6. homework assignment; JAVA, Academic year 2014/2015; FER

First: read last page. I mean it! You are back? OK. This homework consists of two problems.

1. Izrada jednostavnog računalnog sustava

Sve razrede koje napišete u okviru rješenja ovog zadatka potrebno je smjestiti u odgovarajuće podpakete paketa `hr.fer.zemris.java.simplecomp.impl`.

1.1 Priprema

Na Ferku u repozitoriju postoji kategorija "Dodatci domaćim zadaćama"/"Uz domaću zadaću 06" i tamo ćete pronaći datoteku:

`primjeri.zip`

Preuzmite tu datoteku. Ona sadrži primjere asemblerskih programa za računalo koje gradite u ovom zadatku.

Napravite u Eclipse-u (ili u naredbenom retku, što Vam je lakše) novi Maven projekt. Njegov groupId mora biti `hr.fer.zemris.java.studentVASJMBAG.hw06`. Primjerice, student čiji je JMBAG 001234567890 definirat će da je groupId jednak `hr.fer.zemris.java.student001234567890.hw06`. Kao ime projekta (artifactId) koristite `HW06Problem1`.

Otvorite datoteku pom.xml. Na dno datoteke najprije dodajte referencu na (nestandardni) javni repozitorij u koji sam uploadao biblioteke koje ćete koristiti pri rješavanju ovog zadatka, kako je prikazano u nastavku.

```
<repositories>
  <repository>
    <id>ferko-mvnrepo.internal</id>
    <name>Internal Release Repository</name>
    <url>http://ferko.fer.hr/mvnrepo/repository/internal/</url>
  </repository>
</repositories>
```

Trebat ćete dvije biblioteke:

1) hr.fer.zemris.java.simplecomp:computer-models:1.1
2) hr.fer.zemris.java.simplecomp:parser:1.1

(za obje sam naveo groupId:artifactId:version). Otiđite na Ferka do javne tražilice ovog repozitorija koja je dostupna na adresi:
https://ferko.fer.hr/mvnrepo/
i pronađite isječak koji morate dodati u <dependencies>...</dependencies> sekciju. Snimite pom.xml i po potrebi (ako radite u Eclipse-u) desni klik na naziv projekta u Eclipse-u u Project Navigatoru, pa Maven, Update project....

Jednom kada ste dodali ove ovisnosti, spremni ste za daljnji rad.

Biblioteka `parser` sadrži implementaciju parsera za asemblerski kôd pisan prema formatu koji je opisan u nastavku ovog dokumenta. Biblioteka `computer-models` sadrži definicije potrebnih sučelja. Eclipse možete zatražiti da skine i izvorne kodove biblioteka čime bi prikaz dokumentacije za te biblioteke morao proraditi

(ako ne bude radio odmah).

1.2 Zadatak

Cilj ovog zadatka je upoznati se još malo bolje sa sučeljima i njihovom uporabom. Stoga ćete tijekom izrade ovog zadatka napraviti jednostavnu implementaciju "mikroprocesora" koji može izvršavati jednostavne programe. Primjerice, neki jednostavan program mogao bi izgledati ovako:

```
#Ovaj program 3 puta ispisuje "Hello world!"

            load r0, @brojac        ; ucitaj 3 u registar r0
            load r1, @nula          ; ucitaj 0 u registar r1
            load r7, @poruka        ; ucitaj poruku u r7
@petlja:    testEquals r0, r1       ; je li r0 pao na nulu?
            jumpIfTrue @gotovo      ; ako je, gotovi smo
            decrement r0            ; umanji r0
            echo r7                 ; ispisi na konzolu poruku
            jump @petlja            ; skoci natrag u petlju
@gotovo:    halt                    ; zaustavi procesor

#podaci koje koristimo u programu

@poruka:    DEFSTR "Hello world!\n" ; poruka na jednoj mem. lokaciji
@brojac:    DEFINT 3                ; broj 3 na drugoj mem. lokaciji
@nula:      DEFINT 0                ; broj 0 na trecoj mem. lokaciji
```

U okviru ovog zadatka računalni sustav i njegove komponente modelirane su kroz niz sučelja. Parser za program napisan asemblerskim jezikom prikazanim u prethodnom primjeru nalazi se u biblioteci koju koristite i ne trebate ga sami pisati. Parser, međutim, ne zna koje sve instrukcije postoje i kako ih treba izvršavati; ono što parser razumije je sintaksa asemblerskog jezika: što su komentari, gdje se nalazi naziv instrukcije, gdje su parametri i slično. Primjerice: pogledajte prvi redak koji sadrži naredbu `load`. Analizom tog retka, parser će zaključiti da je potrebno stvoriti naredbu koja se zove `load` te koja ima dva operanda: prvi je registar r0 a drugi je adresa memorijske lokacije (tj. cijeli broj) na kojoj se nalazi zapisana "varijabla" brojač (potražite ga u kodu i uočite da je njegova početna vrijednost postavljena na 3). Jednom kada pročita redak, prevodioc će pokušati pronaći razred koji implementira sučelje `Instruction` i koji predstavlja implementaciju ove instrukcije. Potom će pozvati konstruktor tog razreda kako bi dobio primjerak koji će predstavljati tu konkretnu instrukciju, i njega će pohraniti u memoriju mikroračunala. Vaš će zadatak, između ostaloga, biti i pisanje implementacija svih potrebnih instrukcija.

O mikroračunalu

Naše se računalo sastoji od uobičajenih dijelova: registara i memorije. Od registara, na raspolaganju su nam registri opće namjene, programsko brojilo te jedna zastavica. Za razliku od konvencionalnih procesora, naši registri opće namjene (kao i sama memorija) umjesto jednostavnih brojeva mogu pamtiti proizvoljne Java objekte. Memorija, dakle, nije niz okteta, već niz objekata. Na svaku lokaciju možemo staviti referencu na proizvoljno veliki objekt.

Zadatak 1.2.1.

U paketu `hr.fer.zemris.java.simplecomp.models` nalaze se sučelja `Computer`, `Memory` te `Registers`. Proučite ova sučelja (svako sučelje dodatno je dokumentirano) i za svako napišite jedan razred koji ih implementira. Ova sučelja dostupna su u biblioteci `computer-models`. Razrede nazovite kao i samo sučelje i nadodajte `Impl` na kraju (tako će Vaš razred `ComputerImpl` implementirati sučelje `Computer`). Ove implementacije smjestite u paket `hr.fer.zemris.java.simplecomp.impl`. Za čuvanje registara opće namjene te za memoriju koristite obično polje. Implementacija memorije trebala bi

imati konstruktor koji prima broj memorijskih lokacija (tj. veličinu memorije):

```
public MemoryImpl(int size) { ... }
```

a implementacija registara trebala bi imati konstruktor koji prima broj registara koji će procesoru biti na raspolaganju:

```
public RegistersImpl(int regsLen) { ... }
```

Jednom kada su definirani razredi potrebni za rad našeg računala, možemo pogledati kako se to računalo može koristiti. U paketu `hr.fer.zemris.java.simplecomp` stvorite razred `Simulator` i u njegovu metodu `main` stavite sljedeći kôd.

```
// Stvori računalo s 256 memorijskih lokacija i 16 registara
Computer comp = new ComputerImpl(256, 16);

// Stvori objekt koji zna stvarati primjerke instrukcija
InstructionCreator creator = new InstructionCreatorImpl(
      "hr.fer.zemris.java.simplecomp.impl.instructions"
);

// Napuni memoriju računala programom iz datoteke; instrukcije stvaraj
// uporabom predanog objekta za stvaranje instrukcija
ProgramParser.parse(
   "examples/asmProgram1.txt",
   comp,
   creator
);

// Stvori izvršnu jedinicu
ExecutionUnit exec = new ExecutionUnitImpl();

// Izvedi program
exec.go(comp);
```

Prethodni primjer sastoji se od nekoliko cjelina:

1. stvaranje novog računala,
2. stvaranje objekta pomoću kojeg će parser stvarati primjerke instrukcija,
3. prevođenje programa zapisanog u tekstualnoj datoteci i njegovo punjenje u memoriju računala,
4. stvaranje izvršne jedinice (tj. sklopa koji će izvoditi program) te
5. pokretanja izvršne jedinice.

Ako ste riješili zadatak 1.2.1, prva cjelina će raditi. U drugoj liniji poziva se parser iz JAR datoteke čiji je zadatak obraditi datoteku s programom. Kako sam parser ne zna na koji način stvoriti odgovarajuću instrukciju, potrebna mu je pomoć. Tu u igru ulazi razred `InstructionCreatorImpl` čija Vam je implementacija također unaprijed dana i dostupna je u istom paketu u kojem se nalazi i sam parser. Zadaća ovog razreda je stvoriti primjerak razred koji implementira instrukciju koju je parser pronašao u kodu. Parser s ovim razredom može razgovarati jer sam razred implementira sučelje `InstructionCreator` (pogledajte izvorni kod tog sučelja). Svaki puta kada parser otkrije neku instrukciju, poziva metodu:

```
public Instruction getInstruction(
  String name, List<InstructionArgument> arguments
);
```

Prvi argument je naziv pronađene instrukcije a kao drugi argument se predaje lista argumenata instrukcije koje je parser pronašao u datoteci.

Npr. kod obrade retka u kojem piše:

```
mul r0, r1, r2
```

ime će biti "`mul`", a lista će sadržavati tri objekta (za svaki registar po jedan; za detalje pogledati izvorni kod sučelja `InstructionArgument`).

Kako bi Vam se olakšao rad, uporabom ponuđenog razreda `InstructionCreatorImpl` dovoljno (*i nužno*) je razred koji implementira svaku instrukciju smjestiti u paket koji se predaje u konstruktoru od korištenog razreda `InstructionCreatorImpl`. Ova implementacija očekuje da će razredi koji predstavljaju instrukcije biti nazvani `Instr`*`Ime`*; primjerice, razred koji predstavlja instrukciju `mul` imat će ime `InstrMul`, razred koji predstavlja instrukciju `add` imat će ime `InstrAdd`, itd. Pri tome razred `InstructionCreatorImpl` očekuje da će svaki razred koji predstavlja implementaciju neke instrukcije imati jedan javni konstruktor koji prima listu argumenata. Koristeći taj konstruktor razred `InstructionCreatorImpl` će stvarati primjerke instrukcije predajući mu kao jedini argument upravo onu listu koju je parser pripremio i predao metodi `getInstruction` kao drugi argument. Primjerice, instrukcija `mul r0, r1, r2` koja uzima sadržaje registara `r1` i `r2`, množi ih i rezultat pohranjuje u `r0` može se implementirati ovako:

```
package hr.fer.zemris.java.simplecomp.impl.instructions;

import java.util.List;
import hr.fer.zemris.java.simplecomp.models.Computer;
import hr.fer.zemris.java.simplecomp.models.Instruction;
import hr.fer.zemris.java.simplecomp.models.InstructionArgument;

public class InstrMul implements Instruction {
	private int indexRegistra1;
	private int indexRegistra2;
	private int indexRegistra3;

	public InstrMul(List<InstructionArgument> arguments) {
		if(arguments.size()!=3) {
			throw new IllegalArgumentException("Expected 3 arguments!");
		}
		if(!arguments.get(0).isRegister()) {
			throw new IllegalArgumentException("Type mismatch for argument 0!");
		}
		if(!arguments.get(1).isRegister()) {
			throw new IllegalArgumentException("Type mismatch for argument 1!");
		}
		if(!arguments.get(2).isRegister()) {
			throw new IllegalArgumentException("Type mismatch for argument 2!");
		}
		this.indexRegistra1 = ((Integer)arguments.get(0).getValue()).intValue();
		this.indexRegistra2 = ((Integer)arguments.get(1).getValue()).intValue();
		this.indexRegistra3 = ((Integer)arguments.get(2).getValue()).intValue();
	}

	public boolean execute(Computer computer) {
		Object value1 = computer.getRegisters().getRegisterValue(indexRegistra2);
		Object value2 = computer.getRegisters().getRegisterValue(indexRegistra3);
		computer.getRegisters().setRegisterValue(
				indexRegistra1,
				Integer.valueOf(
					((Integer)value1).intValue() * ((Integer)value2).intValue()
				)
		);
		return false;
	}
}
```

Zadatak 1.2.2.

Napišite implementacije instrukcija:

`load rX, memorijskaAdresa`

koja uzima sadržaj memorijske lokacije (dobit će to kao broj u drugom argumentu) i pohranjuje taj sadržaj u registar `rX` (index će dobiti kao broj u prvom argumentu),

`echo rX`

koja uzima sadržaj registra `rX` i ispisuje ga na ekran (pozivom metode `System.out.print()`), te

`halt`

koja zaustavlja rad procesora.

Zadatak 1.2.3

Napišite razred koji implementira sučelje `ExecutionUnit`. Za pseudo-kod pogledajte u izvorni kod tog sučelja. Modificirajte razred `Simulator` tako da kao jedini argument naredbenog retka prima stazu do datoteke s asemblerskim kodom programa koji je potrebno prevesti i pokrenuti. Ako taj argument nije prisutan, onda program treba korisnika pitati da utipka putanju do datoteke.

Prilagodite metodu `main` tako da implementira tu logiku (u trenutnom primjeru putanja je bila "hardkodirana").

Stvorite datoteku `examples/prim1.txt` sljedećeg sadržaja:

```
          load r7, @poruka    ; ucitaj poruku u r7
          echo r7             ; ispisi na konzolu poruku
          halt                ; zaustavi procesor

@poruka:  DEFSTR "Hello world!\n"
```

Prilikom prevođenja ovog programa, parser će u memoriju na lokaciju 0 pohraniti instrukciju `load` (odnosno primjerak vašeg razreda `InstrLoad`), na lokaciju 1 instrukciju `echo`, na lokaciju 2 instrukciju `halt` te na lokaciju 3 string `"Hello world!\n"` (za ovo posljednje je zaslužna direktiva `DEFSTR` – *define string*; integeri se u memoriju pohranjuju s `DEFINT`). Direktive `DEFSTR` i `DEFINT` ne pišete Vi – to parser sam zna obaviti. Također, kod instrukcija koje primaju memorijsku lokaciju, u argumentima Vašeg konstruktora parser Vam neće dati ime te lokacije (tipa `@poruka`) već ćete dobiti stvarnu lokaciju (broj).

Sada biste trebali moći pokrenuti program `Simulator`, i na ekranu dobiti ispis `"Hello world!"`. Ako nešto ne radi, sada je pravo vrijeme za otkriti u čemu je problem.

Konačni cilj

Cilj je napraviti procesor koji će moći "izvrtiti" kodove priložene u primjerima u ZIP datoteci; primjere (odnosno čitav direktorij `examples` iskopirajte u Eclipseov projekt, tako da u direktoriju projekta uz `src` i `target` imate i `examples`). Prvi primjer nekoliko puta ispisuje istu poruku a drugi generira dobro poznati slijed brojeva. Da bi to ostvarili, još Vam trebaju neke instrukcije koje morate ostvariti.

Zadatak 1.2.4

Instrukcija	*Opis*
`add rx, ry, rz`	`rx` ← `ry` + `rz`
`decrement rx`	`rx` ← `rx` - 1
`increment rx`	`rx` ← `rx` + 1
`jump lokacija`	`pc` ← lokacija
`jumpIfTrue lokacija`	ako je `flag=1` tada `pc` ← lokacija
`move rx, ry`	`rx` ← `ry`
`testEquals rx, ry`	postavlja zastavicu `flag` na `true` ako su sadržaji registara `rx` i `ry` isti, odnosno na `false` ako nisu.

Ostvarite ove instrukcije i provjerite rad programa oba priložena programa.

Napišite još i sljedeću instrukciju:

Instrukcija	*Opis*
`iinput lokacija`	Čita redak s tipkovnice. Sadržaj tumači kao `Integer` i njega zapisuje na zadanu memorijsku lokaciju. Dodatno postavlja zastavicu `flag` na `true` ako je sve u redu, odnosno na `false` ako konverzija nije uspjela ili je drugi problem s čitanjem. `[lokacija]` ← pročitani `Integer`

Sada u datoteku `examples/prim2.txt` napišite asemblerski kod programa čijim ćete pokretanjem dobiti ponašanje prikazano u sljedećem primjeru (korisnikovi unosi su prikazani crveno), ispisi programa crno.

```
Unesite početni broj: perica
Unos nije moguće protumačiti kao cijeli broj.
Unesite početni broj:
Unos nije moguće protumačiti kao cijeli broj.
Unesite početni broj: 3.58
Unos nije moguće protumačiti kao cijeli broj.
Unesite početni broj: -23
Sljedećih 5 brojeva je:
-22
-21
-20
-19
-18
```

Problem 2.

Create new Maven project. Use groupId `hr.fer.zemris.java.studentVASJMBAG.hw06` and as name and artifactId `Multistack`.

Create an implementation of `ObjectMultistack`. You can think of it as a `Map`, but a special kind of `Map`. While `Map` allows you only to store for each key a single value, `ObjectMultistack` must allow the user to store multiple values for same key and it must provide a stack-like abstraction for these values. Keys for your `ObjectMultistack` will be instances of the class `String`. Values that will be associated with those keys will be instances of class `ValueWrapper` (you will also create this class). Let me first give you an example.

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

import hr.fer.zemris.java.custom.scripting.exec.ObjectMultistack;
import hr.fer.zemris.java.custom.scripting.exec.ValueWrapper;

public class ObjectMultistackDemo {

	public static void main(String[] args) {
		ObjectMultistack multistack = new ObjectMultistack();

		ValueWrapper year = new ValueWrapper(Integer.valueOf(2000));
		multistack.push("year", year);

		ValueWrapper price = new ValueWrapper(200.51);
		multistack.push("price", price);

		System.out.println("Current value for year: "
		                                + multistack.peek("year").getValue());
		System.out.println("Current value for price: "
		                                + multistack.peek("price").getValue());

		multistack.push("year", new ValueWrapper(Integer.valueOf(1900)));
		System.out.println("Current value for year: "
		                                + multistack.peek("year").getValue());

		multistack.peek("year").setValue(
		            ((Integer)multistack.peek("year").getValue()).intValue() + 50
		);
		System.out.println("Current value for year: "
		                                + multistack.peek("year").getValue());

		multistack.pop("year");
		System.out.println("Current value for year: "
		                                + multistack.peek("year").getValue());

		multistack.peek("year").increment("5");
		System.out.println("Current value for year: "
		                                + multistack.peek("year").getValue());
		multistack.peek("year").increment(5);
		System.out.println("Current value for year: "
		                                + multistack.peek("year").getValue());
		multistack.peek("year").increment(5.0);
		System.out.println("Current value for year: "
		                                + multistack.peek("year").getValue());
	}

}
```

This short program should produce the following output:

```
Current value for year: 2000
Current value for price: 200.51
Current value for year: 1900
Current value for year: 1950
Current value for year: 2000
Current value for year: 2005
Current value for year: 2010
Current value for year: 2015.0
```

Your `ObjectMultistack` class must provide the following methods:

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

public class ObjectMultistack {

	public void push(String name, ValueWrapper valueWrapper) {...}
	public ValueWrapper pop(String name) {...}
	public ValueWrapper peek(String name) {...}
	public boolean isEmpty(String name) {...}

}
```

Of course, you are free to add any private method you need. The semantic of methods `push`/`pop`/`peek` is as usual, except they are bounded to "virtual" stack defined by given name. In a way, you can think of this collection as a map that associates strings with stacks (yes, please do use some O(1) `Map` implementation from *Java Collection Framework* for this). And these virtual stacks for two different string are completely isolated from each other. Please also observe: what is pushed onto and poped from stack are `ValueWrapper` objects (which encapsulate some other object; e.g. integer, double, string).

Your job is to implement this collection. However, **you are not allowed** to use instances of existing class `Stack` from Java Collection Framework. Instead, **you should define your inner static class** `MultistackEntry` that acts as a node of a single-linked list. Then use some implementation of interface `Map` to map names to first instance of `MultistackEntry` class (which has reference to next one for the same key, etc, creating a linked list of `MultistackEntry`-es). Using `MultistackEntry` class you can efficiently implement simple stack-like behaviour that is needed for this homework.

Methods `pop` and `peek` should throw an appropriate exception if called upon empty stack (i.e. map contains no such key).

Finally, you must implement `ValueWrapper` class whose structure is as follows.

- It must have a read-write property `value` of type `Object`.
- It must have a single public constructor that accepts initial value as `Object`.
- It must have four arithmetic methods:
  - `public void increment(Object incValue);`
  - `public void decrement(Object decValue);`
  - `public void multiply(Object mulValue);`
  - `public void divide(Object divValue);`
- It must have additional numerical comparison method:
  - `public int numCompare(Object withValue);`

All four arithmetic operation modify current value. However, there is a catch. Although instances of `ValueWrapper` do allow us to work with objects of any types, if we call arithmetic operations, it should be

obvious that some restrictions will apply at the moment of method call (e.g. we can not multiply a network socket with a GUI window). Here are the rules. Please observe that we have to consider three elements:

1. what is the type of currently stored value in `ValueWrapper` object,
2. what is the type of argument provided, and
3. what will be the type of new value that will be stored as a result of invoked operation.

We define that allowed values for current content of `ValueWrapper` object and for argument are `null` and instances of `Integer`, `Double` and `String` classes. If this is not the case, throw a `RuntimeException` with explanation.

Further, if any of current value or argument is `null`, you should treat that value as being equal to `Integer` with value 0.

If current value and argument are not `null`, they can be instances of `Integer`, `Double` or `String`. For each value that is `String`, you should check if `String` literal is decimal value (i.e. does it have somewhere a symbol '.' or 'E'). If it is a decimal value, treat it as such; otherwise, treat it as an `Integer` (if conversion fails, you are free to throw `RuntimeException` since the result of operation is undefined anyway). Please be careful: for conversion configure a static helper `DecimalFormat` (or something similar) which can be instructed to always use decimal point (so that "33.52" is always parsed correctly, regardless of localization settings active on users platform). Please note: this will not have any effect on string output which is produced when you call `.toString()` on `Double` object – this procedure uses localization settings.

Now, if either current value or argument is `Double`, operation should be performed on `Double`s, and result should be stored as an instance of `Double`. If not, both arguments must be `Integer`s so operation should be performed on `Integer`s and result stored as an `Integer`.

If you carefully examine the output of program `ObjectMultistackDemo`, you will see this happening!

Please note, you have four methods that must somehow determine on which kind of arguments it will perform the selected operation and what will be the result – please do not copy&paste appropriate code four times; instead, isolate it in one (or more) private methods that will prepare what is necessary for these four methods to do its job.

Rules for `numCompare` method are similar. This method does not perform any change. It perform numerical comparison between currently stored value in `ValueWrapper` and given argument. The method returns an integer less than zero if currently stored value is smaller than argument, an integer greater than zero if currently stored value is larger than argument or an integer 0 if they are equal.

- If both values are `null`, treat them as equal.
- If one is `null` and the other is not, treat the `null`-value being equal to an integer with value 0.
- Otherwise, promote both values to same type as described for arithmetic methods and then perform the comparison.

Important notes

This homework is the first homework in which you had to create several projects. In preparation for this homework you had to create three maven projects. For this homework you had to create another two maven projects. You must create a ZIP archive containing all five projects (each in its own folder), and than upload this single ZIP. ZIP arhive must have name `HW06-`*yourJMBAG*`.zip`.

Considering the fact that you have exams this week, you are required to junit-test only the second problem from this homework. I do recommend to junit test first problem as well, but it is not a requirement.

Please note. You can consult with your peers and exchange ideas about this homework *before* you start actual coding. Once you open you IDE and start coding, consultations with others (except with me) will be regarded as cheating. You can not use any of preexisting code or libraries which is not part of Java standard edition (Java SE) unless explicitly allowed or provided by me. You can use Java Collection Framework classes and its derivatives (moreover, I recommend it; except for the Stack in the second problem in which detailed implementation requirements are described). Document your code!

Upload final ZIP archive to Ferko before the deadline. Do not forget to lock your upload or upload will not be accepted. Deadline is April 25th 2015. at 07:00 AM.