DIZAJN I RAZVOJ SOFTVERA

Nasljeđivanje

Polimorfizam

Interfejsi

Uvod u nasljeđivanje

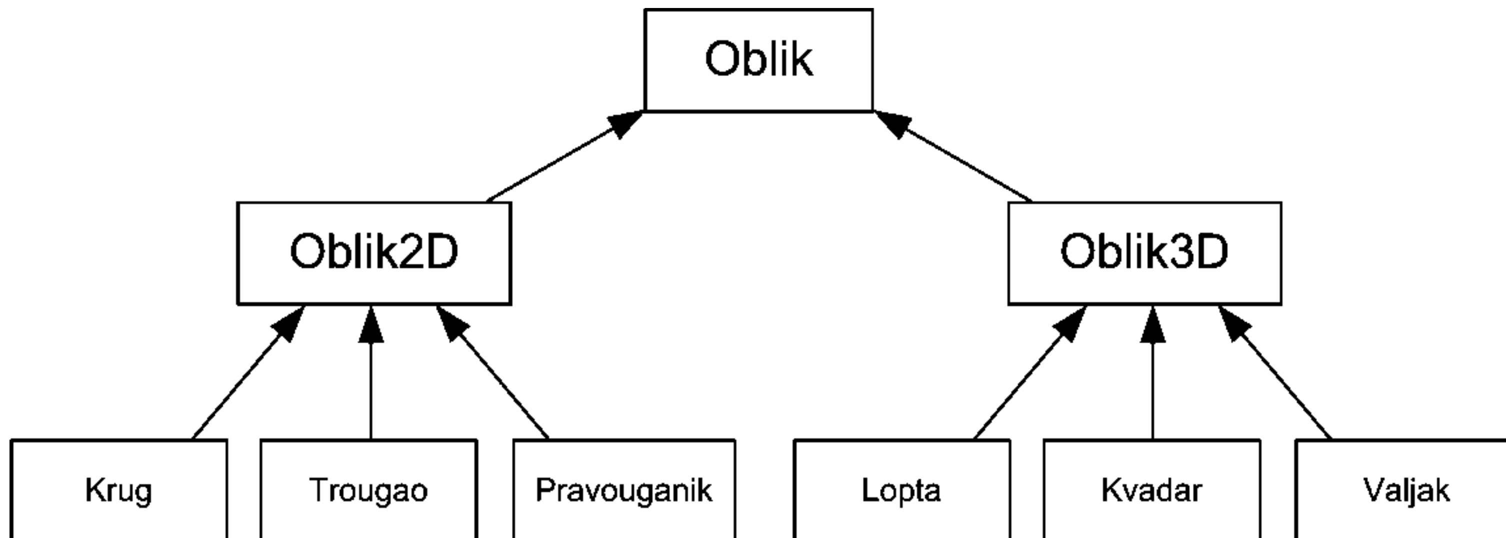
- Kreiranje novog softvera rijetko kad počinje od nule, već se zasniva na postojećem softveru.
- Najbolji i najbrži način kreiranja je imitirajući postojeći softver, poboljšavajući ga i kombinujući sa drugim softverom.
- Tradicionalni softverski dizajn je dobrim dijelom zanemarivao ovaj aspekt razvoja softvera, dok je u OO dizajnu on od suštinskog značaja.
- **Nasljeđivanje** predstavlja oblik ponovne upotrebe softvera kojim se, iz postojeće klase, kreira nova klasa.
- Nova klasa nasljeđuje članove postojeće klase, uz mogućnost da doda nove članove ili modifikuje postojeće. Na ovaj način se smanjuje vrijeme razvoja aplikacije korišćenjem postojećeg provjerenog programskog koda.
- Postojeća klasa se naziva **superklasa**, a nova **potklasa**. Proces nasljeđivanja se može nastaviti u smislu da nova klasa može predstavljati superklase budućih klasa.

Uvod u nasljeđivanje

- Pored postojećih podataka i metoda, potklasa može definisati svoje podatke i metode, na ovaj način postajući specifičnija od svoje superklase.
- Potklasa može i modifikovati naslijeđene podatke i metode, prilagođavajući ih svom tipu. Zbog ovoga se nasljeđivanje još naziva i specijalizacija. Superklase su opštije, a potklase specifičnije.
- Za potklasu se još kaže da proširuje (eng. *extends*) superklasu, jer može dodati nove članove u odnosu na superklasu.
- U hijerarhiji nasljeđivanja, potklasa može imati direktnu superklasu i indirektne superklase. Indirektna superklasa je bilo koja klasa u hijerarhiji nasljeđivanja iznad direktne superklase.
- Najviša klasa u Javinoj klasnoj hijerarhiji je `Object` (deklarisana u paketu `java.lang`) i sve Javine klase direktno ili indirektno nasljeđuju klasu `Object`.

Uvod u nasljeđivanje

- Nasljeđivanje odražava **vezu tipa jeste** (eng. *is-a relationship*), nasuprot kompozicije koja odražava vezu tipa ima.
- U vezi tipa jeste, objekat potklase se može tretirati kao objekat superklase.
- Primjer nasljeđivanja klase `Oblik` je dat na slici ispod.
- Svaka strelica na slici predstavlja vezu tipa jeste. Prateći strelice možemo reći da `Krug` jeste `Oblik2D`, jeste `Oblik`, i na, kraju, jeste `Object`.
- Potklasa može modifikovati (**redefinisati**) naslijeđene metode, radi prilagođavanja specifičnostima potklase.



protected pravo pristupa

- Pored `public` i `private` modifikatora pristupa, postoji i `protected`, koji pruža prava pristupa između `public` i `private`.
- Ukoliko se član klase deklariše kao `protected`, njemu mogu pristupiti svi članovi te klase, članovi potklasa te klase, kao i ostale klase u istom paketu.
- Ako nije specificiran modifikator pristupa `public`, `protected` ili `private` za određeni član klase, taj član ima pravo pristupa paketa (eng. *package access*), tj. mogu mu pristupiti sve klase iz istog paketa.

	Public	Protected	Bez modifikatora	Private
Ista klasa	Da	Da	Da	Da
Potklasa iz istog paketa	Da	Da	Da	Ne
Klasa iz istog paketa koja nije potklasa	Da	Da	Da	Ne
Potklasa iz drugog paketa	Da	Da	Ne	Ne
Klasa iz drugog paketa koja nije potklasa	Da	Ne	Ne	Ne

protected i proširivost softvera

- Jedan od ključnih faktora kvaliteta softvera je **proširivost** (eng. *extendibility*) ili **skalabilnost** koji predstavlja mogućnost prilagođavanja softvera promjenama u njegovoj specifikaciji.
- U direktnoj vezi sa proširivošću softvera je **kriterijum modularnog kontinuiteta** (eng. *modular continuity*) po kome bi mala promjena u specifikaciji problema kojim se bavi određeni modul trebala da se odrazi samo na taj modul ili na mali broj modula.
- Ako bismo matematički predstavili funkciju promjene softvera (y osa) u zavisnosti od promjene specifikacije (x osa), ta funkcija ne bi smjela da ima naglih skokova.
- U vezi sa proširivošću softvera i modularnim kontinuitetom, korišćenje `protected` podataka superklase je u suprotnosti sa njima, dok im upotreba `set` i `get` metoda za manipulaciju podacima superklase ide u prilog.

Potpuna klasa Knjiga

```
public class Knjiga {
   private String ime;
   private int brojStrana;
   public Knjiga(){
      this(null, 0);
   }
   public Knjiga(String imeKnjige){
      this(imeKnjige, 0);
   }
   public Knjiga(int brStr){
      this(null, brStr);
   }
   public Knjiga(String imeKnjige, int brStr){
      postaviKnjigu(imeKnjige, brStr);
   }
   public void postaviKnjigu(String imeKnjige, int brStr){
      setIme(imeKnjige);
      setBrojStrana(brStr);
   }
   public String getIme() {
      return ime;
   }
```

```
   public void setIme(String S) {
      ime = S;
   }
   public int getBrojStrana() {
      return brojStrana;
   }
   public void setBrojStrana(int N) {
      if (N > 0) brojStrana = N;
      else throw new
         IllegalArgumentException("Broj strana negativan!");
   }
   public String toString(){
      return String.format("Knjiga: %s\nbroj strana: %d\n",
           getIme(), getBrojStrana());
   }
}
```

Bacanje izuzetka

Klasa KnjigaSaCijenom

nasljeđivanje

```
public class KnjigaSaCijenom extends Knjiga {

  private double cijena;
  public KnjigaSaCijenom(String imeKnjige, int brStr, double cijenaKnjige){
    super(imeKnjige, brStr);
    setCijena(cijenaKnjige);
  }
  public double getCijena() {
    return cijena;
  }
  public void setCijena(double cijenaKnjige) {
    if (cijenaKnjige > 0)
      cijena = cijenaKnjige;
    else
      throw new IllegalArgumentException("Cijena mora biti pozitivan broj");
  }

  @Override
  public String toString(){
    return String.format("%scijena: %.2f \u20AC\n", super.toString(),
getCijena());
  }
}
```

Pozivanje konstruktora superklase

Redefinisanje metode

Pozivanje metode superklase

Klasa KnjigaSaCijenomTest

```
public class KnjigaSaCijenomTest {

  public static void main(String[] args) {
    KnjigaSaCijenom knjiga = new KnjigaSaCijenom("Robinzon Kruso", 234,
45.6);
    System.out.print(knjiga.toString());
  }

}
```

```
Knjiga: Robinzon Kruso
broj strana: 234
cijena: 45.60 €
```

Ispis

Ključna riječ extends

- Ključna riječ `extends` označava nasljeđivanje.
- Klasa `KnjigaSaCijenom` nasljeđuje promjenljive i metode klase `Knjiga`, ali su samo `public` i `protected` članovi superklase dostupni potklasi, `private` nisu.
- Konstruktor superklase se ne nasljeđuje.
- Sve Javine klase direktno ili indirektno nasljeđuju klasu `Object`. U tom smislu, mogli smo navesti u deklaraciji klase `Knjiga` da ona nasljeđuje klasu `Object` na sljedeći način:

```
public class Knjiga extends Object
```

  ali se to implicitno podrazumijeva, tj. ako ne navedemo koju klasu nasljeđuje data klasa, podrazumijeva se da nasljeđuje klasu `Object`.
- Iz klase `Object` se mogu naslijediti samo metode jer ona nema podataka.

Konstruktor potklase

- Konstruktori se ne nasljeđuju. Ipak, konstruktoru superklase se može pristupiti iz potklase.
- Prva stvar koju uradi konstruktor potklase je pozivanje konstruktora direktne superklase, bilo eksplicitno (navede se poziv konstruktora) ili implicitno (ne navede se poziv konstruktora). Ovo se radi da bi se obezbijedilo da su naslijeđene promjenljive instanci pravilno inicijalizovane.
- U superklasi `Knjiga`, nismo navodili poziv konstruktora njene superklase `Object`.
- U slučaju kad se ne navede poziv konstruktora superklase, Java implicitno poziva podrazumijevani ili konstruktor bez argumenata direktne superklase i to prvo izvršava u konstruktoru potklase. Podrazumijevani konstruktor klase `Object` ne radi ništa.
- Čak i ako potklasa nema svoje konstruktore, Java će kreirati podrazumijevani konstruktor koji će prvo pozvati podrazumijevani ili konstruktor bez argumenata direktne superklase.

Ključna riječ super

- Konstruktor direktne superklase se može pozvati pomoću ključne riječi `super`, što je u klasi `KnjigaSaCijenom` urađeno naredbom

```
super(imeKnjige, brStr);
```

- Eksplicitni poziv konstruktora superklase mora biti prva naredba u tijelu konstruktora potklase.
- Da nismo eksplicitno naveli poziv konstruktora superklase, Java bi implicitno pozvala konstruktor bez argumenata superklase `Knjiga`. Da u klasi `Knjiga` nismo deklarisali konstruktor bez argumenata, ovo bi dovelo do greške.
- Ključna riječ `super` se može iskoristiti da se pristupi `public` ili `protected` članu superklase. `private` podacima se ne može pristupiti iz potklase.
- U metodi `toString` klase `KnjigaSaCijenom`, pristupili smo istoimenoj metodi njene superklase `Knjiga` sa `super.toString()`. O ovome će biti riječi u nastavku.

Redefinisanje metoda i @Override anotacija

- `@Override` anotacija se koristi kad klasa treba da redefiniše neku metodu superklase.
- Metoda `toString()` klase `Knjiga` se mora prilagoditi klasi `KnjigaSaCijenom` tako da se pri ispisu uključi i cijena knjige.
- Kad kompajler naiđe na metodu deklarisanu sa `@Override` provjerava da li potpis metode potklase odgovara potpisu metode superklase. Ukoliko oni nisu potpuno isti, dolazi do greške.
- Greška je pokušati promijeniti modifikator pristupa redefinisane metode u modifikator sa manje prava pristupa.
- Drugim riječima, `public` metoda ne može postati `protected` ili `private` u potklasi, dok `protected` metoda ne može postati `private` u potklasi. Ukoliko bi to bilo dozvoljeno, narušila bi se veza tipa jeste - `KnjigaSaCijenom` jeste `Knjiga`, i moramo biti u mogućnosti da sve javne servise koje obezbeđuje klasa `Knjiga` koristimo i kod klase `KnjigaSaCijenom`.
- Dakle, metoda deklarisana kao `public` ostaje `public` u svim direktnim i indirektnim potklasama.

Ponovna upotreba koda u metodama potklase

- Zapisom `super.imeMetode`, iz potklase se može pozvati bilo koja javna zasjenjena metoda superklase.
- Ovo se ponekad može efikasno inkorporirati u metode potklase, dajući elegantno rješenje za dati problem.
- Konkretno, u redefinisanoj metodi `toString`, nismo iznova kreirali `String` reprezentaciju objekta, već smo pozvali istoimenu metodu superklase koja je kreirala dio stringa koji se odnosi na zajednički dio podataka (ime knjige i broj strana), a zatim nadovezali dio stringa karakterističan za potklasu.

```
public String toString() {
    return String.format("%scijena: %.2f \u20AC\n",
super.toString(),
                getCijena());
}
```

- Na ovaj način se izbjegava dupliranje koda u superklasi i potklasi, i olakšava se održavanje koda i ispravljanje grešaka.

Klasa Object

- Klasa `Object`, deklarisana u paketu `java.lang`, predstavlja superklasu svih Javinih klasa, tj. klasa `Object` predstavlja korijen u Javinoj klasnoj hijerarhiji.
- Ova klasa nema podataka, već samo metode, i to 11 metoda. Svi objekti nasljeđuju metode ove klase. U tabeli ispod su date sve metode klase `Object`. Metoda `wait` je preklopljena.

Metoda	Opis
`clone`	Kopira objekat koji je poziva. `Protected` metoda bez argumenata koja vraća referencu na `Object`. Podrazumijevana implementacija metode obavlja tzv. **plitko kopiranje** (eng. *shallow copy*) koje podrazumijeva da se vrijednost referencijskih podataka kopira, tako da reference originalnog objekta i kopije ukazuju na isti objekat u memoriji. Nasuprot toga imamo **duboko kopiranje** (eng. *deep copy*) kojim se kreira novi objekat za svaku promjenljivu referencijskog tipa. Duboko kopiranje može biti problematično jer referencijske promjenljive mogu sadržati reference, koje opet mogu sadržati reference. Duboko kopiranje podrazumijeva da se kreira novi objekat za svaku takvu referencu. Zbog ovoga se ne preporučuje redefinisanje metode `clone` da vrši duboko kopiranje, već da se takvo kopiranje vrši korišćenjem serijalizacije objekta (prevođenje objekta u sekvencu bitova).

Metoda	Opis
`equals`	Poredi dva objekta i vraća `true` ako su jednaki i `false` u suprotnom. Ima jedan argument tipa `Object`. Podrazumijevana implementacija metode poredi reference objekata, tj. da li reference ukazuju na isti objekat u memoriji. Ukoliko želimo da poredimo sadržaje dva objekta, potrebno je da redefinišemo ovu metodu.
`finalize`	`protected` metoda koju poziva garbage collector kad se utvrdi da nema referenci na predmetni objekat.
`getClass`	Tokom izvršavanja programa, svaki objekat zna svoj tip. Ova metoda vraća objekat klase `Class` (paket `java.lang`) koji sadrži informaciju o tipu objekta, kao što je ime klase (ime klase vraća metoda `getName` klase `Class`).
`hashCode`	U Javi, svakom objektu se dodjeljuje `int` kod, poznat kao hash kod. Hash kod identifikuje objekat i koristi se za brzo manipulisanje instancama. Od velike važnosti je i za smeštanje i povraćaj informacija smještenih u strukturi podataka poznatoj kao hash tabela. Klasa `Object` obezbeđuje podrazumijevano računanje hash kodova, koje se može redefinisati u korisničkoj klasi.

`wait`
`notify`
`notifyAll`
	Ove tri metode (zapravo pet, jer je `wait` preklopljena) se koriste kod multithreading-a.

`toString`	Metoda koja vraća `String` reprezentaciju objekta. Podrazumijevana implementacija ove metode vraća ime paketa (ako nije podrazumijevani paket), ime klase i heksadecimalnu vrijednost hash koda koji vraća metoda `hashCode`.

Uvod u polimorfizam

- Polimorfizam je pojam vezan za nasljeđivanje. Omogućava da se objekti izvedeni iz zajedničke superklase, bilo direktne ili indirektne, procesiraju kao da su u pitanju objekti superklase, što značajno pojednostavljuje programiranje.
- Posmatrajmo klasu Oblik2D, koja predstavlja superklasu klasa Krug, Trougao, Pravougaonik itd. Pretpostavimo da svaka od ovih klasa ima metodu za rotiranje koja se naziva rotiraj.
- Rotiranje svih objekata svih potklasa klase Oblik2D se elegantno može izvršiti koristeći polimorfizam na sledeći način: Kreiramo niz (ili bilo koju drugu kolekciju podataka) referencijskih promjenljivih superklase Oblik2D i u njih upišimo reference svih objekata klasa koje su izvedene iz Oblik2D. Nakon toga, dovoljno je da prođemo kroz taj niz i za svaku referencu pozovemo metodu rotiraj.
- Svaka izvedena klasa ima svoj način rotiranja. Na primjer, rotiranje kruga i kvadrata u 2D ravni nije isto.
- Pozivanjem metode rotiraj, mi svakom objektu šaljemo istu poruku, a u zavisnosti od klase, objekat će se rotirati na ovaj ili onaj način.

Uvod u polimorfizam

- Objekat „zna“ kako da reaguje da poslatu poruku.
- Ovo je ključni koncept polimorfizma - ista poruka, poslata mnoštvu različitih objekata, ima više formi rezultata.
- Zbog ove višeobličnosti rezultata, sam princip se naziva **polimorfizam**.
- Polimorfizam značajno pojednostavljuje dodavanje novih klasa postojećoj aplikaciji, uz minimalnu modifikaciju koda.
- Na primjer, ukoliko želimo da uvedemo nove 2D oblike u našu aplikaciju, i da njih rotiramo, potrebno je izvesti odgovarajuću klasu i realizovati njenu metodu `rotiraj`.
- Nakon toga, novokreirana klasa se može „uključiti“ u aplikaciju bez značajnije modifikacije postojećeg kôda, iako ta klasa nije bila planirana u početku. Modifikuje se samo klijentski kôd novokreirane klase, koji treba da uključi kreiranje instanci te klase i slanje odgovarajućih poruka.
- Na ovaj način, polimorfizam promoviše proširivost kôda.

Uopšteno vs. specifično programiranje

- Polimorfizam omogućava „uopšteno programiranje“, nasuprot „specifičnom programiranju“ karakterističnom za proceduralno programiranje.
- Koristeći polimorfizam, možemo programirati generalno, dok izvršnom okruženju (u slučaju Jave - JVM) prepuštamo da vodi računa o pojedinostima.
- Sve dok objekti pripadaju istoj hijerarhiji nasljeđivanja, možemo upravljati objektima bez da znamo kojem konkretno tipu pripadaju. Pod upravljanjem se ovdje podrazumijeva pozivanje metoda definisanih za predmetnu klasu.
- Tehnički, polimorfizam se implementira kreiranjem referencijskih promjenljivih superklase i upisivanjem referenci potklasa u te promjenljive. Za to je najjednostavnije koristiti niz referencijskih promjenljivih superklase.
- Prolaskom kroz niz i pozivanjem metoda svake reference, neće se pozivati metoda superklase, već verzija metode koja odgovara tipu referenciranog objekta.

Tip objekta, a ne promjenljive!

- **Tip referenciranog objekta, a ne tip referencijske promjenljive, određuje koja će se metoda pozvati!**
- Dodjeljivanje vrijednosti reference potklase promjenljivoj tipa superklase je dozvoljeno zbog veze tipa jeste. Objekat potklase jeste objekat superklase. Obrnuto ne važi.
- Ukoliko referencijska promjenljiva superklase sadrži referencu na objekat potklase, preko te promjenljive možemo pristupiti samo dijelu potklase naslijeđenom iz superklase, tj. ne može se pristupiti dijelu potklase koji je karakterističan samo za potklasu.
- Ako želimo da preko referencijske promjenljive superklase pristupimo članovima karakterističnim za potklasu, referenca superklase se mora eksplicitno konvertovati u tip potklase.
- Ovakva konverzija se naziva **konverzija naniže** (eng. *downcasting*).

Primjer polimorfizma

```
public class TestPolimorfizma {

  public static void main(String[] args) {

    Knjiga knjiga1 = new Knjiga("Robinzon Kruso", 234);
    KnjigaSaCijenom knjiga2 = new KnjigaSaCijenom("Grof Monte Kristo", 189, 
31.5);
    Knjiga knjiga3 = new KnjigaSaCijenom("Životinjska farma", 111, 15.8);
    KnjigaSaCijenom knjiga4 = (KnjigaSaCijenom) knjiga3;
    // greška bi se desila ako umjesto knjiga3 stavimo knjiga1

    System.out.println(knjiga1.toString());
    System.out.println(knjiga2.toString());
    System.out.println(knjiga3.toString());
    System.out.println(knjiga4.toString());
  }

}
```

```
Knjiga: Robinzon Kruso
broj strana: 234

Knjiga: Grof Monte Kristo
broj strana: 189
cijena: 31.50 €

Knjiga: Životinjska farma
broj strana: 111
cijena: 15.80 €

Knjiga: Životinjska farma
broj strana: 111
cijena: 15.80 €
```

Downcasting

Ispis

Dinamičko vezivanje

- Tokom izvršavanja, tip objekta na koji ukazuje promjenljiva, a ne tip promjenljive, određuje koja verzija metode će se zvati.
- Ovaj proces se naziva **dinamičko vezivanje** (eng. *dynamic binding*) ili **kasno vezivanje** (eng. *late binding*).
- U naredbi

```
KnjigaSaCijenom knjiga4 = (KnjigaSaCijenom) knjiga3;
```

  bez dowcasting-a `(KnjigaSaCijenom)` bi došlo do greške.
- U prethodnoj naredbi, greška bi se desila i ako `knjiga3` ne sadrži referencu na objekat tipa `KnjigaSaCijenom`. Tad bi se bacio izuzetak tipa `ClassCastException`.
- Greška bi se desila i ako pokušamo da pozovemo metodu `getCijena` pomoću promjenljive superklase `knjiga3`.
- Preko `knjiga3` možemo da pozovemo samo članove koje ima superklasa `Knjiga`.

Statičko vezivanje

- Vezivanje predstavlja povezivanje definicije metode sa pozivom metode. Postoje dvije vrste vezivanja, statičko i dinamičko (prethodni slajd).
- **Statičko** ili **rano vezivanje** (eng. *early binding*) je suprotno od dinamičkog vezivanja.
- Statičko vezivanje je vezivanje koje se može razriješiti tokom kompajliranja (eng. *compile time*).
- Sve `static`, `private` i `final` metode se vezuju statički. Zašto?
- Dinamičko vezivanje se razrješava tokom izvršavanja (eng. *run time*).

Apstraktne klase i metode

- Postoje klase koje ne mogu imati instance, i takve klase se nazivaju **apstraktnim**.
- U hijerarhiji nasljeđivanja, ovakve klase se koriste isključivo kao superklase, pa se ponekad nazivaju i **apstraktnim superklasama**.
- Svrha apstraktne klase je da posluži kao superklasa drugim klasama, tj. da se iz nje izvedu druge klase koje bi dijelile zajednički dizajn.
- Izvedene klase, koje mogu imati svoje instance, se nazivaju **konkretnim klasama**. U tom smislu, apstraktne klase su nepotpune.
- Potklase izvedene iz apstraktne klase, da bi mogle postojati njihove instance, moraju implementirati svaku deklarisanu metodu (implementacija nekih metoda se može naslijediti). Ako to ne urade, i same postaju apstraktne!
- Apstraktne klase specificiraju samo ono što je zajedničko svim izvedenim potklasama.

Apstraktne klase i metode

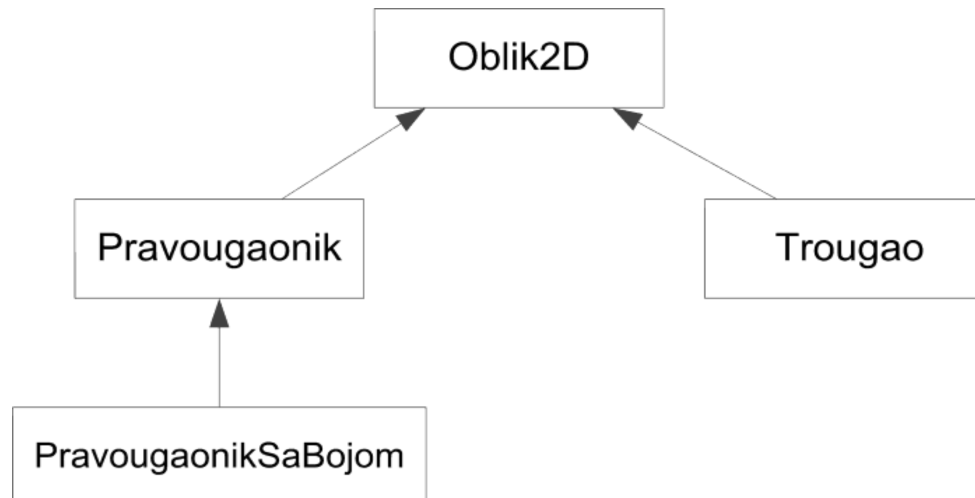
- U hijerarhiji nasljeđivanja, apstraktne klase se mogu naći na nekoliko nivoa. Na primjer, apstraktne klase `Oblik2D` i `Oblik3D` mogu naslijediti apstraktnu klasu `Oblik`.
- Klasa se deklariše apstraktnom pomoću ključne riječi **`abstract`**.
- Apstraktna klasa obično sadrži jednu ili više apstraktnih metoda.
- Metoda se deklariše apstraktnom takođe pomoću ključne riječi `abstract`, kao na primjer:

```
public abstract double povrsina();
```

- Apstraktne metode se ne implementiraju.
- Ako klasa sadrži bar jednu apstraktnu metodu, ona se mora eksplicitno deklarisati kao `abstract`, čak iako ta klasa sadrži neke implementirane metode.
- Da bi potklasa apstraktne superklase bila konkretna, ona mora implementirati sve apstraktne metode naslijeđene iz superklase.

Konstruktori i statičke metode. Primjer

- Konstruktori i statičke metode ne mogu biti apstraktni!
- Konstruktori se ne nasljeđuju, pa se apstraktni konstruktor ne bi mogao implementirati.
- Statičke metode se mogu naslijediti (ako su `public` ili `protected`), ali se ne mogu redefinisati!
- Dajmo primjer polimorfnog procesiranja na klasama čija hijerarhija nasljeđivanja je data na slici ispod.



Primjer polimorfnog procesiranja

```
public abstract class Oblik2D {

   double dim1, dim2;

   public Oblik2D(double x, double y) {
      dim1 = x;
      dim2 = y;
   }

   abstract double povrsina();

}
```

```
public class Pravougaonik extends Oblik2D {

   Pravougaonik(double x, double y){
      super(x,y);
   }

   public double povrsina(){
      return dim1 * dim2;
   }

}

public class Trougao extends Oblik2D {

   Trougao(double x, double y){
      super(x,y);
   }

   public double povrsina(){
      return dim1 * dim2 / 2;
   }

}
```

Primjer polimorfnog procesiranja

```
public class PravougaonikSaBojom extends Pravougaonik {

  byte rgb[];

  public PravougaonikSaBojom(double x, double y, int r, int g, int b) {
    super(x, y);
    setRgb(r, g, b);
  }

  public void setRgb(int r, int g, int b) {
    rgb = new byte[3];
    rgb[0] = (byte) (r - 128);  // prilagođavanje opsegu tipa byte (-128 do 127)
    rgb[1] = (byte) (g - 128);  // podatak tipa int se mora cast-ovati u byte
    rgb[2] = (byte) (b - 128);
  }

  public String getBoja(){
    return String.format("(R,G,B) = (%d,%d,%d)",
        rgb[0] + 128, rgb[1] + 128, rgb[2] + 128);
  }

}
```

Primjer polimorfnog procesiranja

```
public class DinamickoVezivanjeTest {

  public static void main(String[] args) {

    Oblik2D nizOblik2D[] = new Oblik2D[6];
    nizOblik2D[0] = new Pravougaonik(2,3);
    nizOblik2D[1] = new Pravougaonik(3,5);
    nizOblik2D[2] = new Trougao(3,1);
    nizOblik2D[3] = new Trougao(4,4);
    nizOblik2D[4] = new PravougaonikSaBojom(2,2,221,99,100);
    nizOblik2D[5] = new PravougaonikSaBojom(5,6,76,145,10);

    System.out.println("Štampanje svih objekata:");
    for(Oblik2D x: nizOblik2D)
      System.out.printf("Tip %s, površina %.2f\n",
         x.getClass().getName(), x.povrsina());

    System.out.println("\nŠtampanje boja objekata PravougaonikSaBojom:");
    for(Oblik2D x: nizOblik2D)
      if(x instanceof PravougaonikSaBojom){
        PravougaonikSaBojom p = (PravougaonikSaBojom) x;
        System.out.println(p.getBoja());
      }
  }
}
```

Niz referenci superklase

Polimorfno procesiranje svih objekata potklasa

Dinamičko određivanje klase

Primjer polimorfnog procesiranja

```
Štampanje svih objekata:
Tip Pravougaonik, površina 6.00
Tip Pravougaonik, površina 15.00
Tip Trougao, površina 1.50
Tip Trougao, površina 8.00
Tip PravougaonikSaBojom, površina 4.00
Tip PravougaonikSaBojom, površina 30.00

Štampanje boja objekata PravougaonikSaBojom:
(R,G,B) = (221,99,100)
(R,G,B) = (76,145,10)
```

Komentari primjera

- Tokom izvršavanja programa, ime klase pojedinačnih instanci možemo dobiti na sledeći način:

  ```
  x.getClass().getName()
  ```

  gdje:

  - x sadrži referencu na objekat,
  - `getClass()` je metoda koju vraća objekat tipa `Class` (iz paketa `java.lang`), koja sadrži informacije o tipu objekta.
  - `getName()` je metoda koja vraća ime klase `Class` objekta.

- Pomoću operatora **`instanceof`** možemo provjeriti da li predmetni objekat pripada nekoj klasi.

- Zbog veze tipa jeste, ako operator `instanceof` vrati `true` za određenu klasu, vratiće `true` i za klase izvedene iz te klase. U našem primjeru,

  ```
  x instanceof PravougaonikSaBojom
  ```

  će vratiti `true` za objekte klase `PravougaonikSaBojom`, kao i objekte svih izvedenih klasa iz te klase.

final metode i klase

- Pored deklarisanja konstanti, ključna riječ `final` se može koristiti i pri deklaraciji metoda, parametara metoda i klasa.
- Finalna metoda superklase se ne može redefinisati u potklasi. Ovo garantuje da će istu implementaciju metode koristiti sve potklase, bilo direktne ili indirektne, date superklase.
- Metode koje su deklarisane kao `private` su takođe finalne, jer im se ne može pristupiti u potklasi. Implicitno finalne su i statičke metode.
- Pozivi finalnih metoda se razrješavaju tokom kompajliranja, što je poznato kao statičko ili rano vezivanje (pomenuto ranije).
- Sve potklase zovu istu finalnu metodu, jer potklase ne mogu imati svoje redefinisane verzije finalne metode.
- **Finalna klasa**, tj. klasa deklarisana ključnom riječju `final`, se ne može naslijediti, tj. ne može biti superklasa. Sve metode u finalnoj klasi su implicitno finalne.
- Tipičan primjer finalne klase je klasa String.

Interfejsi

- Polimorfizam omogućava jednostavno procesiranje objekata koji imaju zajedničku superklasu, indirektnu ili direktnu.
- Kako procesirati objekte koji ne dijele zajedničku superklasu, tj. nemaju nikakve veze jedni sa drugim?
- Uzmimo, na primjer, program za obračunavanje troškova koji ima firma. Firma ima zaposlene, koji primaju platu, a ima i račune za nabavljenu robu. Što se tiče obračunavanja troškova, poželjno je da se iznos plata i računa može dobiti uniformnim zapisom, a ne da vodimo računa o tipu svih objekata koji mogu biti uključeni u račun. U tom smislu, ne možemo koristiti polimorfizam kako smo ga do sad koristili, jer objekti tipa `Radnik` i `Racun` najverovatnije nemaju ništa zajedničko.
- **Interfejsi** omogućavaju da klase koje nisu povezane implementiraju skup zajedničkih metoda.
- Kada proglasimo da klase implementiraju određeni interfejs, objekti tih klasa mogu biti uniformno procesirani u smislu da se mogu pozivati metode koje su navedene u interfejsu.

Deklarisanje interfejsa

- Deklaracija interfejsa započinje ključnom riječju **`interface`**, i sadrži samo konstante i apstraktne metode!
- Za razliku od klasa, svi članovi interfejsa moraju biti `public` i interfejsi ne mogu sadržati nikakvu specifikaciju metoda.
- Sve metode deklarisane u interfejsu su implicitno `public abstract`, dok su svi podaci implicitno `public static final`.
- U skladu sa specifikacijom Java jezika, ne navode se ključne riječi `public`, `abstract` i `final` pri deklaraciji metoda i podataka interfejsa.

```
public interface Ime {

   tip metoda1(parametri);
   tip metoda2(parametri);
   ...
   tip metodaN(parametri);

   tip prom1 = vrijednost1;
   tip prom2 = vrijednost2;
   ...
   tip promM = vrijednostM;

}
```

Korišćenje interfejsa

- Da bi koristili interfejs, mora se navesti da konkretna klasa implementira interfejs, i klasa mora da realizuje svaku metodu iz deklaracije interfejsa.
- Navođenje da klasa implementira interfejs se vrši pomoću ključne riječi **implements** praćene imenom interfejsa u deklaraciji klase, na primjer:

```
public class ABC implements DEF {
     ...
}
```

- Klasa može implementirati proizvoljno mnogo interfejsa. U tom slučaju, imena interfejsa se razdvajaju zarezima u zaglavlju klase:

```
public class ABC implements Interf1, Interf2, ...{
     ...
}
```

Interfejsi i apstraktne klase

- Klasa koja ne implementira sve metode interfejsa je apstraktna i mora se deklarisati pomoću ključne riječi `abstract`.
- Implementiranje interfejsa je, u OO žargonu, potpisivanje ugovora između klase i kompajlera po kome klasa tvrdi da će realizovati sve metode navedene interfejsom ili će sebe proglasiti apstraktnom.
- Funkcionalnost interfejsa i apstraktnih klasa je slična. I jedni i drugi omogućavaju polimorfno procesiranje objekata.
- Za razlika od apstraktnih klasa, interfejs može da procesira raznorodne objekte, koji nemaju zajedničku superklasu.
- Jedino što ti objekti treba da urade je da implementiraju metode iz interfejsa.
- Na ovaj način se eliminiše potreba da postojeće apstraktne superklase mijenjamo kako bi obuhvatile više raznorodnih objekata, a što nekada i nema smisla.

Interfejsi i apstraktne klase

- Ukoliko apstraktna superklasa implementira interfejs, svaka njena konkretna potklasa mora implementirati sve metode interfejsa da bi ispunila ugovor koji je superklasa potpisala sa kompajlerom.
- Ukoliko to ne uradi, mora biti deklarisana kao `abstract`.
- Kad klasa implementira interfejs, primenjuje se veza tipa jeste. Tako, na primjer, ako klasa `ABC` implementira interfejs `DEF`, možemo reći da `ABC` jeste `DEF` objekat.
- Ova se veza nastavlja i dublje u hijerarhiji nasljeđivanja. Ako klasa `ABC1` nasljeđuje klasu `ABC`, a `ABC` implementira interfejs `DEF`, može se reći da `ABC1` jeste `DEF` objekat.
- Isto važi i prilikom implementiranja više interfejsa.

Nasljeđivanje interfejsa

- Jedan interfejs može naslijediti drugi interfejs, što se radi na potpuno isti način kao kod klasa, pomoću ključne riječi `extends`.

```
public interface Drugi extends Prvi {
     tip metoda1(lista parametara);
     tip metoda2(lista parametara);
     tip prom1 = vrijednost1;
     tip prom2 = vrijednost2;
}
```

- Za razliku od klasa, gdje se može naslijediti samo jedna klasa, interfejsi mogu naslijediti proizvoljno mnogo interfejsa, na primjer:

```
public interface Peti extends Prvi, Drugi, Treci, Cetvrti {
     tip metoda1(lista parametara);
     tip metoda2(lista parametara);
     tip prom1 = vrijednost1;
     tip prom2 = vrijednost2;
}
```

Interfejs i klasa

- Česta je zabuna da se interfejs poistovjećuje sa klasom. Interfejs nije klasa!
- U pitanju su dva različita koncepta. Klasa opisuje osobine (atribute) i ponašanje objekata (metode). Interfejs navodi koje metode će klasa implementirati.
- Sličnosti interfejsa i klase se ogledaju u sljedećem:
  - Interfejs i klasa mogu da sadrže proizvoljan broj metoda, u ovom slučaju apstraktnih.
  - Izvorni kod interfejsa se piše u fajlu sa `.java` ekstenzijom, pri čemu ime interfejsa odgovara imenu fajla.
  - Bajt kod interfejsa se pojavljuje u `.class` fajlu.
  - Interfejsi se nalaze u paketu.
- Razlike intefejsa i klase se ogledaju u sledećem:
  - Ne može se kreirati instanca interfejsa.
  - Interfejs nema konstruktore.
  - Sve metode interfejsa su apstraktne.
  - Svi podaci interfejsa su `static` i `final`.
  - U Javi, klasa može naslijediti samo jednu klasu, dok interfejs može naslijediti proizvoljno mnogo interfejsa.

Primjer korišćenja interfejsa

- Kao primjer korišćenja interfejsa, uzmimo primjer klasa `Oblik2D`, `Pravougaonik`, `Trougao` i `PravougaonikSaBojom` realizovanih ranije.
- Pretpostavimo da, pored ovih klasa, imamo i klasu `Prozor`, koji će predstavljati grafički prozor, gdje, recimo, možemo da iscrtavamo objekte `Pravougaonik`, `Trougao` i `PravougaonikSaBojom`.
- Klasa `Prozor` nije srodna klasi `Oblik2D`, ali ima jednu zajedničku karakteristiku, a to je površina. Oblici imaju površinu i grafički prozor ima površinu.
- Zajednička akcija koja se može izvesti za klase izvedene iz `Oblik2D` i klasu `Prozor` je, na primjer, pomjeranje. Oblici se mogu pomjerati u okviru grafičkog prozora, a grafički prozor se može pomjerati po monitoru.
- Posmatrajmo samo računanje površina oblika i prozora. Da bi ovu zajedničku akciju obavili uniformno, definisaćemo interfejs `Povrsina` koji sadrži samo jednu apstraktnu metodu `izracunaj()`.

Primjer korišćenja interfejsa

```java
public interface Povrsina {
   double izracunaj();
}
```

```java
public class Prozor implements Povrsina {
   double sirina, visina;
   public Prozor(){ this(0, 0); }
   public Prozor(double x){ this(x, 0); }
   public Prozor(double x, double y){
      setSirina(x); setVisina(y); }
   public double getSirina() { return sirina; }
   public void setSirina(double x) { sirina = x; }
   public double getVisina() { return visina; }
   public void setVisina(double y) { visina = y; }

   // Implementacija metode izracunaj()
   // iz interfejsa Povrsina
   @Override
   public double izracunaj(){
      return getSirina() * getVisina(); }

   public String toString(){
      return String.format("Prozor širine %.2f i visine
%.2f)", getSirina(), getVisina());
   }
}
```

Primjer korišćenja interfejsa

- Da bi klasa `Oblik2D` i sve njene izvedene klase implementirale interfejs `Povrsina`, potrebno je navesti da `Oblik2D` implementira taj interfejs:

```
public abstract class Oblik2D implements Povrsina
```

- Realizacija ove klase se ne mijenja. Pošto ova klasa neće implementirati metodu `izracunaj()`, nećemo navoditi tu metodu u okviru klase, iako se to može uraditi sa `abstract public double izracunaj();`
- Klase `Pravougaonik` i `Trougao` implementiraju metodu `izracunaj()`, što će uraditi dodavanjem sljedećeg koda unutar svoje realizacije:

```
@Override
public double izracunaj() {
     return povrsina();
}
```

- Nema potrebe navoditi u zaglavlju klasa `Pravougaonik` i `Trougao` da one implementiraju interfejs `Povrsina`, pošto je njihova superklasa to navela.
- Konačno, ništa se neće promijeniti u klasi `PravougaonikSaBojom`, jer ona nasljeđuje implementaciju metode `izracunaj()` iz direktne superklase `Pravougaonik`, koja se za svrhu računanja površine ne mora redefinisati.

Testiranje interfejsa

```
public class InterfejsTest {

  public static void main(String[] args) {
    Povrsina nizPovrsina[] = new Povrsina[4];
    nizPovrsina[0] = new Pravougaonik(2,3);
    nizPovrsina[1] = new Trougao(3,1);
    nizPovrsina[2] = new PravougaonikSaBojom(2,2,221,99,100);
    nizPovrsina[3] = new Prozor(11, 8);

    System.out.println("Štampanje svih objekata:");
    for(Povrsina x: nizPovrsina)
      System.out.printf("Tip %s, površina %.2f\n",
         x.getClass().getName(), x.izracunaj());
  }

}
```

```
Štampanje svih objekata:
Tip Pravougaonik, površina 6.00
Tip Trougao, površina 1.50
Tip PravougaonikSaBojom, površina 4.00
Tip Prozor, površina 88.00
```

Podrazumijevane i statičke metode interfejsa

- Prije Jave SE 8, dodavanje novih metoda u interfejs bi onemogućilo upotrebu svih klasa koje implementiraju interfejs jer klase nemaju implementaciju novih metoda interfejsa. Takve klase bi se onda morale proglasiti apstraktnim.
- **Podrazumijevane** i **statičke** metode u interfejsima uvodi Java SE 8.
- Podrazumijevane metode se deklarišu ključnom riječju `default`. Podrazumijevane metode moraju biti implementirane u okviru interfejsa!
- Dodavanje podrazumijevanih metoda u interfejs ne narušava rad klasa koje su implementirale interfejs. Klase dobijaju još jednu metodu koju mogu da koriste jer interfejs sadrži implementaciju metode. Ukoliko je potrebno, klasa može redefinisati novododatu metodu.
- Problem koji se može javiti kod podrazumijevanih metoda interfejsa je kada klasa implementira više interfejsa koji imaju istu podrazumijevanu metodu. Ovaj se problem razrješava tako što klasa redefiniše konfliktnu metodu.
- Statičke metode u interfejsima se deklarišu kao statičke metode klasa, ključnom riječju `static`, i moraju biti implementirane u okviru interfejsa.
- Pozivaju se standardno, preko imena interfejsa. Štaviše, statičku metodu interfejsa možemo pozvati u okviru druge statičke ili podrazumijevane metode.

Podrazumijevane i statičke metode - primjer

```
public class DefaultStatic implements Interfejs {

  public static void main(String[] args) {
    DefaultStatic ds = new DefaultStatic();
    System.out.println(ds.redefinisana());
    System.out.println(ds.podrazumijevana());
    System.out.println(Interfejs.staticka());
  }

  @Override
  public String redefinisana() {
    return "redefinisana metoda";
  }
}

interface Interfejs {
  String redefinisana();
  default String podrazumijevana() { return "podrazumijevana" +
metoda(); }
  static String staticka() { return "statička" + Interfejs.metoda(); }
  static String metoda() { return " metoda"; }
}
```

```
redefinisana metoda
podrazumijevana
metoda
statička metoda
```

Ispis