**INTERFEJSI**

Interfejs je konstrukt nalik klasi koji sadrzi samo konstante i abstraktne meode.

U mnogo nacina interfejs je slican abstraktnoj klasi ali njegova namjena je da specifise zajednicka ponasanja za objekte povezanih ili ne povezanih klasa. Ma primjer, koristeci odgovarajuci interfejs, moze se specificirati da su objekti uporedivi, jedivi ili da se mogu klonirati. Da bi se napravila razlika izmedju interfejsa i klase koristi se sljedeca sintaksa da se definise interfejs:

**Modifikator interface nazivInterfejsa { }**

**public interface Edible {**

**/\*\* Describe how to eat \*/**

**public abstract String howToEat();**

**}**

U Javi interfejs se tretira kao specijalna klasa. Svaki interfejs se kompajlira u zaseban bytecode fajl bas kao i regularna klasa, moze se koristiti vise manje isto kao i abstraktna klasa. Na primjer moze se koristiti kao tip podataka za referencnu varijablu, kao rezultat rasporedjivanja i itd. Bas kao i sa abstraktnom klasom ni sa interfejsom ne moze kreirati instanca koristeci se NEW operatorom. Nasljedjivanje interfejsa je je u sustini isto kao i kod klasa. Kada zelimo koristiti neki interfejs korisimo kljucnu rijec **implements**.

**class Chicken extends Animal implements Edible {**

**@Override**

**public String howToEat() {**

**return "Chicken: Fry it";**

**}**

**@Override**

**public String sound() {**

**return "Chicken: cock-a-doodle-doo";**

**}**

**}**

Klasa chicken implementira Edible interfejs da bi specificiralo da je kokos jestiva zivotinja. Kada klasa implementira interfejs ona implementira sve metode definisane u interfejsu sa tacnom signaturom i tipm podataka koji vracaju. Chicken u ovom slucaju extenda i Animal klasu radi zvuka koje proizvode.

**abstract class Fruit implements Edible {**

**// Data fields, constructors, and methods omitted here**

**}**

**class Apple extends Fruit {**

**@Override**

**public String howToEat() {**

**return "Apple: Make apple cider";**

**}**

**}**

**class Orange extends Fruit {**

**@Override**

**public String howToEat() {**

**return "Orange: Make orange juice";**

**}**

**}**

Fruit klasa implementira Edible ali obzirom da ne implementira howToEat metodu ova klasa se mora modifikovati kao abstraktna. Konkretne subklase Fruit klase moraju implementirati howToEat metodu, Apple i Orange klase.

**public class TestEdible {**

**public static void main(String[] args) {**

**Object[] objects = {new Tiger(), new Chicken(), new Apple()};**

**for (int i = 0; i < objects.length; i++) {**

**if (objects[i] instanceof Edible)**

**System.out.println(((Edible)objects[i]).howToEat());**

**if (objects[i] instanceof Animal) {**

**System.out.println(((Animal)objects[i]).sound());**

**}**

**}**

**}**

**}**

Main metoda kreira niz sa tri objekta Tiger, Chicken i Apple i poziva howToEat metodu ukoliko je elemenat jestiv i sound metodu ukoliko je elemenat zivotinja. U sustini Edible interfejs definise zajednicko ponasanje za jestive objekte, te svi jestivi objekti imaju howToEat metodu.

Obzirom da su svi data fieldovi u interfejsu automatski definisani kao **public static final** i sve metode su **public abstract** Java dozvoljava da se modifikatori izostavljaju te umjesto

**Public static final int K = 1;**  moze se koristiti **int K = 1;**

**UPOREDIVI INTERFEJSI**

**Comparable** interfejs definise compareTo metodu za poredjenje objekata.

Uzmimo za primjer da zelimo da dizajniramo genericnu metodu koja pronalazi veci od dva objekta istoga tipa, npr. dva studenta, dva datuma, dva kruga, dva pravougaonika ili dva kvadrata. Da bi ovo bilo izvedivo dva objekta moraju biti uporediva, znaci tajednicko ponasanje za objekte mora biti uporedivo. Java pruza **Comparable** interfejs za ove svrhe. Interfejs se definise kao:

**// Interface for comparing objects, defined in java.lang**

**package java.lang;**

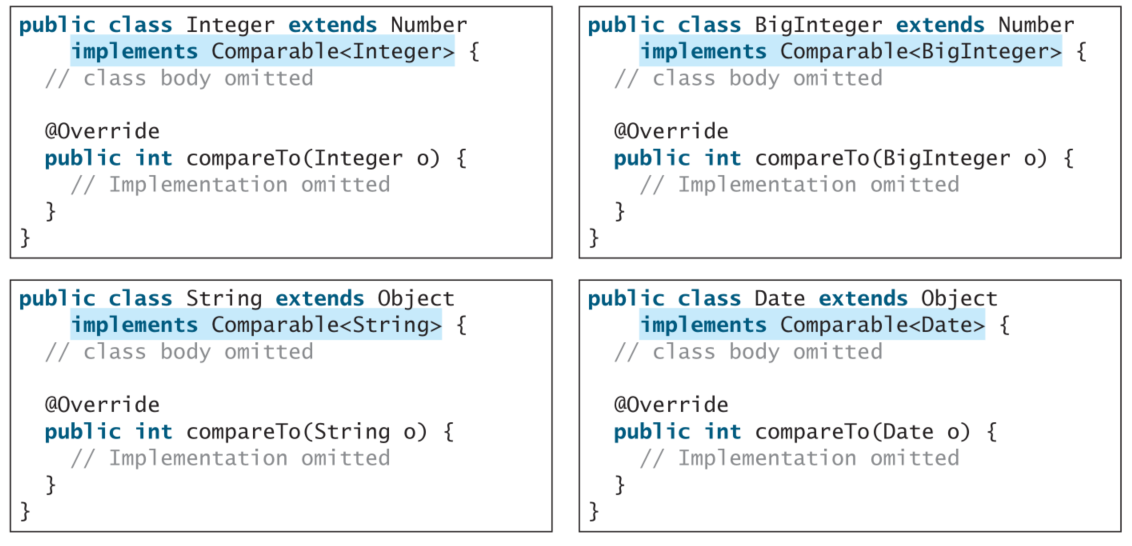
**public interface Comparable<E> {**

**public int compareTo(E o);**

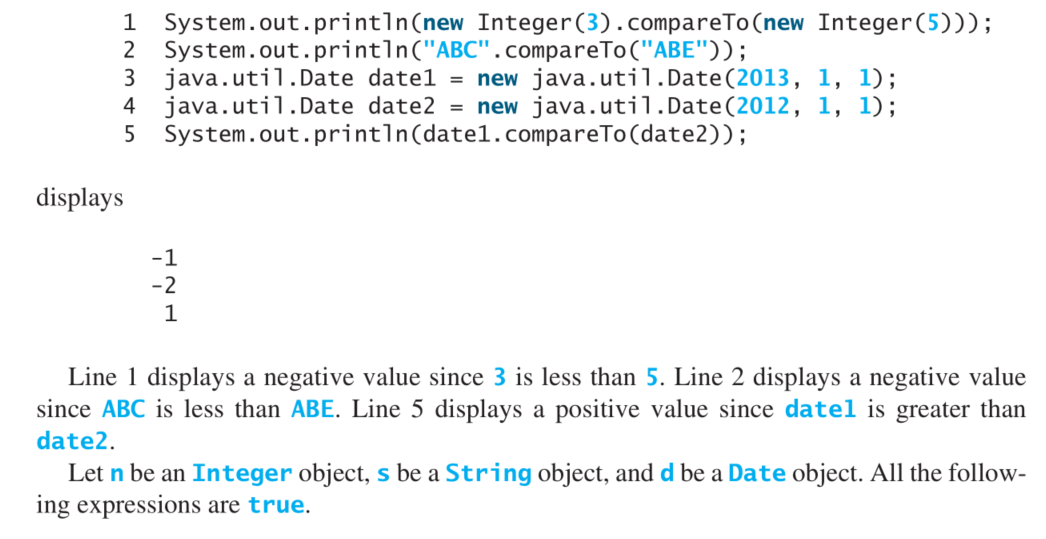
**}**

**compareTo** metoda utvrdjuje redosljed ovog objekta sa specificiranim objektom **o** i vraca negativni integer, nulu ili pozitivni integer ukoliko je ovaj objekat manji, jednak ili veci od **o**.

**Comparable** interfejs je genericki interfejs u Javi. Genericki tip **E** ce biti zamijenjen konkretnim tipm kada se implementira ovaj interfejs. Mnoge klas eu Java biblioteki implementiraju **Comparable** kako bi definirale prirodni poredak za objekte. Klase **Byte, Short, Integer, Long, Float, Double, Character, BigInteger, BigDecimal, Calendar, String i Date** sve implementiraju **Comparable** interfejs. Na primjer **Integer, BigInteger, String i Date** klase su definirane u Java API-u na sljedeci nacin:



Obzirom na navedeno brojevi su uporedivi, stringovi su upordivi kao i datumi. Mozete koristiti **compareTo**  metodu da uporedite dva broja, dva stringa i dva datuma. Na primjer:



Obzirom da svi **Comparable** objekti imaju **compareTo** metodu, **java.util.Arrays.sort(Object[])** metoda u Java Api-ju koristi **compareTo** metodu kako bi uporedila i sortirala objekte u nizu, podrazumjevajuci da su objekti instance **Comparable** interfejsa.

**INTERFEJS KOJI SE MOZE KLONIRATI**

Specificira da se objekat moze klonirati.

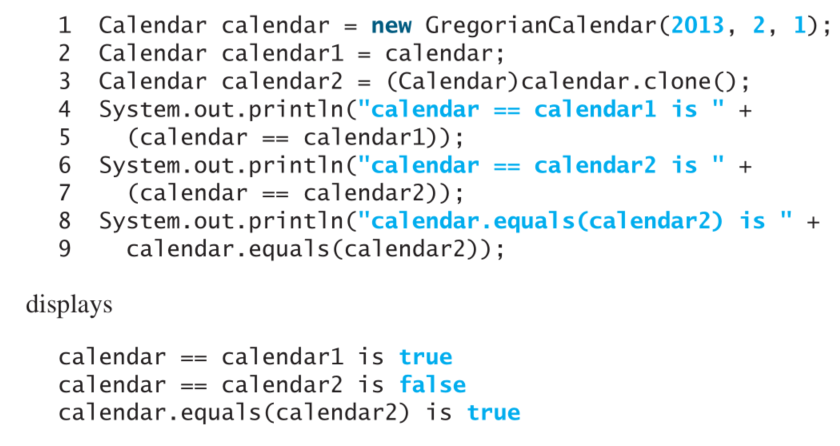
Cesto je pozeljno da se kreira kopija nekog objekta. Da bi se ovo izvelo potrebno je koristiti **clone** metodu i razumiti **Cloneable** interfejs. Interfejsi sadrze konstante i metode ali **Cloneable** interfejs je specijalan slucaj i definisan je na sljedeci nacin:

**package java.lang;**

**public interface Cloneable {**

**}**

Interfejs je prazan. Interfejs sa praznim tijelom se naziva i **marker interfaceB.** Marker interface ne sadzikonstante i metode vec se koristi kako bi se ukazalo da klasa posjeduje odredjena pozeljna svojstva. Klasa koja implementira **Cloneable** interfejs je oznacena kao cloneable i njeni objekti mogu biti klonirani koristeci **clone()** metodu definiranu u **Object** klasi. Mnoge klase u Java biblioteci (Date, Calendar i ArrayList) implementiraju **Cloneable** te instance tih klasa mogu biti klonirane. Primjer:

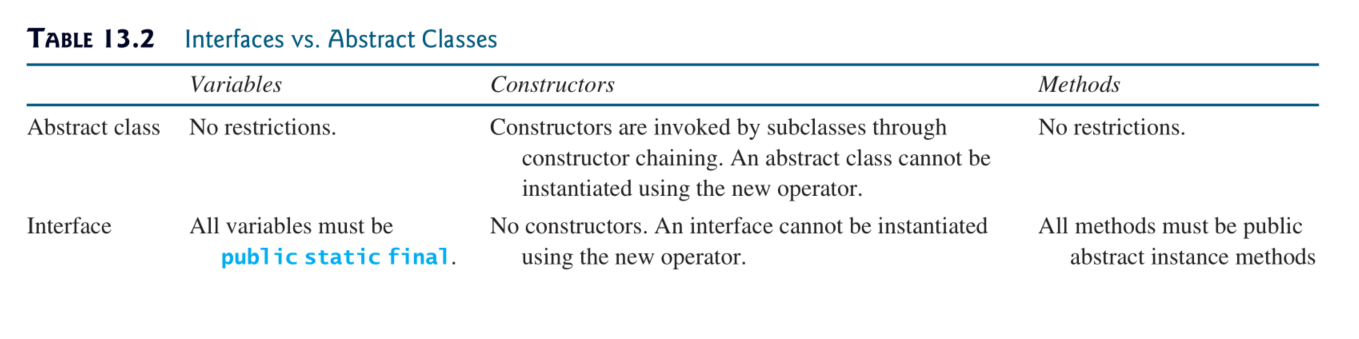


U navedenom primjeru linija 2 koda kopira referencu calendar na calendar1 tako da calendar i calendar1 pokazuju na isti **Calendar** objekat. Linija 3 kreira novi objekat koji je klon calendar i dodijeljuje referencu novog objekta na calendar2. calendar2 i calendar su razlicini objekti sa istim sadrzajem.

**INTERFEJSI vs ABSTRAKTNE KLASE**

Klasa moze implementirati vise interfejsa ali moze extendati samo jednu superklasu.

Interfejs se moze koristiti, vise ili manje, na isti nacin kao i abstraktna klasa ali definisanje interfejsa je drugacije od abstraktne klase. Sljedeca tabela pokazuje razlike:



Java dozvoljava samo jedno naslijedjivanje za klase ali dozvoljava naslijedjivanje vise interfejsa. Na primjer:

**public class NewClass extends BaseClass**

**implements Interface1, ..., InterfaceN {**

**...**

**}**

Interfejs moze naslijediti drugi interfejs koristeci se **extends** kljucnom rijecju i takav interfejs se naziva **subinterface**. Na primjer **NewInterface** u kodu ispod je subinterface od Interface1 .... i InterfaceN.

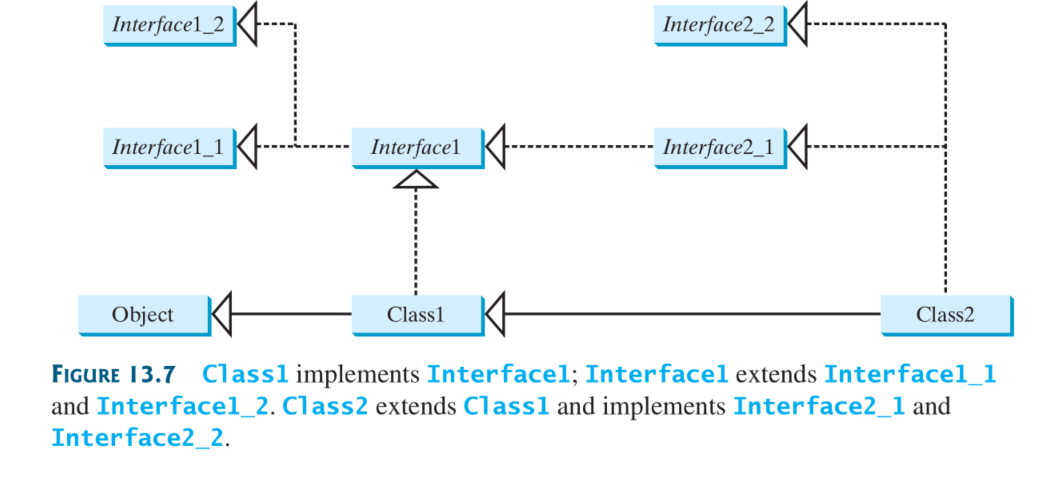
**public interface NewInterface extends Interface1, ... , InterfaceN {**

**// constants and abstract methods**

**}**

Klasa koja implementira **NewInterface** mora implementirati i abstraktne metode definirane u **NewInterface, Interface1, ... InterfaceN**. Interface moze extendati druge interfejse ali ne i klase. Klasa moze extendati svoju superklasu i implementirati vise interfejsa.

Sve klase dijele zajednicki korijen, **Object** klasu ali ne postoji zajednicki korijen za interfejse. Bas kao i klasa i interfejs takoidje definise tip. Varijalba interfejs tipa moze ukazivati bilo koju instancu klase koja implementira interfejs. Ukoliko klasa implementira interfejs interfejs postaje kao superklasa za tu klasu. Interfejs se moze koristiti kao tip podataka i bacati varijablu interfejs tipa njegovoj subklasi i obratno. Na primjer ukoliko pretpostavimo da je **c** instanca **Class2** u sljedecem prikazu **c** je takodje instanca **Object, Class1, Interface1, Interface1\_1, Interface1\_2, Interface2\_1 i Interface2\_2**.



Imena klasa su imenice dok imena interfejsa mogu biti pridjevi ili imenice.

I abstraktne klase i interfejsi se mogu koristiti kako bi se specificiralo zajednicko ponasanje objekata. Kako onda odluciti da li koristiti interfejs ili klasu? U sustini jak **is-a** odnos koji ocigledno opisuje rodtelj-dijete odnos bi trebao biti uredjivan koristenjem klasa. Na primjer Gregorijanski kalendar je kalendar tako da veza izmedju klasa **java.util.GregorianCalendar** i  **java.util.Calendar** ce biti uredjivana putem klasa i naslijedjivanja. Slaba **is-a** veza takodje poznata kao na neki nacin je veza, ukazuje da objekat posjeduje odredjena svojstva. Ovakva veza moze se uredjivati putem interfejsa. Na primjer svi stringovi su uporedivi tako da **String** klasa implementira **Comparable** interfejs.

Generalno gledajuci interfejsi su pozeljniji ispred abstraktnih klasaiz razloga sto interfejs moze definisati supertip za nepovezane klase. Interfejsi su dosta fleksibilniji naspram klasa. Pretpostavimo da **howToEat** je definisana u **Animal** klasi kako slijedi:

**abstract class Animal {**

**public abstract String howToEat();**

**}**

Dvije subklase od **Animal su definisane na sljedeci nacin:**

**class Chicken extends Animal {**

**@Override**

**public String howToEat() {**

**return "Fry it";**

**}**

**}**

**class Duck extends Animal {**

**@Override**

**public String howToEat() {**

**return "Roast it";**

**}**

**}**

Gledajuci predstavljeu hijerarhiju naslijedjivanja, polimorfizam nam omogucava da drzimo referencu na **Chicken** objekat ili **Duck** objekat u varijabli **Animal** tipa kao sto se vidi u sljedecem kodu:

**public static void main(String[] args) {**

**Animal animal = new Chicken();**

**eat(animal);**

**animal = new Duck();**

**eat(animal);**

**}**

**public static void eat(Animal animal) {**

**animal.howToEat();**

**}**

JVM dinamicki odlucuje koju **howToEat** metodu da pozove bazirajuci se na objekat koji je pozvao metodu.

Moze se definisati subklasa od **Animal**. Ali postoje ogranicanja u tom slucaju: Subklasa mora biti za neku drugu zivotinju (npr. curku).

Interfejsi nemaju ova ogranicenja, oni dozvoljavaju vise fleksibilnosti naspram klasa zato sto se ne mora voditi racuna da se sve strpa u jedan tip klase. Mozete definisati **howToEat()** metodu pomocu interfejsa i pustiti da sluzi kao zajednicki tip za druge klase. Npr:

**public static void main(String[] args) {**

**Edible stuff = new Chicken();**

**eat(stuff);**

**stuff = new Duck();**

**eat(stuff);**

**stuff = new Broccoli();**

**eat(stuff);**

**}**

**public static void eat(Edible stuff) {**

**stuff.howToEat();**

**}**

**interface Edible {**

**public String howToEat();**

**}**

**class Chicken implements Edible {**

**@Override**

**public String howToEat() {**

**return "Fry it";**

**}**

**}**

**class Duck implements Edible {**

**@Override**

**public String howToEat() {**

**return "Roast it";**

**}**

**}**

**class Broccoli implements Edible {**

**@Override**

**public String howToEat() {**

**return "Stir-fry it";**

**}**

Kako bi se definirala klasa koja predstavlja jestive objekte, jednostavno dopustite klasi da implementira **Edible** interfejs. Klasa je sada subtip **Edible** tipa i bilo koji **Edible** objekta moze se koristiti da se pozove **howToEat.**

**SMJERNICE ZA DIZAJN KLASA**

1. Kohezija

Klasa bi trebala da opisuje jedan pojav/entitet i sve operacije unutar klase bi trebale logicki da s euklope kako bi podrzavale povezanu svrhu. Na primjer mozemo koristiti klasu za studente ali ne bi trebali kombinirati klasu za studente sa osobljem zato sto su studenti i osoblje razliciti entiteti.

Jedan entitet sa mnogim odgovornostima moze biti razbijen u nekoliko klasa kako bi se odvojile odgovornosti. Klase String, StringBuilder i StringBuffer sve rade sa stringovima ali imaju razlicite odgovornosti. String klasa sluzi za **immutable** stringove, StringBuilder klasa za kreiranje **mutable** stringova, te StringBuffer klasa koja sadrzi sinhronizovane metode za izmijenu i nadogradnju stringova.

1. Konsistentnost

Pratite standardne Java programerske stilove i konvencije za imenovanje. Koristite deskriptivna imena za klase, data fieldove i metode. Popularan stil je da se postavi deklaracija podatka prije konstruktora i postavi konstruktor prije metode.

Neka imena budu konsistentna. Nije dobra praksa birati razlicita imena za sluicne operacije. Na primjer **length()** metoda vraca velicinu Stringa, StringBuilder i StringBuffer-a. Bilo bi nekonsistentno da su razlicita imena koristena za ovu metodu za razlicite klase.

U globalu konsistentno bi trebali kreirati **public** no-arg konstruktor za kreiranje defaultnih instanci. Ukoliko klasa ne podrzava no-arg konstruktor dokumentujte razlog. Ukoliko konstruktori nisu definisani eksplicitno preuzima se defaultni no-arg konstruktor sa praznim tijelom.

Ukoliko zelite da sprijecite korisnike u kreiranju klase mozete kreirati **private** konstruktor u klasi kao sto je slucaj u **Math** klasi.

1. Enkapsulacija

Klasa bi trebala koristiti **private** modifikatore kako bi sakrila svoje podatke od korisnika. Ovo olaksava odrzivost klase.

Pruzite getter metode samo ukoliko zelite da podaci budu citljivi i pruzite setter metode samo ukoliko zelite da podaci budu izmmjenjivi. Na primjer **Rational** klasa pruza gettere za **numerator i denominator** ali ne setter metode zato sto je **Rational** objekt immutable.

1. Jasnoća

Kohezija, konstitentnost i enkapsulacija su dobre smjernice pri postizanju jasnog dizajna. Osim toga klasa bi trebala imati jasan ugovor koji je lako objasnjiv i razumnjiv.

Korisnici mogu inkorporirati klase u razlicitim kombinacijama, redosljedima, i okruzenjima. Zbog toga trebali bi dizajnirati klasu koja nema ogranicenja na nacin kako ili kada je korisnici mogu koristiti, dizajnirati svojstva na nacin da dozvoljavaju korisniku da ih podesi u bilo kojem redosljedu i sa bilo kojom kombinacijom vrijednosti i dizajnirati metode koje funkcioniraju nezavisno od njihovog redosljeda pojavljivanja.

Na primjer **Loan** klasa sadrzi svojstva **loanAmount, numberOfYears** i  **annualInterestrate.** Vrijednosti ovih svojstava semogu podesiti u bilo kojem redosljedu.

Metode bi trebale biti dizajnirane intuitivno bez da uzrokuju konfuziju. Na primjer **substring(int beginIndex, int andIndex)** metoda u **String** klasi je pomalo zbunjujuca. Ona vraca substring od **beginIndex** do **endIndex – 1**, a ne do **endIndex**. Bilo bi intuitivnije da vraca substring od beginIndex do endIndex.

Ne biste trebali deklarirati data field koji se moze izvesti iz drugih data fieldova. Na primjer **Person** klasa ima dva data fielda **birthdate i age**. Obzirom da se age moze izvesti od birthdate-a ne bi trebalo da se deklarise kao data field.

**public class Person {**

**private java.util.Date birthDate;**

**private int age;**

**...**

**}**

1. Kompletentnost

Klase su dizajnirane kako bi ich koristilo vise musterija. Kako bi bile korisne u vecem broju aplikacija klase bi trebale da pruze vise nacina kostumizacije kroz svojstva i metode. Na primjer String klasa sadrzi vise od 40 metoda koje su korisne za razlicite aplikacije.

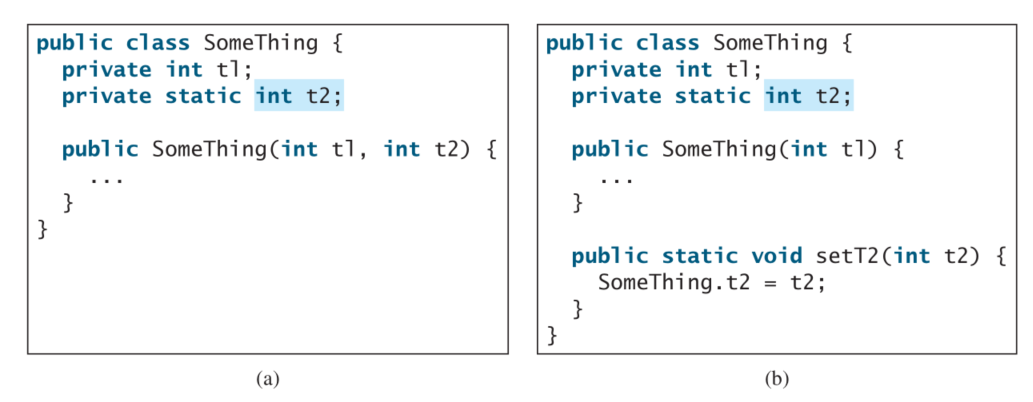
1. Instanca vs Static

Varijabla ili metoda koje su ovisne o specificnoj instance klase muraju biti instancne varijable I metode. Varijabla koju dijele sve instance klase bi trebala biti deklarisana kao **static**. Na primjer barijabla **numberOfObjects** u **CircleWithPrivateDataFields** u primjeru u knjizi (primjer 9.8) je dijeljena od strane svih objekata navedene klase I zato je deklarisana kao static**.**

Metoda koja nije ovisna o specificnoj instance bi trebala biti definisana kao **static**. Na primjer metoda **getNumberOfObjects()** u **CircleWithPrivateDataFields** klasi nije vezana ni za jednu specificnu instance i deklarisana je kao static.

Uvijek upućujte/referencirajte staticne varijable i metode iz imena klasa (radije nego referencnih varijabli) kako bi se poboljsala citljivost i izbjegle greske.

Ne prosljedjujte parameter iz konstruktora kako bi inicijalizirali staticni data field. Bolje je da se koristi setter metoda kako bi se promijenio staticni data field. Tako da je klasa u primjeru a je bolje da se zamijeni klasom iz primjera b:



Instanca i static su integralni dijelovi objekto orijentisano programiranja. Data field ili metoda sui li instance ili static. Nemojte slucajno preci preko staticnih data fieldova ili metoda. Cesta greska je definistai instancnu metodu koja je trebala biti static. Na primjer **factorial(int n)**  metoda za izracunavanje faktorijala od n bi trebala biti definisana kao static zato sto je neovisna o bilo kakvoj instanci.

Konstruktor je uvijek instanca zato sto se koristi kako bi se kreirala specificna instanca. Staticna varijabla ili metoda moze biti pozvana id instancne metode ali instancna metoda ili varijabla ne moze biti pozvana iz staticne metode.

1. Naslijedjivanje vs kompozicija (agregacija)

Razlika izmedju naslijedjivanja i kompozicije je razlika u **is-a** i **has-a** odnosu. Na primjer jabuka je voce tako da bi smo koristili naslijedjivanje da oblikujemo odnos izmedju klasa **Apple** i **Fruit**. Osoba ima ime te bi se koristila kompozicija da se oblikuje odnos izmedju klasa **Person** i **Name.**

1. Interfejsi vs Abstraktne klase

Oboje i interfejsi i abstraktne klase se koriste kako bi se specificiralo zajednicko ponasanje za objekte. Kako onda odluciti da li koristiti interfejs ili abstraktnu klasu. U globalu jak **is-a** odnos koji jasno opisuje roditelj dijete odnos bi trebao da bude uredjen koristeljem klasa. Na primjer obzirom da je narandza voce njihov odnos bi trebao da bude uredjen koristenjem naslijedjivanja. Slab **is-a** odnos takodje poznat i kao **is-kind-of** odnos ukazuje da object posjeduje neka odredjena svojstva. Slab **is-a** odnos moze biti uredjen interfejsima. Na primjer svi stringovi su uporedivi tako da String klasa implementira Comparable interfejs. Krug ili pravougaonik je geometrijski objekat tako da Circle moze biti dizajniran kao subklasa od GeometricObjekt. Krugovi su drugaciji i uporedivi bazirano na njihovom radijusu tako da Circle moze implementirati Comparable interfejs.

Interfejsi su fleksibilniji od abstraktnih klasa zato sto subklasa moze extendati samo jednu superklasu ali moze implementirati bilo koji broj interfejsa. Medjutim interfejsi ne mogu sadrzavati konkretne metode. Vrline interfejsa i abstraktnih klasa mogu biti kombinirane u jednu cjelinu na nacin da se kreira interfejs sa abstraktnom klasom koja ga implementira. Tada se moze koristiti interfejs ili abstraktna klasa sta god je zgodnije.