3. domaća zadaća; OPRPP2

Ako još niste, napravite novi radni prostor (engl. *workspace*) u okolini Eclipse. U njemu potom napravite prazan Maven projekt:

- u vršnom direktoriju radnog prostora napravite direktorij `hw03-0000000000` (zamijenite nule Vašim JMBAG-om),
- u tom novom direktoriju oformite Mavenov projekt `oprpp2.jmbag0000000000:hw03-0000000000` (zamijenite nule Vašim JMBAG-om),
- u projekt dodajte ovisnost prema biblioteci `junit` pa
- importajte projekt u Eclipse.

Sada možete nastaviti s rješavanjem zadataka.

Zadaća ima 8 zadataka. Sve ih rješavate u ovom istom projektu. Svaki zadatak nadograđuje funkcionalnost koju ste razvili u prethodnom zadatku. Za rješavanje određenih dijelova trebat ćete iz vaših prethodnih zadaća (OPRPP1!) u ovaj projekt **iskopirati** određene pakete koje ste ranije razvili.

Problem 1.

Create a package `hr.fer.zemris.java.webserver` and in it a class `RequestContext`. This class has a single inner public static class entitled `RCCookie`. `RCCookie` has read-only String properties `name`, `value`, `domain` and `path` and read-only Integer property `maxAge`.
The class `RequestContext` has following private properties `OutputStream outputStream` and `Charset charset;` following public write-only properties `String encoding` (defaults to `"UTF-8"`), `int statusCode` (defaults to `200`), `String statusText` (defaults to `"OK"`), `String mimeType` (defaults to `"text/html"`), `Long contentLength` (defaults to `null`); following private collections `Map<String,String> parameters`, `Map<String,String> temporaryParameters`, `Map<String,String> persistentParameters`, `List<RCCookie> outputCookies` and private property `boolean headerGenerated` (deafults to `false`). There is a single constructor available:

```
public RequestContext(
      OutputStream outputStream,                // must not be null!
      Map<String,String> parameters,            // if null, treat as empty
      Map<String,String> persistentParameters,  // if null, treat as empty
      List<RCCookie> outputCookies);            // if null, treat as empty
```

The map `parameters` should be treated as read-only map. Maps `temporaryParameters` and `persistentParameters` are readable and writable. Add following methods:

* method that retrieves value from parameters map (or `null` if no association exists):

```
public String getParameter(String name);
```

* method that retrieves names of all parameters in parameters map (note, this set must be read-only):

```
public Set<String> getParameterNames();
```

* method that retrieves value from `persistentParameters` map (or `null` if no association exists):

```
public String getPersistentParameter(String name);
```

* method that retrieves names of all parameters in persistent parameters map (note, this set must be read-only):

```
public Set<String> getPersistentParameterNames();
```

* method that stores a value to `persistentParameters` map:
```
public void setPersistentParameter(String name, String value);
```

* method that removes a value from `persistentParameters` map:
```
public void removePersistentParameter(String name);
```

* method that retrieves value from `temporaryParameters` map (or `null` if no association exists):
```
public String getTemporaryParameter(String name);
```

* method that retrieves names of all parameters in temporary parameters map (note, this set must be read-only):
```
public Set<String> getTemporaryParameterNames();
```

* method that retrieves an identifier which is unique for current user session (for now, implement it to return empty string):
```
public String getSessionID();
```

* method that stores a value to `temporaryParameters` map:
```
public void setTemporaryParameter(String name, String value);
```

* method that removes a value from `temporaryParameters` map:
```
public void removeTemporaryParameter(String name);
```

Add following three methods:
```
public RequestContext write(byte[] data) throws IOException;
public RequestContext write(byte[] data, int offset, int len) throws IOException;
public RequestContext write(String text) throws IOException;
```

All of these `write` methods write its data into `outputStream` that was given to `RequestContext` in constructor. The method that gets `String` argument converts given data into bytes using previously configured `encoding` (i.e. `charset`). However, there is a catch. First time that any of these three `write` methods is called, a special header must be written into the underlying `outputStream` and only then can given data be written. This header is written only once (no matter which `write` method is called of the two available). After the moment the header is created and written, all attempts to change any of properties `encoding`, `statusCode`, `statusText`, `mimeType`, `outputCookies`, `contentLength` must throw `RuntimeException`; since these properties are used for header creating as well as for configuration of `RequestContext` objects, after the header is created there is no point in allowing the change anyway. At the moment of header construction, you should create a value for `charset` property: `charset = Charset.forName(encoding);`.

So how does the header looks like? Properties used for header construction are `encoding`, `statusCode`, `statusText`, `mimeType`, `outputCookies`, `contentLength`.

Header is obtained by serializing a several lines of text into bytes using codepage *`ISO_8859_1`* (see `StandardCharsets`). Lines are separated by "\r\n". First line must be of form:

“`HTTP/1.1`” *statusCode statusMessage*

Second line must be of form:

“`Content-Type:`“ *mimeType*

If mime type starts with “`text/`” (for example, “`text/html`” or “`text/plain`”), you should append on mime-type “`; charset=`” *encoding*.

If property `contentLength` is not null, add next line to headers:

“`Content-Length:`“ *contentLength*

If `contentLength` is null, previous line must not be included; this way, the consumer of the sent data will read all the data we send through the output stream, until the stream is closed.

If list of `outputCookies` is not empty, for each cookie you should emit a single line of form:

'`Set-Cookie: '` *name* '`="`' *value* '`"; Domain='` *domain* '`; Path='` *path* '`; Max-Age='` *maxAge*

domain, *path* and *maxAge* are included only if they are not `null` in given cookie object. For example, for a cookie with only *name* set to 'korisnik' and *value* set to 'perica' you would emit:

```
Set-Cookie: korisnik="perica"
```

If cookie also included *maxAge* set to `3600` you would instead emit a line:

```
Set-Cookie: korisnik="perica"; Max-Age=3600
```

Finally, another empty line should be emitted to signal the end of headers.

I have prepared a simple test case (not actual unit-test but something to demonstrate the usage and expectation) for your implementation of this class.

```
package hr.fer.zemris.java.custom.scripting.demo;

import hr.fer.zemris.java.webserver.RequestContext;
import hr.fer.zemris.java.webserver.RequestContext.RCCookie;

import java.io.IOException;
import java.io.OutputStream;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.HashMap;

public class DemoRequestContext {

	public static void main(String[] args) throws IOException {

		demo1("primjer1.txt", "ISO-8859-2");
		demo1("primjer2.txt", "UTF-8");
		demo2("primjer3.txt", "UTF-8");

	}

	private static void demo1(String filePath, String encoding) throws IOException {
		OutputStream os = Files.newOutputStream(Paths.get(filePath));

		RequestContext rc = new RequestContext(os, new HashMap<String, String>(),
					new HashMap<String, String>(),
					new ArrayList<RequestContext.RCCookie>());
		rc.setEncoding(encoding);
		rc.setMimeType("text/plain");
		rc.setStatusCode(205);
		rc.setStatusText("Idemo dalje");
```

```
		// Only at this point will header be created and written...
		rc.write("Čevapčići i Šiščevapčići.");

		os.close();
	}

	private static void demo2(String filePath, String encoding) throws IOException {
		OutputStream os = Files.newOutputStream(Paths.get(filePath));

		RequestContext rc = new RequestContext(os, new HashMap<String, String>(),
					new HashMap<String, String>(),
					new ArrayList<RequestContext.RCCookie>());
		rc.setEncoding(encoding);
		rc.setMimeType("text/plain");
		rc.setStatusCode(205);
		rc.setStatusText("Idemo dalje");
		rc.addRCCookie(
			new RCCookie("korisnik", "perica", 3600, "127.0.0.1", "/"));
		rc.addRCCookie(new RCCookie("zgrada", "B4", null, null, "/"));

		// Only at this point will header be created and written...
		rc.write("Čevapčići i Šiščevapčići.");

		os.close();
	}
}
```

This program will create three files: `primjer1.txt`, `primjer2.txt` and `primjer3.txt`. The mixed hex-based and textual ***view*** of `primjer1.txt` is show on image below. This is a file-view that can generate any more advanced text viewer so once you generate your text files, open them, activate HEX view and compare result with the following. Such a view is useful since you can easily observe all generated chars; for example, you can easily identify that the first line was terminated by a \r\n sequence (0D 0A hex).

```
00000000: 48 54 54 50 2F 31 2E 31|20 32 30 35 20 49 64 65 | HTTP/1.1 205 Ide
00000010: 6D 6F 20 64 61 6C 6A 65|0D 0A 43 6F 6E 74 65 6E | mo dalje..Conten
00000020: 74 2D 54 79 70 65 3A 20|74 65 78 74 2F 70 6C 61 | t-Type: text/pla
00000030: 69 6E 3B 20 63 68 61 72|73 65 74 3D 49 53 4F 2D | in; charset=ISO-
00000040: 38 38 35 39 2D 32 0D 0A|0D 0A C8 65 76 61 70 E8 | 8859-2....Čevapč
00000050: 69 E6 69 20 69 20 A9 69|B9 E8 65 76 61 70 E8 69 | ići i ©iąčevapči
00000060: E6 69 2E              |                         | ći.
```

The mixed hex-based and textual view of `primjer2.txt` is show on image below. Please observe that although the textual content of file is the same, the file lengths of previous file and this one differ because of different charsets used to encode characters. For example, in above example letter 'Č' is encoded with a single byte `C8` while in example below the same letter is using UTF-8 encoded with a sequence of two bytes: `C4` and `8C`.

```
00000000: 48 54 54 50 2F 31 2E 31|20 32 30 35 20 49 64 65 | HTTP/1.1 205 Ide
00000010: 6D 6F 20 64 61 6C 6A 65|0D 0A 43 6F 6E 74 65 6E | mo dalje..Conten
00000020: 74 2D 54 79 70 65 3A 20|74 65 78 74 2F 70 6C 61 | t-Type: text/pla
00000030: 69 6E 3B 20 63 68 61 72|73 65 74 3D 55 54 46 2D | in; charset=UTF-
00000040: 38 0D 0A 0D 0A C4 8C 65|76 61 70 C4 8D 69 C4 87 | 8....ÄŚevapÄŤiÄ‡
00000050: 69 20 69 20 C5 A0 69 C5|A1 C4 8D 65 76 61 70 C4 | i i Ĺ iĹ˘ÄŤevapÄ
00000060: 8D 69 C4 87 69 2E      |                       | ŤiÄ‡i.
```

For `primjer3.txt` here is only a textual representation. Please observe how "`; charset=UTF-8`" is

automatically added in header since the mime type is one of "`text/*`" types.

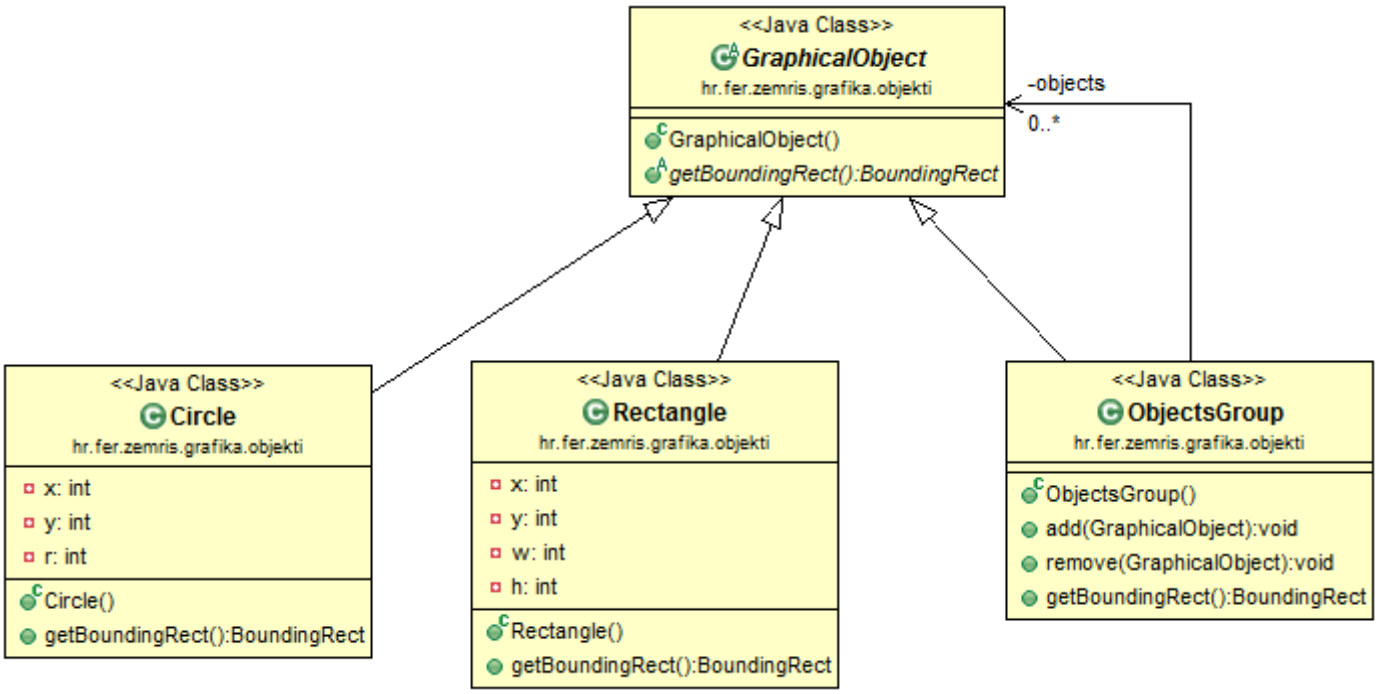
```
HTTP/1.1 205 Idemo dalje
Content-Type: text/plain; charset=UTF-8
Set-Cookie: korisnik="perica"; Domain=127.0.0.1; Path=/; Max-Age=3600
Set-Cookie: zgrada="B4"; Path=/

Čevapčići i Šiščevapčići.
```

Problem 2.

First, let us consider two important design patterns that will be used for this and next problem: Composite pattern and Visitor pattern.

The Composite desing pattern

You have most certainly already worked with the Composite pattern without knowing it. The idea behind the Composite pattern is to allow clients to work with simple objects and with composite objects (i.e. collections of other simple objects or composite objects) equally. See the following class diagram for illustration.



Here we have base abstract class `GraphicalObject` that represents any kind of graphical objects. And we have two such simple examples: the class `Circle` and the class `Rectangle` which both derive from `GraphicalObject`. Please observe that each `GraphicalObject` declares a method `getBoundingRect()` which returns the smallest rectangle that entirely encompasses the whole graphical object. The `BoundingRect` objects are constructed using constructor:

```
public BoundingRect(int left, int top, int right, int bottom);
```

In class `Circle` we can provide an implementation such as this:

```
@Override
public BoundingRect getBoundingRect() {
	return new BoundingRect(x-r,y-r,x+r,y+r);
}
```
and in class `Rectangle` an implementation such as this:

```
@Override
public BoundingRect getBoundingRect() {
	return new BoundingRect(x,y,x+w,y+h);
}
```

Now we can have a client that performs some calculations:

```
public void doStuff(GraphicalObject g) {
	BoundingRect brect = g.getBoundingRect();
	if(brect.right-brect.left > 200) {
		System.out.println("Objekt je preširok!");
	} else {
		System.out.println("Objekt je prihvatljivih dimenzija.");
	}
}
```

and we can call it as:

```
doStuff(new Circle(100, 100, 20));
doStuff(new Rectangle(90, 70, 20, 50));
```

What we would like to do now is to enable our clients (i.e. method `doStuff`) to operate on groups of graphical objects transparently – treating the whole group as a single object. This is important since it allows us to extend the functionality without modifying existing clients, and it simplifies programming.

In order to enable this, we add into picture another class: the so-called composite which is in our case class `ObjectsGroup`. This class derives from `GraphicalObject` so it is (from the viewpoint of client) a graphical object. However, instead of being some actual kind of object, it is an object that allows us to aggregate a collection of other `GraphicalObjects`. For this, this class must maintain a collection of its children (on the previous diagram this is the `objects` property), it must provide methods to manipulate this collection (methods `add` and `remove`) and it must declare and implement all of the actual methods that `GraphicalObject` declares on a meaningful way.

The latter in our case means that it must implement the method `getBoundsRect()` so that it asks all of its children for its bounding-rectangles and it must calculate the final minimal bounding rectangle that encompasses all of them.

The implementation could be as follows:

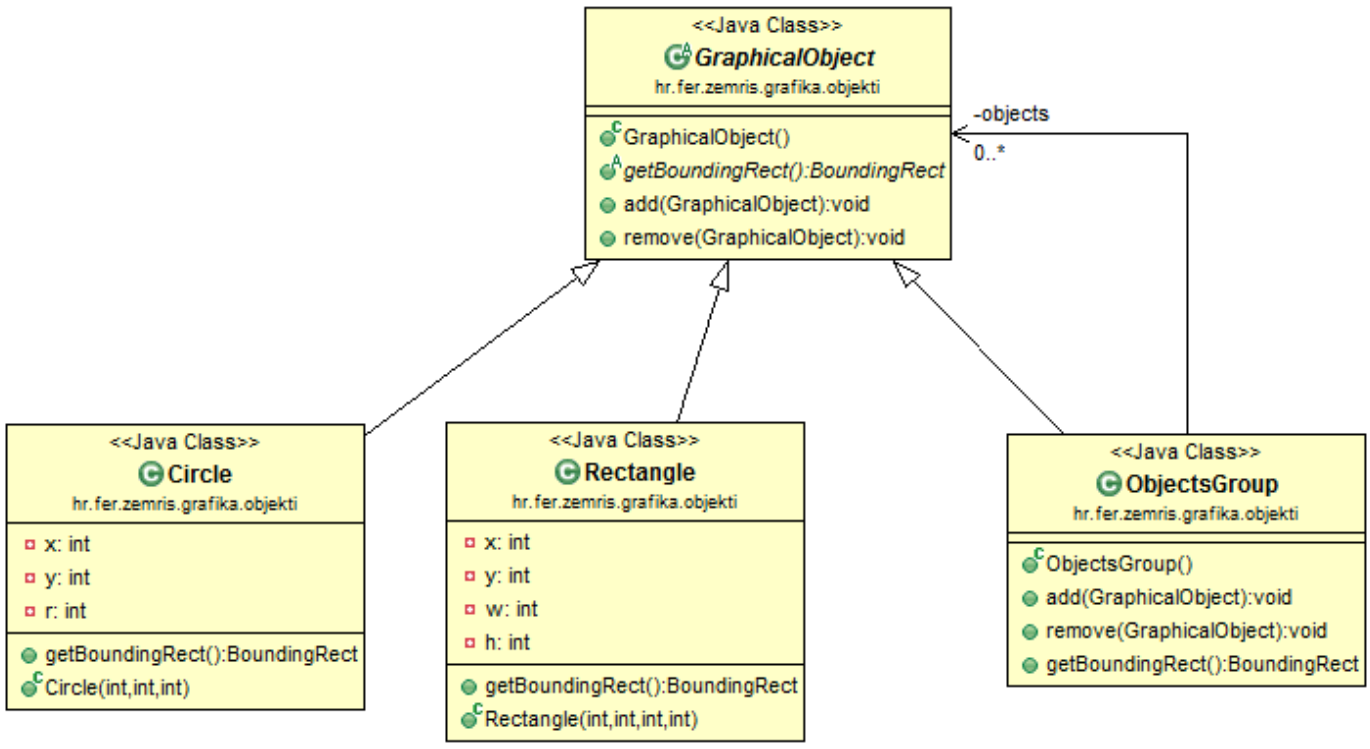
```
public class ObjectsGroup extends GraphicalObject {

	private List<GraphicalObject> objects = new ArrayList<>();

	public void add(GraphicalObject o) {
		objects.add(o);
	}

	public void remove(GraphicalObject o) {
		objects.remove(o);
	}
```

```
	@Override
	public BoundingRect getBoundingRect() {
		Iterator<GraphicalObject> it = objects.iterator();
		BoundingRect result = it.next().getBoundingRect();
		for(;it.hasNext();) {
			BoundingRect r = it.next().getBoundingRect();
			result.left = Math.min(result.left, r.left);
			result.top = Math.min(result.top, r.top);
			result.right = Math.max(result.right, r.right);
			result.bottom = Math.max(result.bottom, r.bottom);
		}
		return result;
	}

}
```

So now we can operate our client either on simple objects or on composite-ones. The next code snippet illustrates this:

```
Circle c = new Circle(100, 100, 20);
Rectangle r = new Rectangle(90, 70, 20, 50);

doStuff(c);
doStuff(r);

ObjectsGroup group = new ObjectsGroup();
group.add(c);
group.add(r);

doStuff(group);
```

Now, with Composite design pattern there are many variations of the same general idea. One of commonly used variants is a variant in which the top-level class is equipped with interface for children management so that from the interface-point-of-view all classes are equals. This is shown on next diagram:

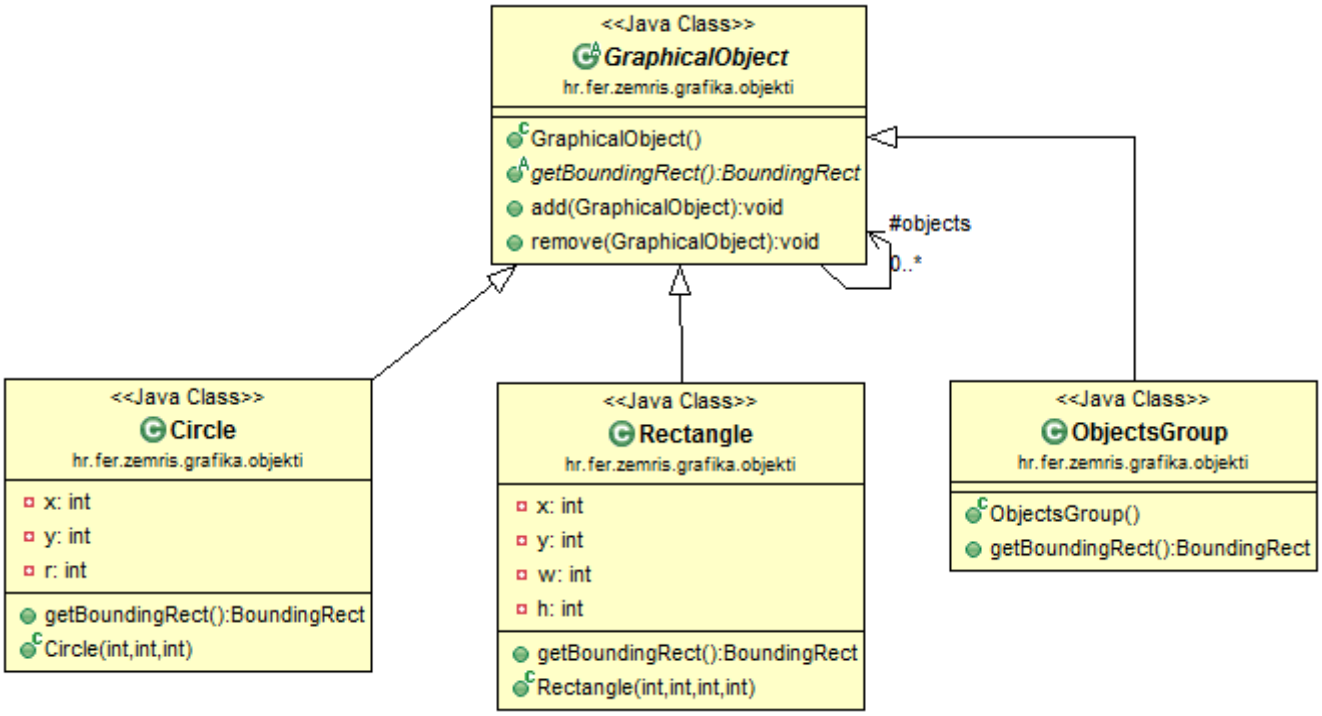


In this scenario, all classes have methods `add` and `remove` since they are declared in `GraphicalObject`. However, in `GraphicalObject` they can be implemented as simply to throw an exception:

```
public abstract class GraphicalObject {

	public abstract BoundingRect getBoundingRect();

	public void add(GraphicalObject o) {
		throw new UnsupportedOperationException("Can not add children!");
	}

	public void remove(GraphicalObject o) {
		throw new UnsupportedOperationException("Can not remove children!");
	}

}
```

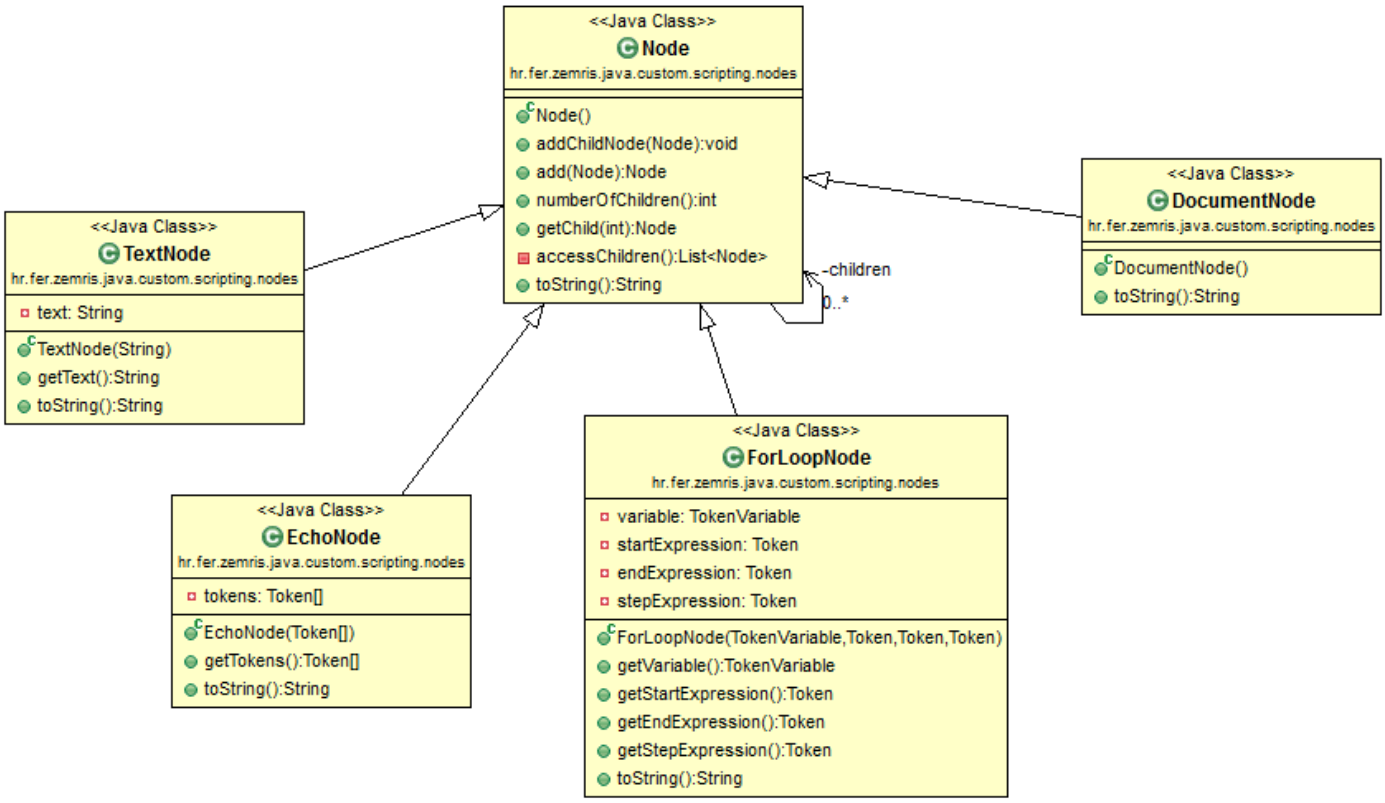
Now `Circle` and `Rectangle` won't override them and only `ObjectsGroup` will declare a private property for actual children storage and override the methods `add` and `remove`.

Finally, there is another variant in which no explicit composite is declared (we do not have out `ObjectsGroup`) but the entire children-management functionality is moved into the top level class (in our case into the `GraphicalObject`) which is illustrated in following diagram:



Now it is the responsibility of leaf-nodes (such as `Circle` and `Rectange`) to disable children addition and removal. Actual operation is still left abstract in `GraphicalObject` and now we can have multiple composites (class `ObjectsGroup1`, class `ObjectsGroup2`, …) which each inherit children management from `GraphicalObject` and only implement concrete operations (in our case `getBoundingRect()`) as appropriate.

When I started the story on Composite design pattern, I sad that “you have most certainly already worked with the Composite pattern (perhaps) without knowing it”. And I wasn't lying: for your 1st homework you have used the composite pattern to store the parsed structure of your script that was written in SmartScript. The class diagram for this case is given next.

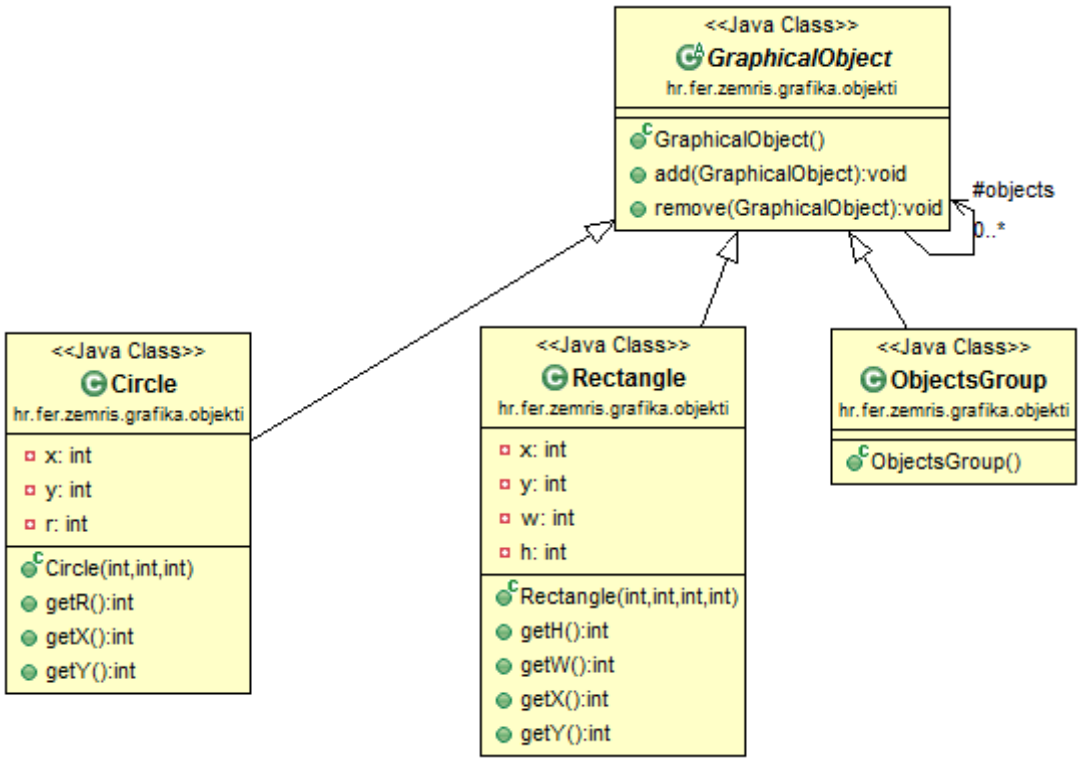


In this case our top-level class `Node` defined and implemented a consistent interface that allowed us to work with any kind of nodes, either the one having children or the one that do not have children.

The Visitor design pattern

Lets return just for a second to our last diagram with `GraphicalObjects`: I allowed the class `ObjectsGroup` to exists simply in order to provide a placeholder for actual implementation of `getBoundingRect()`. Now please note that the calculation of the bounding rectangle is just one of possible operations that we can perform either over a simple object (i.e. circle or rectangle) or over group of objects. There are many other similar operations: finding the graphical object with smallest bounding rectangle and returning that bounding rectangle, finding the graphical object with largest bounding rectangle and returning that bounding rectangle, calculating the total area of union of all objects, calculating the total area of intersection of all objects, calculating the sum of areas for all objects, etc. You may think that some of these operations are just stupid and no one would want to use them. But I will say this: if I'm developing a model of graphical objects, I should be aware that I can not predict all possible ways in which this model could be used. Observe that, in order to add another operation, I should modify all of the classes for any of graphical objects. What we want is to decouple operations from domain objects. And this is place where Visitor design pattern jumps in.

So here is the general idea. We will define an interface that describes our Visitor object: a single Visitor will usually perform a single operation and it will contain a dedicated method for performing this operation on every different domain object. Lets stick to our example with graphical objects. I will work with domain model shown of next image. Observe that I have deleted concrete operations from model (no more `getBoundingRect()` in model).



Having three concrete classes (`Circle`, `Rectangle` and `ObjectsGroup`) we will define an interface:

```
public interface IGraphicalObjectVisitor {

	public void visitObjectsGroup(ObjectsGroup object);

	public void visitCircle(Circle object);

	public void visitRectangle(Rectangle object);

}
```

We will add abstract method `accept(IGraphicalObjectVisitor visitor)` into top-level class of model (i.e. `GraphicalObject`). We will then implement that method in each concrete `GraphicalObject` to call apropriate visitor method with itself as an argument. Here are the modifications:

```
public abstract class GraphicalObject {

	// …

	public abstract void accept(IGraphicalObjectVisitor visitor);

}

public class Circle extends GraphicalObject {

	// …

	@Override
	public void accept(IGraphicalObjectVisitor visitor) {
		visitor.visitCircle(this);
	}

}
```

```
public class Rectangle extends GraphicalObject {

	// …

	@Override
	public void accept(IGraphicalObjectVisitor visitor) {
		visitor.visitRectangle(this);
	}
}

public class ObjectsGroup extends GraphicalObject {

	@Override
	public void accept(IGraphicalObjectVisitor visitor) {
		visitor.visitObjectsGroup(this);
	}

}
```

Now, if we have a reference to some concrete visitor, we can write:

```
IGraphicalObjectVisitor visitor = new ...;

Circle c = new Circle(100, 100, 20);
Rectangle r = new Rectangle(90, 70, 20, 50);

// This will end up as visitor.visitCircle(c)
c.accept(visitor);

// This will end up as visitor.visitRectangle(c)
r.accept(visitor);
```

The only question is how to handle composite objects, i.e. whose responsibility is to traverse recursively into children. Traversal code can be placed into the model itself. This means that we should modify the accept method in `ObjectsGroup`:

```
public class ObjectsGroup extends GraphicalObject {

	@Override
	public void accept(IGraphicalObjectVisitor visitor) {
		visitor.visitObjectsGroup(this);
		for(GraphicalObject g : objects) {
			g.accept(visitor);
		}
	}

}
```

This way visitor has no control on the order in which its visitXXX methods will be called, i.e. on the order the objects of the composite will be visited. However, it is a simple and often utilized solution.

If we have modified the `ObjectsGroup.accept` as above, lets write now a code for a visitor that will calculate the total area of objects.

```
public class CalcAreaVisitor implements IGraphicalObjectVisitor {

	private double area = 0;

	@Override
	public void visitObjectsGroup(ObjectsGroup object) {
	}

	@Override
	public void visitCircle(Circle object) {
		area += object.getR() * object.getR() * Math.PI;
	}

	@Override
	public void visitRectangle(Rectangle object) {
		area += object.getW() * object.getH();
	}

	public double getArea() {
		return area;
	}
}
```

An example to illustrate its usage:

```
void doStuff() {
	Circle c = new Circle(100, 100, 20);
	Rectangle r = new Rectangle(90, 70, 20, 50);

	ObjectsGroup group = new ObjectsGroup();
	group.add(c);
	group.add(r);

	CalcAreaVisitor visitor = new CalcAreaVisitor();
	group.accept(visitor);

	System.out.println("Površina je: "+visitor.getArea());
}
```

If we need better control over the traversal policy, the solution can be to move the traversal code from model into the visitor itself. This can be achieved, of course, only if domain model offers enough information on its children. Assume now that our ObjectsGroup.accept is again simple:

```
public class ObjectsGroup extends GraphicalObject {

	@Override
	public void accept(IGraphicalObjectVisitor visitor) {
		visitor.visitObjectsGroup(this);
	}

}
```

and assume that our model is just a bit more informative:

```
public abstract class GraphicalObject {
	// …

	public int numberOfChildren() {
		return objects.size();
	}
```

```
	public GraphicalObject getChild(int index) {
		return objects.get(index);
	}
}
```

We can write the visitor to have all neccesary traversal code in methods that handle composite objects. Here is the example of our area-calculating visitor again which chooses to traverse children of composite objects from backward:

```
public class CalcAreaVisitor implements IGraphicalObjectVisitor {

	private double area = 0;

	@Override
	public void visitObjectsGroup(ObjectsGroup object) {
		for(int index = object.numberOfChildren()-1; index >= 0; index--) {
			object.getChild(index).accept(this);
		}
	}

	@Override
	public void visitCircle(Circle object) {
		area += object.getR() * object.getR() * Math.PI;
	}

	@Override
	public void visitRectangle(Rectangle object) {
		area += object.getW() * object.getH();
	}

	public double getArea() {
		return area;
	}
}
```

Both of these design patterns are more detailed explained in chapter 17 in book, so I would encourage you to read (and understand) this chapter, before continuing any further.

Actual problem for you to solve

From your second homework copy packages `hr.fer.zemris.java.custom.scripting.nodes` and `hr.fer.zemris.java.custom.scripting.tokens` into your project (and its content, od course). Also copy your implementation of `SmartScriptParser`.

Put an interface `INodeVisitor` in package `hr.fer.zemris.java.custom.scripting.nodes`. It is defined as follows:

```
public interface INodeVisitor {
	public void visitTextNode(TextNode node);
	public void visitForLoopNode(ForLoopNode node);
	public void visitEchoNode(EchoNode node);
	public void visitDocumentNode(DocumentNode node);
}
```

Go through all `Node`-types from package `hr.fer.zemris.java.custom.scripting.nodes` and add appropriate `accept` method in order to build into them a support for Visitor design pattern. Leave traversal logic for Visitors to implement.

Make a package `hr.fer.zemris.java.custom.scripting.demo` and write a program `TreeWriter` that accepts a file name (as a single argument from command line). Your program must open that file (is should be a smart script), parse it into a tree and that reproduce its (aproximate) original form onto standard output. You solved this problem in your first homework but the chances are that you did not use Visitor design pattern. Now you must solve it using the visitor pattern. So create an inner static class `WriterVisitor` for this job. The general usage pattern should be something like this:

```
String docBody = ...;
SmartScriptParser p = new SmartScriptParser(docBody);
WriterVisitor visitor = new WriterVisitor();
p.getDocumentNode().accept(visitor);
// by the time the previous line completes its job, the document should have been
written
// on the standard output
```

Problem 3.

Create a package `hr.fer.zemris.java.custom.scripting.exec`. Copy into it your implementations of `ObjectMultistack` and `ValueWrapper` from your previous homework. In this package add a new class `SmartScriptEngine`. Its job is to actually execute the document whose parsed tree in obtains. Here is the expected usage example.

```
String documentBody = readFromDisk(fileName);
Map<String,String> parameters = new HashMap<String, String>();
Map<String,String> persistentParameters = new HashMap<String, String>();
List<RCCookie> cookies = new ArrayList<RequestContext.RCCookie>();

// put some parameter into parameters map
parameters.put("broj", "4");

// create engine and execute it
new SmartScriptEngine(
      new SmartScriptParser(documentBody).getDocumentNode(),
      new RequestContext(System.out, parameters, persistentParameters, cookies)
).execute();
```

This class should have following structure:

```
public class SmartScriptEngine {

      private DocumentNode documentNode;
      private RequestContext requestContext;
      private ObjectMultistack multistack = new ObjectMultistack();

      private INodeVisitor visitor = new INodeVisitor() {
            // your implementation here...
      };

      public SmartScriptEngine(DocumentNode documentNode, RequestContext
requestContext) {
            // implementation ...
      }

      public void execute() {
            documentNode.accept(visitor);
      }
}
```

So what should your visitor do for each tag?

- For `DocumentNode` it should call accept for all `DocumentNode-s` direct children.
- For `TextNode` it should write the text that node contains using `requestContext`'s `write` method.
- For `ForLoopNode` it should push onto object stack new instance of variable defined in `ForLoopNode` and initialize it with initial value. As long as this value is less then or equal to end value it should make one pass through `ForLoopNode`'s direct children and call accept on them. After a single iteration is done, you should retrieve current value of variable from stack, increment it and update it on stack, then compare it with final value and if it is still less than or equal to final value, proceed to next iteration. Once iterations are done, you should remove one instance of created variable from stack.
- For `EchoNode` create a temporary stack of objects. Go through every `Token` found in this node. If token is some kind of constant, simply push it into stack (tokens value, not the token itself; if token is `TokenConstantString,` you would push the string it contains on the stack). For each token representing a variable you would find the most current variable with that name on object stack (not this temporary stack!), you would peek that variable and on your temporary stack you would push its value. For each token representing operator you would pop two arguments from temporary stack, do the required operation and push the result back onto the temporary stack. You are required to support operation +, -, * and /. Finally, for each token representing a function you would pop required number of arguments from temporary stack, apply the function and push the result back onto the temporary stack. Once you passed through all tokens, you will be left with possibly non-empty temporary stack. For each element found on that temporary stack you would call `requestContext`'s `write` method; you should do this starting with the first element that was pushed on stack (e.g. if you pushed A then B then C, you should call also write with A then B then C although the C will be the topmost element of the stack – the one you would retrieve with pop).

The functions you are required to support are following.

- `sin(x)`; calculates sinus from given argument and stores the result back to stack. Conceptually, equals to: `x = pop(), r = sin(x), push(r)`.
- `decfmt(x,f)`; formats decimal number using given format `f` which is compatible with `DecimalFormat;` produces a string. `X` can be integer, double or string representation of a number. Conceptually, equals to: `f = pop(), x = pop(), r = decfmt(x,f), push(r)`.
- `dup()`; duplicates current top value from stack. Conceptually, equals to: `x = pop(), push(x), push(x)`.
- `swap()`; replaces the order of two topmost items on stack. Conceptually, equals to: `a = pop(), b = pop(), push(a), push(b)`.
- `setMimeType(x)`; takes string `x` and calls `requestContext.setMimeType(x)`. Does not produce any result.
- `paramGet(name, defValue)`; Obtains from `requestContext parameters` map a value mapped for `name` and pushes it onto stack. If there is no such mapping, it pushes instead `defValue` onto stack. Conceptually, equals to: `dv = pop(), name = pop(), value=reqCtx.getParam(name), push(value==null ? defValue : value)`.
- `pparamGet(name, defValue)`; same as `paramGet` but reads from `requestContext persistant parameters` map.
- `pparamSet(value, name)`; stores a `value` into `requestContext persistant parameters` map. Conceptually, equals to: `name = pop(), value = pop(), reqCtx.setPerParam(name, value)`.
- `pparamDel(name)`; removes association for `name` from `requestContext persistentParameters` map.
- `tparamGet(name, defValue)`; same as `paramGet` but reads from `requestContext temporaryParameters` map.
- `tparamSet(value, name)`; stores a `value` into `requestContext temporaryParameters` map.

Conceptually, equals to: `name = pop(), value = pop(), reqCtx.setTmpParam(name, value)`.

- `tparamDel(name)`; removes association for `name` from `requestContext` `temporaryParameters` map.

To help you check if you did the implementation correctly, check the behavior on following scripts.

Script 1. osnovni.smscr

```
{$= "text/plain" @setMimeType $}This is sample text.
{$ FOR i 1 10 1 $}
  This is {$= i $}-th time this message is generated.
{$END$}
{$FOR i 0 10 2 $}
  sin({$=i$}^2) = {$= i i * @sin  "0.000" @decfmt $}
{$END$}
```

With a test program such as this:

```
String documentBody = readFromDisk("osnovni.smscr");
Map<String,String> parameters = new HashMap<String, String>();
Map<String,String> persistentParameters = new HashMap<String, String>();
List<RCCookie> cookies = new ArrayList<RequestContext.RCCookie>();

// create engine and execute it
new SmartScriptEngine(
      new SmartScriptParser(documentBody).getDocumentNode(),
      new RequestContext(System.out, parameters, persistentParameters, cookies)
).execute();
```

you should get output such as:

```
… zaglavlje ...

This is sample text.

  This is 1-th time this message is generated.

  This is 2-th time this message is generated.

  This is 3-th time this message is generated.
  ...

  sin(0^2) = 0.000

  sin(2^2) = 0.070
  ...
```

Script 2. zbrajanje.smscr

```
{$= "text/plain" @setMimeType $}
Računam sumu brojeva:
{$= "a=" "a" 0 @paramGet ", b=" "b" 0 @paramGet ", rezultat=" "a" 0
@paramGet "b" 0 @paramGet + $}
```

With a test program such as this:

```
String documentBody = readFromDisk("zbrajanje.smscr");
Map<String,String> parameters = new HashMap<String, String>();
Map<String,String> persistentParameters = new HashMap<String, String>();
List<RCCookie> cookies = new ArrayList<RequestContext.RCCookie>();
parameters.put("a", "4");
parameters.put("b", "2");

// create engine and execute it
new SmartScriptEngine(
      new SmartScriptParser(documentBody).getDocumentNode(),
      new RequestContext(System.out, parameters, persistentParameters, cookies)
).execute();
```

You should get result:

```
… zaglavlje ...

Računam sumu brojeva:
a=4, b=2, rezultat=6
```

Script 3. brojPoziva.smscr

```
{$= "text/plain" @setMimeType $}
Ovaj dokument pozvan je sljedeći broj puta:
{$= "brojPoziva" "1" @pparamGet @dup 1 + "brojPoziva" @pparamSet $}
```

With a test program such as this:

```
String documentBody = readFromDisk("brojPoziva.smscr");
Map<String,String> parameters = new HashMap<String, String>();
Map<String,String> persistentParameters = new HashMap<String, String>();
List<RCCookie> cookies = new ArrayList<RequestContext.RCCookie>();
persistentParameters.put("brojPoziva", "3");
RequestContext rc = new RequestContext(System.out, parameters, persistentParameters,
cookies);
new SmartScriptEngine(
      new SmartScriptParser(documentBody).getDocumentNode(), rc
).execute();
System.out.println("Vrijednost u mapi: "+rc.getPersistentParameter("brojPoziva"));
```

You should get result:

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=UTF-8


Ovaj dokument pozvan je sljedeći broj puta:
3
Vrijednost u mapi: 4
```

Observe how the value of parameter in persistent map after the execution of program has changed since the program first obtains the old value, then increments it and then stores it back into persistent map.

Script 4. fibonacci.smscr

```
{$= "text/plain" @setMimeType $}Prvih 10 fibonaccijevih brojeva je:
{$= "0" "a" @tparamSet
    "1" "b" @tparamSet
    "0\r\n1\r\n" $}{$FOR i 3 10 1$}{$=
"b" "0" @tparamGet @dup
"a" "0" @tparamGet +
"b" @tparamSet "a" @tparamSet
"b" "0" @tparamGet "\r\n"
$}{$END$}
```

With a test program such as this:

```
String documentBody = readFromDisk("fibonacci.smscr");
Map<String,String> parameters = new HashMap<String, String>();
Map<String,String> persistentParameters = new HashMap<String, String>();
List<RCCookie> cookies = new ArrayList<RequestContext.RCCookie>();

// create engine and execute it
new SmartScriptEngine(
      new SmartScriptParser(documentBody).getDocumentNode(),
      new RequestContext(System.out, parameters, persistentParameters, cookies)
).execute();
```

you should get output such as:

```
… zaglavlje ...

Prvih 10 fibonaccijevih brojeva je:
0
1
1
2
3
5
8
13
21
34
```

In this example we are using temporary parameters for storage of local variables *a* and *b*, which are used for storage of *fibo(i)* and *fibo(i+1)*.

Script 5. fibonaccih.smscr

This is again calculation of Fibonacci numbers, but output is formatted as HTML document.

```
{$= "text/html" @setMimeType $}
<html>
 <head>
  <title>Tablica Fibonaccijevih brojeva</title>
 </head>
 <body>
  <h1>Fibonaccijevi brojevi</h1>
  <p>U nastavku je prikazana tablica prvih 10
     Fibonaccijevih brojeva.</p>

{$= "0" "a" @tparamSet
    "1" "b" @tparamSet $}

  <table>
    <thead>
      <tr><th>Redni broj</th><th>Fibonaccijev broj</th></tr>
    </thead>
    <tbody>
      <tr><td>1</td><td>0</td></tr>
      <tr><td>2</td><td>1</td></tr>
      {$FOR i 3 10 1$}
      {$=
           "b" "0" @tparamGet @dup
           "a" "0" @tparamGet +
           "b" @tparamSet "a" @tparamSet
      $}
      <tr><td>{$= i $}</td><td>{$= "b" "0" @tparamGet $}</td></tr>
      {$END$}
    </tbody>
  </table>
 </body>
</html>
```

You can test it with the program almost identical to the one given with script 4. Output should be an HTML document.

Problem 4.

In package `hr.fer.zemris.java.webserver` you previously created add a new class `SmartHttpServer`. Now you will start to implement a simple but functional web server. We will start by defining several configuration files we will use.

server.properties

```
# On which address server listens?
server.address = 127.0.0.1

# What is the domain name of our web server?
server.domainName = www.localhost.com

# On which port server listens?
server.port = 5721

# How many threads should we use for thread pool?
server.workerThreads = 10

# What is the path to root directory from which we serve files?
server.documentRoot = D:/eclipse_workspaces/tecaj112C/Zadaca11/webroot

# What is the path to configuration file for extension to mime-type mappings?
server.mimeConfig = D:/eclipse_workspaces/tecaj112C/Zadaca11/config/mime.properties

# What is the duration of user sessions in seconds? As configured, it is 10 minutes.
session.timeout = 600

# What is the path to configuration file for url to worker mappings?
server.workers = D:/eclipse_workspaces/tecaj112C/Zadaca11/config/workers.properties
```

mime.properties

```
html = text/html
htm = text/html
txt = text/plain
gif = image/gif
png = image/png
jpg = image/jpg
```

workers.properties

```
/hello = hr.fer.zemris.java.webserver.workers.HelloWorker
/cw = hr.fer.zemris.java.webserver.workers.CircleWorker
```

You can read property files either by using class `java.util.Properties` and its method `load` or you can write your own implementation. Your server should be startable from command line; place main configuration file is in `config` subdirectory, so we can write:

```
java -cp target/classes hr.fer.zemris.java.webserver.SmartHttpServer
./config/server.properties
```

Create in your root project directory (on the same level as the src and pom.xml) directories `config` and `webroot` with appropriate files (all needed configuration files and web content). When uploading your homework, you MUST pack these directories as well, so reviewer can edit paths in

`config/server.properties` and then run your server without and further modifications.

For now, just be aware of properties that can be found in configuration files and of the syntax of those files. Lines that start with '#' are comments. Empty lines are ignorable as well.

Write a skeleton of your web server as follows.

```
public class SmartHttpServer {

	private String address;
	private String domainName;
	private int port;
	private int workerThreads;
	private int sessionTimeout;
	private Map<String,String> mimeTypes = new HashMap<String, String>();
	private ServerThread serverThread;
	private ExecutorService threadPool;
	private Path documentRoot;

	public SmartHttpServer(String configFileName) {
		// … do stuff here …
	}

	protected synchronized void start() {
		// … start server thread if not already running …
		// … init threadpool by Executors.newFixedThreadPool(...); …
	}

	protected synchronized void stop() {
		// … signal server thread to stop running …
		// … shutdown threadpool …
	}

	protected class ServerThread extends Thread {
		@Override
		public void run() {
			// given in pesudo-code:
			// open serverSocket on specified port
			// while(true) {
			//   Socket client = serverSocket.accept();
			//   ClientWorker cw = new ClientWorker(client);
			//   submit cw to threadpool for execution
			// }
		}
	}

	private class ClientWorker implements Runnable {

		private Socket csocket;
		private InputStream istream;
		private OutputStream ostream;
		private String version;
		private String method;
		private String host;
		private Map<String,String> params = new HashMap<String, String>();
		private Map<String,String> tempParams = new HashMap<String, String>();
		private Map<String,String> permPrams = new HashMap<String, String>();
		private List<RCCookie> outputCookies =
                                    new ArrayList<RequestContext.RCCookie>();
		private String SID;
```

```
		public ClientWorker(Socket csocket) {
			super();
			this.csocket = csocket;
		}

		@Override
		public void run() {
		}
	}
}
```

And here is a pseudo-code for `ClientWorker`'s run method:

```
public void run() {
  // obtain input stream from socket
  // obtain output stream from socket
  // Then read complete request header from your client in separate method...
  List<String> request = readRequest();
  // If header is invalid (less then a line at least) return response status 400
  String firstLine = request.get(0);
  // Extract (method, requestedPath, version) from firstLine
  // if method not GET or version not HTTP/1.0 or HTTP/1.1 return response status 400
  // Go through headers, and if there is header “Host: xxx”, assign host property
  //   to trimmed value after “Host:”; else, set it to server’s domainName
  //   If xxx is of form some-name:number, just remember “some-name”-part
  String path; String paramString;
  // (path, paramString) = split  requestedPath to path and parameterString
  // parseParameters(paramString); ==> your method to fill map parameters
  // requestedPath = resolve path with respect to documentRoot
  // if requestedPath is not below documentRoot, return response status 403 forbidden
  // check if requestedPath exists, is file and is readable; if not, return status 404
  // else extract file extension
  // find in mimeTypes map appropriate mimeType for current file extension
  // (you filled that map during the construction of SmartHttpServer from mime.properties)
  // if no mime type found, assume application/octet-stream
  // create a rc = new RequestContext(...); set mime-type; set status to 200
  // If you want, you can modify  RequestContext to allow you to add additional headers
  // so that you can add “Content-Length: 12345” if you know that file has 12345 bytes
  // open file, read its content and write it to rc (that will generate header and send
  // file bytes to client)
}
```

Here are some clarifications. If your server listens on address 127.0.0.1 and on port 5721, you can request something like this (e.g. by writing it in address bar of Mozilla Firefox):

```
http://127.0.0.1:5721/abc/def?name=joe&country=usa
```

The first line of clients request will then look like this:

```
GET /abc/def?name=joe&country=usa HTTP/1.1
```

You should bind this to the variables mentioned above as follows:

```
firstLine = "GET /abc/def?name=joe&country=usa HTTP/1.1"
method = "GET"
requestedPath = "/abc/def?name=joe&country=usa"
version = "HTTP/1.1"
path = /abc/def
paramString = name=joe&country=usa
```

Method `parseParameters` should analyze `paramString`, determine there are two mappings and call:

```
params.put("name", "joe");
params.put("country", "usa");
```

Put in `webroot` folder a sample text file (`sample.txt`), a sample html file (`index.html`) and a sample png image (`fruits.png`).

You have successfully finished this problem if you can open a browser, enter following URLs (assuming host 127.0.0.1 and port 5721) and if you get correct response. Your text file must be displayed as is, your html file must be processed and rendered (you do not want to see HTML tags) and your image should be displayed as image. URLs are:

```
http://127.0.0.1:5721/sample.txt
http://127.0.0.1:5721/index.html
http://127.0.0.1:5721/fruits.png
http://www.localhost.com:5721/sample.txt
http://www.localhost.com:5721/index.html
http://www.localhost.com:5721/fruits.png
```

Important. Please do this before starting the server and before proceeding to next problems.

Linux users: edit file /etc/hosts as root; it probably contains several lines of text, one of which is:

127.0.0.1 localhost

Add below that line new line:

127.0.0.1 www.localhost.com

and save the file.

Windows users: open c:\windows\system32\drivers\etc\hosts in notepad (run as administrator) and do the same. See https://www.howtogeek.com/howto/27350/beginner-geek-how-to-edit-your-hosts-file/ for help.

This will allow us to use symbolic name in addresses instead of IP addresses, so following links should also work:

```
http://www.localhost.com:5721/sample.txt
http://www.localhost.com:5721/index.html
http://www.localhost.com:5721/fruits.png
```

Problem 4b (some refactoring).

Define an interface:

```
package hr.fer.zemris.java.webserver;

public interface IDispatcher {
      void dispatchRequest(String urlPath) throws Exception;
}
```

Modify class `ClientWorker` to implement this interface. Add the following methods:

```
private void internalDispatchRequest(String urlPath, boolean directCall)
     throws Exception {
}

public void dispatchRequest(String urlPath) throws Exception {
  internalDispatchRequest(urlPath, false);
}
```

Refactor the method `run()` from `ClientWorker` so that after it performs all the steps up to (including) parameter extraction and isolation of requested `urlPath` (so that after, for example, `/x/y?a=b&c=d` it determines that `urlPath` is `/x/y`), it calls method `internalDispatchRequest` with determined `urlPath` and `true` as the second parameter. This method, then, will analyze the path and determine how to process it. You will later further expand this method.

Expand the class `RequestContext`. Add to it a read-only property `IDispather` dispatcher, and add the second constructor which accepts all the arguments as the one already present, and then two additional: `Map<String,String> temporaryParameters` and `IDispatcher dispatcher`. This constructor should not make a copy of given temporary parameters map but instead use the given map. Refactor both constructors so that code duplication is avoided.

Problem 5.

Modify the way your web server processes the client request (in method `internalDispatchRequest`). But first, in your document root folder create a subfolder `scripts`. Now find five scripts you used for testing in problem 3:

```
osnovni.smscr
zbrajanje.smscr
brojPoziva.smscr
fibonacci.smscr
fibonaccih.smscr
```

and copy them into that folder `scripts`. This way, these scripts will be accessible to your web server with URL such as:

```
http://www.localhost.com:5721/scripts/osnovni.smscr
http://www.localhost.com:5721/scripts/zbrajanje.smscr?a=3&b=7
http://www.localhost.com:5721/scripts/brojPoziva.smscr
http://www.localhost.com:5721/scripts/fibonacci.smscr
```

Remember the step in which we have extracted the `path` from `requestedPath`? You should check if `path` has extension `smscr`. If it has, instead of treating it as a simple file, you will instead open that file, read it in memory, produce a string out of it, parse it as a *SmartScript* to obtain a document tree and create an instance of `SmartScriptEngine` that will execute your script. When creating `RequestContext`, you will not pass it a `System.out` as output stream but instead a reference to output stream toward your client. This way engine will interpret the script and write response directly to client! How cool is that? :-) Use the newly defined constructor (from 4b) for creating `RequestContext`. Pass `this` (the ClientWorker itself) as `IDispatcher`.

If done correctly, you should observe dynamically generated content right in front of you. Please note that for now, *brojPoziva.smscr* script will always write 1 as result. This is OK (for now).

Problem 6.

Writing *SmartScript* is one way to extend a capabilities of your web server. Now you will focus your attention to another approach. Add a new interface as shown below:

```
package hr.fer.zemris.java.webserver;

public interface IWebWorker {

	public void processRequest(RequestContext context) throws Exception;

}
```

What we did here is we declared an interface toward any object that can process current request: it gets `RequestContext` as parameter and it is expected to create a content for client.

Now create a package `hr.fer.zemris.java.webserver.workers`. Create in it a class `HelloWorker`, as given on next page.

```
package hr.fer.zemris.java.webserver.workers;

import java.io.IOException;
import java.text.SimpleDateFormat;
import java.util.Date;

import hr.fer.zemris.java.webserver.IWebWorker;
import hr.fer.zemris.java.webserver.RequestContext;

public class HelloWorker implements IWebWorker {

	@Override
	public void processRequest(RequestContext context) {
		SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
		Date now = new Date();

		context.setMimeType("text/html");

		String name = context.getParameter("name");
		try {
			context.write("<html><body>");
			context.write("<h1>Hello!!!</h1>");
			context.write("<p>Now is: "+sdf.format(now)+"</p>");
			if(name==null || name.trim().isEmpty()) {
				context.write("<p>You did not send me your name!</p>");
			} else {
				context.write("<p>Your name has "+name.trim().length()
							+" letters.</p>");
			}
			context.write("</body></html>");
		} catch(IOException ex) {
			// Log exception to servers log...
			ex.printStackTrace();
		}
	}

}
```

Do you see what this program is supposed to do? It will create a HTML page with current time displayed and it will give a different message depending if a parameter called “`name`” was provided in URL that started this worker.

In the same package create another worker: `CircleWorker`. Its job is to produce an PNG image with dimensions 200x200 and with a single filled circle. The pseudocode you can use is here:

```
BufferedImage bim = new BufferedImage(200, 200, BufferedImage.TYPE_3BYTE_BGR);

Graphics2D g2d = bim.createGraphics();
// do drawing...
g2d.dispose();

ByteArrayOutputStream bos = new ByteArrayOutputStream();
try {
      ImageIO.write(bim, "png", bos);
      context.write(bos.toByteArray());
} catch (IOException e) {
      e.printStackTrace();
}
```

Now lets go back into the `SmartHttpServer` class – in the construction phase of it. Add to class `SmartHttpServer` another private property:

```
private Map<String,IWebWorker> workersMap;
```

When you process `server.properties` file, observe there is a directive `server.workers` and I have provided you an example of such file. During construction of `SmartHttpServer`, you should open the referenced file, parse each line of it (that is not empty or comment). Each line you will split into `path` and `FQCN` (fully qualified class name). If there are multiple lines with same path you should throw an appropriate exception. When you have `FQCN`, you will assume that instances of that class can be casted to `IWebWorker`. So ask Java Virtual Machine to create a new instance of that class and to return you a reference to it; then you will cast it to `IWebWroker` and put it in `workersMap`: path will be a key and reference to this object will be a value. Here is how you can do it:

```
String path = "...some...path...";
String fqcn = "hr.fer...etc...SomeWorker";

Class<?> referenceToClass = this.getClass().getClassLoader().loadClass(fqcn);
Object newObject = referenceToClass.newInstance();
IWebWorker iww = (IWebWorker)newObject;

workersMap.put(path, iww);
```

In the light of multithreading, please observe that although we will access `workersMap` from multiple threads, we do construction of it in single-threaded environment and later we only read from it so we are safe. However, our implementations of `IWebWorker` are not thread-safe: multiple threads can at the same time call `IwebWorker.processRequest` so our workers should not use class properties without explicit synchronization.

Now you will modify the processing of client's request once more. Add to the `ClientWorker` private `RequestContext context` which is `null` by default. Go into client's `internalDispatchRequest` method and modify this method as follows. Everywhere you need the `RequestContext` instance, pass previously declared private `context`; check if it is `null` and create it when first needed (but remember it so that you can reuse it if needed during the proccessing of current client request. Before you check the extension in requested URL insert a code that checks if the requested `path` is mapped to some `IWebWorker` (consult `workersMap`). If it is, call that worker's `processRequest` and you are done; if it is not, proceed as usual.

With the given configuration I prepared in `workers.properties`, you should now see the results when

accessing:

```
http://www.localhost.com:5721/hello
http://www.localhost.com:5721/hello?name=john
http://www.localhost.com:5721/cw
```

Try this and do not proceed further if this does not work.

The approach I described here is known as *configuration-based*. There is additional variant you will now implement and it is known as *convention-over-configuration* approach. The idea is simple: if we have predetermined conventions that we will obey, we do not have to write configuration files since everything will be exactly there were it is expected to be. So let us agree (you and me) that if the requested URL is of the form such as:

```
http://www.localhost.com:5721/ext/XXX
http://www.localhost.com:5721/ext/XXX?name1=value1&...&namen=valuen
```

then XXX is name of a worker whose class is in package `hr.fer.zemris.java.webserver.workers`. So, for example, if a request is:

`http://www.localhost.com:5721/ext/EchoParams?name1=value1&...&namen=valuen`
we will assume that a class `hr.fer.zemris.java.webserver.workers.EchoParams` exists and that implements `IWebWorker` interface. So the modification I want you to do is this:

- write worker `EchoParams`; it simply outputs back to the user the parameters it obtained formatted as an HTML table
- modify the way the client's requests are processed so that you first check if the request is of the form `/ext/XXX`; if it is, ask JVM to load that class, create an instance of it, cast it to `IWebWorker` and use it to process the request. Otherwise process it as before.

Observe that now, without any change in the configuration files you will be able to call:

```
http://www.localhost.com:5721/ext/EchoParams?name1=value1&...&namen=valuen
```

as well as older workers:

```
http://www.localhost.com:5721/ext/HelloWorker
http://www.localhost.com:5721/ext/CircleWorker
```

However, you can not call:

```
http://www.localhost.com:5721/EchoParams
```

since you did not explicitly map the path `/EchoParams` to any worker.

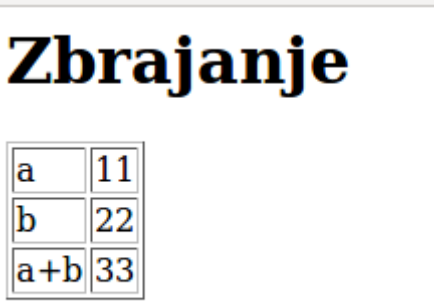
These two approaches are two sides of the same coin: by using approach *convention-over-configuration* you obtain a freedom – easy extensibility without any configuration changes (this is exactly what Maven does with standard directory layout: you are not required to configure in pom.xml where are sources; as long as they are in src/main/java, your project will automatically compile). However, you pay the price: now for each request you are using reflection API to communicate directly with JVM and you instantiate a new instance of your worker for each client's request. Configuration based approach did not have the mentioned penalty since we did the instantiation part only once, at the beginning. If you are considered that this approach means slow server startup, that can be alleviated by using lazy-loading technique: worker could be loaded first time it is needed.

Problem 6b.

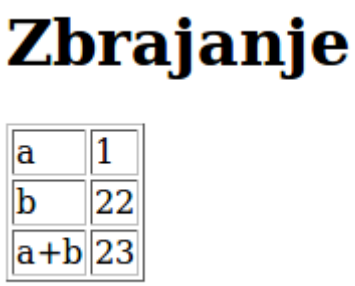
Now you will bind the two approaches of processing request together. First, create a directory `images` in `${server.documentRoot}`. Find on internet two images (in jpg, png or gif format) and put them in this directory.

1. Create a worker `SumWorker` and map it to `/calc`. It accepts two parameters: `a` and `b` and treat their values as integers. If user passed something that is not integer, or if user did not send any of them, use the default value of `1` for `a` and `2` for `b`. This worker will calculate their sum, convert it into string, and place it as *temporary* parameter "`zbroj`" into `RequestContext`. It will also create a temporary parameter `varA` and `varB` with values that were actually used for calculation. Finally, it will create additional *temporary* parameter `imgName` which will hold name of first image if the sum is even, or of the second image, if the sum is odd. Then it will call the dispatcher and ask it to delegate the further processing to the "`/private/pages/calc.smscr`"; the worker will not generate any HTML output by itself!
2. Create the subfolder `private` into `${server.documentRoot}`, in it `pages` subfolder, and in it, create a script `calc.smscr`. This script must create a HTML page with a table, showing `varA`, `varB` and the result that `SumWorker` prepared as temporary parameters. After the table, selected image must be shown.

For example, if user asks for http://127.0.0.1:5721/calc?a=11&b=22, the browser should render something like (only start of the page is shown below; image is missing on screenshot):



If user asks for something like http://127.0.0.1:5721/calc?b=22, the browser should render something like:



Modify the method `internalDispatchRequest` so that, if the requested path is `/private`, or if starts with `/private/` substring, it generates to the client the error status `404` if `directCall` is set to `true`. This way, we will prevent the user from directly calling (from browser) any path which starts with `private`. However, it is OK that some worker performs some computations, and then forwards the html rendering to some script placed in this subfolder. This way, we can separate the processing part (for which workers are excellent) and the rendering part (for which scripts are excellent), which is the basic idea of Model-View-Controller (MVC) Design Pattern applied to web.

Problem 7.

And finally, there remains one more problem to solve. Have you asked yourself why we did not so far speak anything about persistent parameters map? Why is it there? Well, here is the behavior I would like to accomplish. When my browser contacts the server, server serves the request. When the browser contacts the server again, the server does not know that it talked to me just a moment before. What I would like to do is to find some mechanism that will allow the server to track me and the requests I'm issuing (for example, I might want to implement shopping cart for my web shop).

One of mechanisms that HTTP protocol defines exactly for this purposes is a mechanism known as Cookies. Cookie is a small amount of information that server can return to browser and that will browser remember and add to each subsequent request it makes toward the server. Each cookie has it name and it value (it behaves similar to parameters in URL). When a server wishes to store a cookie in client's browser, is adds a `Set-Cookie` directive in response it sends to client. Here is example of such directive:

```
HTTP/1.0 200 OK
Set-Cookie: wishes="strawberry,lettuce"; Domain=www.localhost.com; Path=/webshop; Max-Age: 600
… other headers...
```

With this directive server told the browser to store a cookie named "`wishes`" whose value is "`strawberry,lettuce`"; this cookie is only valid for domain `www.localhost.com`; it should be sent back to server only with requests whose path starts with `/webshop` and is valid for 10 minutes (600 seconds) measuring from this exact moment. For cookies that must not expose to any other channel except HTTP (and HTTPS), server can request that a cookie must be treated as http-only cookie. To indicate this, server must include `HttpOnly` attribute with no value in `Set-Cookie` directive, so the previous example will look like this:

```
HTTP/1.0 200 OK
Set-Cookie: wishes="strawberry,lettuce"; Domain=www.localhost.com; Path=/webshop; Max-Age: 600; HttpOnly
… other headers...
```

HttpOnly-cookies won't be available to JavaScript scriptlets (which can access other cookies using DOM methods and `document.cookie` variable) which makes them preferred choice for cookies which store sensitive data.

If now user clicks to link with address `http://www.localhost.com/webshop/list`, the browser will as part of the request send to server a header `Cookie,` so the request will be something like this:

```
GET /webshop/list HTTP/1.1
Host:  www.localhost.com
Cookie: wishes="strawberry,lettuce"
… other headers...
```

If the server previously set more than one cookie, they will usually be returned in a single `Cookie` header but delimited by ';'. For example:

```
Cookie: wishes="strawberry,lettuce";name="John";country="usa"
```

And now here is what you should do. Go to `SmartHttpServer` and add static inner class `SessionMapEntry`.

```
private static class SessionMapEntry {
	String sid;
	String host;
	long validUntil;
	Map<String,String> map;
}
```

Additionally, add to `SmartHttpServer` two more properties:

```
private Map<String, SessionMapEntry> sessions =
                    new HashMap<String, SmartHttpServer.SessionMapEntry>();
private Random sessionRandom = new Random();
```

We would like to achieve the following. Each time we encounter a new client, we will generate for it a large random identifier that we will call `sid` (session id). It should be, for example, a string that is a concatenation of 20 uppercase letters. For that client we will instantiate one `SessionMapEntry` object, store in it a generated session id, the time until this object is valid (it will be `now` + `session.timeout)` and a new dedicated `map` (pick some thread-safe implementation of map) for storing that clients data. We will store that entry into our `sessions` map we just added as private property of `SmartHttpServer` (store it using `sid` as key). Additionally, we will add a cookie with name `sid` and value of generated session id in our response that will tell browser to remember it and to include it in subsequent request. To achieve this, just add it in a list of `RCCookie`-s that you give to the constructor of `RequestContext.` In this cookie, you will set *domain* to *host*-property you calculated based on the received "Host:" header, and *path* to "/". Leave cookie's *Max-Age* to `null`. This way you are creating what is known as *session cookie* – client's web browser will send it back to the server in each consequent request as long as the client keeps the browser open. Once the client terminates the browser, the browser will forget about all session cookies it had temporarily stored. Make sure to declare this cookie as HttpOnly-cookie. Modify `RequestContext` so that in `getSessionID()` returns `sid` value (pass it in `RequestContext`'s constructor as last parameter).

A note on *Domain* value. If your server can be contacted on several different ways, ie. http://127.0.0.1:port/ and http://someName:port/, you should set the *Domain* value in Set-Cookie to the "Host:" the user has given to the browser, in order for cookies to work properly. And DO NOT include port – just the host. For this, we have already described the algorithm: you inspect the Host: header the browser has sent (and strip the port, if present). Only if the Host: header is not present, use the domainName configured in property file which the server user during the booting.

Now, when you process clients request, before doing anything else (before calling `parseParameters`) call the method `checkSession` with a list of header lines. That method should do the following:

- go through header lines
- if line does not starts with `"Cookie:"`, skip it
- look what cookies you have got
- if there is a cookie named `"sid"`, remember its value in tmp variable `sidCandidate`

If you did not find a `sidCandidate`, create a new unique `sid` and store new object in `sessions` map; add a cookie to response.

If you did found a `sidCandidate`, go into `sessions` map and obtain associated `SessionMapEntry` object.

- If stored host does not match host calculated based on "Host:"-header, proceed just as if you did not find a `sidCandidate` (described previously); otherwise check second bullet
- If that object is invalid (valid field is too old), remove this object and proceed just as if you did not find a `sidCandidate` (described previously).

Finally, if you do have a valid `SessionMapEntry` object, update its property `validUntil` by setting it to `now + session.timeout`.

In any case, at this point, you have in your `sessions` map a valid `SessionMapEntry` object. Set ClientWorker's `permPrams` property to the map from the `SessionMapEntry` object you just retrieved (lets repeat: set it to point to toward to map from the `SessionMapEntry` object you just retrieved, do not create a new map object and copy content from map from the `SessionMapEntry` object; this is important since it will assure that is worker or script modifies something in `permPrams`, our session will remember these modificiations and they will be visible during subsequent client requests).

Important: this whole process of checking if we have registered `SessionMapEntry` object in `sessions` map, creating a new one if needed, generating a new random `sid` by using `sessionRandom` and similar must be treated as a single atomic operation: you can not allow two clients to simultaneously access and modify `sessions` map, so take appropriate care!

How to check implementation

If you implemented this correctly, when you point your browser to address:

```
http://www.localhost.com:5721/scripts/brojPoziva.smscr
```

and when you press reload several times, your script will correctly start incrementing the number that page has been called. Of course, if you open new browser (*not new tab in the same browser*!) and point it to the same address (lets say the first one was Firefox and now you have opened Chrome) the counter for that new client will start from 1; each client will have its own session id and our server will keep a separate map for their data, so on each reload, only the counter for one of browser will be updated. This is also true for using different server “addresses” in same browser. Addresses:

```
http://127.0.0.1:5721/scripts/brojPoziva.smscr
http://www.localhost.com:5721/scripts/brojPoziva.smscr
```

should have independent counters since `127.0.0.1` and `www.localhost.com` do not share sessions.

Important

Please note: in order to avoid excessive memory consumption by expired sessions, add a new daemonic thread that will periodically (e.g. each 5 minutes) go through all session records and that will remove records for expired sessions from `sessions` map.

Warning (once again!): for cookie management to work as expected, what you set for cookie domain must be the same as what you use for accessing your web server from web browser. For example, if you use URL as:

```
http://127.0.0.1/something
```

you must set cookie's domain to `127.0.0.1`; but if you use:

```
http://localhost/something
```

you must set cookie's domain to `localhost`.

Problem 8.

Create new worker `Home` and map it to `/index2.html`. This worker should check if in persistent map exists “”`bgcolor`”: if it does, it should take its value and copy it under name “`background`” into temporary parameters. If it does not exist, it should put value “`7F7F7F`” into temporary parameters under name “`background`”. Then, it should delegate further processing to script `/private/pages/home.smscr`.

Create file `home.smscr` to be HTML document which background color is determined by the value of the temporary parameter “`background`”. For this, use CSS, so that the structure of your document is like the following:

```
<!DOCTYPE html>
<html>
  <head>
    ...
    <style>
      body {background-color: #00FF00;}
    </style>
  </head>
  <body>
   ...
  </body>
</html>
```

Your script will insert correct color here

On this page, display HTML links (tag A) to scripts:

```
osnovni.smscr
brojPoziva.smscr
fibonaccih.smscr
```

and to workers:

```
HelloWorker
CircleWorker
```

Additionally, on that page, add an html form with attribute `method="GET"` (see `FORM` tag), that will allow the user to provide two numeric parameters *a* and *b* (see `INPUT` tag), and add *submit* button (see `INPUT` tag) which will pass given parameters to `SumWorker`.

On the same HTML page add another html form which will offer user to select one color from a predetermined list of colors (for example, white, orange, red, green) and which will, when submit button is activated, send the hex-encoded RGB values of selected color as a parameter named *bgcolor* to `BgColorWorker` mapped on `/setbgcolor`. To accomplish this, you will need to use:

```
<SELECT name="bgcolor">
```

and a number of something like:

```
<OPTION value="FF0000">Red</OPTION>
```

(see `SELECT` and `OPTION` tags).

Then create `BgColorWorker`, and implement it to performs the following:

1. check if parameter *bgcolor* was received and if it is valid hex-encoded color (a string containing exactly 6 hex-digits)
    1. if yes, store the received value in persistent map under name *bgcolor*; generate HTML document with link to `/index2.html` and message that the color is updated
    2. if no, ignore everything; generate HTML document with link to `/index2.html` and message that the color is **not** updated

How to check implementation

If you implemented this correctly, as long as users current session is not expired, when loading `/index2.html`, page will be rendered with background color that was last set with `BgColorWorker`. To test this, open two browsers (eg. Mozilla Firefox and Chrome) and visit `/index2.html`. In each browser update the color. If everything is implemented correctly, the two browsers should not interfere: color set in first browser will be used only in that browser; color set in other browser will be used only in other browser when rendering `/index2.html`.

If you have done all this, you are, finally, done. And, you are ready for next lesson.

Važno. Smijete se konzultirati s Vašim kolegama i razmijenjivati ideje od trenutka kada ste pročitali uputu pa sve do trenutka kada krenete rješavati zadatake. Od trenutka kada napišete prvu liniju koda rješenja daljnja komunikacija je zabranjena. Naravno, u slučaju bilo kakvih nejasnoća, mene uvijek možete kontaktirati.

Dokumentirajte Vaš kod. Projekt i sav izvorni kod mora biti pohranjen uporabom kodne stranice UTF-8.

Kada ste završili sa zadaćom, zapakirajte projekt u ZIP-arhivu imena `hw03-0000000000.zip` (zamijenite nule Vašim stvarnim JMBAG-om). U ZIP-arhivu **ne pakirate** datoteke koje nastaju prevođenjem (tipa: direktorij `target`). Međutim, u arhivu trebate zapakirati direktorij s konfiguracijom te sa svim skriptama koje ste napisali. Ideja je osigurati da se Vaša zadaća može raspakirati, pokrenuti i odmah provjeriti iz bilo kojeg preglednika. To znači da staze koje ćete upisivati u konfiguracijske datoteke trebaju biti relativne s obzirom na korijenski direktorij projekta domaće zadaće.

Arhivu na Ferka uploadajte **prije** isteka roka. Ako ne stignete predati rješenje do tog roka, na zasebnom mjestu u Ferku bit će omogućena zakašnjela predaja koja će se prihvaćati još najviše 72 sata nakon izvornog roka (uvjeti su specificirani u početnoj prezentaciji).

Nemojte zaboraviti zaključati Vaš upload jer ga inače nećemo prihvatiti.