

Advanced Java Programming

Generičke klase

ITAcademy

Šta su to generičke klase

- Generička klasa je klasa koja dopušta apstrakciju tipova kojima rukuje
- Ova osobina omogućava povećavanje apstrakcije i u okolini generičke klase
- Praktična prednost generičke klase je to što možemo kreirati logiku ne znajući koji će konkretno tipovi u njoj biti zastupljeni
- Takođe, generička klasa omogućava korisniku klase da odabere koji će tip klasa obrađivati
- Generičke klase često srećemo kod kolekcija, jer su odlično rješenje za izbjegavanje box-inga i unbox-inga.

Šta se rješava generičkom klasom

- Prilikom prolaska kroz kolekciju, enhanced for petljom nije potrebno vršiti konverziju (unboxing)

Mora konverzija

```
ArrayList list1 = new ArrayList();  
ArrayList<Integer> list2 = new ArrayList<>();  
list1.add(10);  
list2.add(20);  
for(Object i : list1){ System.out.println((Integer)i+2); }  
for(Integer i : list2){ System.out.println(i+2); }
```

Ne mora konverzija

ITAcademy

Korisnički definisane generičke klase

- Da bi klasa bila generička mora joj se između oznaka manje i veće, nakon imena, dodati identifikator tipa. Ovaj identifikator može biti bilo koja riječ koja poštuje pravila imenovanja promjenljivih
- Objekat ovakve klase između ostalog zavisi od načina na koji smo je inicijalizovali
- Ako generičku klasu inicijalizujemo bez tipa, tada se objekat naziva raw type objektom

```
//Valid
MyClass mc = new MyClass("Hello");
//Invalid
MyClass<Integer> mc1 = new MyClass<>("hello");
//Invalid
MyClass<Integer> mc2 = new MyClass<Integer>("hello");
//Valid
MyClass<Integer> mc3 = new MyClass<>(10);
//Valid
MyClass<Integer> mc4 = new MyClass<Integer>(20);
System.out.println(mc3);
System.out.println(mc4);
```

Zašto bi koristili generičke klase?

- Pored implementacija generičkih tipova koji su već na raspolaganju u Javi, jedna od čestih upotreba generičke klase je kreiranje DAO objekata

```
public class HibernateDao<E, K extends Serializable> {  
    private SessionFactory sessionFactory;  
    public void add(E entity) {  
        currentSession().save(entity);  
    }  
    public void update(E entity) {  
        currentSession().saveOrUpdate(entity);  
    }  
    public void remove(E entity) {  
        currentSession().delete(entity);  
    }  
}
```

Generičke metode

- Osim klasa, i metode mogu biti generičke

```
public static<A,B> String myMethod(A a, B b){  
    String left = String.valueOf(a);  
    String right = String.valueOf(b);  
    return left+right;  
}  
  
public static void main(String[] args) {  
    System.out.println(myMethod(10,20));  
    System.out.println(myMethod("Hello","World"));  
}
```

Ograničenja generičkih tipova

- Moguće je ograničiti tipove na neku određenu nadklasu.
- Sljedeći primjer ograničava prosljeđeni tip na podklasu klase Numeric

```
package simpleboundedtypes;

public class MyClass<T extends Number> {
    public MyClass(T t){
        System.out.println(t.getClass().getName());
    }
}
```

Nasljeđivanje generičkih klasa

- Generičke klase je moguće naslijediti
- Prilikom nasljeđivanja, generički tip može biti specijalizovan, ili ostati generički:

```
public class MyParentClass<T> {  
    public MyParentClass(T t) {  
        System.out.println(t.getClass().getName());  
    }  
}
```

```
public class MyChildClass<T> extends MyParentClass<T> {  
    public MyChildClass(T t) {  
        super(t);  
    }  
}
```

```
new MyChildClass("hello");
```

```
new MyChildClass(10);
```


Restrikcije u generičkim klasama

<http://docs.oracle.com/javase/tutorial/java/generics/restrictions.html>

- Ne mogu se koristiti sa prostim tipovima
- Generički parametri se ne mogu instancirati
- Ne mogu se definisati statička polja tipa generičkog parametra
- ...

Vježba 1

- Kreirati klasu generičku **Game** koja će imati cijelobrojna polja **width** i **height**, listu različitih objekata tipa `GameObject`, i metod **run**.
- Kreirati apstraktnu klasu `GameObject`, sa konstruktorom koji prihvata objekat klase `Game`, apstraktnim metodama `update` i `draw`, i cjelobrojnim poljima **posx**, **posy**, **width**, **height** i **speed**.
- Kreirati klase **Ball** i **Bat**, koje nasljeđuju klasu `GameObject`
 - U klasi `Ball`, metod `update` pomjera objekat u svim pravcima
 - U klasi `Bat`, metod `update` pomjera objekat lijevo desno
 - `Draw` metode obje klase, treba da prikazuju trenutnu poziciju objekta (x i y) u konzoli
- Nakon aktivacije `run` (klase `Game`), startuje se glavna petlja aplikacije u kojoj se konstantno aktiviraju metode `update` i `draw` objekata u listi (`Ball` i `Bat`)
- **Opciono:** Uz pomoć biblioteke **lanterna** nacrtati i animirati kreirane objekte (<https://code.google.com/p/lanterna/>)

Vježba 2

- Potrebno je kreirati interfejs ICard koji ima metode **getMoney(double amount):void** i **validate:boolean**.
- Potrebno je kreirati klase Master i Visa koje implementiraju interfejs ICard
- Potrebno je kreirati generičku klasu Bank, čiji će generički parametar biti nasljednik interfejsa ICard. Klasa Bank treba da ima metod **pay**, koji poziva metode **validate** i **getMoney**.
- Treba instancirati dva puta klasu Bank. Jednom za Master, a drugi put za Visa tip, a zatim izvršiti metod pay, na obje instance

Događaji

- Događaji su pojam kome će biti posvećena pažnja u ovom, ali i u narednim kursevima. To je jedan od ključnih koncepata u programiranju GUI aplikacija, kako u Javi, tako i na ostalim platformama, odnosno programskim jezicima
- Programiranje koje podrazumjeva praćenje i obradu događaja u toku izvršenja aplikacije naziva se **Event Based** programiranje. Ovaj način rukovanja programom prepoznatljiv je u svim aplikacijama koje sadrže korisničke kontrole (tastere, prozore, tekst boksove...).
- Recimo da želimo da se vozimo automobilom kome rezervoar nije pun. Imali bismo dva načina da budemo sigurni da se nećemo zaustaviti. Jedan je da konstantno provjeravamo stanje u rezervoaru, a drugi da se oslonimo na indikator rezerve goriva. Naravno, druga varijanta je daleko galantnija od prve i to je upravo način na koji funkcionišu i događaji objekata u programiranju.



Događaji unutar objekta

- Analizirajmo sljedeći program

Reservour.java

```
package simplelocalevent;
public class Reservour {
    private int reserveLimit = 10;
    private int totalAmount = 100;
    private void reserveIndicator(){
        System.out.println("Hey, I am on reserve! Please refill me!");
    }
    public void getFuel(){
        if(--totalAmount<=reserveLimit){
            reserveIndicator();
        }
        System.out.println(totalAmount);
    }
}
```

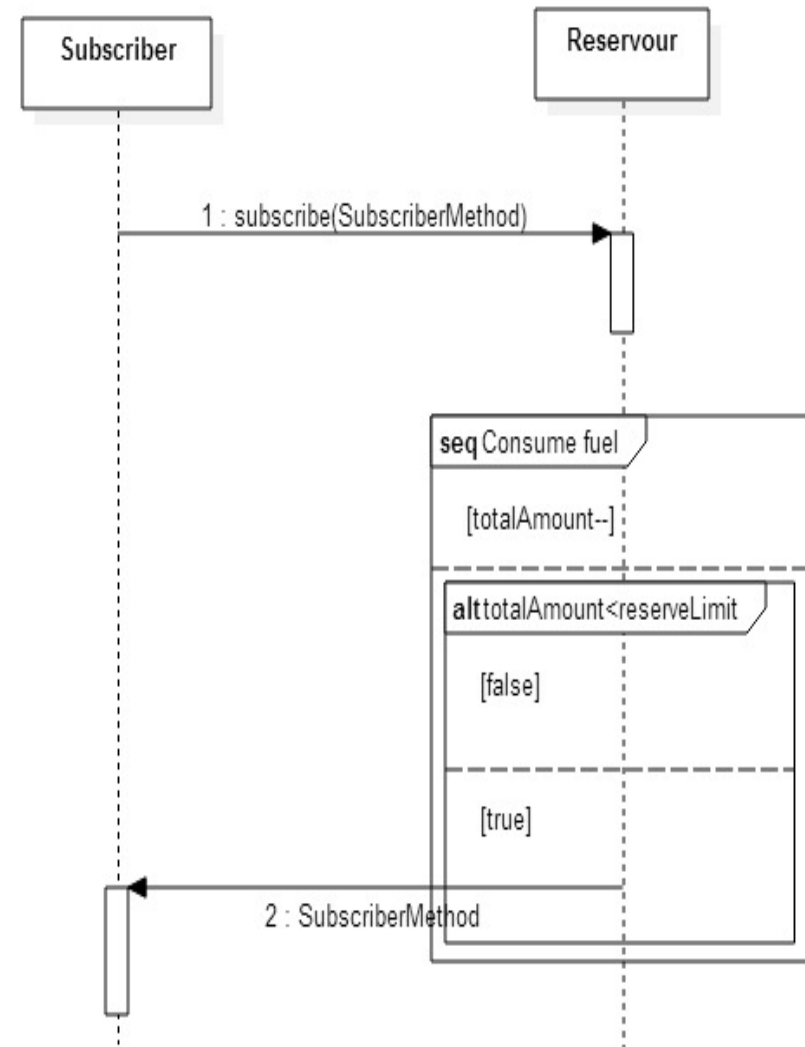
SimpleLocalEvent.java

```
package simplelocalevent;
public class SimpleLocalEvent {
    public static void main(String[] args) {
        Reservour res = new Reservour();
        for(int i=0;i<100;i++){
            res.getFuel();
        }
    }
}
```

- Primjećujemo da je rezervoar svjestan ulaska u rezervu, ali da svijet oko njega nije. Unutar main metode, ne znamo da je rezervoar „na rezervi“ jer je događaj identifikovan i obrađen unutar objekta

Distributer/subscriber model

- Da bi događaj bio vidljiv za objekte izvan objekta u kome se dogodio (**distributera**), oni se moraju pretplatiti na njega
- Da bi se objekat pretplatio na događaj drugog objekta, mora ispuniti uslove. Ti uslovi podrazumjevaju postojanje odgovarajućeg metoda na objektu pretplatniku
- U trenutku pretplate, objekat distributer prijavljuje pretplatnika na događaj (stavlja ga u listu pretplatnika)
- Prilikom detektovanja događaja, distributer prolazi kroz listu pretplatnika i svima im šalje informaciju da je došlo do događaja
- Ovaj koncept poznat je i pod nazivom **observer pattern**



Slušać događaja

- Prvi korak u procesu definicije događaja je konstrukcija slušača događaja. Slušać događaja je najčešće interfejs koji ima jedan ili više metoda za koje će znati distributer i pretplatnici. Slušać događaja je nešto što treba da bude logički vezano za klasu koja će generisati događaj (ili više događaja)

```
package carevents;  
import java.util.EventObject;  
public interface ReservoirListener {  
    public void reserveReached(EventObject evt);  
}
```

Distributer događaja

- Distributer je klasa u kojoj se događaj dogodio. Ova klasa mora imati **listu slušača**, mehanizam za pridruživanje slušača listi (**addListener**), mehanizam za uklanjanje slušača (**removeListener**) i mehanizam za obavješćavanje slušača.

```
import java.util.ArrayList;
import java.util.EventObject;
import java.util.List;
public class Reservoir {
    private List<ReservoirListener> listeners;
    public Reservoir(){ listeners = new ArrayList<>(); }
    public void addEventListener(ReservoirListener lis){ listeners.add(lis); }
    public void removeEventListener(ReservoirListener lis){ listeners.remove(lis); }
    public void distributeEvent(){
        for(ReservoirListener lis : listeners){
            lis.reserveReached(new EventObject(this));
        }
    }
}
```


Aktivacija događaja

- Kada postoji kompletan mehanizam za upravljanje događajima unutar klase, samu aktivaciju događaja treba vršiti po potrebi

```
package carevents;
import java.util.ArrayList;
import java.util.EventObject;
import java.util.List;
public class Reservoir {
    private int currentState;
    private int reserveLimit;
    private List<ReservoirListener> listeners;
    public Reservoir(){
        currentState = 100;
        reserveLimit = 10;
        listeners = new ArrayList<>();
    }
    public void addEventListener(ReservoirListener lis){ listeners.add(lis); }
    public void removeEventListener(ReservoirListener lis){ listeners.remove(lis); }
    public void distributeEvent(){
        for(ReservoirListener lis : listeners){
            lis.reserveReached(new EventObject(this));
        }
    }
    public void consumeFuel(){
        System.out.println("Fuel consumed. " + currentState + " liters remaining");
        if(--currentState<reserveLimit){
            distributeEvent();
        }
    }
}
```

Pretplata na događaj

- Kada je objekat u stanju da detektuje događaj i distribuirati ga pretplatnicima, same pretplatnike možemo (a ne moramo) dodavati prema potrebi

```
package carevents;
import java.util.EventObject;
public class CarEvents {
    public static void main(String[] args) throws InterruptedException {
        Reservoir res = new Reservoir();
        res.addEventListener(new ReservoirListener() {
            @Override
            public void reserveReached(EventObject evt) {
                System.out.println("No more fuel in car. Please refill!");
            }
        });
        for(int i=0;i<100;i++){
            res.consumeFuel();
            Thread.sleep(100);
        }
    }
}
```

Vježba 3

- Aplikacija traži od korisnika x poziciju tenka
- Nakon unosa, računar odabira poziciju svog tenka
- Pozicije oba tenka se zatim ispisuju na izlazu
- Korisnik unosi jačinu i ugao svog sljedećeg hica
- Hitac se ispaljuje i i prati u realnom vremenu, provjeravajući da li je:
 - o Projektil udario u tlo
 - Hitac se smatra završenim i ne dodjeljuju se poeni za pogodak
 - o Projektil udario u neprijateljski tenk
 - Hitac se smatra završenim i dodjeljuju se poeni za pogodak
 - o **Opciono:**
 - Sistem generiše random pozicije na „nebu“, koje predstavljaju ptice
 - Projektil udario u pticu
 - U ovom slučaju, broj poena za taj hitac, povećava se za svaku pticu po jedan
- Pomenute dvije (tri) situacije riješiti pomoću event-a

Refleksija

- Refleksija je sistem koji omogućava pristup klasama i njihovu modifikaciju tokom izvršavanja programa
- Refleksija omogućava dinamičko instanciranje klasa i startovanje njihovih metoda
- Refleksija se smatra sporim sistemom u svim okruženjima u kojima postoji
- Refleksija nije naročito bezbijedna, jer interveniše na strukturi klasa

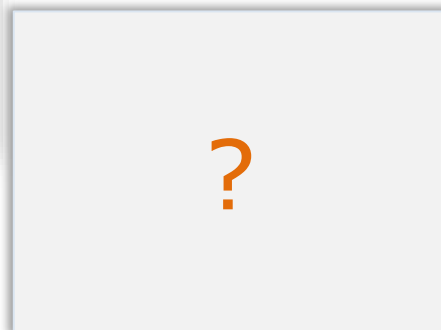
Zašto nam treba refleksija

- Refleksija je korisna kada na primjer želimo da instanciramo klasu i aktiviramo njen metod na osnovu evaluacije string parametara. Ovo je čest slučaj kada podaci dolaze spolja



new BlackJack();

new Slot();



Nova klasa igre, ne može se instancirati prije nego što se predstavi sistemu

Klase i klasa Class

- Glavna klasa refleksije je klasa **Class**
- Ova klasa obezbeđuje informacije o nekoj klasi
- Instancu klase **Class** na osnovu postojeće klase možemo dobiti na osnovu **postojeće klase** ili **postojećeg objekta**, ili samo na osnovu **imena klase**

```
class MyClass {  
    public int x;  
    public int y;  
}  
  
MyClass mc = new MyClass();  
Class cl = MyClass.class;  
Class cl1 = mc.getClass();  
Class cl2 = Class.forName("simplereflection.SimpleReflection$1MyClass");  
System.out.println(cl==cl1&&cl1==cl2);
```

**Preuzimanje iz
statičke klase**



**Preuzimanje iz
postojećeg objekta**



**Kreiranje pomoću
Imena klase**



my

Rukovanje Class objektom

- Kada je jednom dobavljen Class objekat, interesovaće nas da pregledamo i eventualno koristimo članove klase

- Pregled svih polja klase:

```
for(Field f : cl.getFields()){  
    System.out.println("Field name: " + f.getName());  
}
```

- Pregled svih metoda klase

```
for(Method m : cl.getMethods()){  
    System.out.println("Method name: " + m.getName());  
}
```

Instanciranje klase u refleksiji

- Refleksija omogućava instanciranje klase i startovanje njenih metoda na osnovu stringova. Ovo omogućava da klasa za nas bude potpuno nepoznata, sve dok znamo parametre koji su nam potrebni za njeno instanciranje, i eventualno pozivanje njenih metoda

```
Class gameClass = Class.forName("reflectioninstancing.GameClass");  
Object game_class_object = gameClass.newInstance();  
Method game_start_method = gameClass.getMethod("StartGame", null);  
game_start_method.invoke(game_class_object, null);  
Method game_hit = gameClass.getMethod("hit", int.class);  
game_hit.invoke(game_class_object, 10);  
game_hit.invoke(game_class_object, 20);  
game_hit.invoke(game_class_object, 30);
```

**Ne moramo
znati ove
informacije**



Vježba 4

- Potrebno je kreirati dvije jednostavne igre (nije neophodno kreirati logiku igara, već samo klase i prazne metode)
- 1: Gamble (Korisnik bira da li će sljedeći broj biti veći ili manji od broja koji je odabrao računar)
- 2: Crvena crna (Korisnik bira da li će sljedeća karta biti crvena ili crna)
- Obje igre su realizovane kroz interfejs IGame, koji ima metod pick (u prvom slučaju, ovaj metod će uzeti input od korisnika i prikazati rezultat)
- Korisnik prilikom startovanja igre, unosi tip (klasu) igre koju hoće da igra (Gamble ili BlackRed)
- Za instanciranje igre treba koristiti refleksiju

Anotacije

- Anotacije su meta podaci klasa ili njenih elemenata
- Koriste se da bi dali neku informaciju sistemu prilikom kompajliranja, ili izvršavanja, ali da pri tom ne utiču direktno na funkcionalnost i strukturu same klase




Korištenje anotacija

<https://docs.oracle.com/javase/tutorial/java/annotations/index.html>

- Anotaciju možemo prepoznati po oznaci @ ispred naziva, i poziciji, koja je obično iznad klase ili njenog člana
- Na primjer:

```
@Override  
public String toString(){  
    return "Hello";  
}
```

Anotacija



- Anotacije se takođe mogu pojaviti i na specifičnim pozicijama (ispred ciljnog tipa prilikom konverzije, ispred generičkog tipa,)
 - ***new @Interned MyObject();***
 - ***myString = (@NonNull String) str;***
 - ***class UnmodifiableList<T> implements @ReadOnly List<@ReadOnly T> { ... }***
 - ***void monitorTemperature() throws @Critical TemperatureException { ... }***

Parametri anotacije

- Anotacija se može pojaviti u neparametrizovanom obliku (kao na prethodnom slajdu), ali takođe može biti i parametrizovana

```
@WebServlet(urlPatterns = {"/hello"})  
public class NewClass extends HttpServlet {
```

Ugrađene anotacije

<https://docs.oracle.com/javase/tutorial/java/annotations/predefined.html>

- Java sadrži mnoštvo ugrađenih/predefinisanih anotacija (neke su već korištene u programima)
- Predefinisane anotacije se naročito često upotrebljavaju unutar Java EE
- <https://docs.oracle.com/javase/tutorial/java/annotations/predefined.html>

Korisnički definisane anotacije

- Sopstene anotacije nije komplikovano kreirati. Ovo su obični (tačnije skoro obični) interfejsi
- Sljedeća linija predstavlja validno kreiran interfejs

```
public @interface MyAnnotation {}
```

- Ovako kreiran interfejs, predstavlja validnu anotaciju, koja se može postaviti na klasu (ali bez naročite svrhe)

```
package simpleuserannotation;  
import java.lang.annotation.ElementType;  
import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;  
import java.lang.annotation.Target;
```

```
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.TYPE)  
public @interface MyAnnotation { }
```

Korištenje korisnički definisanih anotacija

- Jedini način da koristimo korisnički definisane anotacije jeste putem refleksije. Na primjer:

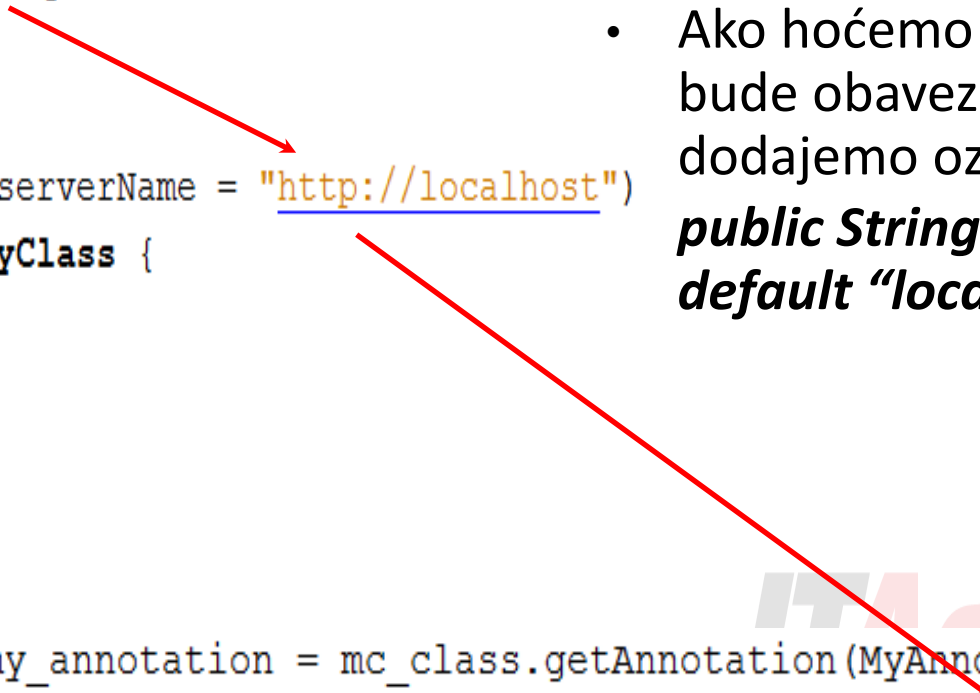
```
Class<MyClass> mc_class = MyClass.class;
System.out.println("All annotations: ");
for(Annotation annotation : mc_class.getAnnotations()){
    System.out.println("Annotation: " + annotation.toString());
}
```

Parametrizacija korisnički definisane anotacije

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface MyAnnotation {
    public String serverName();
}

@MyAnnotation(serverName = "http://localhost")
public class MyClass {

}
```



```
MyAnnotation my_annotation = mc_class.getAnnotation(MyAnnotation.class);
System.out.println("Server name is: " + my_annotation.serverName());
```

- Parametri anotacije realizuju se pomoću potpisa metoda interfejsa anotacije
- Ako hoćemo da parametar ne bude obavezan, u interfejsu dodajemo oznaku default:
***public String serverName()
default "localhost";***

Vježba 5

- Potrebno je kreirati anotaciju pod nazivom **Locale**
- Anotacija mora da ima jedan parametar **language** koji je tipa **String**
- Potrebno je kreirati klasu **UserMessageService** koja
- Klasa UserMessageService sadrži polje messages koje je tipa Map<String,String> i sadrži poruke dobrodošlice na različitim jezicima pri čemu su ključevi oznake jezika (en,fr...), a vrijednosti same poruke
- Klasa treba da ispisuje sadržaj na izlaz, pomoću metode **showWelcomeMessage**
- Klasa mora da ima anotaciju Locale, u kojoj će biti kao parametar naveden jezik pozdravne poruke
- Klasu treba instancirati pomoću refleksije, a zatim prikazati pozdravnu poruku na jeziku određenom u anotaciji

Lambda izrazi

- Lambda izrazi su sistem kojim se olakšava kreiranje anonimnih klasa u svrhu implementacije funkcionalnih interfejsa
- Lambda izrazi podrazumjevaju sljedeću sintaksu:

Parametri **Lambda oznaka** **Tijelo**

```
MyInterface11 impl2 = ()->{  
    System.out.println("Hello from anonymous class!!!");  
};
```

- Efekat je isti kao i sljedeći kod:

```
MyInterface11 impl1 = new MyInterface11() {  
    @Override  
    public void onlymethod() {  
        System.out.println("Hello from anonymous class!!!");  
    }  
};
```

Academy

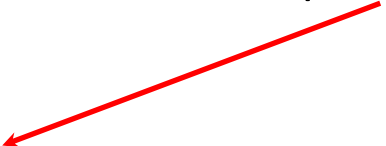
Rad sa regularnim izrazima

<http://docs.oracle.com/javase/tutorial/essential/regex/index.html>

- Regularni izrazi predstavljaju način za definisanje seta karaktera kojima se formira takozvani šablon
- Za kreiranje regularnih izraza koristi se posebna sintaksa
- Regularne izraze možemo sresti/upotrebljavati na različitim mjestima:
 - o Veoma često u validaciji podataka
 - Validacija email adrese
 - Validacija šablona unijetog podatka (na primjer broj telefona ili datum)
 - ...
 - o Prilikom izolovanja ključnih podataka iz nekog većeg podatka (Stringa)
 - Preuzimanje host-a iz url-a
 - Preuzimanje naslova iz teksta
 - Mijenjanje jedne riječi u tekstu drugom rječju
 - ...

Rad sa regularnim izrazima

- Regularni izrazi predstavljaju, sami po sebi, jedan mali programski jezik. Ovaj jezik obično je implementiran u neku biblioteku jezika „domaćina“ kojim se njime upravlja. U Javi ova biblioteka naziva se `java.util.regex`. Zbog toga je za korištenje regularnih izraza neophodno implementirati pomenuti paket.



```
import java.util.regex.*;
public class SimpleRegEx {
    public static void main(String[] args) {

    }
}
```

Kreiranje šeme (izraza) i provjera Stringa

- Glavni učesnici u radu sa regularnim izrazima su klase **Pattern** i **Matcher**
- Klasom Pattern predstavlja se šablon po kome treba testirati String
- Klasom Matcher vrši se samo testiranje i eventualno procesira testirani string
- Kreiranje instance Pattern klase obavlja se pomoću statičke metode compile:

```
Pattern p = Pattern.compile("bong");
```

- Kao parametar metode, proslijeđuje se šema po kojoj će biti vršeno poređenje
- Metodom matcher, generiše se instanca klase Matcher

```
Matcher m = p.matcher("bing bang bong bung");
```

- Metod matcher, kao parametar prihvata string nad kojim će biti izvršen izraz
- Nad objektom klase Matcher, možemo izvršavati različite metode za provjeru ali je to najčešće jedna od dvije (**find/match** i **replace**)

```
System.out.println("String contains word: " + m.find());
```

```
System.out.println(m.replaceAll("HELLO"));
```

Šabloni regularnih izraza

- Prilikom krieranja šablona regularnih izraza, koristi se posebna notacija kojom se označavaju eventualne specifične situacije u tekstu za koje želimo da testiramo tekst.
- Ove situacije ne podrazumjevaju fiksne vrijednosti u stringu, već različite opsege vrijednosti, sekvence karaktera, serije karaktera koje su približne očekivanim (fuzzy) i slično

Opsezi

- Opseg podrazumjeva poklapanje za bilo koji iz serije slučajeva. Serija slučajeva označava se uglastim zagradaama:
 - **[abc]** – traže se karakteri **a**, **b** ili **c**
- Ako želimo opseg (od do) koristimo crticu (minus)
 - **[d-h]** – traže se inkluzivno karakteri od d do h (dakle d,e,f,g i h)
 - **[a-ce-g]** – traže se karakteri od a do c i od e do f
 - **[0-9]** – traže se brojevi od 0 do 9

```
System.out.println("Match if string contains numbers from 0 to 9");  
p = Pattern.compile("[0-9]");  
matcher = p.matcher("I don't have any numbers");  
System.out.println(matcher.find());
```



Predefinisani karakteri u regularnim izrazima

- Za neke popularnije šablone, postoje specijalni karakteri u Javi

Konstrukcija	Zamena	Značenje
<code>\d</code>	<code>[0-9]</code>	bilo koja cifra
<code>\D</code>	<code>[^0-9]</code>	sve osim cifre
<code>\s</code>	<code>[\t\n\r\x0B\f]</code>	prazan karakter
<code>\S</code>	<code>[^\s]</code>	sve osim praznog karaktera
<code>\w</code>	<code>[a-zA-Z_0-9]</code>	karakter reči: slovo ili cifra
<code>\W</code>	<code>[^\w]</code>	sve osim slova i cifara

Vježba 6

- Treba kreirati jednu listu stringova koja sadrži rečenice

```
List<String> sentences = new ArrayList<>();
sentences.add("A tank is a large type of armoured fighting vehicle with tracks, designed for front-line combat.");
sentences.add("Modern tanks are strong mobile land weapons platforms, mounting a large-calibre cannon in a rotating gun turret.");
sentences.add("They combine this with heavy vehicle armour providing protection for the crew of the weapon and operational mobility↵
, which allows them to position on the battlefield in advantageous locations.");
sentences.add("These features enable the tank to have enormous capability to perform well in a tactical situation: the combination ↵
of strong weapons fire from their tank gun and their ability to resist enemy fire means the tank can take hold of and control an area of th↵
e battle and prevent other enemy vehicles from advancing, for example.");
```

(<http://pastie.org/pastes/9777818/text>)

- Potrebno je kreirati program koji uzima riječ od korisnika (ili koristi hard kodiranu riječ) i prikazuje rečenice u kojima se riječ pojavljuje (korišćenjem regularnih izraza)

Meta karakteri

- U okviru samog izraza, mogu se pojaviti karakteri koji imaju posebno značenje za evaluaciju izraza. Ovi karakteri nazivaju se metakarakter
- Osim dva koja smo već vidjeli (uglaste zagrade i minus), postoje i drugi metakarakter `<([{\^-= $!|]})? *+ .>`,

Specijalni karakteri

- `.` - Jedno pojavljivanje bilo kog karaktera na mjestu na kome se tačka nalazi (`hel.o` = `hello`, `helao`)
- `*` - Ni jedno ili više pojavljivanja prethodnog karaktera (`hel*o` = `helo`, `hellllllllo`)
- `+` - Jedno ili više pojavljivanja prethodnog karaktera (`hel+o` = `hello`, `hellllllllo` (ali ne i `helo`))
- `?` – Karakter se može pojaviti jednom, ili ni jednom (`hel?o` = `hello`, `helo`)
- `{}` – Prvi parametar označava minimalan broj pojavljivanja, a drugi maksimalan (`hell{2,3}o` = `helllo`, `helllllo` (ali ne i `hello`, niti `helllllo`))
- `\` - Oznaka se može koristiti za ukidanje funkcionalnosti specijalnog (meta) karaktera. Specijalni karakter iza oznake backslash, biće tretiran kao običan karakter (`hel\o` = `hel.o` (ali ne i `hello`, `helao` i slično))
- `^` - Označava početak stringa (`^hello` = `hello` (ali ne i `world hello`))
- `$` - Označava kraj stringa (`hello$` = `world hello` (ali ne i `hello world`))

Kvantifikator	Značenje
<code>X?</code>	jednom ili nijednom
<code>X*</code>	nijednom ili više puta
<code>X+</code>	jednom ili više puta
<code>X{n}</code>	tačno n puta
<code>X{n, }</code>	najmanje n puta
<code>X{n, m}</code>	najmanje n puta, ali ne više od m puta

Vježba 7 – validacija broja telefona

- Potrebno je izvršiti validaciju broja telefona po sljedećem šablonu: ###/###-####, tako da na primjer, sljedeći string bude validan: 123/456-7890

Zadatak

- Pokušajte samostalno da kreirate i primjenite šablon za validaciju email adrese

Dizajn šabloni

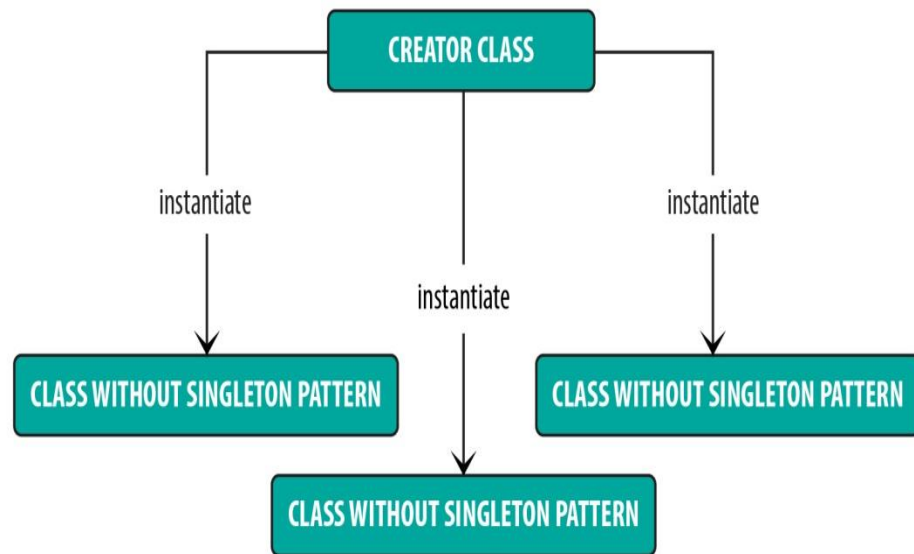
- Dizajn šablon je standardno rješenje za neki programski problem. Do tog rješenja se došlo evolutivno i empirijski
- Korištenjem dizajn šablona znatno ubrzavamo proizvodnju, naročito ukoliko se ona odvija u timskom okruženju.
- Šabloni nisu nešto što možemo upotrijebiti tako što ćemo uraditi Copy/Paste. Oni se implementiraju u objektnu strukturu i čine njen sastavni dio, sa akcentom na riječi „objektnu“, jer su dizajn šabloni usko vezani baš za objektno-orijentisani programerski koncept.
- Šabloni se dijele na tri osnovne kategorije: šablone za **kreaciju**, šablone **strukturu** i šablone **ponašanja**, pri čemu svaki od ovih naziva intuitivno oslikava upotrebno okruženje šablona.
- Količina dostupnih šablona je velika i mi ćemo obraditi samo neke od najpoznatijih i češće korištenih u Javi

Singleton pattern

- Singleton Pattern je šablon kreacije koji garantuje da ćete u kodu napraviti samo jednu instancu određene klase.
- Pogledajmo primjer:

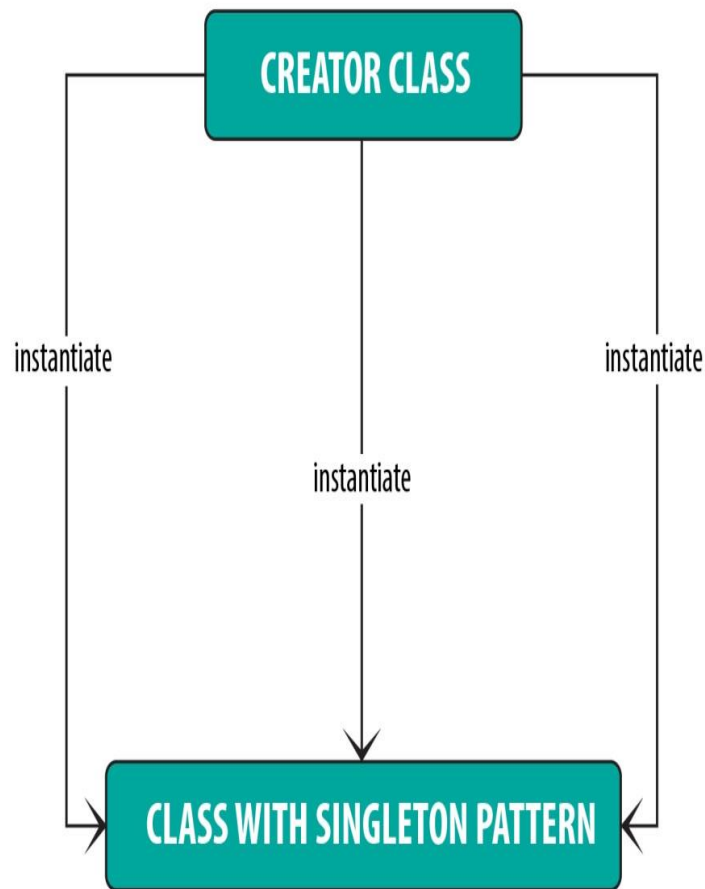
```
MyClass a = new MyClass();  
MyClass b = new MyClass();  
System.out.println(a.equals(b));
```

- Ovaj program će emitovati rezultat false jer promjenljive a i b ne predstavljaju jedan isti objekat. Ovo je potpuno logično ponašanje navedenog programa, ali šta da se u klasi MyClass nalazi sistem za rukovanje bazom podataka? Onda bismo, možda, željeli da se obezbijedimo od toga da korisnik može da napravi više od jedne instance objekta jer bi u tom slučaju imao i više od jedne konekcije.



Singleton pattern

- U tom slučaju najbolje rješenje bi bila upotreba Singleton Patterna. Tada bismo mogli biti sigurni da će svaki pokušaj instanciranja neke klase rezultirati vraćanjem uvijek istog objekta



Singleton pattern

- Za singleton, neophodno je ostvariti dvije stvari. Prije svega, onemogućiti direktno instanciranje klase.
- Ovo je moguće postići uvođenjem privatnog konstruktora i metode specijalizovane samo za dobavljanje odgovarajuće instance

```
public class MyClass {  
    private static MyClass instance;  
    private MyClass() { }  
    public static MyClass getInstance() {  
        if(instance==null){  
            instance = new MyClass();  
        }  
        return instance;  
    }  
}
```

```
MyClass a = MyClass.getInstance();  
MyClass b = MyClass.getInstance();  
System.out.println(a.equals(b));
```



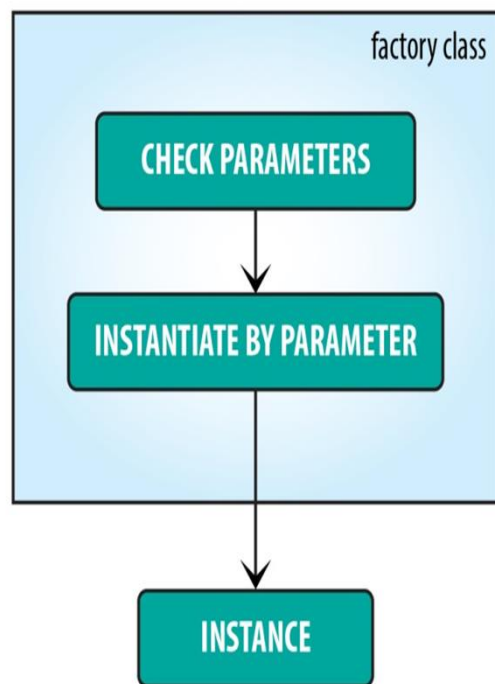
Ovaj kod sada na izlazu prikazuje vrijednost true

Observer

- Observer, šablon ponašanja, omogućava da se stanje jednog objekta (Subject) emituje različitim objektima (Subscribers) koji taj objekat nadgledaju. Ovaj jednostavni koncept može se uporediti sa pretplatom za novine. Dok pretplata važi, novine stižu. Kada otkažemo pretplatu, novine više ne stižu.
- Ono što tehnički možemo da izdvojimo kao jasne cjeline u primjeru novina jeste:
 - lista svih pretplatnika koju posjeduje izdavač
 - način na koji će izdavač prosljeđivati izdanje pretplatniku

Factory

- Ovaj kreacioni šablon koristimo kada želimo da koristimo posredničku klasu uz pomoć koje ćemo kreirati instancu određene klase.



Decorator

- Decorator je strukturni Pattern koji omogućava izmjenu (dodavanje) funkcionalnosti postojećoj klasi, pri čemu, za razliku od klasičnog nasljeđivanja, Decorator klasa nije u direktnoj hijerarhijskoj vezi sa klasom koju „dekorirše“, već služi kao omotač za osnovnu verziju klase.

