Manipulating fields and methods

Getting fields values

```
class Item {
    public static final int maxItems = 100;
    public static int inStock = 19;

    private String name;
    protected int basePrice;

    public Item(String name, int basePrice) {
        this.name = name;
        this.basePrice = basePrice;
    }

    public String getName() {
        return name;
    }

    public int getPrice() {
        return (int) (basePrice * getMarkUp());
    }

    protected double getMarkUp() {
        double markUp = 0.1;
        // ... connecting to the remote server
        return 1 + markUp;
    }
}
```

With the help of the Field object we can find out the value of some field of the object. Basically, this is the main purpose of this class. It has a get method that takes one argument, which is an object whose field's value we want to get.
Note that the Field object is not bound to any object of the Item class.

That's why we need to pass the object directly as an argument. To get the value of the static field you can pass null as an argument.

```
Item item = new Item("apples", 500);
Class itemClass = item.getClass();
Field[] fields = itemClass.getDeclaredFields();
```

Now let's try to use the get method for all its fields:

```
for (Field field : fields) {
   System.out.println(field.getName() + " " + field.get(item));
}
```

java.lang.IllegalAccessException: cannot access a member with modifiers "private"

Java checks if you can access this field, but you can change the accessibility just by calling setAccessible(true) method.
Let's improve the code a little:

```
for (Field field : fields) {
   field.setAccessible(true);
   System.out.println(field.getName() + " " + field.get(item));
}
```

maxItems 100
inStock 19
name apples
basePrice 500

Setting values to the fields

Field's set method works the same way. It takes two arguments: an object and a new value for the field. Again, if you want to set a static field you can pass null as the first argument. Below is an example of calling the set method. What we try to do here is to set the value to itself:

```
for (Field field : fields) {
   field.setAccessible(true);
   field.set(item, field.get(item));
   System.out.println(field.getName() + " " + field.get(item));
}
```

java.lang.IllegalAccessException: Can not set static final int field to java.lang.Integer

The final fields in Java cannot be changed, it is true. But now there is no workaround for this: it would be an even bigger crime in the world of Java if someone changed a final field of an object.
To correct the code, we should make sure that the field is not final by checking its modifier with isFinal() method. Since the example above is somewhat useless, we'll also make it more sensible:

```
for (Field field : fields) {
   field.setAccessible(true);
   if (field.getType() == int.class && !Modifier.isFinal(field.getModifiers()))
{
      field.set(item, 0);
   }
```

```
}
```

This code resets all non-final integer fields in the instance of some class to 0.

Invoking methods
Invoking methods is similar, but this time the invoke method of the Method object is used. This method can take a different number of arguments: one more than the called method has. The first argument is the object whose method we want to call or, as you might expect, in case of static methods it's null.

```
Method[] methods = itemClass.getDeclaredMethods();
for (Method method : methods) {
   method.setAccessible(true);
   System.out.println(method.invoke(item));
}
```

All three methods had zero arguments, so invoke was called with only one argument. The output may differ because the elements in the returned array are not sorted and are not in any particular order. Here's one of the possible outputs:
apples
1.1
550

If you want to invoke a static method, you need to pass the class to the invoke method: **method.invoke(YourClass.class, ...).**

When it works

One of the typical problems that can be solved by using reflection is the **serialization** of objects. If some class does not implement the Serializable interface, it cannot be serialized without using reflection. With reflection, however, all the class fields, even private ones, become visible, so you can write them into an external file.

Never change the values of private fields in all cases except for deserialization, because this way there is a high probability to crash the program. To deserialize an object, you must first create an instance of it.

```
System.class
        .getField("out").getType()
        .getMethod("println", String.class)
        .invoke(new PrintStream(System.err), "Hello!");
```

public Method getMethod(String name, Class<?>... parameterTypes) throws NoSuchMethodException, SecurityException

- name: A String specifying the simple name of the desired method.
- parameterTypes: An array of Class objects that represent the parameter types of the desired method.

This method searches for a public method with the specified name and parameter types in the class or interface represented by this Class object. If the method is found, it returns a Method object that reflects the specified public method.

```
import java.lang.reflect.Method;

public class Main {
   public static void main(String[] args) throws NoSuchMethodException, SecurityException {
      Class<String> stringClass = String.class;
      Method lengthMethod = stringClass.getMethod("length");
      System.out.println("Method name: " + lengthMethod.getName());
      System.out.println("Return type: " + lengthMethod.getReturnType());
   }
}
```

In this example, getMethod("length") is used to obtain the Method object representing the length() method of the String class. Then, getName() and getReturnType() are used to retrieve information about the method.

```
public class Main {

   public static void main(String[] args) throws Exception {
      Main.class.getMethod("hello").invoke(null);
   }

   static void hello() {
      System.out.println("Hello world!");
   }

}
```

Throws NoSuchMethodException as it is not public.

More examples:

```
class FieldGetter {

    public Object getFieldValue(Object object, String fieldName) throws
IllegalAccessException {
        try{
            return object.getClass().getField(fieldName);
        }catch (NoSuchFieldException e){
            return null;
        }
    }

}
```