

Apstraktne klase

- Apstraktne klase su šabloni za kreiranje drugih klasa
- Njihova osnovna osobina je da se ne mogu instancirati, već jedino naslediti ili koristiti u statičkom kontekstu
- Da bi učinili klasu apstraktnom, dodajemo joj modifikator **abstract**

Instanciranje apstraktnih klasa

- Apstraktne klase NIJE moguće instancirati

Sledeći kod funkcioniše

```
package abstractclasses;
public class Person {
    public String firstName;
    public String lastName;
}

package abstractclasses;
public class AbstractClasses {
    public static void main(String[] args) {
        Person person = new Person();
    }
}
```

Ali sledeći ne

```
package abstractclasses;
public abstract class Person {
    public String firstName;
    public String lastName;
}

package abstractclasses;
public class AbstractClasses {
    public static void main(String[] args) {
        Person person = new Person();
    }
}
```

Person is abstract; cannot be instantiated

(Alt-Enter shows hints)

Apstraktne klase i interfejsi

LINKgroup

Nasleđivanje osobina od apstraktnih klasa

- To što klasa ne može da bude instancirana, ne znači da ne može sadržati funkcionalnosti i polja:
- Svaka klasa koja nasledi apstraktnu klasu može se normalno upotrebljavati, i imaće na raspolaganju sve elemente klase koju je nasledila

```
package abstractclasses;  
public class Student extends Person { }
```

```
package abstractclasses;  
public abstract class Person {  
    public String firstName;  
    public String lastName;  
    public void show() {  
        System.out.println(firstName + " " + lastName);  
    }  
}
```

- I naravno, može se instancirati

```
package abstractclasses;  
public class AbstractClasses {  
    public static void main(String[] args) {  
        Student student = new Student();  
        student.firstName = "Peter";  
        student.lastName = "Jackson";  
        student.show();  
    }  
}
```

Apstraktne klase i interfejsi

LINKgroup

Nasleđivanje apstraktne klase apstraktnom klasom

- Klasa koja nasleđuje apstraktnu klasu, takođe može biti apstraktna, i tada za nju takođe važe pravila apstraktne klase

```
package abstractclasses;  
public abstract class Student extends Person { }
```

```
package abstractclasses;  
public class AbstractClasses {  
    public static void main(String[] args) {  
        Student student = new Student();  
        student.firstName = "Peter";  
        student.lastName = "Jackson";  
        student.show();  
    }  
}
```

Student is abstract; cannot be instantiated

(Alt-Enter shows hints)

Apstraktne metode

- Apstraktna klasa može sadržati apstraktne metode
- Ovo su metode koje nemaju telo, već sadrže samo potpis:

```
package abstractclasses;  
public abstract class Person {  
    public String firstName;  
    public String lastName;  
    public abstract void show();  
}
```

Apstraktan metod **show**



- Čemu služi metod koji ne može da uradi ništa, i koji u stvari i ne postoji?

Apstraktne metode

- Apstraktan metod predstavlja neku vrstu ugovora između apstraktne klase i klase koja je nasleđuje
- On garantuje da će klasa koja nasleđuje ispoštovati uslove za učešće u ostatku sistema

Na primer:
Svaka benzinska pumpa ima isti sistem za ubacivanje goriva u automobil. Prilikom izgradnje pumpe, kreator pumpe mora ispoštovati standarde. Krajnji cilj je taj, da kada se pumpa jednom nađe u sistemu, vozači ne razmišljaju o tome da li će crevo odgovarati ulazu rezervoara



Primer apstraktne metode (jcex082014 GasStation)

- Recimo da postoji šablon za kreiranje benzinskih stanica, i da on zahteva metod **fillCar**

```
package gasstation;  
public abstract class BaseStation {  
    public abstract void fillCar();  
}
```

- Ako bismo nasledili ovu klasu sopstvenom praznom klasom (za sopstvenu stanicu), Java bi odmah prijavila problem, jer nismo ispoštovali ugovor/dogovor i nismo implementirali apstraktan metod fillCar

```
1 package gasstation;  
2 public class ShellGasStation extends BaseStation {  
3  
4 }
```

ShellGasStation is not abstract and does not override abstract method fillCar() in BaseStation

(Alt-Enter shows hints)

Primer apstraktne metode (jcex082014 GasStation)

- Kada implementiramo metod fillCar, Java više ne pravi problem

```
package gasstation;
public class ShellGasStation extends BaseStation {
    public void fillCar() {
        System.out.println("Filling car with water");
    }
}
```

- Primećujemo da je sve u redu, iako će automobil zapravo biti napunjen vodom (što svakako nije u redu)
- Ovaj deo sistema nije „zainteresovan“ za taj problem. Njemu je jedino važno da je ispoštovan ugovor, te da je metod obezbeđen.

Primer apstraktne metode (jcex082014 RpgHeroes)

- Recimo da kreiramo rpg igru i da u igru treba periodično dodavati nove heroje
- Koncept igre je takav, da svaki heroj ima određeni set akcija, koje se aktiviraju određenim metodama: **firePrimary** i **fireSecondary**
- Takođe, heroji trebaju da imaju mogućnost da prime određeni udarac i ta da na osnovu toga izvrše neku reakciju (izračunaju štetu, prekinu da postoje, izvrše automatski kontra udarac...). Ovo bi mogao biti metod: **receiveHit**
- Takođe, želimo da postoje situacije u kojoj svi heroji reaguju primanjem udarca. Na primer, prilikom neprijateljskog area of effect napada.



Apstraktne klase i interfejsi

LINKgroup

Primer apstraktne metode (jcex082014 RpgHeroes)

Prilikom postavke sistema, svakako ćemo znati da će svaki heroj moći da ima neke opšte karakteristike. Na primer, ima određenu količinu zdravlja i mane, može biti mrtav ili živ. Ali ne možemo znati na koji način će se do tih podataka doći, jer to zavisi od nekih karakteristika koje su specifične za heroja. Dakle, osnovna klasa kojom bi mogli predstaviti heroja, mogla bi u osnovi biti:

```
package rpgheroes;

public abstract class BaseHero {
    public double health, mana;
    public boolean dead;
    public void areaOfEffectHit() {
    }
}
```

Kreirana klasa je u stanju da nosi podatke koji su nam potrebni u igri, ali imamo problem sa metodom **areaOfEffectHit**. Znamo da je to situacija kada heroji koji su u zoni napada, trpe štetu. Ali šta ta šteta znači za te heroje? Koliko će im zdravlja oduzeti? I koje će druge posledice doneti? Ne znamo, jer još uvek nemamo ni jednog heroja, niti znamo kako će se on ponašati u datoj situaciji. Takođe, čak i da imamo heroje, svi oni će imati različito ponašanje.

Primer apstraktne metode (jcex082014 RpgHeroes)

- Da bi rešili problem opisan na prethodnom slajdu, možemo u metodi `areaOfEffectHit`, pozvati metod **`receiveHit`** i kompletnu odgovornost prebaciti na autora klase **konkretnog** heroja.
- Nije nas briga da li će autor odlučiti da heroj preživi ovakav udarac, to je više stvar pravila igre, nego sistema
- Ono što nas jeste briga to je da ne dodemo u situaciju da metod `receiveHit` uopšte ne postoji, i zbog toga ga označavamo apstraktno, kako bi naterali autora heroja da ga implementira.
- *Da li smo mogli i sami da izvršimo neku generičku implementaciju ovih metoda (`receiveHit`, `primaryFire` i `secondaryFire`), a autoru heroja ponudimo da ih prepíše samo ukoliko želi? Jesmo, i to bi takođe bilo sasvim ispravno rešenje ovog problema.*

```
package rpgheroes;

public abstract class BaseHero {
    public double health, mana;
    public boolean dead;
    public void areaOfEffectHit() {
        receiveHit();
    }
    public abstract void receiveHit();
    public abstract void primaryFire();
    public abstract void secondaryFire();
}
```

Primer apstraktne metode (jcex082014 RpgHeroes)

- Sada autor heroja ima slobodu prilikom kreiranja, a mi (kao eventualni autori sistema) znamo da će se taj deo uklopiti u ostatak. U suprotnom, program neće moći da bude ni preveden, zbog čega će biti lako identifikovati eventualni problem
- Jedino što moramo da uradimo, jeste da kažemo autoru da će heroj morati da nasledi klasu **BaseHero**.
Bez ovoga, naravno, kompletan koncept ne bi imao smisla
- Možemo takođe autoru napraviti i „sheet“ za heroja.
Na primer:

Tip heroja DwarfWarrior
Pri udarcu gubi 10 zdravlja
Primarno oružje troši 5 mane
Sekundarno oružje troši 10 mane

```
package rpgheroes;
public class DwarfWarrior extends BaseHero {
    @Override
    public void receiveHit() {
        this.health -= 10;
        this.dead = this.health <= 0;
    }
    @Override
    public void primaryFire() {
        this.mana -= 5;
        System.out.println("Firing primary!!!!");
    }
    @Override
    public void secondaryFire() {
        this.mana -= 10;
        System.out.println("Firing secondary!!!!");
    }
    public void show() {
        System.out.println(
            "Health: " + this.health
            + " Mana: " + this.mana
            + " Dead: " + this.dead
        );
    }
}
```

Primer apstraktne metode (jcex082014 RpgHeroes)

- Nakon kreirane klase, posao autora klase je završen. Pokušajte da napišete i startujete sledeći kod
- Samo su prve četiri linije bitne, ostatak može biti proizvoljan i njime samo simuliramo dinamiku igre

```
package rpgheroes;
public class RpgHeroes {
    public static void main(String[] args) {
        DwarfWarrior dwarfWarrior = new DwarfWarrior();
        dwarfWarrior.dead = false;
        dwarfWarrior.health = 25;
        dwarfWarrior.mana = 25;
        dwarfWarrior.show();
        dwarfWarrior.receiveHit();
        dwarfWarrior.show();
        dwarfWarrior.receiveHit();
        dwarfWarrior.show();
        dwarfWarrior.primaryFire();
        dwarfWarrior.show();
        dwarfWarrior.secondaryFire();
        dwarfWarrior.show();
        dwarfWarrior.receiveHit();
        dwarfWarrior.show();
    }
}
```

Mora postojati

Ne mora da postoji.
Simulira tok igre

Zadatak:

- Pokušajte samostalno da kreirate heroja po sledećem sheet-u:
 - 1. Tip heroja ElfMage**
 - 2. Pri udarcu gubi 30 zdravlja**
 - 3. Primarno oružje troši 15 mane**
 - 4. Sekundarno oružje troši 50 mane**
- Pokušajte da kreirate konstruktor za ovog heroja, tako da se osnovne karakteristike mogu dodati prilikom instanciranja

Interfejsi

- Apstraktna klasa može sadržati sve komponente obične (konkretne) klase, ali takođe može sadržati apstraktne metode koje predstavljaju „ugovor“ između apstraktne klase i klase koja je nasleđuje.
- Ponekad ćemo hteti da kreiramo samo „ugovor“, bez ostatka logike. Tada možemo kreirati apstraktnu klasu, koja će sadržati isključivo apstraktne metode
- Ipak, mnogo češće, u pomenutoj situaciji, kreiraćemo **Interfejs**
- Interfejs je ugovor, čijim „potpisivanjem“ se „potpisnik“ obavezuje da će ispoštovati njegove stavke (ono što svaki ugovor zapravo i jeste), pri čemu su stavke ovog ugovora **metode**, odnosno, njihova implementacija.

Kreiranje interfejsa

- Interfejs se kreira na isti način kao i klasa.
 - Kreira se u zasebnom .java fajlu
 - Poštuje se ista konvencija imenovanja, pri čemu se najčešće stavlja veliko slovo I ispred naziva (IChampion, IEntity, Iterator...). Ova praksa nije baš tako česta u Javi pa većinu ugrađenih interfejsa nećete sretati sa velikim slovom I ispred naziva.
- Interfejs (treba da) sadrži samo metode. Iako postoji tehnička mogućnost pridruživanja polja interfejsima, ona se ređe koristi (i sama polja nemaju uticaj na objekte već samo direktno na interfejs (dakle, statička su))


```
package interfaces;  
public interface IChampion {  
    public void receiveHit();  
    public void primaryFire();  
    public void secondaryFire();  
}
```


Implementacija interfejsa

- Interfejs pridružujemo klasi pomoću ključne reči **implements**
- Kada smo pridružili interfejs klasi, moramo implementirati sve njegove metode

Ovim smo potpisali ugovor

```
package interfaces;
public class UndeadMage implements IChampion {
    @Override
    public void receiveHit() { System.out.println("I'm hit"); }
    @Override
    public void primaryFire() { System.out.println("Firing primary"); }
    @Override
    public void secondaryFire() { System.out.println("Firing secondary"); }
}
```



Implementacija interfejsa

- Jedna od važnih razlika između klasa i interfejsa je i to što jedna klasa može naslediti samo jednu roditeljsku klasu, ali može implementirati neograničen broj interfejsa
- Svaki novi interfejs, dodaje se u listu interfejsa oznakom zarez (,)

```
public class MyClass extends UndeadMage implements IChampion, ItemListener,
    Iterable<Object>, Cloneable, Readable, Runnable
{
    @Override
    public void itemStateChanged(ItemEvent e) {
        throw new UnsupportedOperationException("Not supported yet."); //To
Templates.
```

Implementacija interfejsa

- Interfejs ne možemo instancirati, ali možemo „u hodu“ kreirati novi objekat koji implementira njegove metode.
- Ovo je isto kao da smo tokom izvršavanja programa napravili novu klasu koja implementira interfejs, a zatim je instancirali

```
package interfaces;
public class Interfaces {
    public static void main(String[] args) {
        UndeadMage undeadMage = new UndeadMage();
        IChampion champion = new IChampion() {
            @Override
            public void receiveHit() { }
            @Override
            public void primaryFire() { }
            @Override
            public void secondaryFire() { }
        };
    }
}
```

Dodatne funkcionalnosti interfejsa

- Interfejs može sadržati statičke članove:

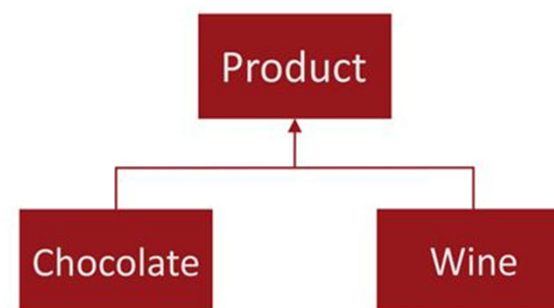
```
public interface MyInterface {  
    public static void interfaceMethod() {  
        System.out.println("Hello");  
    }  
}
```

- Interfejs može imati podrazumevanu implementaciju metode:

```
default public void instanceMethod() {  
    System.out.println("Hello from default method");  
}  
}
```

Zadatak

Zamislite da je potrebno da modelujete informacijski sistem jedne trgovine. Potrebno je da napravite klasu **Product** koja će predstavljati osnovu za dalje nasleđivanje i neće se moći instancirati. Ovu klasu nasleđuju dve klase koje predstavljaju konkretne grupe proizvoda: **Chocolate** i **Wine**. Ova hijerarhija prikazana grafikom izgleda ovako:



Svaki proizvod poseduje određene osobine:
naziv proizvoda,
bar-kod,
osnovnu cenu,
porez.

Takođe, svaki proizvod poseduje i metodu za računanje cene, koja se izračunava kada se osnovna cena i uveća za iznos poreza. Porez (PDV) za svaki proizvod je 20% i ovo je podatak koji je konstantan i neće se menjati. Ipak, proizvodi iz grupe vina, imaju i dodatni porez, pošto spadaju u grupu alkoholnih pića i on iznosi 10% od cene već uvećane za iznos PDV-a. I ovo je podatak koji je konstantan i neće se menjati. Zbog ovoga je potrebno redefinisati metodu za računanje cene u okviru klase Wine. Pored ovoga, klasa *Wine* treba da poseduje atribut koji definiše zapreminu boce, a klasa *Chocolate* atribut koji definiše težinu. U klasama *Chocolate* i *Wine*, potrebno je napraviti parametrizovane konstruktore za kreiranje objekata. Potrebno je, takođe, u klasama redefinisati metodu `toString` za prikaz informacija o objektu.



Distance Learning System

Core Java Programming

Paketi

Paketi

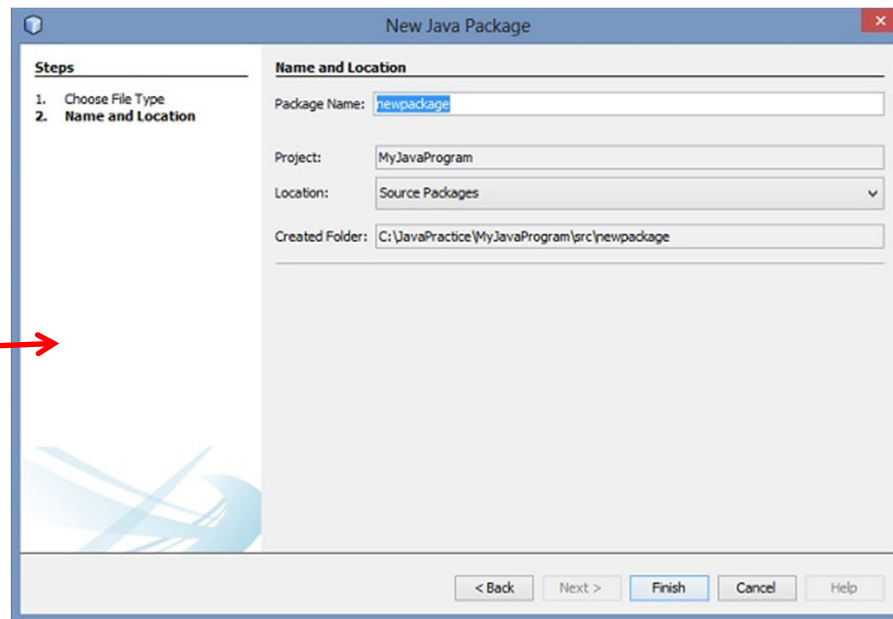
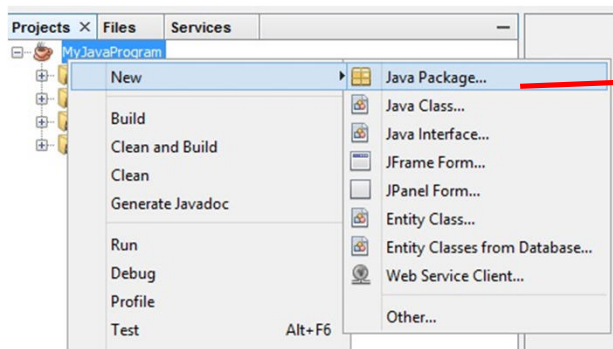
- Paketi su sistem za grupisanje sistemskih komponenti po srodnosti
- Pakete možemo tretirati kao „kutije“ u kojima se nalaze klase. Takođe, unutar kutija, mogu se nalaziti i druge kutije
- Jedna od ključnih osobina paketa je to što omogućavaju istoimene klase unutar istog projekta

Kreiranje paketa

- Paket se fizički, na fajl sistemu, realizuje kroz direktorijum, tako da kreiranjem novog direktorijuma unutar aplikacije zapravo kreiramo novi paket
- Da bi u NetBeans-u kreirali novi paket, koristimo opciju kontekstnog menija projekta:

New->Java Package

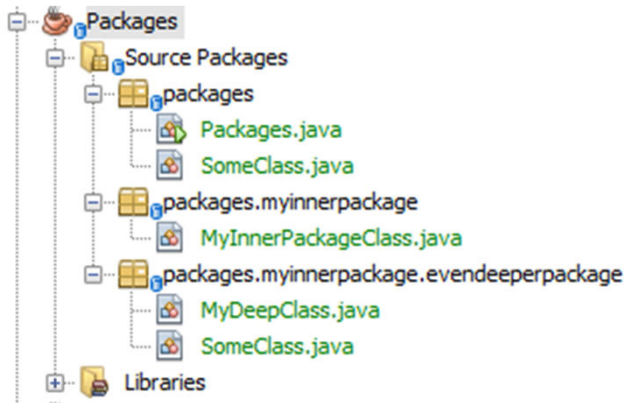
a zatim unosimo željeni naziv paketa



Opseg vidljivosti unutar paketa (jcex082014 Packages)

- Sve što se kreira unutar jednog paketa, vidljivo je implicitno dok smo unutar tog paketa
- Sve što se ne nalazi u paketu, nije vidljivo bez tačne putanje
- Evo kako bi izgledala jedna struktura aplikacije, i pristupanje njenim klasama

Struktura projekta



Pristupanje

```
package packages;
public class Packages {
    public static void main(String[] args) {
        packages.myinnerpackage.evendeepackage.MyDeepClass myDeepClass
            = new packages.myinnerpackage.evendeepackage.MyDeepClass();
        packages.myinnerpackage.MyInnerPackageClass myInnerPackageClass =
            new packages.myinnerpackage.MyInnerPackageClass();
        packages.myinnerpackage.evendeepackage.SomeClass someClass1 =
            new packages.myinnerpackage.evendeepackage.SomeClass();
        SomeClass someClass = new SomeClass();
    }
}
```

Učitavanje paketa ili klasa

- Da bi kod bio pregledniji i kraći, možemo umesto navođenja kompletne putanje svaki put, učitati klasu ili kompletan paket ključnom rečju **import**
- Ako nameravamo da koristimo istoimene klase iz različitih paketa, tada ipak moramo eksplicitno navesti putanju do željene klase:

```
package packages;
import packages.myinnerpackage.MyInnerPackageClass;
import packages.myinnerpackage.evendeepackage.*;
public class Packages {
    public static void main(String[] args) {
        MyDeepClass myDeepClass = new MyDeepClass();
        MyInnerPackageClass myInnerPackageClass = new MyInnerPackageClass();
        SomeClass someClass1 = new SomeClass();
        packages.myinnerpackage.evendeepackage.SomeClass someClass =
            new packages.myinnerpackage.evendeepackage.SomeClass();
    }
}
```

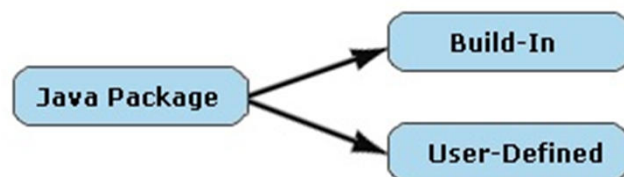
Učitava se klasa
iz drugog paketa

Učitava se kompletan paket

Putanja se navodi eksplicitno

Ugrađeni i korisnički definisani paketi

- Java razlikuje ugrađene i korisnički definisane pakete
- Ugrađeni paketi su oni koje dobijamo na raspolaganju instaliranjem Java Developers Kit-a
- Korisnički definisani su oni koje sami napravimo ili koje preuzmemo, a ne spadaju u sastavni deo jezika



- Kada kažemo paketi, mislimo na klase koje se u njima nalaze

Ugrađeni java paketi

- Svi ugrađeni Java paketi nalaze se u paketu java (njegovi su podpaketi)

- Kao Java developeri, trebalo bi da budemo upoznati sa većim delom ovih paketa i njihovih klasa

- Paket java.lang je podrazumevano dostupan svakoj Java aplikaciji, i njega ne moramo eksplicitno učitavati naredbom import

Naziv paketa	Opis
java.lang	osnovne funkcionalnosti jezika Java, kao što su ugrađeni tipovi
java.util	kolekcije, događaji, internacionalizacija
java.io	rad sa ulaznim i izlaznim tokovima podataka
java.math	BigInteger, BigDecimal
java.nio	rad sa baferima
java.net	podrška za mrežu, URL, socket programiranje, IP adrese
java.security	podrška sigurnosti, enkripcija, dekripcija, generisanje ključeva
java.sql	podrška za rad sa bazama podataka
java.awt	integrisani set klasa za stvaranje korisničkog interfejsa
java.applet	podrška za kreiranje applet-a
java.time	rukovanje datumom i vremenom