Modul 2

Koncepti OO programiranja na primjeru Java aplikacije.

Modul pokriva:

- Objekte i klase
- Intefejse
- Nasljeđivanje
- Pakete
- Polimorfizam
- Abstraktne klase
- Unutrašnje i anonimne klase
- Kolekcije
- Često korištene design patterne: singleton, observer i chain of responsibility

Objekti i klase

Objekti u Javi, kao i u drugim programskim jezicima, se mogu posmatrati kao modeli objekata iz stvarnosti. Objekat karakterišu stanje i ponašanje. Stanje opisuju instance varijable a ponašanje određuju metode.

Klase su nacrti po kojima može da se izradi više objekata.

Klase mogu da sarže tri tipa varijabli:

- **varijable klase**

Deklarisane unutar klase, van svih metoda, s static ključnom riječi. Dostupne svakom objektu koji se kreira iz te klase.

- **instance varijable**

Deklarisane van svih metoda unutar klase. Ove varijable se instanciraju kada se kalsa učitava i dostupne su unutar svake metode ili konstruktora unutar kalse.

- **lokalne varijable**

Deklarisane unutar metoda, konstruktora ili drugih blokova koda unutar klase. Instaciraju se pozivanjem metode koja ih sadrži i uništavaju se kada metoda završi s radom.

Konstruktori kalse

Konstruktori su najvažniji dio svake Java klase i svaka Java klasa ima bar jedan konstruktor. Ako nijedan konstruktor nije deklarisan, Java kompajler koristi osnovni konstruktor.

Klase mogu da imaju više konstruktora.

Osnovno pravilo za konstruktore je da moraju biti istog imena kao klasa u kojoj se nalaze.

Obično se deklarišu s public modifierom ali i druge opcije su validne kao u slučaju kada klasa treba da ima samo jednu instancu, odnosno da se ponaša kao singleton.

```java
package company.people;

public class Request {

	// Varijabla klase
	static int requestNumber;

	// Instance varijable
	private int type;
	private String desription;

	/*
	 * Osnovni konstruktor
	 * Postoji uvijek, bez obzira da li je neki drugi konstruktor deklarisan ili ne
	 */
	public Request(){
		requestNumber++;
	}

	// Konstruktor
	public Request(int type, String description){
		this.type = type;
		this.desription = description;
		requestNumber++;
	}

	// Override toString() metode Object clase
	@Override
	public String toString() {

		//Lokalna varijabla
		String desc = "Request [type=" + type + ", desription=" + desription + "]";

		return desc;
	}

	// getteri i setteri

	public int getType() {
		return type;
	}

	public void setType(int type) {
		this.type = type;
	}

	public String getDesription() {
		return desription;
	}

	public void setDesription(String desription) {
		this.desription = desription;
	}

	public static int getRequestNumber() {
		return requestNumber;
	}
}
```

Interfejsi

Interfejsi su skupovi abstraktnih metoda. Ove abstraktne metode su implementirane u klasi koja implemenitra određeni intefejs. Intefejsi su dosta slični klasama ali imaju nekoliko bitnih razlika.

Osobina	Interfejs	Klasa
Ekstenzija fajla	`.java`	
Broj metoda	proizvoljan	
Ključna riječ	`interface`	`class`
Instancijacija	Ne	Da
Konstruktor	Ne	Da
Modifieri metoda	sve abstract	proizvoljno
Instance varijable	Ne	Da
Višestruko nasljeđivanje	Da	Ne

Interfejsi su implicitno abstraktni, abstrakt ključna riječ se izostavalja prilikom njihove deklaracije. Sve metode unutar interfejsa su implicitno public.

```
package company.people;

public interface ReponsibleManager {
	public abstract void registerWorker(ResponsibleWorker worker);
	public abstract void unRegisterWorker(ResponsibleWorker worker);
	public abstract void vacationForAll(int numOfDays);
	public abstract void allComeToWork(String date);
}
```

Implementacija interfejsa

- Svaka metoda koja je deklarisana u interfejsu i koja se implementira u klasi mora da ima isti potpis što se tiče tipa podataka koje motoda vraća i modifiera koji su korišteni za deklaraciju metode.
- Ako je klasa koja implementira interfejs abstraktna onda se metode interfejsa ne moraju implementirati.
- Klasa može da implementira više interfejsa odjednom.
- Klasa može da naslijedi samo jendu klasu ali da implementira više interfejsa.
- Intefejsi mogu nasljeđivati interefejse.

Postoje i interfejsi bez deklarisanih metoda. Oni se označavaju kao tagging interfesji i najčešće služe za kreiranje zajedničkog interfejsa za ostale interfejse i za proširivanje tipa klase koja ih implementira kroz polimorfizam.

Nasljeđivanje i polimorfizam

Nasljeđivanje je proces u kome jedan objekat preuzima ili nalježuje osobine drugog objekta.
Ovo nasljeđivanje je definisano ključnim riječima **`extends`** i **`implements`** koje dovode dva objekta u IS-A vezu.

```
public class Zivotinja{ }

public class Sisar extends Zivotinja{ }

public class Reptil extends Zivotinja{ }

public class Pas extends Sisar{ }
```

- `Zivotinja` je super klasa za `Sisar` klasu.
- `Zivotinja` je super klasa za `Reptil` klasu.
- `Sisar` i `Reptil` su potklase `Zivotinja` klase.
- `Pas` je potklasa za `Sisar` i `Zivotinja` klasu.

Na osnovu navedenih relacije imamo sljedeće:

- `Sisar` je (IS-A) `Zivotinja`
- `Reptil` je `Zivotinja`
- `Pas` je `Sisar` > `Pas` je `Zivotinja` jer je `Sisar Zivotinja`

Nasljeđivanje i interfejsi

IS-A veza među objektima može se postići i imlementacijom interfejsa.

```
public interface Zivotinja { }

public class Sisar implements Zivotinja{ }

public class Pas extends Sisar{ }
```

Paketi

Svi java source fajlovi su organizovani u pakete tako da paket predstavlja logičku jedinicu unutar aplikacije. Struktura paketa odgovara strukturi foldera na fajl sistemu od kojih se sastoji aplikacija.

Sve klase unutar jednog paketa su međusobno vidljive i dostupne.
Klase koje se ne nalaze u istom paketu mogu pristupiti jedna drugoj preko ključne riječi **`import`** u zaglavlju klase.

Importovati se može i čitav paket, bez navođenja imena pojedinačnih fajlova koristeći znak asterisk (*) koji označava bilo koje ime fajla.

```
// Import samo Developer klase
import company.people.Developer;

// Import svega iz paketa people
import company.people.*;
```

Abstraktne klase

Abstraktne klase su nacrt za neki objekat koji je previše generalan da bi se mogao predstaviti konkretnom klasom.

Abstraktne klase se deklarišu kao i obične klase s tim da se koristi **`abstract`** ključna riječ i takve klase se ne mogu instancirati.

Abstraktne klase idalje mogu imati konstruktore i sve ostale pojedinosti obične klase.

Najčešće se koriste kao parent klase za grupu drugih konkretnih klasa ali su previše abstraktne da se koriste same, kao konkretne klase.

Potklase neke abstraktne klase poprimaju njene osobine i ostvaruju pristup metodama tako što je naslijede.

```
public abstract class Worker {
	protected String name;
	protected String surname;
	protected String phoneNumber;

	// Abstraktna metoda koju sve konkretne child klasemoraju implementirati
	public abstract String getResponsibilities();

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public String getSurname() {
		return surname;
	}

	public void setSurname(String surname) {
		this.surname = surname;
	}

	public String getPhoneNumber() {
		return phoneNumber;
	}

	public void setPhoneNumber(String phoneNumber) {
		this.phoneNumber = phoneNumber;
	}
}
```

Implementacija `Worker` klase preko `Developer` klase

```
public class Developer extends Worker {

	private Language language;

	public Developer(String devName, String devSurname, String devPhoneNumber, Language
language){
		// name, surname i phoneNumber su instance varijable Worker super klase
		name = devName;
		surname = devSurname;
		phoneNumber = devPhoneNumber;
		this.language = language;
	}

	// Implementacija getResponsibilities() abtraktne metode
	@Override
	public String getResponsibilities() {
		return "Doing dev suff";
	}

	public Language getLanguage() {
		return language;
	}

	public void setLanguage(Language language) {
		this.language = language;
	}

	@Override
	public String toString() {
		return "Developer [language=" + language + ", name=" + name
				+ ", surname=" + surname + ", phoneNumber=" + phoneNumber + "]";
	}
}
```

Pravila:

- Ako neka klasa sadrži abstraktnu metodu ta klasa mora biti deklarisana kao abstraktna.
- Ako abstraktnu klasu nasljeđuje neka druga abstraktna klasa abstraktne metode ne moraju biti implementirane.
- Konkretne klase moraju implementirati sve abstraktne metode parent klase.

Unutrašnje i anonimne klase

Unutrašnje klase su klase definisane unutar neke druge klase i mogu imati iste modifiere kao i bilo koje varijable.

```
public class Developer extends Worker {

	private Language language;

	public Developer(String devName, String devSurname, String devPhoneNumber, Language
language){
		name = devName;
		surname = devSurname;
		phoneNumber = devPhoneNumber;
		this.language = language;
	}
```

```
	@Override
	public String getResponsibilities() {
		return "Doing dev stuff";
	}

	public Language getLanguage() {
		return language;
	}

	public void setLanguage(Language language) {
		this.language = language;
	}

	@Override
	public String toString() {
		return "Developer [language=" + language + ", name=" + name
				+ ", surname=" + surname + ", phoneNumber=" + phoneNumber + "]";
	}

	// Unutrašnja klasa
	public static class GenericDeveloper{
		private String name;
		private String surname;

		public GenericDeveloper(String name, String surname){
			this.name = name;
			this.surname = surname;

		}
		public String getName() {
			return name;
		}
		public void setName(String name) {
			this.name = name;
		}
		public String getSurname() {
			return surname;
		}
		public void setSurname(String surname) {
			this.surname = surname;
		}
	}
}
```

Anonimne klase su klase koje se deklarišu u izrazu. Koriste se ako takva klasa treba samo jednom i čine kod preglednijim i kraćim.

Pravila:

- Anonimne klase moraju uvijek nasljediti super klasu ili implementirati interfejs.
- Anonimna klasa mora implemenitrati sve abstraktne metode super klase ili interfejsa.
- Anonimna klasa uvijek koristi konstruktor super klase pri kreiranju instance.

`ResponsibleWorker` interfejs

```
public interface ResponsibleWorker {

	public abstract void takeVacation(int numOfDays);
	public abstract void comeToWork(String date);
}
```

Anonimna klasa koja implementira interfejs

```
public ResponsibleWorker getResponibleWorker(){

	return new ResponsibleWorker(){

		String name = "RW1";

		@Override
		public void takeVacation(int numOfDays) {

		}

		@Override
		public void comeToWork(String date) {

		}

		public String getName() {
			return name;
		}

		public void setName(String name) {
			this.name = name;
		}
	};
}
```

Kolekcije

Kolekcije su jednostavno objekti koji sadrže grupu drugih objekata i predstavljaju ih kao jednu cjelinu.

Kolekcije u javi su framework koji sadrži sljedeće:

- **Interfejse**

Abstraktni tipovi podataka koji predstavljaju kolekcije i omogućuju da se kolekcijama upravlja na sličan način, bez obzira na konkretnu implementaciju, npr. List, Map

- **Implementacije**

Konkretne implementacije interfejsa, strukture podataka koje se mogu koristi na više različitih mjesta u aplikaciji, npr ArrayList, HashMap

- **Algoritme**

Metode koje izvode neke operacije nad grupom objekata, tipa sortiranja i pretrage

Pregeld interfejsa i implementacija

Interfaces	Hash table Impl.	Resizable array Impl.	Tree Implementations	Linked list Implementations	Hash table + Linked list Impl.
`Set`	`HashSet`		`TreeSet`		`LinkedHashSet`
`List`		`ArrayList`		`LinkedList`	
`Queue`					
`Deque`		`ArrayDeque`		`LinkedList`	
`Map`	`HashMap`		`TreeMap`		`LinkedHashMap`

Design patterni

Design patterni su šabloni za rješavanje problema koji se često ponavljaju u programiranju. Oni, također, predstavljaju i skup najboljih praksi u programiranju i razvoju softvera.

Singleton

Koristi se u slučajevima kada je potrebna globalna dostupnost instance neke klase koja je ujedno i jedina instanca te klase.

```
public class StaffManager implements {

	private static StaffManager mStaffManager;

	protected StaffManager(){
		Log.d(TAG, "StaffManager is working...");
	}

	public static StaffManager getManager(){
		if(mStaffManager == null)
			mStaffManager = new StaffManager();

		return mStaffManager;
	}
}
```

Observer pattern

Obeserver pattern predstavlja one-to-many vezu između objekata tako da, kada jedan objekat (subject), promijeni neko od svojih svojstava svi ostali objekti koji prate tu promjenu budu obavješteni automatski.

`ReponsibleManager` interfejs kojeg će implementirati objekat koji obavještava ostale objekte

```
public interface ReponsibleManager {
	public abstract void registerWorker(ResponsibleWorker worker);
	public abstract void unRegisterWorker(ResponsibleWorker worker);
	public abstract void vacationForAll(int numOfDays);
	public abstract void allComeToWork(String date);
}
```

`ResponsibleWorker` interfejs kojeg implementira objekat koji treba da bude obaviješten

```
public interface ResponsibleWorker {

```

```
	public abstract void takeVacation(int numOfDays);
	public abstract void comeToWork(String date);
}
```

Implementacija `ResponsibleManager` interfejsa u `StaffManager` klasi

```
public class StaffManager implements ReponsibleManager {
	public static final String TAG = StaffManager.class.getSimpleName();

	private static StaffManager mStaffManager;
	List<ResponsibleWorker> workers;

	protected StaffManager(){
		workers = new ArrayList<ResponsibleWorker>();
		Log.d(TAG, "StaffManager is working...");
	}

	public static StaffManager getManager(){
		if(mStaffManager == null)
			mStaffManager = new StaffManager();

		return mStaffManager;
	}

	@Override
	public void registerWorker(ResponsibleWorker worker) {
		if(!workers.contains(worker)){
			workers.add(worker);
			Log.d(TAG, "Registered worker "+(Worker)worker);
		}

	}

	@Override
	public void unRegisterWorker(ResponsibleWorker worker) {
		if(workers.contains(worker)){
			workers.remove(worker);
			Log.d(TAG, "unregistered worker "+((Worker)worker).getName());
		}

	}

	@Override
	public void vacationForAll(int numOfDays) {
		for(ResponsibleWorker worker : workers)
			worker.takeVacation(numOfDays);

	}

	@Override
	public void allComeToWork(String date) {
		for(ResponsibleWorker worker : workers)
			worker.comeToWork(date);
	}
}
```

Implementacija `ResponsibleWorker` interfejsa u `JavaDeveloper` klasi

```
public class JavaDeveloper extends Developer implements ResponsibleWorker {

	private String technology;

	public JavaDeveloper(String devName, String devSurname, String devPhoneNumber, Language
```

```
language, String technology) {
		super(devName, devSurname, devPhoneNumber, language);

		this.technology = technology;

		StaffManager.getManager().registerWorker(this);
	}

	@Override
	public String getResponsibilities(){
		return "Java developer programming in "+getTechnology();
	}

	public String getTechnology() {
		return technology;
	}

	public void setTechnology(String technology) {
		this.technology = technology;
	}

	@Override
	public void takeVacation(int numOfDays) {
		Log.d(TAG, getName()+" ("+getLanguage()+") is going to vacations for "+numOfDays);
	}

	@Override
	public void comeToWork(String date) {
		Log.d(TAG, getName()+" ("+getLanguage()+") comming to work at "+date);
	}
}
```

Chain of Responibility pattern

Ovaj pattern se koristi kada je potrebno sekvencijalno procesiranje poruka među objektima tako da su objekti koji procesiraju poruke poredani po nekom prioritetu.

Interfejs kojeg implementiraju klase koje su dio chain-a

```
public interface RequestHandler {

	public abstract void setSuccessor(RequestHandler handler);
	public abstract void handleRequest(Request request);

}
```

```
public class Developer extends Worker implements RequestHandler {

	private Language language;

	RequestHandler successorHandler;

	public Developer(String devName, String devSurname, String devPhoneNumber, Language language){
		name = devName;
		surname = devSurname;
		phoneNumber = devPhoneNumber;
		this.language = language;
	}

	@Override
	public String getResponsibilities() {
		return "Doing dev suff";
	}

	public Language getLanguage() {
		return language;
```

```
	}

	public void setLanguage(Language language) {
		this.language = language;
	}


	@Override
	public String toString() {
		return "Developer [language=" + language + ", name=" + name
				+ ", surname=" + surname + ", phoneNumber=" + phoneNumber + "]";
	}

	@Override
	public void setSuccessor(RequestHandler handler) {
		successorHandler = handler;
	}

	@Override
	public void handleRequest(Request request) {

		if(request.getType() == Request.Type.JS_FIX && this instanceof JavaScriptDeveloper){
			Log.d("Developer", "request "+request+" handled by "+getName()+"
"+this.getClass().getSimpleName());
		} else if(request.getType() == Request.Type.JAVA_FIX && this instanceof JavaDeveloper){
			Log.d("Developer", "request "+request+" handled by "+getName()+"
"+this.getClass().getSimpleName());
		}else {
			if(successorHandler != null)
				successorHandler.handleRequest(request);
		}
	}
}
```

Konstrukcija chain-a

```
public class Company {
	public static final String TAG = Company.class.getSimpleName();

	String name = "";

	public Company(String name){
		this.name = name;
		Log.d(TAG, "Company created");

		Developer javaDev1 = new JavaDeveloper("John1", "Doe", "061123456", Language.JAVA, "Android");

		Developer jsDev1 = new JavaScriptDeveloper("JS dev #1 name", "JS dev #1 surname", "062896543",
Language.JAVASCRIPT, "AngularJS", true);

		ProjectManager manager1 = new ProjectManager("Mujo", "Mujic", "062341234", "Paromlinska bb");

		Worker ceo = new CEO("CEO name", "CEO surname", "033 123 124", 10);

		((RequestHandler)ceo).setSuccessor(manager1);

		manager1.setSuccessor(StaffManager.getManager());

		StaffManager.getManager().setSuccessor(javaDev1);

		javaDev1.setSuccessor(jsDev1);

		((RequestHandler)ceo).handleRequest(new Request(Request.Type.JAVA_FIX, "Popravi bug 18"));
	}

	public static void main(String[] args){

		new Company("Holo");

	}
}
```