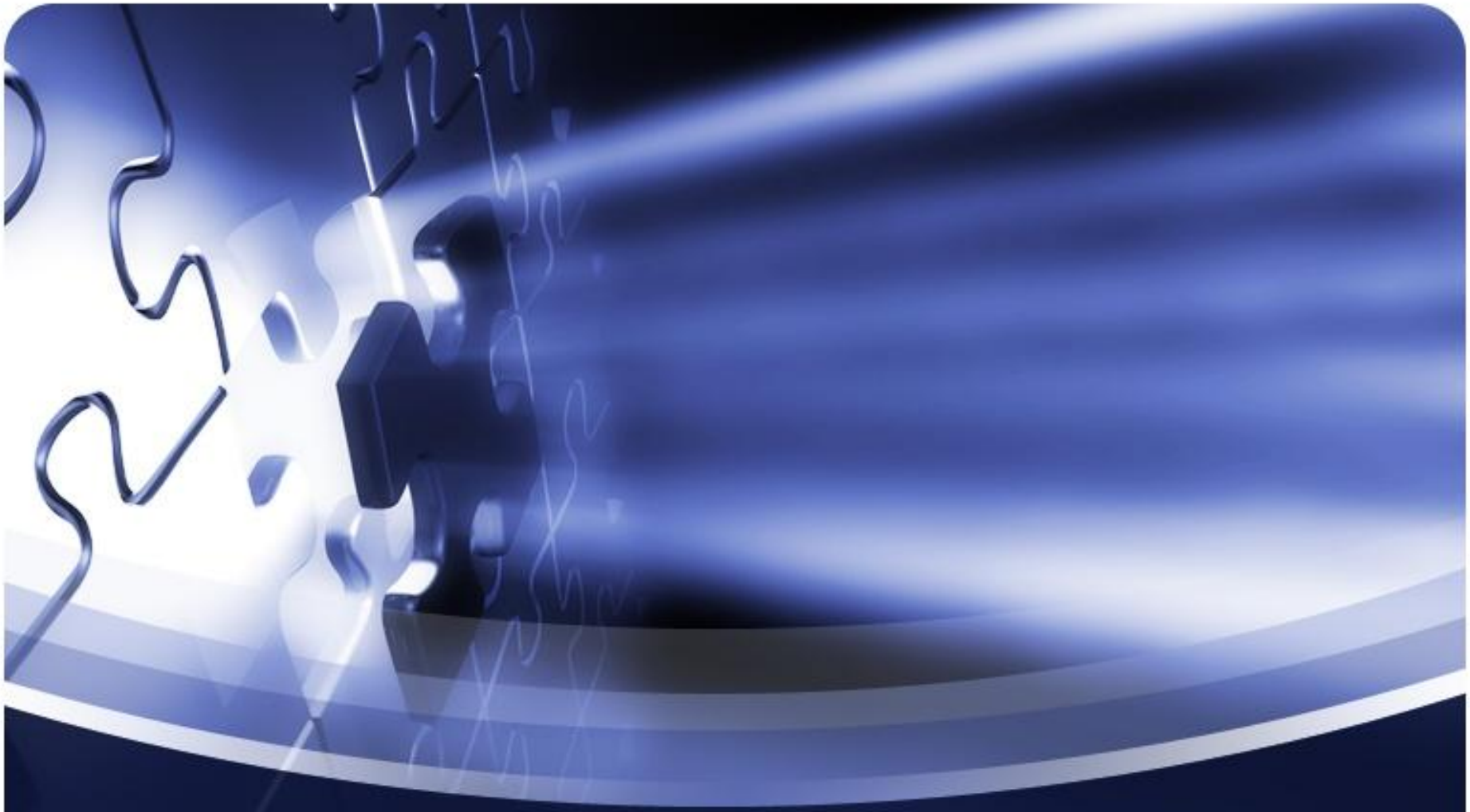


# OOP u Java programskom jeziku

Programski jezici II

# Osnovni mehanizmi kreiranja novih klasa na bazi postojećih

- Postoje dva osnovna mehanizma kreiranja novih klasa na bazi postojećih:
  - nasljeđivanje – definiše relaciju „je vrsta“ (eng. is-a, is-a-kind-of) između osnovne i klase nasljednice – objekat klase nasljednice je „vrsta“ objekta roditeljske klase i može biti korišten gdje god se očekuje objekat roditeljske klase. Ova tvrdnja se može koristiti i kao test pri odlučivanju o kreiranju klase nasljednice u procesu objektno-orijentisanog dizajna. Relacija „je vrsta“ definiše hijerarhiju klasa.
  - agregacija – definiše relaciju „sadrži“ (eng. has-a), odnosno cjelina/dio („whole/part“), između instance klase i njenih dijelova – u Javi, objekti ne mogu sadržavati druge objekte, moguće je samo sadržavanje vrijednosti reference. Sadržavani objekti (dijelovi) u agregaciji mogu biti dijeljeni između različitih objekata (cjelina). Agregacija ne govori ništa o uzajamnom odnosu životnog vijeka cjeline i životnog vijeka dijela, tj. životni vijek cjeline i životni vijek dijela mogu, ali ne moraju, biti zavisni. Ova relacija definiše hijerarhiju objekata. Kompozicija predstavlja jaku agregaciju, kod koje je cjelina odgovorna za životni vijek svojih dijelova.

# Nasljeđivanje

- Nasljeđivanje:
  - jedan od osnovnih mehanizama za ponovnu upotrebu programskog koda u objektno-orijentisanom programiranju.
  - omogućava klasama da naslijede funkcionalnost iz drugih klasa. Na ovaj način smanjuje se kompleksnost velikih softverskih sistema.
- Nova klasa (podklasa, izvedena klasa ili klasa nasljednica) nasljeđuje članove iz osnovne klase (superklasa ili roditeljska klasa)
- Klasa nasljednica može dodati nove osobine i ponašanja i, pod određenim uslovima, modifikovati ponašanje osnovne klase
- Nasljeđivanje članova zavisi od njihove dostupnosti. Ako je članu osnovne klase moguće pristupiti navođenjem naziva tog člana, bez bilo kakve dodatne sintakse (poput ključne riječi super), onda se taj član smatra nasljeđenim. Iz tog razloga privatni, redefinisani i sakriveni članovi osnovne klase nisu nasljeđeni.

# Nasljeđivanje

```
public class Kalkulator {
	protected int operand1;
	protected int operand2;

	Kalkulator(int op1, int op2){
		operand1 = op1;
		operand2 = op2;
	}

	public int zbir(){
		return operand1 + operand2;
	}

	public int razlika(){
		return operand1 - operand2;
	}
}
```

```
public class ProsireniKalkulator extends Kalkulator {

	ProsireniKalkulator(int op1, int op2) {
		super(op1, op2);
	}
	public int proizvod() {			// 1
		return operand1 * operand2;
	}
	public int kolicnik() {			// 2
		return operand1 / operand2;
	}
}
```

# Nasljeđivanje

- Instance klase ProsireniKalkulator će se ponašati identično instancama klase Kalkulator, s tim što će imati dodatne mogućnosti – računanje proizvoda i količnika dva operanda

```
ProsireniKalkulator kalkulator1 = new
ProsireniKalkulator(1,2);
int zbir = kalkulator1.zbir();
int proizvod = kalkulator1.proizvod();
int kolicnik = kalkulator1.kolicnik();
```

# Nasljeđivanje

- Ako u deklaraciji klase nije navedena ključna riječ extends, onda klasa implicitno nasljeđuje Object klasu iz java.lang paketa – na ovaj način sve klase nasljeđuju Object klasu, direktno ili indirektno
  - klasa Kalkulator implicitno nasljeđuje klasu Object, dok klasa ProsireniKalkulator indirektno, nasljeđivanjem klase Kalkulator, nasljeđuje i klasu Object
- klasa Object nema roditeljsku klasu
- Privatni članovi osnovne klase se ne nasljeđuju – moguće im je pristupiti jedino indirektno, definisanjem metoda osnovne klase koje će se koristiti za čitanje i mijenjanje vrijednosti privatnih članova
  - izvedena klasa nasljeđuje sve *public* i *protected* članove roditeljske klase, bez obzira u kojem paketu je izvedena klasa – ako je izvedena klasa u istom paketu kao i roditeljska, nasljeđuje i *friendly* članove
  - članovi koji imaju podrazumijevanu dostupnost u osnovnoj klasi ne mogu biti nasljeđeni od strane klasa u drugim paketima
- Konstruktori i inicijalizacioni blokovi ne mogu biti nasljeđeni, jer nisu članovi klase

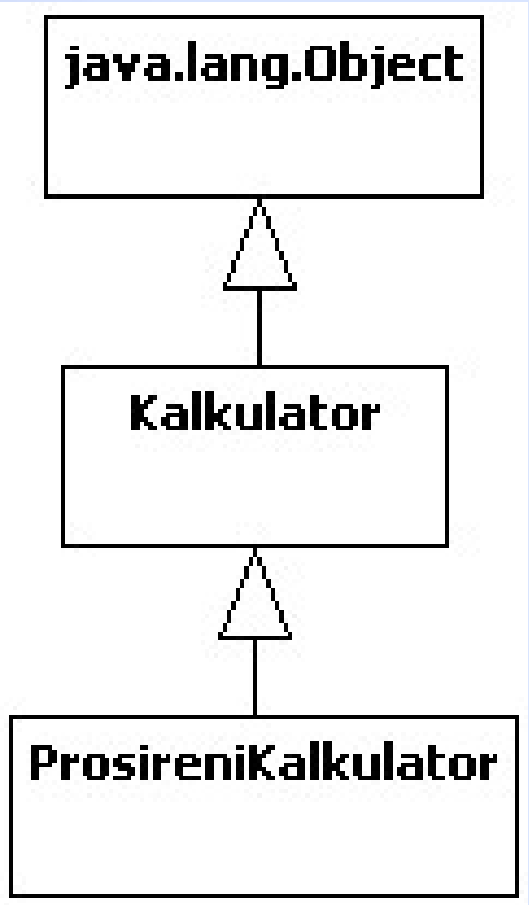
# Nasljeđivanje – klasa Object

- klasa Object (java.lang paket) se nalazi na vrhu hijerarhijskog stabla
- sve klase nasljeđuju klasu Object, direktno ili indirektno
- metode klase Object:
  - protected Object clone() throws CloneNotSupportedException
    - kreira i vraća kopiju objekta
  - public boolean equals(Object obj)
    - vrši poređenje dva objekta
  - protected void finalize() throws Throwable
    - poziva je *garbage collector*
  - public final Class getClass()
    - vraća runtime klasu objekta
  - public int hashCode()
    - vraća hash code reprezentaciju objekta
  - public String toString()
    - vraća string reprezentaciju objekta
  - public final void notify()
  - public final void notifyAll()
  - public final void wait()
  - public final void wait(long timeout)
  - public final void wait(long timeout, int nanos)

# Jednostruko nasljeđivanje

- U Java programskom jeziku nije dozvoljeno višestruko nasljeđivanje, tj. svaka klasa može imati samo jednu roditeljsku klasu – jednostruko nasljeđivanje
- Na ovaj način nastaje hijerarhija klasa, gdje su klase koje se nalaze više u hijerarhiji više generalizovane, dok su klase niže u hijerarhiji više specijalizovane
- Na vrhu hijerarhije klasa uvijek se nalazi Object klasa, jer je sve druge klase nasljeđuju, direktno ili indirektno

# Jednostruko nasljeđivanje



# Proširivanje reference

- nasljeđivanje definiše relaciju „je vrsta“ između osnovne i klase nasljednice – ova relacija podrazumijeva da se vrijednost reference objekta klase nasljednice može dodijeliti referenci objekta osnovne klase, jer objekat klase nasljednice može da se pojavi gdje god se očekuje objekat osnovne klase

- ova dodjela podrazumijeva proširivanje reference (eng. *widening reference conversion*)

# Proširivanje reference

```
Kalkulator kalkulator1 = new ProsireniKalkulator(1,2); // 1
int zbir = kalkulator1.zbir();                         // 2
// int proizvod = kalkulator1.proizvod();              // 3
```

- referenca kalkulator1 se može koristiti za poziv metoda nad objektom klase ProsireniKalkulator koje su nasljeđene iz klase Kalkulator
- bitno je primijetiti da kompajler u vrijeme prevođenja programa ima saznanje o deklarisanom tipu reference
  - na osnovu toga što je tip reference kalkulator1 Kalkulator, kompajler ograničava da samo metode iz klase Kalkulator mogu biti pozivane putem ove reference
  - za vrijeme izvršavanja referenca kalkulator1 referenciraće objekat klase ProsireniKalkulator – iako referenca kalkulator1 referencira objekat klase ProsireniKalkulator nije moguće pozvati metodu proizvod, jer kompajler u vrijeme prevođenja nema saznanje o tome koji objekat će referenca kalkulator1 referencirati, tj. kompajer ima saznanje samo o deklarisanom tipu reference (poziv metode proizvod u ovom slučaju rezultiraće greškom pri kompajliranju)

# Redefinisanje metoda

- klasa nasljednica može redefinisati metode osnovne klase
- redefinisanje metode omogućava da klasa nasljednica obezbijedi vlastitu implementaciju metode koju bi inače naslijedila iz osnovne klase
- redefinisana metoda osnovne klase se u ovoj situaciji ne nasljeđuje, a nova metoda u klasi nasljednici mora da ispunjava sljedeće:
  - mora da ima isti potpis (naziv metode, tip, broj i redoslijed parametara),
  - povratni tip nove metode može biti podtip povratnog tipa redefinisane metode (kovarijantni povratni tip),
  - nova metoda ne može smanjiti dostupnost metode, ali je može povećati,
  - nova metoda može baciti sve, nijedan ili podskup provjerenih izuzetaka (uključujući i njihove podklase) koji su specificirani u throws klauzuli redefinisane metode
- `@Override` napomena obavještava kompajler o namjeri redefinisanja metode iz roditeljske klase – ako kompajler detektuje da metoda ne postoji u roditeljskoj klasi prijaviće grešku

# Redefinisanje metoda

```
public class Osnovna {
	public void metoda1(){                          // 1
		System.out.println("metoda1, klasa Osnovna");
	}
	public void metoda2(){                          // 2
		System.out.println("metoda2, klasa Osnovna");
	}
	protected void metoda3() throws IOException,
SQLException{                                       // 3
		System.out.println("metoda3, klasa Osnovna");
	}
}
```

# Redefinisanje metoda

```
public class Nasljednica extends Osnovna {
	public void metoda1() {                                    // 1
		System.out.println("metoda1, klasa
Nasljednica");
	}
//	public void metoda3() throws Exception{                    // 2
//		System.out.println("metoda3, klasa
Nasljednica");
//	}
//	public void metoda3(){                                     // 3
//		System.out.println("metoda3, klasa
Nasljednica");
//	}
//	public void metoda3() throws IOException,
SQLException{                                                  // 4
//		System.out.println("metoda3, klasa
Nasljednica");
//	}
//	public void metoda3() throws IOException {                 // 5
//		System.out.println("metoda3, klasa
Nasljednica");
//	}
	public void metoda3() throws SQLException {                // 6
		System.out.println("metoda3, klasa
Nasljednica");
	}
}
```

# Redefinisanje metoda

- Metoda instance u klasi nasljednici ne može redefinisati statičku metodu osnovne klase – pokušaj kompajliranja dovešće do greške
  - statička metoda u klasi nasljednici može sakriti statičku metodu osnovne klase
- Metode u čijoj deklaraciji se nalazi modifikator final ne mogu biti redefinisane – pokušaj redefinisanja ovakve metode dovešće do greške pri kompajliranju
- Apstraktne metode u klasama nasljednicama (koje nisu apstraktne) moraju biti redefinisane, tj. implementacija ovih metoda mora biti obezbjeđena
- Modifikator private u deklaraciji metode označava da metoda nije dostupna za klase nasljednice, pa iz tog razloga ova metoda ne može biti redefinisana – klasa nasljednica može specificirati vlastitu metodu koja ima isti potpis kao metoda osnovne klase

# Nasljeđivanje – apstraktne klase

- mješavina interfejsa i klasa
  - mogu imati definisane metode, mogu imati polja
- ako ima samo apstraktne metode – onda treba biti interfejs, a ne apstraktna klasa
- klase koje ne mogu imati svoje objekte
- klase koje su deklarisane kao *abstract* mogu, ali ne moraju, imati apstraktne metode
- koriste se kada klase nasljednice imaju zajedničke segmente koda
- apstraktna klasa može imati statička polja i statičke metode
- apstraktna klasa može implementirati interfejs

```
abstract class X implements Y { }
class XX extends X { }
```

# Kovarijantni povratni tip

- povratni tip nove metode može biti podtip povratnog tipa redefinisane metode – ovaj povratni tip naziva se kovarijantnim povratnim tipom
- klasa Osnovna:

```
public Osnovna kreirajInstancu(){
        return new Osnovna();
}
```

- klasa Nasljednica:

```
public Nasljednica kreirajInstancu(){
        return new Nasljednica();
}
```

- kovarijantni povratni tip ne važi za primitivne tipove
- klasa nasljednica mora koristiti ključnu riječ super kako bi pozvala redefinisanu metodu roditeljske klase

# Preklopljene metode

- preklopljene metode su metode koje imaju isti naziv, ali različitu listu parametara
- lista parametara se može razlikovati po tipu, redoslijedu i/ili broju parametara
- kako povratni tip nije dio potpisa metode, nije dovoljno da se metode razlikuju samo po povratnom tipu da bi bile preklopljene
- i metode instance i statičke metode mogu biti preklopljene u osnovnoj ili klasi nasljednici

# Preklopljene metode

```
public class PreklapanjeMetoda {
	public long zbir(int x, int y){
		return x + y;
	}
	public long zbir(long x, long y){
		return x + y;
	}
	public long zbir(int x, long y){
		return x + y;
	}
//	public double zbir(int x, int y){
//		return x + y;
//	}
}
```

# Preklopljene metode

- poput metoda, i konstruktori mogu biti preklopljeni
- ako klasa posjeduje više konstruktora oni su uvijek preklopljeni, jer svi konstruktori imaju isti naziv (koji odgovara nazivu klase), pa se njihovi potpisi razlikuju samo po listi parametara

```
Kalkulator(){
	operand1 = 0;
	operand2 = 0;
}
Kalkulator(int op1, int op2){
	operand1 = op1;
	operand2 = op2;
}
```

# Redefinisanje vs. preklapanje metoda

| | redefinisanje | preklapanje |
|---|---|---|
| naziv metoda | mora biti identičan | mora biti identičan |
| lista parametara | mora biti identična | ne smije biti identična |
| povratni tip | identičan ili kovarijantan | nema ograničenja |
| izuzeci | svi, nijedan ili podskup izuzetaka redefinisane metode | nema ograničenja |
| dostupnost | ne može se smanjiti | nema ograničenja |
| lokacija | samo u klasi nasljednici | u istoj ili u klasi nasljednici |
| izvršavanje metode određuje | tip objekta referenciran u vrijeme izvršavanja | deklarisani tip reference |

# Maskiranje (sakrivanje) polja

- polja osnovne klase ne mogu biti redefinisana u klasi nasljednici, ali mogu biti maskirana
- polja se maskiraju tako što se u klasi nasljednici definiše polje sa istim nazivom kao u osnovnoj klasi – u ovakvom slučaju poljima osnovne klase se ne može pristupiti direktno, pa je jasno da se ona ne nasljeđuju – pristup poljima osnovne klase koja se ne nasljeđuju, uključujući i maskirana polja, može se ostvariti korištenjem ključne riječi super
- za razliku od redefinisanja metoda, gdje metoda instance ne može redefinisati statičku metodu, kod maskiranja polja ne postoji takvo ograničenje – polje instance u klasi nasljednici može maskirati statičko polje osnovne klase
- tip maskiranih polja nije bitan, važan je jedino naziv polja
- sakrivanje polja se ne preporučuje

# Maskiranje statičkih metoda

- statička metoda u klasi nasljednici može maskirati statičku metodu osnovne klase, ako su uslovi identični uslovima za redefinisanje metoda instance ispunjeni
- maskirana statička metoda osnovne klase se ne nasljeđuje
- ako su potpisi metoda identični, a drugi zahtjevi poput povratnog tipa, throws klauzule i dostupnosti nisu ispunjeni, kompajler će prijaviti grešku
- ako su potpisi metoda različiti, onda je riječ o preklapanju metoda, a ne o maskiranju
- maskirana statička metoda uvijek može biti pozvana preko imena osnovne klase u kodu klase nasljednice, kako iz statičkog tako i iz nestatičkog konteksta – moguće je koristiti ključnu riječ super za poziv maskirane statičke metode, u nestatičkom kontekstu u kodu klase nasljednice
- statička metoda u klasi nasljednici ne može maskirati metodu instance osnovne klase – pokušaj ovakvog maskiranja dovešće do greške pri kompajliranju

# Maskiranje (sakrivanje) članova

```
public class MaskiranjeOsnovna {
	int broj = 0;                                   // 1
	String string = "test";                         // 2
	public static void statickaMetoda(){            // 3
		System.out.println("staticka metoda osnovne
klase");
	}
}

class MaskiranjeNasljednica extends MaskiranjeOsnovna{
	int broj = 1;                                   // 4
	static int string = 2;                          // 5
	public void metoda(){
		int tmp = super.broj + broj;            // 6
		String s = super.string + string;       // 7
		MaskiranjeOsnovna.statickaMetoda();     // 8
		super.statickaMetoda();                 // 9
	}
	public static void statickaMetoda(){            // 10
		MaskiranjeOsnovna.statickaMetoda();     // 11
		System.out.println("staticka metoda klase
nasljednice");
	}
}
```

# Nasljeđivanje

| | **Nestatička metoda roditeljske klase** | **Statička metoda roditeljske klase** |
|---|---|---|
| **Nestatička metoda izvedene klase** | redefinisanje | greška pri kompajliranju |
| **Statička metoda izvedene klase** | greška pri kompajliranju | sakrivanje |

# Referenca this

- ključna riječ this označava referencu koja je dostupna u nestatičkom kontekstu i koja referencira tekući objekat
- u nestatičkom kontekstu this referenca se može koristiti kao i bilo koja druga referenca na objekte, i pomoću nje se može pristupati članovima objekta
- referenca this ne može biti mijenjana, ona je final referenca
- referenca this se često koristi u situacijama gdje lokalna promjenljiva maskira polje sa istim nazivom

# Referenca super

- ključna riječ super označava referencu koju je moguće koristiti u nestatičkom kontekstu, ali samo u klasi nasljednici, radi pristupa poljima i pozivanju metoda osnovne klase
- ova referenca referencira tekući objekat kao instancu osnovne klase
- pri pozivu metoda korištenjem ključne riječi super, metoda osnovne klase se poziva bez obzira na stvarni tip objekta i bez obzira na to da li je metoda redefinisana u klasi nasljednici
- referenca super se obično koristi za pristup članovima koji su sakriveni i pozivanje metoda koje su redefinisane

# Ulančavanje konstruktora

- konstruktori ne mogu biti nasljeđeni niti redefinisani, ali mogu biti preklopljeni unutar iste klase
- ulančavanje: korištenjem this i korištenjem super
- konstrukcija this() sa odgovarajućom listom parametara rezultira pozivom odgovarajućeg konstruktora
- obavezno je da poziv konstruktora korištenjem konstrukcije this() bude prva naredba u tijelu konstruktora
- konstrukcija super() se koristi u konstruktoru klase nasljednice radi poziva odgovarajućeg konstruktora neposredne roditeljske klase – koji konstruktor roditeljske klase će biti izvršen zavisi od liste parametara super() konstrukcije – preporuka je da se konstrukcija super() koristi za inicijalizaciju nasljeđenog stanja
- konstrukcija super() mora biti prva naredba u konstruktoru i moguće ju je koristiti samo u konstruktoru

# Ulančavanje konstruktora

- kako moraju biti prve naredbe u konstruktoru, jasno je da se konstrukcije this() i super() ne mogu pojaviti u istom konstruktoru
- ako se u konstruktoru klase nasljednice ne nalazi eksplicitan poziv konstrukcije super(), kompajler će implicitno ubaciti poziv ove konstrukcije (bez argumenata) kako bi pozvao podrazumijevani konstruktor roditeljske klase
  - preciznije, ako konstruktor ne posjeduje ni konstrukciju this(), ni konstrukciju super(), kompajler će implicitno ubaciti poziv konstrukcije super() (bez argumenata)
- ako roditeljska klasa ne posjeduje podrazumijevani konstruktor, tj. ako roditeljska klasa posjeduje samo konstruktore sa parametrima, onda ubacivanje poziva konstrukcije super() od strane kompajlera u konstruktore klasa nasljednica neće imati efekta – u ovakvim situacijama desiće se greška pri kompajliranju – da bi se ova greška izbjegla u konstruktorima klasa nasljednica potrebno je eksplicitno pozvati konstruktor roditeljske klase, korištenjem konstrukcije super() sa odgovarajućim argumentima

# Ulančavanje konstruktora

```
public class B {                    // 1
      int x = 0, y = 0;
      B(int a, int b){
            x = a;
            y = b;
      }
      public int zbir(){
            return x + y;
      }
      public static void main(String args[]){
            B b = new B(1,2);
            C c = new C();
      }
}

class C extends B{                  // 2
      public int zbir(){
            return y+x;
      }
}
```

# Ulančavanje konstruktora

```
public class KonstruktorOsnovna {
    public KonstruktorOsnovna() {
        System.out.println("Konstruktor osnovne klase...");
    }

    public KonstruktorOsnovna(String s){
        System.out.println("Konstruktor osnovne klase s argumentom...");
    }

    public static void main(String[] args) {
        KonstruktorNasljednica kn = new KonstruktorNasljednica();
    }
}

class KonstruktorNasljednica extends KonstruktorOsnovna {
    public KonstruktorNasljednica(){
//      super("test");
        System.out.println("Konstruktor klase nasljednice...");
    }
}
```

# Interfejsi

- interfejsi kao API
- interfejsi nisu dio “klasne” hijerarhije

# Deklaracija interfejsa

```
<modifikator dostupnosti> interface <naziv interfejsa>
<extends klauzula>        // zaglavlje interfejsa
{                         // tijelo interfejsa
<deklaracije konstanti>
<deklaracije apstraktnih metoda>
<deklaracije ugnježđenih klasa>
<deklaracije ugnježđenih interfejsa>
}
```

<deklaracije default metoda>

# Deklaracija interfejsa

- interfejs nema implementaciju, te je po definiciji apstraktan, tj. ne može biti instanciran
  - interfejs nema konstruktor
- u deklaraciji interfejsa može se koristiti i ključna riječ abstract, ali je to potpuno nepotrebno
- interfejse implementiraju klase, pa iz tog razloga članovi interfejsa implicitno imaju public dostupnost – iz tog razloga modifikator public može biti izostavljen u deklaraciji članova interfejsa
- da bi klasa implementirala interfejs, mora da implementira sve njegove metode
- moguće je da postoje i interfejsi bez tijela – oni se obično koriste da označe klasu koja ima određenu osobinu ili ponašanje
- sve metode (koje nisu default) interfejsa su implicitno public i abstract
- sve apstraktne, default i statičke metode interfejsa su implicitno public
- sve konstante definisane u interfejsu su implicitno public, static i final

# Default metode

- dodavanje nove metode (koja nije default) u interfejs, dovešće do toga da se moraju modifikovati sve klase koje implementiraju dati interfejs
- default metode omogućavaju dodavanje nove funkcionalnosti u interfejs, bez potrebe da se modifikuju klase koje implementiraju dati interfejs (binarna kompatibilnost koda napisanog za prethodnu verziju interfejsa biće očuvana)
- Kod nasljeđivanja interfejsa koji sadrži default metodu, postoje sljedeće mogućnosti:
  - ne pominjati default metodu, čime prošireni interfejs nasljeđuje default metodu
  - redeklarisati default metodu i učiniti je apstraktnom
  - redefinisati default metodu

# Implementacija interfejsa

- Java programski jezik podržava višestruko nasljeđivanje tipa, što je mogućnost klase da implementira više od jednog interfejsa
  - objekt može imati višestruke tipove – tip klase i tipove interfejsa koje klasa implementira
- klase ne nasljeđuju interfejse, već ih implementiraju
- klasa može da implementira jedan ili više interfejsa
  - kod klase koja implementira više interfejsa nazivi interfejsa se specificiraju u zaglavlju klase, nakon ključne riječi implements, i razdvajaju se znakom „ , “
- metode interfejsa moraju imati public dostupnost pri implementaciji u klasi (ili njenim klasama nasljednicama)
- klasa ne može smanjiti dostupnost metode interfejsa, niti može specificirati novi izuzetak u throws klauzuli metode, jer će to dovesti do narušavanja ugovora interfejsa – što nije legalno
  - kriterijumi za redefinisanje metode važe i pri implementaciji metoda interfejsa
- klasa može implementirati samo neke od metoda interfejsa, tj. klasa može obezbijediti parcijalnu implementaciju interfejsa – u ovom slučaju klasa mora biti deklarasana kao apstraktna

# Implementacija interfejsa

- metode interfejsa uvijek moraju biti implementirane kao metode instance, a ne kao statičke metode
- klasa koja nasljeđuje drugu klasu i implementira jedan ili više interfejsa ima ponašanje i roditeljske klase i interfejsa – slično višestrukom nasljeđivanju
  - roditeljska klasa i interfejsi, jednim imenom, nazivaju se supertipovi klase
  - s druge strane, klasa koja nasljeđuje roditeljsku klasu i implementira jedan ili više interfejsa jeste podtip svojih supertipova
- bez obzira koliko interfejsa klasa implementira, direktno ili indirektno, ona posjeduje samo jednu implementaciju metode koji može biti deklarisana u više interfejsa

# Nasljeđivanje interfejsa

- jedan interfejs može da naslijedi jedan ili više drugih interfejsa, tj. kod interfejsa postoji višestruko nasljeđivanje – u deklaraciji interfejsa koji nasljeđuje jedan ili više drugih interfejsa nazivi interfejsa koji se nasljeđuju se specificiraju u zaglavlju interfejsa, nakon ključne riječi extends, i razdvajaju se znakom „ , “
- interfejsi koji su nasljeđeni, direktno ili indirektno, od strane nekog interfejsa, nazivaju se superinterfejsi – s druge strane, interfejs koji nasljeđuje druge interfejse naziva se podinterfejsom svojih superinterfejsa
- kako interfejsi definišu nove referencne tipove, superinterfejsi i podinterfejsi su i supertipovi i podtipovi, respektivno
- podinterfejs nasljeđuje sve metode svojih superinterfejsa, jer su metode interfejsa javne (public)
- deklaracije apstraktnih metoda mogu biti preklopljene, analogno preklapanju metoda kod klasa

# Nasljeđivanje interfejsa

```
interface Instrument {
	void sviraj();
	void nastimaj();
}

interface GlasanInstrument extends Instrument {
	void pojacaj();
}
```

# Reference tipa interfejsa

- iako interfejsi ne mogu biti instancirani, moguće je deklarisati referencu nekog tipa interfejsa
- vrijednost reference nekog objekta može biti dodjeljena referencama supertipova datog objekta

# Konstante u interfejsima

- u interfejsu se mogu deklarisati konstante koje su implicitno public, static i final
- ove konstante moraju biti inicijalizovane
- konstantama deklarisanim u interfejsu može pristupiti bilo koji klijent (klasa ili interfejs) putem punog kvalifikovanog imena
- ako klijent (klasa ili interfejs) implementiraju, odnosno nasljeđuju interfejs, onda je konstanti moguće pristupiti navođenjem njenog jednostavnog imena
- bilo koji konflikt imena konstanti, do kojeg može doći usljed višestrukog nasljeđivanja interfejsa, može biti razriješen korištenjem punih kvalifikovanih imena konstanti

# Konstante u interfejsima

```
interface Brzina {
	int PRVA = 1;
	int DRUGA = 2;
	int TRECA = 3;
	int CETVRTA = 4;
	int PETA = 5;
	int SESTA = 6;
}
class Motocikl implements Brzina{
	int trenutnaBrzina;

	public static void main(String args[]){
		Motocikl moto = new Motocikl();
		moto.trenutnaBrzina = Brzina.PRVA;
		moto.trenutnaBrzina = DRUGA;
	}
}
```

# Konstante u interfejsima

```
interface KosmickaBrzina {
	String PRVA = "Warp 1";
	String DRUGA = "Warp 2";
	String TRECA = "Warp 3";
}

class NLO implements Brzina, KosmickaBrzina{
	String trenutnaBrzina;

	public static void main(String args[]){
		NLO nlo = new NLO();
		nlo.trenutnaBrzina = "" + Brzina.DRUGA;
		nlo.trenutnaBrzina = KosmickaBrzina.DRUGA;
	}
}
```

# Operator instanceof

- binarni operator instanceof može se koristiti za poređenje tipova

```
<referenca> instanceof <odredišni tip>
```

- operator instanceof vraća istinitu vrijednost (true) ako operand s lijeve strane (vrijednost reference koju sadrži <referenca>) referencira objekat čija klasa može biti tip s desne strane (<odredišni tip>) ili podtip tipa s desne strane – u suprotnom, operator instanceof vraća false vrijednost – operator instanceof uvijek vraća false ako je operand s lijeve strane null
- operator instanceof zahtjeva provjeru za vrijeme kompajliranja, kao i provjeru za vrijeme izvršavanja
  - neka je <referenca> određenog tipa <izvorišni tip> - provjera za vrijeme kompajliranja podrazumijeva provjeru postojanja podklasa-superklasa veze između izvorišnog i odredišnog tipa – ako ovo nije slučaj, jasno je da primjena instanceof operatora nije moguća
  - za vrijeme izvršavanja <referenca> sadrži vrijednost reference određenog objekta – riječ je o tipu stvarnog objekta na osnovu kojeg će biti određen rezultat operacije (rezultat instanceof operatora)

# Operator instanceof

```
class Letjelica {}
class MotornaLetjelica extends Letjelica {}
class Avion extends MotornaLetjelica {}
class LetjelicaBezMotora extends Letjelica {}
class Zmaj extends LetjelicaBezMotora {}

public class Letjelice {
      public static void main(String[] args) {
            Letjelica letjelica = new Letjelica();     //1
//           rezultat = letjelica instanceof String;   //2
            System.out.println(letjelica instanceof
LetjelicaBezMotora);                                   //3
            System.out.println(letjelica instanceof Avion);
                                                       //4
            letjelica = new Zmaj();                    //5
            System.out.println(letjelica instanceof
LetjelicaBezMotora);                                   //6
            System.out.println(letjelica instanceof
Letjelica);                                            //7
            System.out.println(letjelica instanceof
MotornaLetjelica);                                     //8
            System.out.println(letjelica instanceof Avion);
                                                       //9
      }
}
```

# Polimorfizam

- polimorfizam omogućava da reference referenciraju objekte različitih tipova u različito vrijeme tokom izvršavanja
- referenca supertipa ispoljava polimorfno ponašanje jer može referencirati objekte svojih podtipova
- polimorfizam se može definisati kao koncept koji omogućava objektima da ispolje različito ponašanje, zavisno od njihove klase, bez obzira što se oni koriste kao instance nekog zajedničkog roditelja
- polimorfizam omogućava da objekti različitih tipova odgovore na metode istog imena, tj. omogućava redefinisanje funkcionalnosti roditeljske klase
- koja metoda će biti izvršena prilikom poziva metode nekog objekta određuje se za vrijeme izvršavanja na osnovu tipa objekta i potpisa metode - ovaj proces naziva se dinamičko određivanje metode (eng. *dynamic method lookup*)

# Polimorfizam

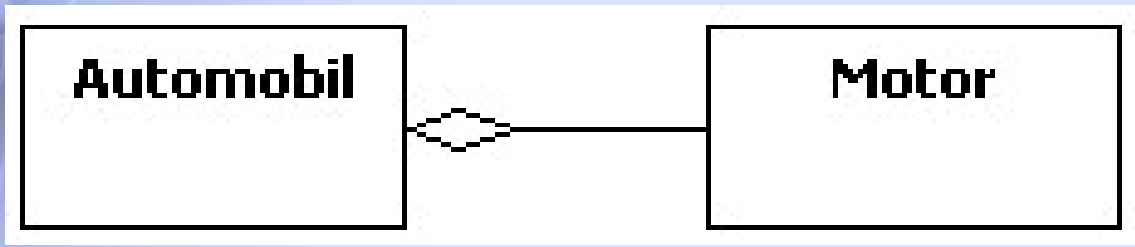
- poziv privatne metode instance nije i ne može biti polimorfan, jer se takav poziv može javiti samo unutar klase i odnosi se na tačno određenu metodu, što može biti utvrđeno za vrijeme kompajliranja
- polimorfizam se može ostvariti kako putem nasljeđivanja, tako i putem implementacije interfejsa
- programski kod koji se oslanja na polimorfno ponašanje će raditi bez bilo kakvih izmjena u slučaju dodavanja novih podklasa roditeljske klase, kao i u slučaju dodavanja klasa koje implementiraju interfejs, pri čemu je polimorfno ponašanje ispoljeno putem roditeljske klase i/ili interfejsa

# Agregacija

- ako se nova klasa kreira na bazi postojećih korištenjem agregacije, onda se objekat nove klase sastoji iz gradivnih dijelova - objekata klasa koje su agregirane u novoj klasi
- agregacija u Javi je bazirana na referencama, jer objekti ne mogu eksplicitno sadržavati druge objekte

# Agregacija

```
public class Automobil {
        Motor motor = new Motor();
}
```



# Osnovni koncepti OO dizajna

- enkapsulacija
- kohezija
- vezivanje

# Enkapsulacija

- objekti imaju osobine i ponašanja koja su enkapsulirana unutar njih
- servisi koje objekti pružaju klijentima čine ugovor ili javni interfejs
- samo ono što je definisano ugovorom dostupno je klijentima, pri čemu implementacija osobina i ponašanja klijentima nije poznata
- enkapsulacija pomaže da se napravi jasna razlika između ugovora i implementacije objekta
- implementacija objekta može biti promijenjena, a da pri tom to nema nikakvog uticaja na klijente
- enkapslacijom se smanjuje i kompleksnost, jer je implementacija sakrivena od klijenata i oni ne mogu ni na koji način da je mijenjaju

# Enkapsulacija

- enkaspulacija se ostvaruje putem sakrivanja informacija
- sakrivanje informacija može biti ostvareno na nivou metode (ili bloka koda), na nivou klase ili na nivou paketa
- sakrivanje informacija na nivou klase postiže se putem modifikatora pristupa članova klase
- preporuka je da se spriječi direktan pristup podacima koje sadrži objekat od strane klijenata koji ga koriste, na takav način da se polja klase deklarišu kao privatna (private), a da se ugovorom definišu javne metode za servise koje objekat pruža
- ovakvom enkaspulacijom razdvaja se korištenje od implementacije klase
- sakrivanje informacija na nivou paketa postiže se na takav način da se klase grupišu u odgovarajuće pakete, a da se međupaketska dostupnost klasa kontroliše putem modifikatora dostupnosti klase

# Kohezija

- stepen sličnosti metoda je glavni aspekt kohezivnosti
- ako klasa ima različite metode koje izvode različite operacije na istom skupu promjenljivih instance onda je klasa kohezivna
- osnovni cilj u ovom pogledu jeste dizajn klasa sa visokim stepenom kohezije, koje izvršavaju dobro definisane i povezane zadatke
- ovakva kohezija naziva se i funkcionalnom kohezijom
- isto tako, metoda jedne klase ne bi trebala da obavlja zadatke koji bi trebali biti implementirani od strane neke druge klase
- nedostatak kohezije znači da namjena klase nije precizno određena
- kao posljedica ovoga klasa će imati različite nepovezane funkcionalnosti, što može dovesti do uticaja na održavanje softvera koji je sačinjen od ovakvih klasa
- ovakva kohezija se naziva slučajnom kohezijom

# Vezivanje

- vezivanje je mjera zavisnosti između klasa
- objekti ostvaruju međusobnu interakciju tako što jedni drugima šalju odgovarajuće poruke, tako da je prirodno da postoje odgovarajuće zavisnosti između klasa
- ove zavisnosti između klasa trebaju biti što je moguće manje, kako bi se postiglo slabo vezivanje koje doprinosi kreiranju proširivih aplikacija
- minimizacija vezivanja i maksimizacija kohezije pomažu u ostvarivanju osnovnih ciljeva objektno-orijentisanog dizajna (održavanje, reupotrebljivost, proširljivost i pouzdanost), tj. pomažu u stvaranju dobro-dizajniranog softvera