LINKgroup

Distance Learning System

# Advanced Java Programming

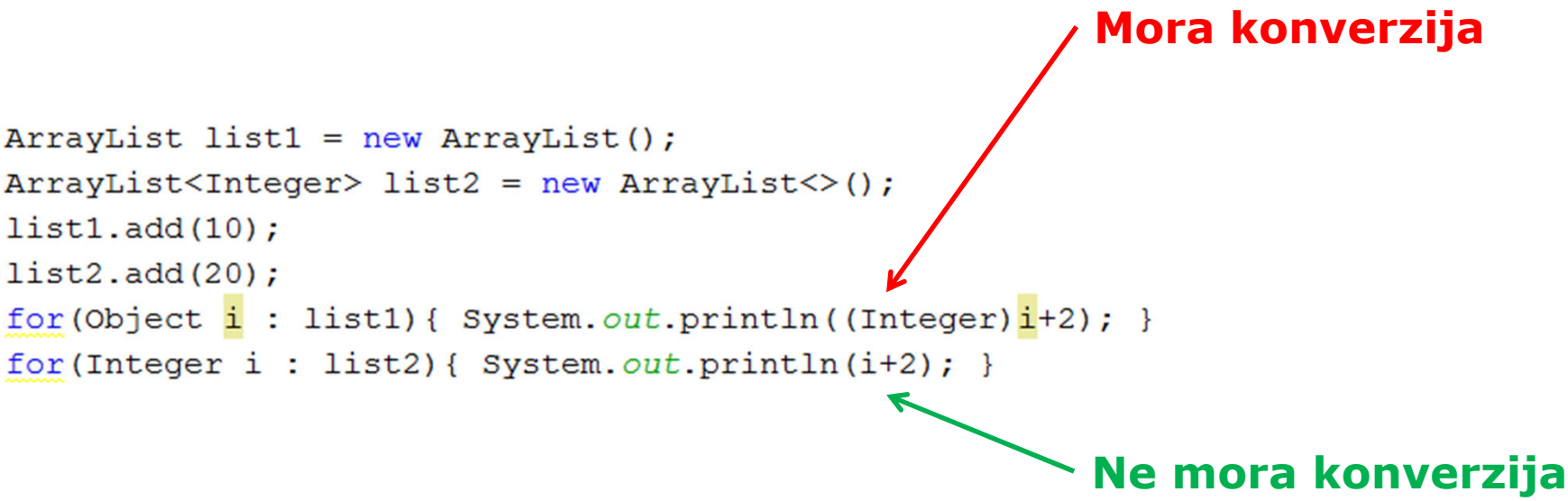
Generičke klase

# 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 rešenje za izbegavanje box-inga i unbox-inga.

# Šta se rešava generičkom klasom

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



# Korisnički definisane generičke klase
## (jaex022014 SimpleGenerics)

- Da bi klasa bila generička mora joj se između oznaka manje i veće, nakon imena, dodati identifikator tipa. Ovaj identifikator može biti bilo koja reč koja poštuje pravila imenovanja promenljivih
- 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);
```

# Zašo 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
## (jaex022014 SimpleGenericMethods)

- 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
## (jaex022014 SimpleBoundedTypes)

- Moguće je ograničiti tipove na neku određenu nadklasu.
- Sledeći primer ograničava prosleđeni tip na podklasu klase Numeric

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

# Nasleđivanje generičkih klasa (jaex022014 SimpleGenericExtending)

- Generičke klase je moguće naslediti
- Prilikom nasleđivanja, 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
- …

# Vežba 1 (jaex022014 BatAndBall)

- Kreirati klasu generičku **Game** koja će imati celobrojna 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 celobrojnim poljima **posx**, **posy,width,height** i **speed**.
- Kreirati klase **Ball** i **Bat**, koje nasleđuju klasu GameObject
  - U klasi Ball, metod update pomera objekat u svim pravcima
  - U klasi Bat, metod update pomera objekat levo desno
  - Draw metode obe 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/)*

# Vežba 2 (jaex022014 GenericBank)

- 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 naslednik 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 obe 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 podrazumeva 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 proveravamo 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 (jaex022014 SimpleLocalEvent)

- Analizirajmo sledeći 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);
    }
}
```

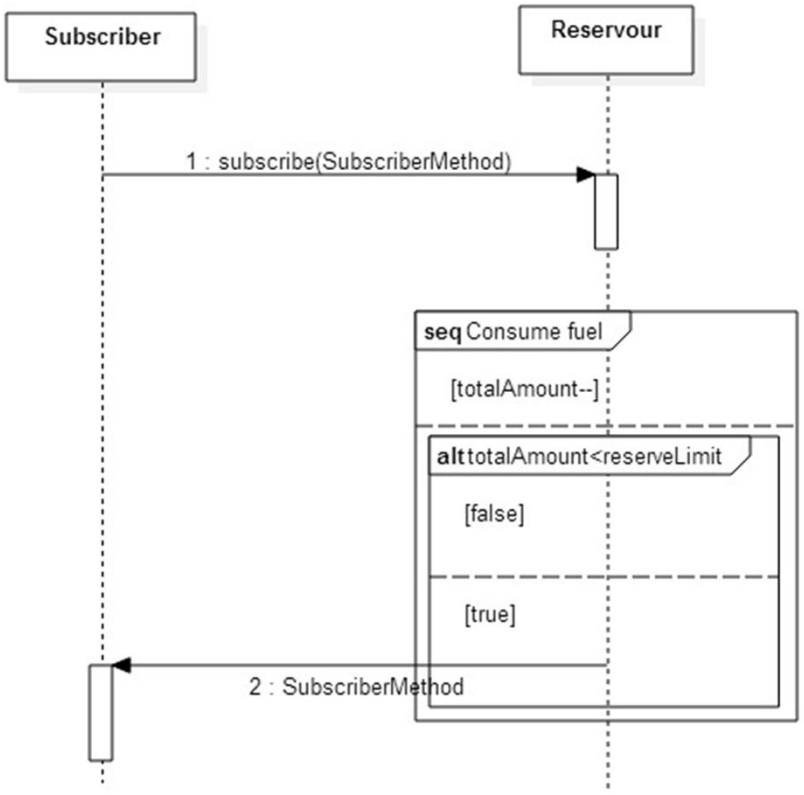
**SimpleLocalEvent.java**

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

- Primećujemo da je rezervoar svestan ulaska u rezervu, ali da svet oko njega nije. Unutar main metode, ne znamo da je rezervoar „na rezervi" jer je događaj identifikovan i obrađen unutar objekta

# Distributer/subscriber model (jcex 022014 CarEvents)

- 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 podrazumevaju 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**



# Slušač događaja

- Prvi korak u procesu definicije događaja je konstrukcija slušača događaja. Slušač događaja je najčešće interfejs koji ima jedan ili više metoda za koje će znati distributer i pretplatnici. Slušač 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 obaveštavanje 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 distribuira 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);
        }
    }
}
```

# Vežba 3 (022014 HitMe)

- 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 sledećeg hica
- Hitac se ispaljuje i i prati u realnom vremenu, proveravajući za svaki pomeraj, da li je:
  - Projektil udario u tlo
    - Hitac se smatra završenim i ne dodeljuju se poeni za pogodak
  - Projektil udario u neprijateljski tenk
    - Hitac se smatra završenim i dodeljuju se poeni za pogodak
  - 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 dve (tri) situacije rešiti 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 bezbedna, jer interveniše na strukturi klasa