LINKgroup

Distance Learning System

Core Java Programming

Objektno Orjentisano Programiranje
Klase i Objekti

Objektno Orjentisano programiranje

- Objektno orjentisano programiranje je programiranje koje omogućava programerima da razvijaju programe, kreirajući objekte i relacije između tih objekata.

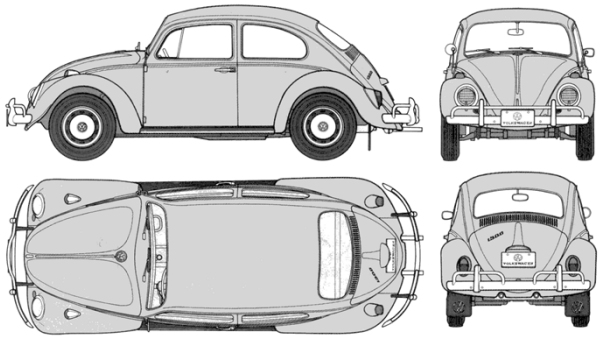
Java u svetu OOP-a

- Iako nije prvi ni jedini objektno orjentisani programski jezik, Java predstavlja simbol objektnog programiranja današnjice
- Java je potpuno objektno orjentisana, što znači da zapravo ne može postojati bez objekata
- Jezici koji mogu da se koriste u objektnom kontekstu ili bez njega, nazivaju se **hibridnim jezicima**
  - Poznatiji hibridni jezici su: C++, php, ...
  - Poznatiji potpuno objektno orjentisani jezici su: C#, Visual Basic
- Karakteristike objektnih jezika su: klase, objekti, metode, svojstva

Klase

- Klasa predstavlja šablon po kome se proizvode objekti koji su još poznati i kao instanca klase.
- Klase generalizuju entitete realnog sveta, dok objekti predstavljaju specifične manifestacije ovih entiteta.
- Na klasu se može misliti kao na modlicu (kalup) po kojoj se prave kolačići.

Klasa BUBA



Objekti klase BUBA

Klase

- Objekat ili instanca jeste jedan primerak nečega, napravljen prema određenom uzorku (klasi). Svaka ptica jeste jedan objekat (instanca) pomenute klase ptica, dok je svaki čovek, objekat klase čovek.
- Da li je ptica instanca klase avion? Verovatno ne. Ali i ptica i avion mogu biti instanca klase: leteći objekat

Kreiranje klasa i objekata

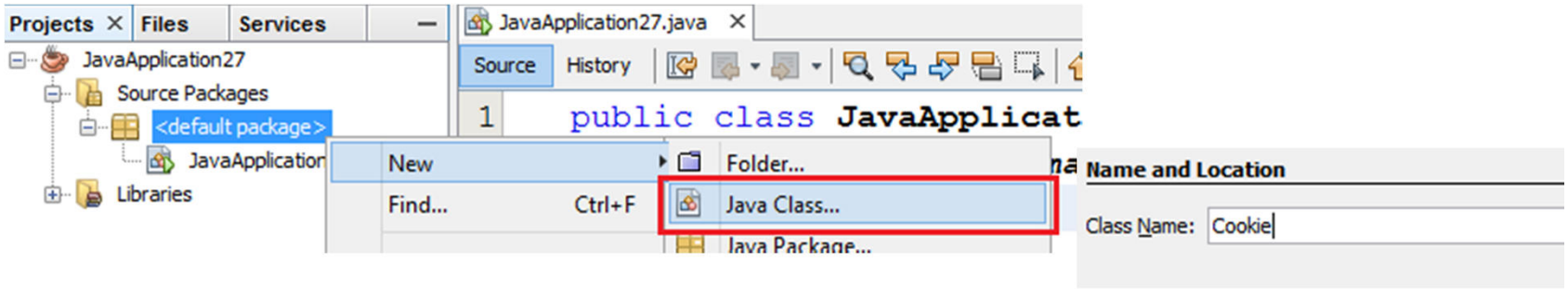
- Prvi korak u procesu kreiranja klasa i objekata jeste kreiranje klase.
- Iako je u nekim jezicima moguće kreirati objekat bez prethodnog kreiranja klase, u Javi to nije moguće
- Da bi u Javi kreirali klasu, kreiramo novi fajl (koji se zove kao i klasa koju hoćemo da kreiramo), u njemu pišemo:

```
class Cookie {
// various member declarations
}
```

- Ovo nam je već poznato, zar ne?

Kreiranje klasa i objekata

- U NetBeansu već postoji šablon za kreiranje ovakvog fajla, pa treba samo aktivirati kontekstni meni iznad odgovarajućeg paketa, a zatim odabrati opciju **new->Java Class**



- Kod imenovanja klasa poštujemo ista pravila kao i kod imenovanja promenljivih, pri čemu postoji još jedno „nepisano" pravilo: Nazive klasa pišemo početnim velikim slovom ukoliko je to moguće, a ako naziv ima više reči, onda svaki početak reči takođe.
- Na primer: class Person, class MyClass, class SomeOtherClass, class Car...

Kreiranje klasa i objekata

- Kada kreiramo klasu, možemo je koristiti za kreiranje objekata

```
Cookie cookie = new Cookie();
```

- *Napisana linija znači: napravi jedan objekat po šablonu* ***Cookie*** *i smesti ga u promenljivu koja se zove* ***cookie****.*
- Takođe smo mogli napisati i:

```
Cookie cookie;
cookie = new Cookie();
```

Kreiranje klasa i objekata

- Klasa koja je kreirana u prethodnom primeru nema naročit smisao jer, iako se zove Cookie, ne sadrži nikakve informacije o bilo kakvim kolačima. Štaviše, ona ne sadrži nikakve informacije.

```
class Person {
    String name;
    String surname;
    int age;
    int height;
    int weight;
}
```

- Klasa Person sadrži neke informacije. Tačnije, ona sadrži polja u koja možemo smestiti informacije. Ime, prezime, godište, visina i težina su karakteristike kojima se može opisati jedan čovek

Kreiranje klasa i objekata

- Sledećim kodom bi mogli instancirati jednu osobu po šablonu Person:

```
Person person = new Person();
person.name = "Peter";
person.surname = "Jackson";
person.age = 50;
person.height = 176;
person.weight = 80;
```

- Ako ovako nešto uradimo u kodu, ništa se neće dogoditi na izlazu, ali će program znati da postoji promenljiva (objekat) sa nazivom person, koja sadrži podakte koje karakterišu jednog čoveka.

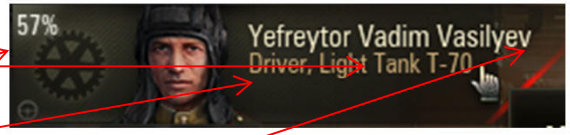
Kreiranje klasa i objekata

- Iako Klase / objekti oslikavaju objekte iz realnog sveta, oni ih ne moraju mapirati u potpunosti, već treba da odgovaraju potrebama sistema
- Na primer, za osobu sa slike, mogli bi kreirati klasu Person, ali za nju nam ne bi bila važna težina i visina, dok bi nam sa druge strane, bilo važno iskustvo

Kreiranje klasa i objekata

- Pogledajmo kako bi izgledala implementacija ove klase

```
public class TankCrewPerson {
    int vehicle_id;
    int experience;
    byte category;
    String name_and_surname;
}
```



- Kreiranje objekta bi moglo izgledati ovako:

```
TankCrewPerson tc_person = new TankCrewPerson();
tc_person.experience = 57;
tc_person.category = 1;
tc_person.name_and_surname = "Yefreytor Vadim Vasilyev";
tc_person.vehicle_id = 5;
```

Kreiranje klasa i objekata

- Pored toga što imaju karakteristike, ljudi mogu nešto i da rade
- Mogućnost objekata da nešto rade, ostvaruje se pomoću metoda
- Metode su takođe sastavni delovi objekta, ali se malo drugačije ponašaju i označavaju od običnih podataka

```
class Person {
    String name;
    String surname;
    int age;
    int height;
    int weight;
    void sleep() {}
    void run() {}
}
```

Polja klase

Metode klase

Tri najbitnija koncepta objektnog programiranja

- Enkapsulacija
- Nasleđivanje
- Polimorfizam

Enkapsulacija

- Pojam enkapsulacije podrazumeva sakrivanje kompleksne logike neke funkcionalnosti.

Za ovo je odličan primer automobil. Kada okrenemo ključ u automobilu, dolazi do startovanja motora. Ono što mi, kao vozači, znamo, je da treba da okrenemo ključ da bismo aktivirali motor. Ali, ono što ceo proces aktivacije motora podrazumeva, mnogo je kompleksnije od okretanja ključa. Kreator automobila sakrio je kompleksnu logiku aktivacije motora od nas i dao nam pristup njegovim najbitnijim funkcijama.

Nije ga briga ko je za volanom

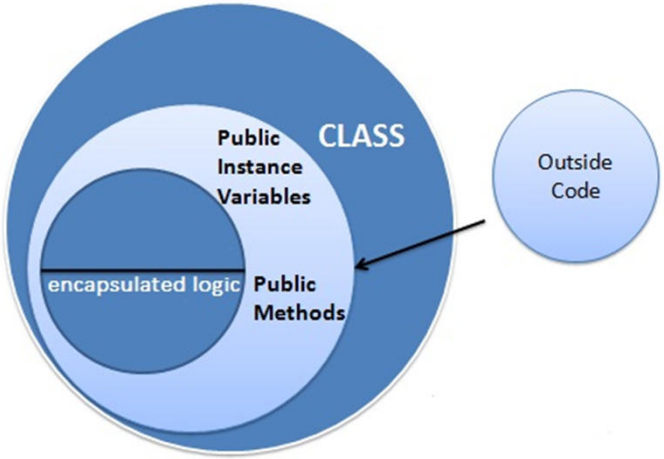
Vozač

Automobil

Nije ga briga šta se događa u pozadini

Enkapsulacija

- Ako postoji klasa Person, i poseduje metodu run, tada najverovatnije nećemo hteti da znamo šta se u toj metodi događa, već hoćemo samo da vidimo krajnji rezultat, a to je trčanje.




```
Person somePerson = new Person();
somePerson.run();
```

Ne zanima nas kako ovaj metod funkcioniše, sve dok nam daje željene rezultate

Enkapsulacija

- Metode i svojstva ugrađenih Java klasa upravo oslikavaju primer enkapsulacije. Mi u našim programima koristimo ove metode i svojstva bez ikakvog znanja o načinu njihovog funkcionisanja.

```java
System.out.println("What is happening inside me?");
```

```java
public void println(String x) {
    synchronized (this) {
        print(x);
        newLine();
    }
}
```

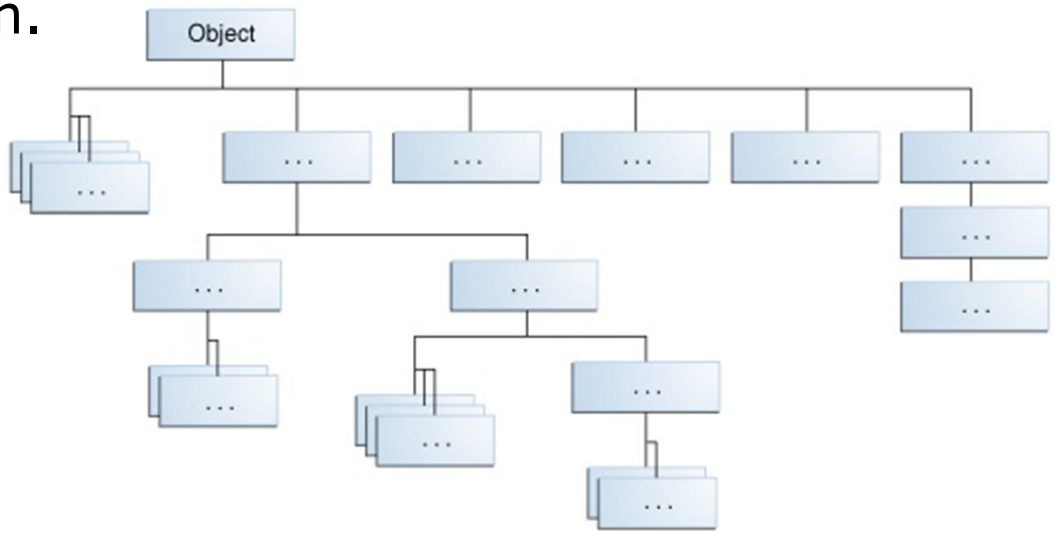
```java
public void print(String s) {
    if (s == null) {
        s = "null";
    }
    write(s);
}
```

```java
private void write(String s) {
    try {
        synchronized (this) {
            ensureOpen();
            textOut.write(s);
            textOut.flushBuffer();
            charOut.flushBuffer();
            if (autoFlush && (s.indexOf('\n') >= 0))
                out.flush();
        }
    }
```

...

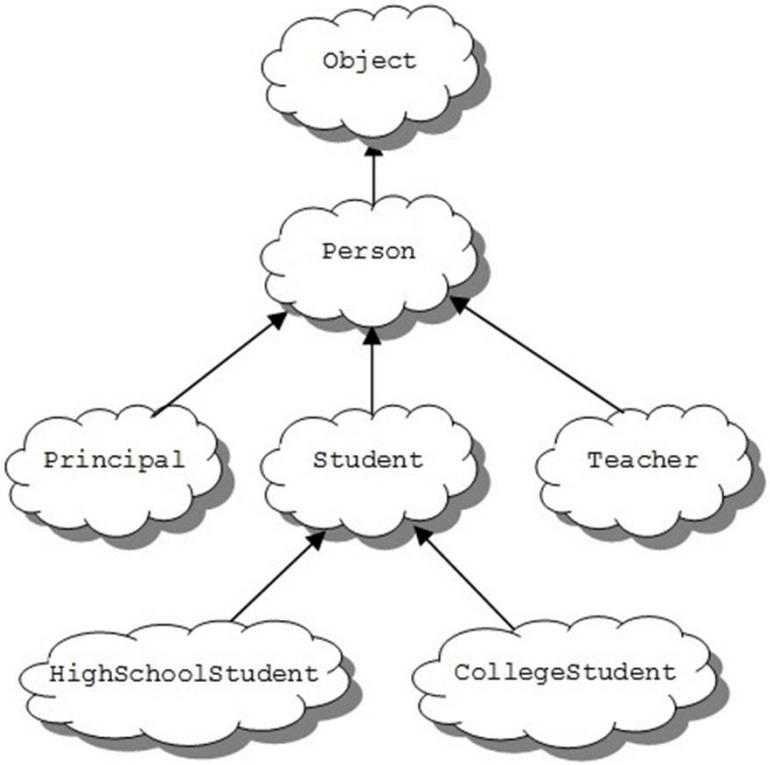
Nasleđivanje (inheritance)

- Nasleđivanje definiše odnos između klasa, a samim tim i objekata u jednom objektno-orijentisanom sistemu.
- Sve klase u Javi imaju neku drugu klasu iznad, koja se može nazvati njihovim roditeljom.
- Specijalna klasa, jedina koja nema svog roditelja je klasa koja se zove Object.
- Svi drugi objekti u Javi predstavljaju naslednike ovog tipa.



Nasleđivanje (inheritance)

- Možemo reći da smo svi mi instance jedne klase Osoba (Person). Ipak svako od nas može obavljati različite poslove, pa se na toj osnovi mogu kreirati novi skupovi osoba u zavisnosti od posla koji obavljaju (Principal, Student, Teacher).
  - Student može biti na visokoj školi ili koledžu. Ako bismo to preveli na klasni model objektno-orijentisanog jezika, može se reći da klasa HighSchoolStudent nasleđuje klasu Student, a da klasa Student nasleđuje klasu Person.
- Svaka klasa koja nema kreiranog roditelja, kao svoju nadklasu ima klasu Object. Ova struktura je prikazana na ilustraciji.



Nasleđivanje - implementacija

- Da bismo naglasili da klasa u Javi nasleđuje neku drugu klasu koristimo ključnu reč extends. Pogledajmo to na primeru klasa Person i Student. Jasno je da klasa Student nasleđuje klasu Person.

```
class Person {

}
class Student extends Person {

}
```

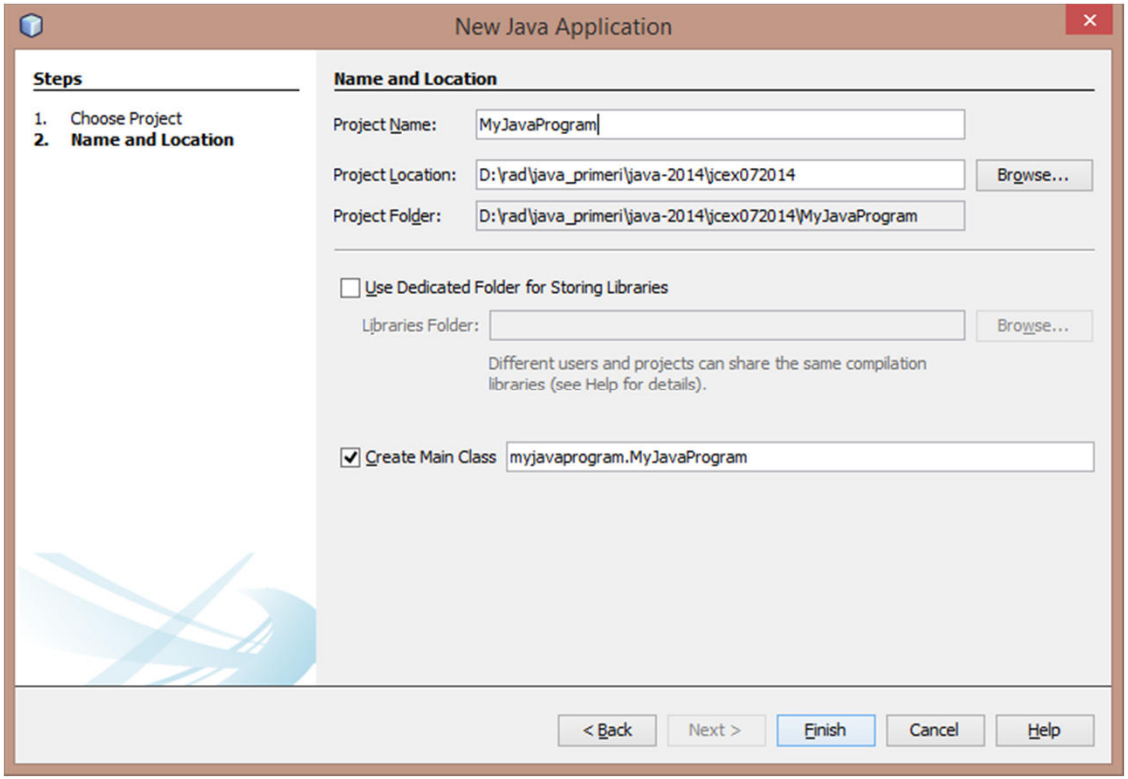
LINKgroup

Distance Learning System

Core Java Programming

Članovi klase

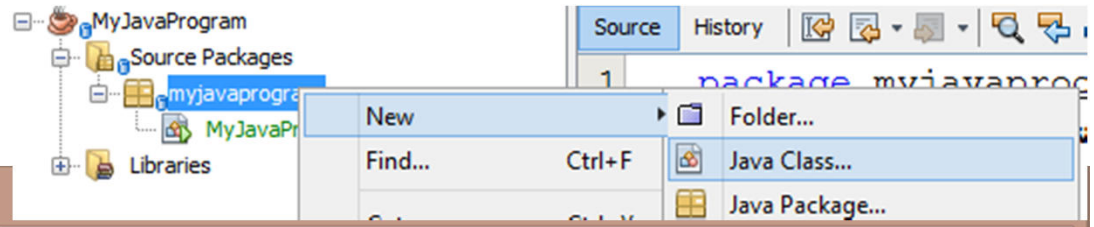
„Prva“ Java klasa
(jcex072014 MyJavaProgram)



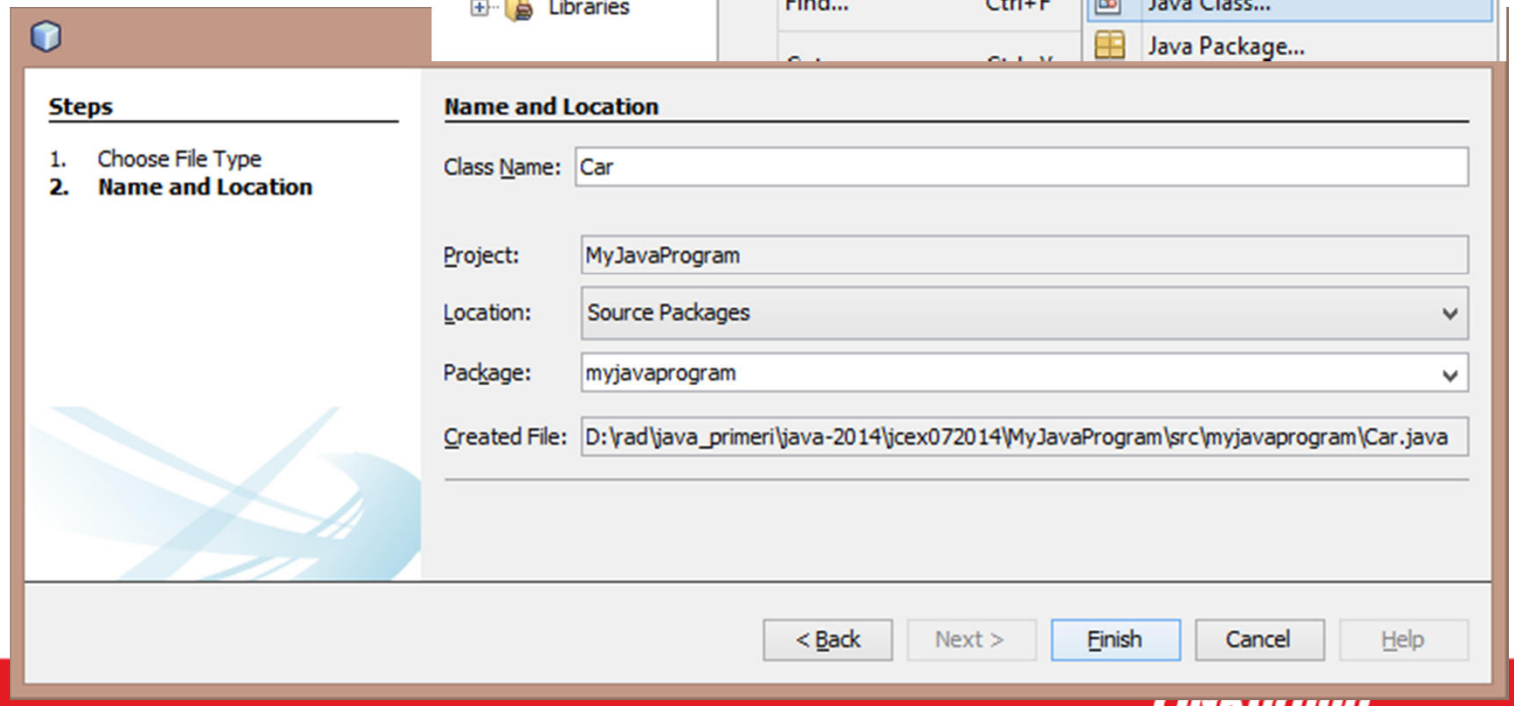
- Da bi ispratili primere u nastavku, treba kreirati novi Java Application projekat pod nazivom MyJavaProgram
- Projekat kreiran po navedenim parametrima daje sledeću strukturu

Prva Java klasa

- Da bi kreirali novu klasu, koristimo opciju **New->Java Class**



- U prozoru koji se otvara upisujemo naziv klase: Car (pod uslovom da pravimo klasu Car)



Struktura klase (jcex072014 MyJavaProgram)

- Prvom linijom definisan je paket u kome se nalazi klasa definisana u kodu ispod.
- Sledi definicija klase.
  - Zaglavlje
    - Public modifikator
    - Ključna reč **class**
    - Naziv klase
  - Telo
    - Predstavljeno je vitičastim zagradama
    - Trenutno je prazno

```
package myjavaprogram;
public class Car {

}
```

Članovi klase

- Dele se na **instancne** i **statičke**
- Članovi su:
  - **Polja**
  - **Metode**
  - **Druge klase**

Modifikatori pristupa

- Modifikatori pristupa predstavljaju opseg dostupnosti određenih članova klase
- Modifikatori pristupa su:
  - **Private** - članovi su vidljivi samo u okviru klase u kojoj se nalaze
  - **Package-private** (često se naziva samo package) svi drugi članovi istog paketa imaju pristup komponenti
  - **Protected** elementima se može pristupiti iz same klase gde su definisani i iz izvedenih klasa
  - **Public** bilo koji objekat ima pristup nad elementom

Polja klase

- Polja su zapravo promenljive koje se nalaze u okviru klase
- Dele se na **instancna** i **statička**

```
package myjavaprogram;
public class Car {
    String make;
    String model;
    int numDoors;
    static int wheels = 4;
}
```

Instancno

```
public static void main(String[] args) {
    Car c = new Car();
    c.make = "BMW";
}
```

Statičko

```
public static void main(String[] args) {
    System.out.println(Car.wheels);
}
```

Private modifikator pristupa

- Na prethodnom primeru ne postoje modifikatori pristupa, i zato su polja javna.
- Ako bi postavili privatni modifikator na neko od polja, ono ne bi bilo dostupno izvan klase:

```
package myjavaprogram;
public class Car {
    private String make;
    String model;
    int numDoors;
    static int wheels = 4;
}
```

Pokušaj pristupanja polju izvan klase izaziva grešku

```
public sta                                  tring[] args) {
    Car ca
    car.make = "BMW";
}
```

make has private access in Car

(Alt-Enter shows hints)

Metode

- Pored polja, među članovima klase nalaze se i **metode**
- Metode nisu ništa drugo do funkcije koje se nalaze unutar klasa
- Obzirom da je Java poptuno objektan jezik, funkcija kao pojam i ne postoji, već samo metoda
- Metod koji se aktivira automatski prilikom kreiranja klase, naziva se **konstruktor**
- Unutar konstruktora, možemo postaviti neke inicijalne vrednosti instanciranom objektu

Konstruktor (jcex072014 Constructor)

- Konstruktor se automatski aktivira prilikom instanciranja klase

1
```
package myjavaprogram;
public class Car {
    String make;
    String model;
    int numDoors;
    static int wheels = 4;
    public Car(){
        System.out.println("The car is being created");
    }
}
```

2
```
public static void main(String[] args) {
    Car car = new Car();
}
```

3
```
run:
The car is being created
BUILD SUCCESSFUL (total time: 0 seconds)
```

Metod main

- Pored konstruktora, još jedan veoma važan metod je: **main**
- Ovaj metod već jako dobro poznajemo, jer bez njega nije moguće startovati Java program (to ne znači da program mora imati ovaj metod)
- Podsetimo se metode main. Sada bi trebalo da nam neke stvari budu mnogo jasnije:

Javno dostupan

Ne vraća vrednost

```
package myjavaprogram;
public class MyJavaProgram {
    public static void main(String[] args) {
        Car car = new Car();
    }
}
```

Potpis metode

Telo metode

Statički metod

Prihvata parametre

Statičke metode

- Statičke metode nemaju veze sa objektom klase, već sa samom klasom. Zbog toga ne mogu pristupati instancnim članovima, već samo statičkim

```
package methods1;
public class Car {
    String make;
    String model;
    int numDoors;
    static int wheels = 4;
    public static void HowManyWheels(){
        System.out.println(wheels);
    }
}
```

Poziv

```
Car.HowManyWheels();
```

Instancne metode (jcex072014 Methods1)

- Karakteristika instancnih metoda je da postoje u kontekstu objekta. To znači da su im dostupni svi članovi objekta klase u kojoj se nalaze
- Ove metode nije moguće startovati bez prethodno kreiranog objekta

```java
package methods1;
public class Car {
    String make;
    String model;
    int numDoors;
    static int wheels = 4;
    public static void HowManyWheels(){
        System.out.println(wheels);
    }
    public Car(){
        model = "I5";
    }
}
```

```java
Car car = new Car();
System.out.println(car.model);
```

Isti podatak

```
run:
I5
BUILD SUCCESSFUL (total time: 0 seconds)
```

Instancne metode (jcex072014 Methods1)

```
public static void HowManyWheels(){...3 lines }
void printDetails(){
    System.out.println("Make " + make);
    System.out.println("Model " + model);
    System.out.println("Number of doors " + numDoors);
}
public Car(){...3 lines }
}
```

- Metoda sa leve strane ima naziv printDetails i služi za prikaz vrednosti polja kreiranog automobila:

- Da bi je aktivirali, kucamo:

```
public static void main(String[] args) {
    Car car = new Car();
    car.printDetails();
```

- Izlaz programa je:

```
run:
Make null
Model I5
Number of doors 0
BUILD SUCCESSFUL (total time: 0 seconds)
```

- Pokušajte da modifikujete program tako da prikaže smislene rezultate

Instancne metode (jcex072014 Methods1)

- Pokušajte da ispravite program iz prethodnog primera, tako da prikaže smislene rezultate

Parametrizacija metoda

- Jedna od ključnih osobina metoda je mogućnost prihvatanja i vraćanja parametara
- Parametri koji ulaze u metod, nazivaju se **ulazni parametri**
- Parametri koje metod vraća, nazivaju se **izlazni parametri** (u pitanju je zapravo samo jedan izlazni parametar)
- Prilikom kreiranja metode, moramo u njegovom potpisu naznačiti koje će parametre vratiti i koje će prihvatiti
- Metod ne mora vratiti ništa, niti prihvatiti ništa, ali onog trenutka kada ga kreiramo, u obavezi smo da poštujemo njegovu definiciju (njegov potpis)
- Za vraćanje vrednosti iz metode, koristimo ključnu reč **return** kojoj sledi vrednost

Parametrizacija metoda (jcex072014 MethodParams)

- Jedan parametrizovan statički metod, može izgledati ovako:

Tip izlaznog parametra

```
public static int add(int a, int b){
    int result = a + b;
    return result;
}
```

Slanje na izlaz

Tipovi ulaznih parametara

- Metod kasnije možemo pozvati na sledeći način:

```
public static void main(String[] args) {
    int res = add(2,3);
    System.out.println(res);
}
```

Parametrizacija metoda - void

- Neretko, metod ne vraća nikakav rezultat
- Kod ovakvih metoda moramo dati do znanja sistemu da metod neće vratiti ništa, i to radimo ključnom rečju **void**

```
public static void emptyResult(){
    System.out.println("Hello, nothing for you!");
}
```

Vrednosti i reference

- Parametre možemo proslediti metodama kao reference ili kao vrednosti
- Ovo zavisi od tipa parametara
  - Svi složeni tipovi (objekti i nizovi) prosleđuju se uvek po referenci
  - Svi prosti tipovi prosleđuju se po vrednosti

Prosleđivanje parametara po referenci

- Ukoliko postoji sledeći metod

```
public static void passByReference(Car car){
    car.model = "Beetle";
}
```

- I obratimo mu se na sledeći način

```
Car car = new Car();
car.model = "Renault 4";
passByReference(car);
System.out.println(car.model);
```

- Šta možemo da očekujemo kao rezultat?
  - Pokušajte da samostalno uradite primer i vidite rezultat

Prosleđivanje parametara po vrednosti

- Ukoliko postoji sledeći metod

```
public static void passByValue(int x){
    x = 25;
}
```

- I obratimo mu se na sledeći način

```
int x = 10;
passByValue(x);
System.out.println(x);
```

- Šta možemo da očekujemo kao rezultat?
  - Pokušajte da samostalno uradite primer i vidite rezultat

Proizvoljan broj argumenata

- U nekim slučajevima, hoćemo da metod sadrži proizvoljan broj parametara
- Ovo je moguće uraditi na više načina
  - Korišćenjem **varargs** parametrizacije
  - **Preopterećenjem** metode

Prosleđivanje proizvoljnog broja parametara

- Da bi prosledili proizvoljan broj parametara metodi, parametre metode moramo označiti ulazni parametar na specifičan način

```
public static int useVarargs(int... operands){
    int result = 0;
    for(int i=0;i<operands.length;i++){
        result+=operands[i];
    }
    return result;
}
```

- Prilikom poziva, umesto jednog parametra, prosleđujemo ni jedan ili više.

```
System.out.println(useVarargs(1,5,12));
```

- U samoj metodi, parametre ćemo dobiti u formi **niza**

Vežba – Kalkulator (jcex072014 Calculator)

- Potrebno je kreirati klasu kalkulator koja ima dva svojstva: operand1 i operand2.
- Klasa poseduje metode:
  - **add**, koja kao rezultat vraća zbir dva operanda
  - **sub**, koja kao rezultat vraća razliku dva operanda
  - **mul**, koja kao rezultat vraća proizvod dva operanda
  - **div**, koja kao rezultat vraća količnik dva operanda
- Sve vrednosti su tipa double.

LINKgroup

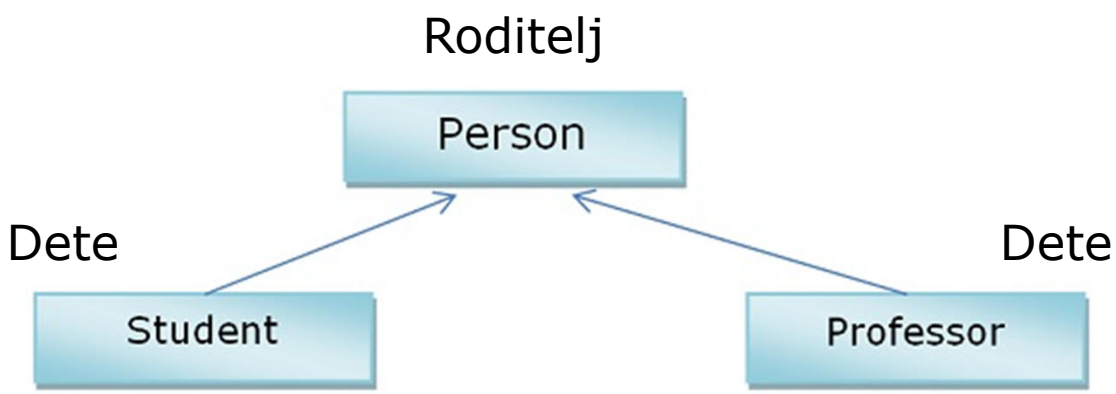
Distance Learning System

Core Java Programming

Nasleđivanje i polimorfizam

Nasleđivanje

- Nasleđivanje predstavlja osobinu klase da prihvati osobine neke druge klase, pri čemu klasa, koja biva nasleđena, ima status roditeljske, dok klasa koja nasleđuje ima status deteta klase



Nasleđivanje

- Prilikom nasleđivanja postoji nekoliko pravila koja treba da budu ispoštovana:
  - Klasa koja nasleđuje može imati samo jednu roditeljsku klasu
  - Sa roditeljske klase na dete klasu se prenose "samo" **public**, **protected** i **package-private** atributi

```
public class Car {
    public String model;
    String make;
    protected int wheels;
    private int doors;
}
```

Nasleđuju se

Ne može se naslediti

Nasleđivanje - primer (jcex082014 Inheritance)

- Za dalje praćenje primera treba kreirati klasu Person:

```
package inheritance;
public class Person {
    public String firstName;
    public String lastName;
    public Person(String firstName, String lastName)
    {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    public void show() {
        System.out.println("Person: " + firstName + " " + lastName);
    }
}
```

- Analizirajte ovu klasu

Nasleđivanje

- Prethodno kreiranu klasu je moguće naslediti. Na primer:

```
package inheritance;
public class Student extends Person {
    public String indexNumber;
    public Student(String firstName, String lastName, String indexNumber){
        super(firstName,lastName);
        this.indexNumber = indexNumber;
    }
}
```

- Klasa ima dve zanimljive komponente:
  - **this** (ukazuje na aktuelni objekat)
  - **super** (ukazuje na roditelja)
- Takođe, primećujemo da klasa nije nasledila konstruktor

Nasleđivanje (instanciranje nasleđene klase)

- Klasu Student možemo koristiti, čak i bez znanja o postojanju klase Person

```
public static void main(String[] args) {
    Student st = new Student("John", "Smith", "10/2014");
    st.
}
```

firstName	String
indexNumber	String
lastName	String
equals(Object obj)	boolean

- Okruženje (NetBeans) je detektovalo da su neka polja nasleđena a neka autentična, pa ih je prikazalo zadebljanim fontom

Zabrana daljeg nasleđivanja

- Ako ne želimo da dozvolimo dalje nasleđivanje klase, među njene modifikatore uključujemo i ključnu reč **final**
- Final kod definicije klase znači da ona ne može da se nasledi, dok kod definicije polja znači da neće moći da mu bude dodeljena vrednost (konstanta)

```
package inheritance;
public final class Person {
    public String firstName;
    public String lastName;
    public Person(String firs
    {
        this.firstName = :
```

Polimorfizam

- Polimorfizam u Objektno Orjentisanom programiranju znači da će se istoimeni metod u jednoj klasi različito ponašati od istoimenog metoda u drugoj klasi, pri čemu, obe klase imaju različitog roditelja
- Na primer, ako bi rekli da su pas i ptica podklase klase životinja, te da oboje imaju metod: kreći se, ova dva metoda, iako se isto zovu, imala bi potpuno drugačiju realizaciju, jer ptica leti, a pas hoda
- Da bi vršili polimorfovanje klase, moramo poznavati pojmove **prepisivanja** i **preopterećenja** (override i overload).

Prepisivanje metoda

- Prepisivanje je pojava u kojoj metoda roditeljske klase biva prepisana od strane dete klase
- Na primer, kada bi u klasi Student, dodali sledeći kod, mogli bi da kažemo da je došlo do prepisivanja

Dodatak klasi Student

```
@Override
public void show(){
    System.out.println("Student: " + firstName + " " + lastName + " " + indexNumber);
}
```

Metod već postoji u klasi Person i biće prepisan

```
public void show() {
    System.out.println("Person: " + firstName + " " + lastName);
}
```

Prepisivanje metoda

- Iako smo prepisali roditeljski metod, to ne znači da je on bespovratno izgubljen, odnosno, da je stvarno prepisan
- Ukoliko nam ustreba, možemo doći do njega pomoću ključne reči super:

```
@Override
public void show(){
    System.out.println("Student: " + firstName + " " + lastName + " " + indexNumber);
    super.show();
}
```

- Rezultat koda je, nakon startovanja sledeći:

```
run:
Student: John Smith 10/2014
Person: John Smith
BUILD SUCCESSFUL (total time: 0 seconds)
```

- Očigledno je da su aktivirani i prvi i drugi metod

Prepisivanje metoda

- Jedan od najčešćih slučajeva prepisivanja jeste prepisivanje metode toString.
- Ako pokušamo da prikažemo objekat klase Student na izlazu, dobićemo rezultat poput sledećeg

```
System.out.println(student);  ──────────────>  inheritance.Student@15db9742
```

- To je zato što Java automatski poziva roditeljski toString metod (i to ne od Klase Person, već od prve roditeljske klase koja ima toString, a to je klasa Object).
- Ako prepišemo metod toString u klasi Student:

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

- Na izlazu ćemo kao rezultat dobiti: `Hello from John`

Polimorfovanje

- Da bi došlo do polimorfizma, mora postojati bar dve dete klase
- Druga dete klasa može biti klasa **Profesor**

```
package inheritance;
public class Professor extends Person {
    public String className;
    public Professor(String firstName, String lastName, String className) {
        super(firstName, lastName);
    }
    @Override
    public void show(){
        System.out.println("Professor " + firstName + " " + lastName + " " + className);
    }
}
```

- Analizirajte metode show u sve tri klase

Polimorfovanje

- Ako sada instanciramo sve tri klase, a zatim aktiviramo njihove metode show, dobićemo tri različita rezultata na izlazu:

```
public static void main(String[] args) {
    Person person = new Person("John", "Davidson");
    Student student = new Student("John", "Smith", "10/2014");
    Professor professor = new Professor("Edward", "Owen", "Java Programming");
    person.show();
    student.show();
    professor.show();
}
```

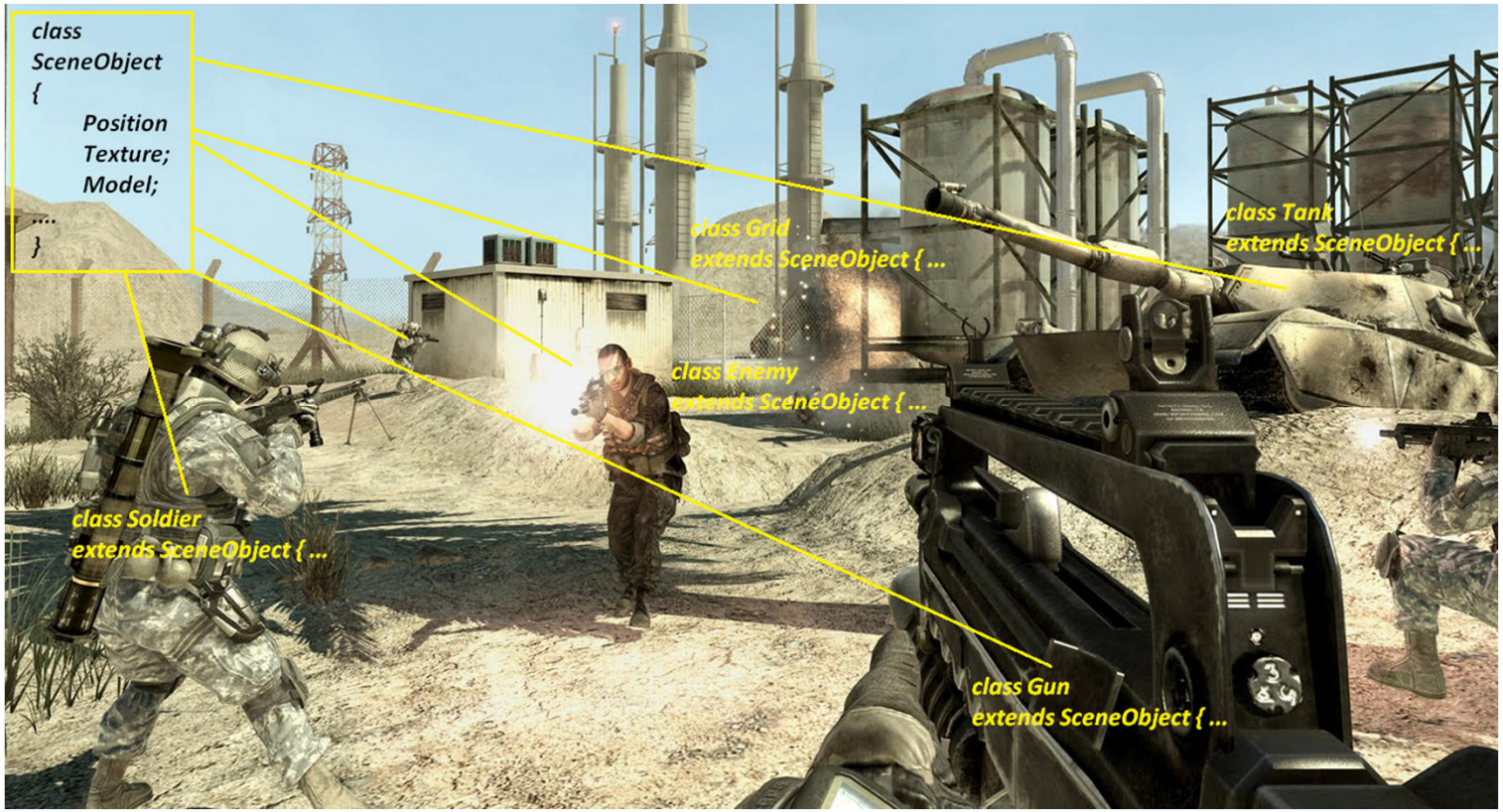
```
run:
Person: John Davidson
Student: John Smith 10/2014
Person: John Smith
Professor Edward Owen null
BUILD SUCCESSFUL (total time: 0 seconds)
```

- Rezultat je različit za istoimene nasleđene metode, te možemo reći da je došlo do **polimorfovanja**

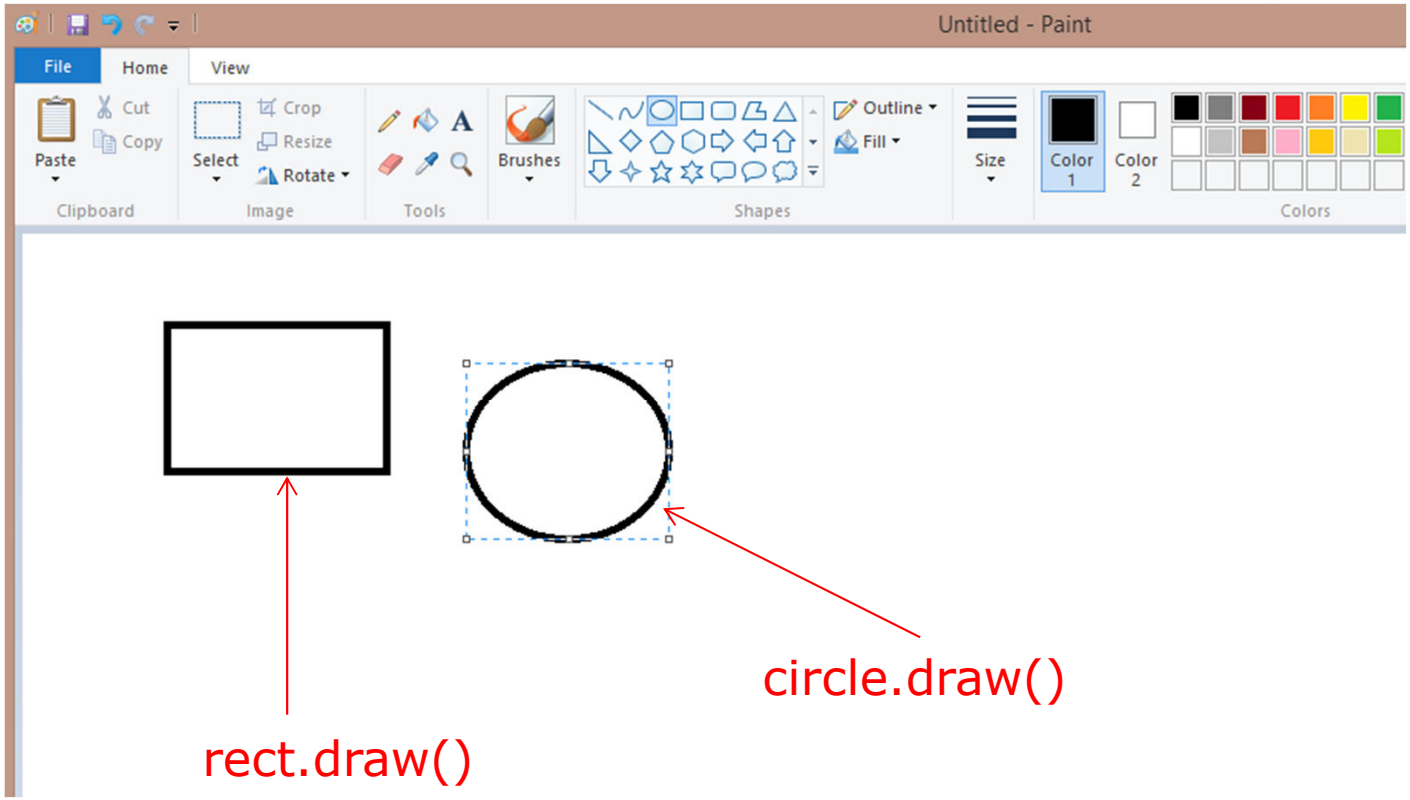
Polimorfovanje i nasleđivanje u praksi?

- Polimorfovanje je prirodna posledica nasleđivanja a u realnom sistemu, pojavljuje se veoma često:

Polimorfovanje i nasleđivanje u praksi?



Polimorfovanje i nasleđivanje u praksi?



Preopterećenje (overload) metoda

- Unutar jedne klase možemo napraviti više istoimenih metoda sa različitim potpisom:

```
@Override
1 public void show(){
      System.out.println("Student: " + firstName + " " + lastName + " " + indexNumber);
      super.show();
  }
2 public void show(String firstName){
      System.out.println(firstName);
  }
3 public void show(String firstName, String lastName){
      System.out.println(firstName + " " + lastName);
  }
```

- Jedini način da specificiramo koji ćemo metod pozvati, jeste da upotrebimo njegov potpis

```
student.show();
student.show("Peter");
student.show("Peter","Jackson");
```

Vežba (jcex082014 Shapes)

- Potrebno je napraviti jednu klasu Shape koja bi sadržala neke podatke o geometrijskom obliku:
  - poziciju (x,y)
  - boju i naziv.
- Potrebno je napraviti tri klase koje nasleđuju ovu klasu, jednu za krug, jednu za kvadrat i jednu za pravougaonik.
- Svaka treba da sadrži metodu za izračunavanje površine, kao i sopstvene atribute koji su neophodni za ovo izračunavanje (strane za pravougaonik i kvadrat i poluprečnik za krug).
- Krug, takođe, treba da poseduje i konstantu PI.

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 asptraktnom, 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(Strin
        Person person = new Person();
    }
}
```

Person is abstract; cannot be instantiated

(Alt-Enter shows hints)

Nasleđivanje osobina od apstraktnih klasa

- To što klasa ne može da bude instancirana, ne znači da ne može sadržati funkcionalnosti i polja:

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

- 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 { }
```

- 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();
    }
}
```

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
        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;
    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.

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 dođemo u situaciju da metod receiveHit uopšte ne postoji, i zbog toga ga označavamo apstraktno, kako bi naterali autora heroja da ga implementira.

```
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();
}
```

- *Da li smo mogli i sami da izvršimo neku generičku implementaciju ovih metoda (receiveHit, primaryFire i secondaryFire), a autoru heroja ponudimo da ih prepiše samo ukoliko želi? Jesmo, i to bi takođe bilo sasvim ispravno rešenje ovog problema.*

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

```java
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:

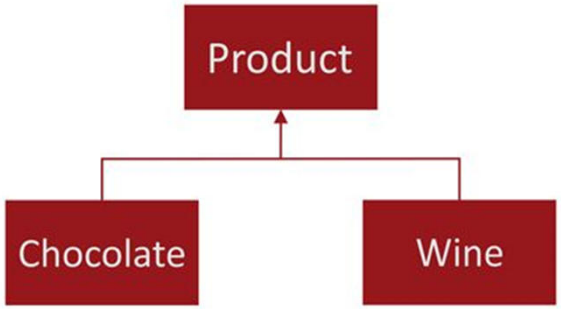
```
public interface MyInterface {
    public static void interfaceMethod(){
        System.out.println("Hello");
    }
```

- Interfejs može imati podrazumevanu implementaciju metode:

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

Zadatak

Zamislite da je potrebno da modelujete informacioni 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 to String za prikaz informacija o objektu.

LINKgroup

Distance Learning System

Core Java Programming

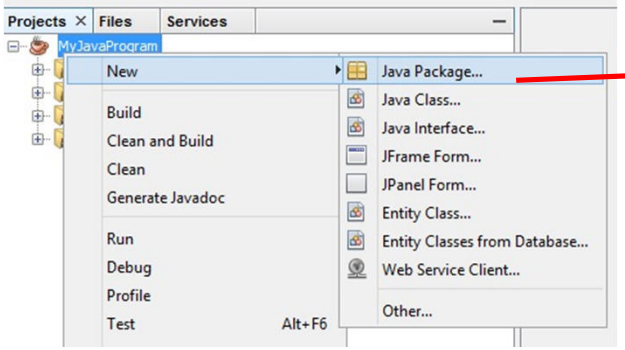
Paketi

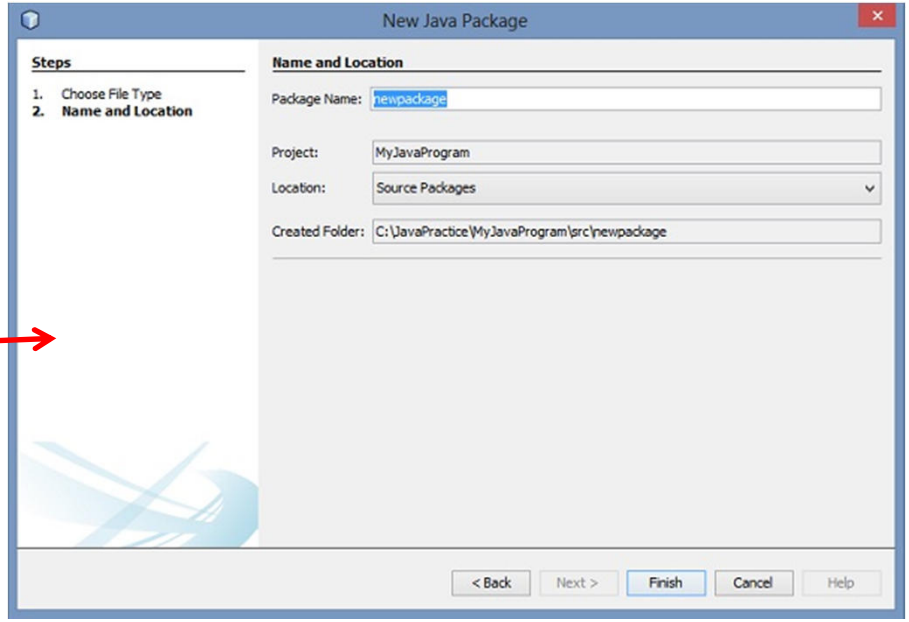
Paketi

- Peketi 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



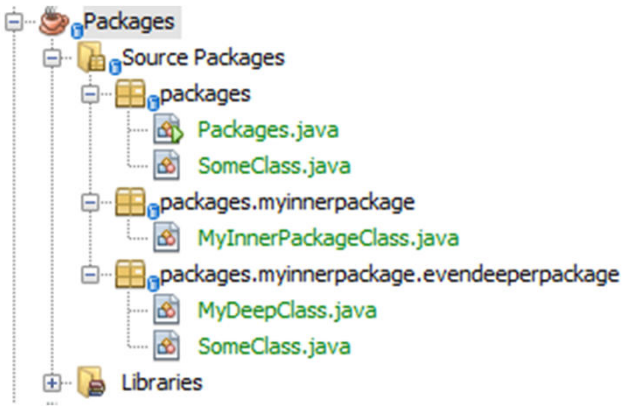


LINKgroup

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

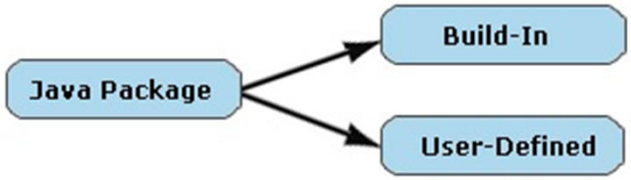
Učitava se klasa iz drugog paketa

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