive types and

reference types. Primitives types hold the actual value and this value will be copied over. So changes that are made in the function will not transfer outside.

When using reference types you will actually be passing the pointer to this object to the function. The pointer will be copied to the stack memory. It will point to the same object. So changes made to this object will retain outside of the function.

Q5 What is the difference between the following two valid Java expressions in which s1 and s2 are both of type String.

```
s1 == s2
s1.equals(s2)
```

When using "==" operator it will compare the reference of the object itself. So when you have two strings objects with the same value but different reference the == will result into False. The .equals() is a method from the class String. This method will compare the value of the String object with the value of another String object. So the data that resides within the object. This is most often the method that you want to use for comparing Strings.

Q6 In your own words, explain the difference between == and .equals() for all types.

For all reference type object it works as I have described in Q5. For primitives there is no equals() method because no methods exist on primitives.

The equals method is from the class Object. All classes extends the Object class. So they will all have this function. The equals function can be overridden for classes that need a different equals method implementation.

Q7 Reflect on the previous questions and discuss for each whether it is about syntactic, semantic or pragmatic aspects of the language.

Question	Aspect	The question is about
1	Syntax	How the syntax of primitives and references are
2	Pragmatic	The compiler automatically converts types and how this works
3	Semantic	Why do primitives exists? What does the code mean
4	Semantic	What does it mean when you are passing a paramater to a function
5	Pragmatic	What happens when you the '==' operator or the equals method
6	Pragmatic	What happens when you the '==' operator or the equals method

Q8 In your own words, explain the concept of definite assignment. Why is it useful and why is it not perfect? Make a comparison with C and/or C++. How does definite assignment relate to the final keyword? When for example you are branching in Java and are declaring a variable in both. like so:

```
int a = 10;
if (a > 10){
   int k = 6;
} else {
   int k = 4;
}
k = 10;
```

k in this case will throw a compiler error. Because it is not sure whether k is initialized. It will not go to all the branches to check. We as humans can immediately see that k will definitely be defined. But the compiler will not go to all the branches and see. To fix this we can do the following:

```
int a = 10;
int k;
if (a > 10){
   k = 6;
} else {
   k = 4;
}
k = 10;
```

Now the compiler is happy and will compile the program. In C it is also not possible to do the following

```
int a =  5;

if (a == 5){
   int k = 10;
}
else {
   int k = 12;
}
k = 10;
```

It works similar as in Java as they are bot static typed languages.

Definite assignment assures the compiler that a variable is defined when it is assigned. When `final` is used you tell the compiler that assigning a final variable is definitely not assigned before. So it is more restrictive then only definite assignment. You could call it single definite assignment.

Classes and inheritance

Q9 If a language supports multiple inheritance, a class can inherit methods and variables from multiple parent classes. However, if multiple parents define the same method, this causes ambiguity in determining which of the versions of the method is actually inherited. Does Java support multiple inheritance? If so, how is the problem of ambiguity resolved or addressed? Explain your answer

Java does not support multiple inheritance.[1] However, a class can implement one or more interfaces, which has helped Java get rid of the impossibility of multiple inheritance[1].

Classes can then implement these interfaces and have to make a custom implementation for the functions of the interface, example[1]:

Another that Java could support multiple inheritance is by nesting classes, see question 11.

```
public interface Hockey extends Sports, Event

interface Event {
   public void start();
}
interface Sports {
   public void play();
}
interface Hockey extends Sports, Event{
   public void show();
}
public class Tester{
   public static void main(String[] args){
      Hockey hockey = new Hockey() {
         public void start() {
            System.out.println("Start Event");
         }
         public void play() {
            System.out.println("Play Sports.");
         }
         public void show() {
            System.out.println("Show Hockey.");
         }
      };

      hockey.start();
      hockey.play();
      hockey.show();
   }
}
```

Q10 What is the difference between a method declared with the static keyword and one without the static keyword within the same class? Refer in your answer to instance variables and explicitly address both syntactic, semantic and pragmatic differences.

A static method (or member variable) is class method that is the same in every instantiation of the class. Thus, a static method can only interact with static members of the class.

An advantages of a static method is that it is bound, and can be optimized, during compilation, so it is more efficient and uses less memory (pragmatic difference). A danger is here that if a static member variable is changed in one object, it will be changed in all existing (and future) objects of this class, which is fine as

long is that is intended behavior (semantic difference). The only syntactical difference is the use of the 'static' keyword.

In a small example:

```
class Animal {
  public static int nrOfSpots = 0;
  public static void showSpots() {
    System.out.println(nrOfSpots);
  }
}
public class Demo {
    public static void main(String args[]) {
        Animal pig = new Animal();
        pig.showSpots(); // Prints 0

        pig.nrOfSpots = 10;

        Animal human = new Animal();
        human.showSpots(); // Prints 10!!
    }
}
```

Q11 In Java it is possible to define classes within classes (i.e. to nest class definitions). Such classes are referred to as inner classes. What kinds of inner classes exist? Explain why inner classes can be useful

Non-static Nested Classes: Inner classes are a security/encapsulation mechanism in Java. In Java a class cannot be associated with the access modifier private, but if we have the class as a member of other class, then the inner class can be made private. This class can also be used to access the private members of a class. Inner classes are of three types depending on how and where you define them. They are:[2]

- Inner Class: can be made private and can access member variables of its parent class.
- Method-local Inner Class: you can even make an entire class within a class method.
- Anonymous Inner Class: similar to a lambda function in functional programming languages you can directly implement a class without a definition using the following syntax[2]:

```
// From within a existing class
AnonymousInner inner = new AnonymousInner() {
   public void mymethod() {
            System.out.println("This is an example of anonymous inner
class");
         }
}
```

Static Nested Classes: A static inner class is like the above a nested class with the 'static' keyword. Being static it has the property that it can be used without instantiating its parent class. Of course it can only use

its parents static member variables and methods.

A few reason to use nested classes:

- Increases encapsulation, if your parent class needs to do something 'difficult' it can use a nested class which only exposes certain things to the outside world. In this sense it is kinda of an interface to other developers and the inside implementation can be easily changed.
- It is a way of grouping related code. In Java you can only have one public class per file, so nesting classes can save a lot of new Java files, which makes sense if functionality is very intertwined.
- In this way you could actually make multiple inheritance work.

Q12 In your own words, explain the overloading concept in Java.

Function overloading This is easily achieved in Java. Methods from the same class with the same name can accept a different number of arguments. As in the following example:

```
// Source: https://www.tutorialspoint.com/what-is-overloading-in-java
public class Sample{
   public static void add(int a, int b){
      System.out.println(a+b);
   }
   public static void add(int a, int b, int c){
      System.out.println(a+b+c);
   }
   public static void main(String args[]){
      Sample obj = new Sample();
      obj.add(20, 40); // prints 60
      obj.add(40, 50, 60); // prints 150
   }
}
```

operator overloading User defined operator overloading is not supported in Java. Some OO counterparts like C++, Python and Kotlin do support this feature.

Q13 In your own words, explain the overriding concept in Java. How does it relate to the final keyword?

In Java inheritance is used a lot, so as a subclass (inherited class) it is possible to use methods from the parent class. But it is often required to slightly adjust the parents' method, for example a to_string method would usually need to be implemented differently. Re-implementing a parents function is called overriding. Overriding is not allowed when the 'final' keyword is present in the parents class. See below example which will give a compilation error:

```
class Boat {
    public void sink() {
      System.out.println("MAYDAY MAYDAY");
   }

   final public void sos() {
```

```
        System.out.println("Sending SOS signal");
    }
}

class Submarine extends Boat {
    public void sink() {
        System.out.println("Setting heading in a downward angle.");
    }

    // Gives compilation error since parent method is final
   public void sos() {
        System.out.println("We are a submarine, we don't send SOS");
   }
}

public class TestBoats {

    public static void main(String args[]) {
        Boat a = new Boat();   // Boat reference and Boat object

        Boat b = new Submarine();   // Boat reference but Submarine object
        a.sink();   // Prints MAYDAY, MAYDAY

        b.sink();   // runs the method in Submarine class, prints "Setting
heading in a downward angle."
    }
}
```

Additionally it is best practice to add the '@Override' annotator when overriding a method. If the override is done incorrectly the compiler will crash on it and it approves readability for other developers.

Additional bonus questions

Q14 Write the shortest syntactically valid Java program you can come up with that sends something to the standard output.

```
public class App
{
    public static void main( String[] args )
    {
        System.out.println( "Hello World!" );
    }
}
```

Q15 Write the shortest syntactically valid Java program you can come up with.

```
public class App
{
```

```
    public static void main( String[] args ) {}
}
```

Q16 Enum declarations simplify working with types of enumerable values. Enum declarations introduce normal classes (and are in that sense redundant) but add certain conveniences to the programmer. What functionality is generated by the Java compiler for Enum declarations? How do Enum declarations simplify working with enumerable types for the programmer?

An enum class is basically a wrapper around a set of constant values. Just like in C enums can be represented under the hood as a simple integer number but have a but more functionality in Java. Enums are used in any language to work with a certain set of constants and improve readability using these constants (like using them in a switch statement), in Java enums also improve type-safety.

The extra functionality that the Java compiler adds is:

- Iterating over enums
- custom fields, methods and constructors can be added
- Implements the following methods[3]:
- value(): returns an array of enum type containing all the enum constants
- valueOf(): takes a string and returns an enum constant having the same string name
- ordinal(): returns the position of an enum constant (index)
- compareTo(): compares two enum constants based on their position (index)
- toString(): returns the string representation of the given enum
- name(): returns the defined name of an enum constant in string form

References

1. https://www.tutorialspoint.com/java-and-multiple-inheritance
2. https://www.tutorialspoint.com/java/java_innerclasses.htm
3. https://www.programiz.com/java-programming/enums