Classes and Objects

Every object is an instance of a class. It is allocated memory. The class of an object serves as the type of the object and as a blueprint, which defines the data that the object stores and the methods for accessing and changing that data. [1] [2]

The critical members of a class in Java are the following: [3] [1]

- Class fields ... are static variables, and are associated with the class itself.
- Class methods ... are static methods, and are associated with the class itself.
- *Instance variables* ... are also called **fields** or **attributes**, and represent the data belonging to an object of a class. They must have a type, which either can be a primitive type or any class type (also known as reference type).
- *Instance methods* ... are blocks of code that perform actions, and are associated to individual instances of the class (with objects).

Let us consider the following example. The listing shows a complete definition of a simple class `Counter`.

```
public class Counter {
   private int count; // a simple integer instance variable
   public Counter() {} // default constructor
   public Counter(int initial) { count = initial; } // an alternate constructor
   public int getCount() { return count; } // an accessor method; a getter method
   public void increment() { count ++; } // an update method
   public void increment(int delta) { count += delta; } // an update method
   public void reset() { count = 0; } // an update method
}
```

This class includes one instance variable named `count`, two constructors named `Counter()` and `Counter(int initial)`, one accessor method, and three update methods.

Creating and Using Objects

Having a class definition, we are able creating and using objects. The following listing illustrates the creating and usage of objects:

```
public class CounterDemo {
   public static void main (String[] args) {
      Counter c; // declares a variable
      c = new Counter(); // constructs a counter; assigns its reference to c
      c.increment(); // increases its value by one
      c.increment(3); // increases its value by 3
      int temp = c.getCount(); // will be 4
      c.reset(); // value becomes 0

      Counter d = new Counter(5); // declares and constructs a counter having value 5
      d.increment(); // value becomes 6

      Counter e = d; // assigns e to reference the same object as d
      temp = e.getCount(); // will be 6 (as e and d reference the same counter)
      e.increment(2); // value of e (also known as d) becomes 8
   }
}
```

In Java, classes are known as **reference types**. A variable of that type is known as **reference variable**. A reference variable is able to store the memory address of an object from the declared class. It can also store a special value, `null`, which represents the lack of an object.

Instantiating a Class (Creating an Object)

An object is an instance of a class. The `new` operator followed by a valid constructor allow creating/instantiating objects in Java. When creating a new instance of a class, the following events occur in Java:

- A new object is dynamically allocated in memory, and all its instance variables are initialized by default. That means reference variables are initially null, and variables of primitive types are initially set to `0` (boolean will be false by default).
- The constructor for the new object is called with the specified parameters.
- After the constructor returns, the new operator returns a reference the newly created object. So, the object variable **refers** to the newly created object.

The Dot Operator

An object reference allows accessing the methods and the instance variables belonging to an object. In Java, this access is realized with the dot (".") operator. We invoke a method associated with an object by using the reference variable name, followed by the dot operator, and the method name and its parameters. For instance, we call `c.increment()` or `c.increment(3)` or `c.getCount()` in the listing above.

If the dot operator is used on a reference that is currently `null`, there will be a `NullPointerException`. A reference variable can be viewed as pointer to an object.

Defining a Class

So far, we are able defining simple classes was well as creating and using objects. In the following, we describe keywords known as modifiers in more detail. [3]

Modifiers

In Java, a modifiers is placed immediately before the definition of a class, an instance variable or a method. A modifier defines an additional constraint about the corresponding definition.

Access Control Modifiers ... control the level of access (visibility, scope):

- The **public** class modifier allows access to all classes. In this case the dot operator can be used to directly access the members.
- The **protected** class modifier allows access to the following groups of other classes: classes that are subclasses of the given class through inheritance, and classes that belong to the same packages as the given class.
- The **private** class modifier allows access to code within that class. Subclasses and any other classes have no access to private members.
- If no explicit access control modifier is given, the **package-private access** level is valid. All classes in the same package have access but no other classes from other packages.

The static Modifier ... can be declared for any variable or method of a class.

Variables, which are declared as static belong to the class, and are used to store global information of a class. They exist also no objects were invoked.

Methods, which are declared as static belong to the class, and exist without invoking any object. A method declared as static is normally called by using the name of the class, instead of an instantiated object.

The abstract Modifier ... may be used in connection with methods of a class.

A method, which is declared as abstract provides its signature without any implementation of its body. A class with one or more abstract methods has to be declared as abstract as well. In case of inheritance, a subclass of an abstract superclass has to deliver the implementation of inherited abstract methods.

The final Modifier ... is used for variables that can be initialized once but can never be assigned to a new value. If the variable is a primitive type, then it is a constant. If the variable is a reference type, then it will always refer to the same object.

If a method is declared as final it cannot be overridden by a subclass. If a class is declared as final it cannot be subclassed.

Defining Constructors

A constructor is used to instantiate / create a new instance of a class. In the course of this creation, each instance variable of the object is typically initialized. Constructors are similar to methods, however, there are some distinctions:
A constructor

- ... cannot be static, abstract or final. Allowed modifiers, defining visibility, are public, protected, private, or by default, package-level visibility.
- ... must be named according to the name of the class it constructs.
- ... does not specify a return type (not even void). The body of a constructor does not return anything. The responsibility of the constructor is only to initialize the state of a new instance.

A class can have many constructors, whereby each constructor has to differ in its signature (different type and/or number of parameters). If no constructor is provided within a class, a user cannot access the syntax using `new` ...

The Keyword this

The keyword `this` automatically refers to the instance upon (dt. auf) which the method was invoked.
The keyword is used for the following reasons:

- Storing the reference in a variable or send it as parameter to another method.
- Differentiating between an instance variable and a local variable with the same name.
- Allowing one constructor body to invoke another constructor body.

The toString Method

Every object type has a method called `toString` that returns a string representation of the object. [1]
By default it displays the type of the object and its address. However, we can override this behavior by providing our own `toString` method.

The equals Method

To check if methods are equal, we can either use the == operator or the `equals` method. [1]
The == operator checks whether objects are identical. That is, whether they are the same object, reference to the same memory.
The `equals` method checks whether they are equivalent. That is, whether they have the same value.

Summary of static and non-static (related to objects) Components

	Object-related Components	Static Components
Declaration	without `static`	with `static`
Exist	in any object	only once in a class
Fields are allocated	when the object is created/instantiated	when the class is loaded (at the beginning of the program)
Fields are released	from the garbage collector, when no pointer refers to the object	when the class is unloaded (at the end of the program)
Constructor is called	when the object is created/instantiated	when the class is loaded
Fields are accessed	`obj.field` `this.field`	`Class.field`
Methods are invoked	`obj.m();` `this.m();`	`Class.m();`

Class Diagrams

A graphical representation can be used to describe classes. [1]
With respect to UML (Unified Model Language) a class diagram serves as documentation for Java code. A class diagram shows the interface of classes.

Fraction

The following example **Fraction.java** illustrates the usage of classes and objects in Java.

The program as follows handles work with fractions. Useful operations are addition, subtraction, multiplication, and division. The data (numerator, denominator) and methods (add, subtract, multiply, divide) build up a unit.

Fraction.java

```
public class Fraction {
   int n;        // numerator
   int d;        // denominator
   static int fractionCounter = 0;
...
}
```

See file Fraction.java

What You Should See

Use Java via Eclipse

```
Fraction@2a139a55
1 / 2
Fraction@15db9742
3 / 5
Multiply obj a with obj b:
Fraction@2a139a55
3 / 10
Divide obj a by obj a:
Fraction@2a139a55
30 / 30
Number of created Objects: 2
```

Do It

1. *Fraction.*

   Extend the class `Fraction`.

   - Write an instance method called `negate` that reverses the sign of a rational number. This method should be a modifier, so it should be `void`. Add lines to `main` to test the new method.
   - Write an instance method called `invert` that inverts the number by swapping the numerator and denominator. Add lines to `main` to test the new method.
   - Write an instance method called `toDouble` that converts the rational number to a double (floating-point number) and returns the result. As always, test the new method.
   - Write an instance method named `reduce` that reduces a rational number to its lowest terms by finding the greatest common divisor of the numerator and denominator and dividing through. This method should not modify the instance variables of the object on which it is invoked.

2. *Times.*

   Write a class `Time` for storing times. Each `Time` object should store a time in form of hours and minutes. There should be the following methods:

   - Reasonable constructor.
   - Adding one time to another time (e.g. 5 hours 42 minutes plus 3 hours 27 minutes gives 9 hours 9 minutes).
   - Computing the difference between two times (e.g. 5 hours 20 minutes minus 3 hours 10 minutes gives 130 minutes).
   - Translation of a time in minutes (e.g. 3 hours 20 minutes gives 200 minutes).

   Present expressive test cases for your implementation.

3. *Dictionary.*

   Write a class `Dictionary`. The dictionary should contain pairs of words (German and English version). The first word serves as key and the second as value. There should be the following methods:

   - Reasonable constructor.
   - `void insert (String key, String value) { ... }`
   - `String lookup (String key) { ... }`

   `insert` adds `key` and `value` into the dictionary. `lookup` searches for `key` in the dictionary and returns the corresponding `value` or `null`, in case that `key` was not found.

   Present expressive test cases for your implementation.

Vocabulary [1]

In the following, we define several terms, which are useful for further understanding.

attribute One of the named data items that make up an object.

dot_notation Use of the dot operator to access attributes or methods of an object.

object-oriented A way of organizing code and data into objects, rather than independent methods.

garbage_collection The process of finding objects that have not references and reclaiming their storage space.

UML Unified Modeling Language, a standard way to draw diagrams for software engineering.

class_diagram An illustration of the attributes and methods for a class.

instance A member of a class. Every object is an instance of some class.

instance_variable An attribute of an object; a non-static variable defined at the class level.

constructor A special method that initializes the instance variables of a newly-constructed object.

client A class that uses objects defined in another class.

getter A method that returns the value of an instance variable.

setter A method that assigns a value to an instance variable.

override Replacing a default implementation of a method such as `toString`.

instance_method A non-static method that has access to `this` and the instance variables.

identical Two values that are the same; in the case of objects, two variables that refer to the same object.

equivalent Two objects that are “equal” but not necessarily identical, as defined by the equals method.

References

[1] Allen B. Downey and Chris Mayfield. *Think Java.* O’Reilly Media, 2016.

[2] Robert Liguori and Patricia Liguori. *Java 8 Pocket Guide.* O’Reilly Media, 2014.

[3] Michael T. Goodrich Roberto Tamassi and Michael H. Goldwasser. *Data Structures and Algorithms in Java.* Wiley, 2014.