1.5 Drugi objektno-orjentisani programski jezici

Cilj ovog poglavlja je da tamo kratak pregled drugih objektno orjentisanih programskih jezika. Činjenica je da je i prije programskog jezika C++ bilo razvijeno nekoliko OO programskih jezika a da sa ovim programskim jezikom razvoj u ovoj oblasti nije završen. Iz različitih razloga uvedeno je desetak novih programskih jezika od kojih je nekoliko u masovnoj upotrebi. Pored ove činjenice, da je razvijen veliki broj programskih jezika, vrijedi istaći i to da mnogi programerski "čistunci" nazivaju programski jezik C++ hibridnim jer je činjenica da on predstavlja hibrid programskog jezika C preko kojega su "nakalemljeni" OO koncepti, odnosno jezik koji posjeduje određenu nefleksibilnost zbog potrebe da zadovolji jednu od osnovnih svojih premisa pod kojim je uvedene a to je kompatibilnost unazad. Kompatibilnost unazad je želja da se obezbjedi izvršavanje svih kodova iz programskog jezika C u kompajlerima programskog jezika C++. Drugi, element koji je iz jednog ugla velika prednost ovog programskog jezika, dok je iz drugog ugla značajno ograničenje, je standardizacija. Naime, C++ stadardizuje komitet WG21 (preciznije ISO/IEC JTC1/SC22/WG21). Ovaj komitet je veoma oprezan i birokratski. Do danas je izdao samo pet revizija standarda (1998, 2003, 2008, 2011 i 2014) dok je u pripremi revizija za 2017 godinu. Prve tri revizije su pravljene na 5 godina što je nedovoljno za dinamiku današnjeg razvoja programiranja što je tjeralo softverske kuće koje prave kompajlere da odstupaju od standarda i prave sopstvena nestandardna proširenja. Ovaj problem se postepeno prevazilazi uvođenjem revizija na tri godine sa idejom da to bude i češće. Sa druge strane programski jezici koji nisu podložni takvoj demokratskoj atmosferi izmjena su kritikovani zbog nestabilnosti i nestandardizacije. Pored ovoga, C++ izdvaja i činjenica da je u pitanju programski jezik naglašenom namjenjen za kompajliranje (efikasnije izvršavanje, veliki izvršni kod, opasnost od korišćenja u mrežnom okruženju. Četvrta karakteristika koju ovdje treba pomenuti da je C++ minimalistički programski jezik. Njegova definicija podrazumijeva mali broj ključnih riječi dok je sve ostalo prepušteno programskim bibliotekama. Drugi programski jezici, čak i kada imaju mali broj ključnih riječi, ipak mnogo više zavise od sadržaja programskih biblioteka nego što je to slučaj kod C++. Ova karakteristika je možda najmanje bitna od svih prethodnih i najviši uticaj ima na brzinu izvršavanja a mnogo manji na ono što se u radu sa programskim jezikom može postići. Bez obzira na mali broj ključnih riječi i programski jezik C++ je snadbjeven sa velikim brojem programskih biblioteka čime se omogućuje korišćenje zatečenog i akumuliranog programerskog znanja i sprečava potreba da se program "gradi" od nule.

Kada planiramo da opišemo druge programske jezike i da poredimo C++ sa njima teško je izbjeći zamku ne-fer poređenja koja je prilično odomaćena u praksi. Na primjer, obično se diskutuje u pravcu da je sa nekim programskim jezikom nešto lako postignuto a sa drugim da to nije tako. Međutim, često je dovoljno potražiti po Internetu i pronaći adekvatno rješenje problema u bilo kom programskom jeziku. Dalje, programski jezici se porede po brzini. Najčešće se želi pokazati da neki programski jezik nije tako spor pa se onda biraju testovi koji su krajnje nerealni tako da pokažu neki programski jezik nije u deficitu po ovom kriterijumu. Neko naše iskustvo pokazuje da su većina ovakvih analiza krajnje pristrasne i da više prikrivaju nego što otkrivaju.

Najpoznatiji OO programski jezici su Java, C#, Eiffel, Objective-C, varijante objektnog Pascal-a, PHP5, Python, Ruby, VisualBasic, kao i programski jezici od istorijskog značaja kao što su Smalltalk i ADA itd. Gotovo svi navedeni programski jezici posjeduju varijante. OO koncepti su danas ugrađeni u gotovo sve druge softverske pakete kao što je na primjer MATLAB.

Pojedini gorenavedeni programski jezici su veoma slični po sintaksi programskom jeziku C++. Prije svega Java, C#, PHP5. Jezici porijeklom iz objektnih varijanti varijanti Pascala ako se i razlikuju po sintaksi u mnogo čemu su slični po osnovnoj strukturi sa C-om a posebno C++-om. Programski jezik Objective-C premda nosi u imenu sličan korjen i povezanost sa programskim jezikom C ipak se dosta sintaksno razlikuje od programskog jezika C++. Objective-C dosta podsjeća na programski jezik Eiffel. Programski jezik Eiffel je nastao kao plod naučne studije sa pitanjem što

bi jedan OO programski jezik trebalo da sadrži. Ovaj programski jezik, nažalost, nije zaživio van akademske zajednice ali predstavlja odličnu teorijsku osnovu po pitanju dizajniranja "optimalnog" skupa koncepata koji čine neki objektno-orjentisani programski jezik. Objective-C po sintaksi je zapravo hybrid programskog jezika C i istorijski veoma bitnog jezika Smalltalk. Python je programski jezik opšte namjene kojim se mogu podržati različite programerske paradigme među kojima je OO paradigma. Podsjeća dobrim dijelom na neku mješavinu MATLAB-a i programskog jezika C ali je ipak po mnogo čemu unikatan. Programski jezik VisualBasic samo po imenu i istorijskom razvoju posjeduje vezu sa starim programskim jezikom Basic i njegovim derivatima. Po OO konceptima on je najsiromašniji od pomenutih. Međutim, popularnost ovog programskog jezika je velika posebno iz razloga jednostavnosti programiranja i činjenice da je predmetni jezik podrška brojnim aplikacijama kao jezik koji se koristi za njihovo programiranje i podešavanja.

Sada ćemo se u kratkim crtama upoznati sa osnovnim elementima nekih od pomenutih OO programskih jezika uz istovremeno njihovo poređenje sa programskim jezikom C++ do nivoa preciznosti kojega smatramo uputnim i do nivoa koji nam obim ovog udžbenika dozvoljava. Jezike koji su po sintaksi sličniji C++ detaljnije ćemo obraditi od onih koji to nisu.

1.5.1 Programski jezik Java

Java programski jezik (čita se džava) počeo je da se razvija ranih 1990 godina. Ovo ime programskog jezika je odabrano nakon nekoliko pokušaja i odomaćeno je od 1995. godine. Premda je sintaksa programskog jezika Java veoma slična C++-u motivacija za većinu koncepata je nađena u OO verzijama programskog jezika Pascal. U početku svog životnog vijeka programski jezik Java je održavala kompanija Sun Microsystems a danas je to preuzela korporacije Oracle.

Prva uočljiva razlika između programskog jezika C++ i Java-e leži u činjenici da u Java-i nema koda van klase odnosno da i glavni program mora da bude metod neke klase. Druga veoma bitna razlika je u namjeni. Programski jezik Java je od samog početka namjenjen za distribuirano izvršavanje, odnosno za situaciju kada se kod programa nalazi na jednoj mašini a program se izvršava na drugoj. Ovo nameće potrebu da se kod u Java jeziku interpretira što se na mašini na kojoj se kod izvršava postiže instalacijom interpretera koji je obično tzv. Java Virtuelna mašina (JVM). Treća uočljiva razlika u odnosu na C++ je činjenica da pošto kompletan kod klase u Java-i se mora nalaziti u jednom fajlu nije moguće kao u C++ razdvojiti specifikaciju klasinog interfejs od implementacije metoda. To je jedan od uzroka nečitljivosti Java koda u odnosu na C++. Druge, manje uočljive ali jako bitne razlike su prije svega u izostanku pokazivača kao i nepostojanju višestrukog nasljeđivanja. Prvi koncept je izbjegnut zbog namjere da se Java izvršava na udaljenom računaru što dovodi do mogućnosti da se preko pokazivača i adresiranja djelova sistema naruši bezbjednost. Drugi koncept je izbjegnut zbog komplikovanosti. Naredni koncept koji je dosta ograničen kod Java-e je izostatak preklapanja operatora. Odnosno, da budemo precizniji, neki operatori su već preklopljeni kao što je to slučaj kod operatora + za stringove (konkatenacija) ali osim nekoliko sličnih nijedan drugi operator nije preklopljen niti je ostavljena prilika programeru da sam izvrši preklapanje.

Što se tiče tipova podataka Java dijeli podatke u približno tri kategorije. Prva kategorija su prosti podaci (cijeli, realni brojevi, karakteri, logičke promjenljive kojih u C++ do skoro nije bilo), nizovi (uključujući stringove sa posebnim pravima, nizove uključujući matrice) sa istim načinom indeksiranja kao što je u C/C++, kao i klasne tipove podataka. Dvije krupne razlike. Java je značajno strožije tipiziran jezik u odnosu na programski jezik C++ odnosno implicitne konverzije sa kojima smo se navikli raditi najčešće neće raditi u Java-i. Druga veoma neobična ali jako bitna razlika je da prilikom svake deklaracije komplikovanih objekata (klasnog tipa) zapravo se deklariše instanca navedene klase. Grubo govoreći praktično radimo sa referentnim tipovima a ne sa samim objektima klasnog tipa što smo u C++ gotovo po pravilu razdvajali na slučajeve kada radimo sa objektima (programske cjeline) i referencama (prilikom prenosa argumenata). Vremenom se C++ programer navikne na ovakav rad dok je onima koji su od početka u Java-i ovo prirodan način rada. Još

istaknimo mada za korisnika to je vjerovatno malo bitno svi prosti podaci su zapravo realizovani putem hijerarhije koja liči na hijerarhiju klasa.

Prikažimo jedan primjer koji uz demonstraciju koliko je Java sličan programski jezik sa C++ ujedno ukazuje na neke od razlika koje smo do sada pomenuli.

```
public class Primjer{
        int a;
        public int b;
        private int c;
        Primjer(int i){a=0; b=0; c=0;}
        Primjer(int i){a=i; b=i; c=i;}
}
```

Ovdje smo deklarisali (klasa je deklarisana kao **public** da bi bila vidjela van klase) klasu koja se sastoji samo od tri podatka člana. U Java-i je podrazumijevano da je vidljivost podataka članova javna ako se drugačije ne naglasi tako da je **a** javno. Dodali smo i konstruktor koji može biti preklopljen praktično na isti način kao u programskom jeziku C++. Deklarišimo sada klasu koja koristi klasu **Primjer**:

```
public class KoristiPrimjer{
        Primjer noviPrimjer=new Primjer();
        Primjer drugiPrimjer=new Primjer(10);
        public static void main(String args[]){
        int i=0,r=1,s=0;
        System.out.println("Zdravo");
        r=noviPrimjer.a+noviPrimjer.b;
        System.out.println(r);
        r=drugiPrimjer.a+drugiPrimjer.b;
        System.out.println(r);
        for(i=0;i<r;i++)
                if(r%i) s+=i;
        System.out.println(r);
                }
        }
```

Ovaj primjer klase je dat sa nekoliko namjena od koji je prva da ukažemo na ogromnu sintaksnu sličnost sa programskim jezikom C++ odnosno činjenicu da su gotovo sve naredbe za kontrolu toka programa iste kao u C/C++ kao i govoro sve ključne riječi, operatori, preklopljeni konstruktori itd. Uočavamo da koristimo funkciju **System.out.println** za prikaz rezultata što ukazuje na činjenicu da je Java programski jezik različit u odnosu na C/C++ jer nije istisnuo ulazno izlazne naredbe u programske biblioteke ili preciznije, u programskom jeziku Java mnogo je bleđa granica između ključnih riječi i ostalih funkcija koji su programeru na raspolagaju. Ovo nije tako loše kao što se čini iz logike C/C++ programera već se od početka praktično programeri privikavaju na činjenicu da će morati koristiti plodove tuđega rada i ono što programsko okruženje nudi. Vidimo da je konstruktor veoma sličan onom u programskom jeziku C++ ali je ovdje bitno ukazati na činjenicu da su podaci klasnih tipova u Java-i od početka svog života instance odnosno reference na objekte a ne u pravom smislu objekte što je isto kao i za ostale složene tipove podataka u Java. Najveći dio rada sa ovakvim objektima se obavlja bez problema ali određeno prilagođavanje je neophodno. Uočili smo nedostatak objekata **cin** i **cout** a ujedno i izostanak operatora >> i <<. Činjenica je da su neki operatori preklopljeni samo za stringove dok se ostali operatori ne mogu preklopiti (što je značajna mana Java-e).

Java nema prijatelje ali postoje načini da se slična funkcionalnost postigne. Što se tiče koncepta nasljeđivanja on je naravno prisutan u programskom jeziku Java sa tim ograničenjem da klasa može da naslijedi samo jednu osnovnu klasu (koncept višestruktog nasljeđivanja nije podržan).

Sintaksa nasljeđivanja se unekoliko razlikuje od one koja postoji u C++. Pretpostavimo da imamo osnovnu klasu:

```
class Osnovna{
…
}
```

Izvedena klasa se iz ove klase se prikazuju:

```
class Izvedena extends Osnovna{
…
}
```

Dakle, uočljiva je ključna riječ **extends** koja znači da izvedena klasa proširuje osnovnu klasu. Posmatrajmo jednostavan primjer klase:

```
class Primjer{
int z;
public void sabiranje(int x, int y){
        z=x+y;
        System.out.println("Suma je "+z);
}
public void oduzimanje(int x, int y){
        z=x-y;
        System.out.println("Razlika je "+z);
}
}
public class IzvedenaKlasa extends Primjer{
public void mnozenje(int x, int y){
        z=x-y;
        System.out.println("proizvod je "+z);
        }
public static void main(String args[]){
int a=20,b=10;
IzvedenaKlasa primjerak=new IzvedenaKlasa();
Primjerak.sabiranje(a,b);
Primjerak.oduzimanje(a,b);
Primjerak.mnozenje(a,b);
}
}
```

Osnovna poruka nasljeđivanja bi trebala da bude jasna iz konteksta primjerna. **IzvedenaKlasa** ima sve što i klasa **Primjer** uz neke dodatke. Ni ovdje se ne nasljeđuju kontruktori iz osnovne klase baš kao i u C++-u. U Java-i je dozvoljeno da imamo zasjenjvanje odnosno da u izvedenoj klasi imamo metode i podatke članove koji se zovu isto kao u osnovnoj klasi. Pristup podacima i metodima iz osnovne klase se onda ne obavlja putem operatora dosega već putem ključne riječi **super** (npr. **super.funk()** ili **super.podatak**). Iz konstruktorske funkcije izvedene klase pristup konstruktoru osnovne klase se obavlja putem ključne riječi **super()** a u zagradi navedeno argumente ovoga konstruktora. Ključna riječ (operator) **instanceof** nam ukazuje da li je neki objekat pripada klasi tako na primjer **a instanceif A** daje tačno (**true**) ako je objekat **a** instanca klase **A**. Ovo se može primjeniti i na objekte izvedenih klasa.

Kao što smo vidjeli Java ima znatna ograničenja vezana za nasljeđivanje u odnosu na C++. Da bi se neka od njih redukovala uvedeni su interfejsi. Interfejsi predstavljaju vezu sa nekim spoljašnjim programerskim elementima koji ih implementiraju a mi ne moramo da znamo koji su to djelovi programskog koda koji nam obezbjeđuju ovu funkcionalnost. Najčešće se prikazuju kao grupe

metoda grupisaneu blok koji liči na klasu sa praznim tijelom i ključnom riječju **interface** umjesto najave klase:

```
interface neki_interface{
        void otvori(int a);
        void zatvori(int b);
        void radi(int c);
}
```

Ako želite da kreirate klasu koja implementira interfejs onda se to najavljuje putem ključne riječ **implements**:

```
class neka_klasa implements neki_interface{
        otvor=0;
        zatvor=1;
        radi=1;
void otvori(int a){
        otvor=a;
        zatvor=0;
}
void zatvori(int b){
        otvor=0;
        zatvor=b;
}
void radi(int c){
        radi=c;
}
}
```

Interfejsi predstavljaju formalniju obavezanost klase da realizuju određene funkcije (ponekad se kaže ugovorna obaveza) odnosno neko ponašanje. Klasa koja najavljuje da će implementirati neki interfejs (**implements**) nije validna sve dok ne definiše i posljednji metod iz interfejsa. Interfejsi ne mogu da budu nasljeđeni od strane klase. Već smo više puta napomenuli da Java ne podržava višestruko nasljeđivanje. Ovaj problem se prevazilazi tako što se koriste interfejsi koje neka klasa garantuje. Stoga neka klasa može da koristi interfejse koje obezbjeđuje više drugih klasa ali ne može da direktno, kako smo uobičavali u C++-u, da naslijedi više klasa.

Kod Java-e svi metodi su podrazuno virtuelni a samo metodi koji su modifikovani sa ključnom riječju **final** a takođe privatni metodi nisu virtuelne funkcije.

U Java-i postoje apstraktne klase i čisti virtuelni metodi koji se nazivaju apstraktnim. Čisti virtuelni metodi se najavljuju ključnom riječju **abstract** pa stoga nije potrebno ove metode izjednačavati sa nulom kao što je to slučaj u C++-u. Klase sa abstraktnim metodima se deklarišu sa ključnim riječju **abstract** ispred naziva klase. Ostala pravila vezana za apstraktne klase i apstraktne metode su ista kao kod C++-a.

Šablonizovati se u Java jeziku mogu klase i interfejsi preko tipova podataka. Deklaracija je veoma slična onoj u C++ osim što nedostaje ključna riječ **template** već se parametri šablona kojih može biti više navode nakon imena šablonske klase:

```
class naziv_klase<T1, T2, …, Tq>{
        /* elementi klasinog interfejsa*/
        }
```

Očigledno je da su u zagradama <> navedeni parametri šablona, odnosno tipovi podataka po kojima se šablonizuje klasa. Na primjer klasa šablonizovana po jednom parametru može da bude:

```
public class Primjer<T> {
   private T t;
   public void set(T t) { this.t = t; }
   public T get() { return t; }
}
```

Konvencija (mada nije obavezna) je da parametri šablona – tipovi počinju slovom **T** ili je prvi tip označen sa **T**, sljedeći sa **S**, pa sa **U**, **V** i dalje po Engleskoj abecedi. Slične konvencije su prihvaćene u Java-i i za imenovanje nekih drugih koncepata. Na primjer izuzeci ili klase koje predstavljaju izuzetke (obično) počinju sa slovom **E** ali ove konvencije nijesu za nas od većeg značaja i danas nisu nešto što je karakteristično samo za ovaj programski jezik. Deklaracija instanca šablonske klase je potpuno ista kao u programskom jeziku C++:

```
Primjer<Integer> CjelobrojniPrimjer;
```

Nova instaca ove klase se kreira ponovo na uobičajeni način:

```
Primjer<Integer> CjelobrojniPrimjer=new Primjer<Integer>;
```

Parametar šablona može biti i drugi šablonizovani tip kao što je na primjer:

```
Primjer <DrugiSablon<Integer>> PrimjerDrSabl;
```

Kao što smo već istakli pored klasa u Java-i se mogu šablonizovati interfejsi. Na primjer posmatrajmo interfejs šablonizovan preko dva parametra:

```
public interface Dva<K, V> {
   public K dajK();
   public V dajV ();
}
```

Klasa koja implementira ovakav interfejs mora takođe da bude šablonska:

```
public class DvaKomada<K, V> implements Dva<K, V> {
   private K k;
   private V v;
   public DvaKomada(K k, V v) {
         this.k = k;
         this.v = v;
   }
   public K dajK()       { return k; }
   public V dajV() { return v; }
}
```

Ovaj kratak pregled sličnosti i razlika između C++ i Java-e ćemo zaključiti pitanjem sistema za obradu izuzetata. Postoje dva osnovna unapređenja koja postoje kod Java-e a odnose se na ključnu riječ **finally** koja ne postoji u programskom jeziku C++ i koju obično implementiraju kompajleri, odnosno na činjenicu da Java obezbjeđuje hijerarhiju postojećih klasa za izuzetke što C++ stavlja na teret programeru ili kompajleru. Java-ina strategija je suštinski prihvaćena u svim savremenim programskim sistemima pa se stoga može smatrati za boljim. Osnovna sintaksa sistema za obradu izuzetaka je slična kao u C++, **try** blok uokviruje zonu u kojoj su se izuzeci mogu pojaviti dok se izuzeci „hvataju“ sa **catch**-evima van uokvirenog bloka. Posmatrajmo sljedeći primjer:

```
try{
        int a[] = new int[2];
        a[0]=0;a[1]=1;
        System.out.println("Element :" + a[2]);
        }catch(ArrayIndexOutOfBoundsException e){
                System.out.println("Van definisanih indeksa:");
        }
```

Ovaj primjer ilustruje čestu praksu u Java-i i drugim sličnim alatima a to je da na prvi pogled unutar bloka nemamo bacanje izuzetka već izuzetak zapravo baca dio programskog u kojem izuzetak nastaje bez eksplicitne kontrole korisnika. Slično je i kod programskih funkcija pa se posao programera dosta često sastoji od podešavanja tipova izuzetaka koje može da baci kod kojega uokviruje **try** blok. U predmetnom slučaju radi se o izuzetku tipa **ArrayIndexOutOfBoundsException** (indeks van granica). Svi tipovi izuzetaka u Javi su izvedeni iz klase **Exception**. Kao i u C++ nakon **try** bloka može biti više **catch**-eva spremnih da obrade izuzetke raznih tipova:

```
try{
        //Blok u kojem se pojavljuju izuzeci
        }catch(TipIzuzetka1 e1){
          //Obrada izuzetka
        }catch(TipIzuzetka2 e2){
          //Obrada izuzetka
        }catch(TipIzuzetka3 e3){
          // Obrada izuzetka
}
```

Kao i u programskom jeziku C++ postoji i ključna riječ **throw** kojom baca izuzetak koja se koristi na isti način. Jedna bitna karakteristika u novijim verzijama Java-e je mogućnost da isti **catch** blok obradi više tipova izuzetaka:

```
catch (IzuzetakTipa1|IzuzetakTipa2 e){
        //obrada izuzetka
}
```

Predmetni **catch** blok može da obradi izuzetke dva tipa koji su u ovom slučaju u listi argumenata razvojeni operatorom | kao da je u pitanju operacija ili. I ovdje važi pravilo da izuzetke obrađuje prvi **catch** blok koji to može a ne onaj koji ima najbolje poklapanje. Ako želite da naglasite da vaš metod baca neki izuzetak putem ključne riječi **throw** (u pojedinim uslovima ste obavezni da to uradite ali o ovome nećemo detaljnije raspravljati) u deklaraciji se može dodati sekcija **throws** koja specificira kojeg je tipa navedeni izuzetak:

```
int funkcija(int a) throws TipIzuzetka{
        //neki kod
        Throw TipIzuzetka();
}
```

Premda se može koristiti i na druge načine **finally** blok koji ne postoji u C++-u se postavlja obično na kraj sekcije sa **catch** blokovima iza **try**-a i uvjek se izvršava bez obzira da li se izuzetak dogodio:

```
try{
        int a[] = new int[2];
        a[0]=0;a[1]=1;
        System.out.println("Element :" + a[2]);
        }catch(ArrayIndexOutOfBoundsException e){
```

```
            System.out.println("Van definisanih indeksa:");
        } finally{
            System.out.println("Izvršava se finally");
            System.out.println("Prvi element :" + a[0]);
        }
}
```

Osnovna prednost kod primjene **finally** iskaza ogleda se u činjenici da se može koristiti za operacije za koje smo sigurni da ih je neophodno obaviti bez obzira da li su se desile vanredne okolnosti ili nisu (npr. zatvaranje fajlova na kraju programa ili zatvaranja konekcije sa Internetom ili bazom podataka što se mora uraditi bez obzira da li su se izuzetne situcije desile ili ne). Za ovakve obavezne operacije u Java postoji specijalni tip **try** bloka koji se naziva i **try** sa resursima kojega ovdje nećemo objašnjavati.

Pored onih izuzetaka koje smo dobili u Java-inom sistemu postoji mogućnost da sami kreirate klasu koja predstavlja izuzetak. Ova klasa mora biti izvedena iz klase **Exception** ili klase koja je izvedena iz ove klase a moguće su još neke varijante. Na primjer:

```
class MojIzuzetakException extends Exception{
        // neki dodaci
}
```

Ovim smo u suštini kompletirali kratak pregled razlika koje su uvedene u programskom jeziku Java u odnosu na one koje postoje u C++. Osnovni komentar je da te razlike nisu veće od onih koje su potrebne da se neko ko je dobro savladao teorijske osnove C++ familijarizuje sa nekim praktičnim C++ radnim okruženjem i sistemom pisanja programa za takav kompajler ili radno okruženje.

1.5.2 Programski jezik C#

Programski jezik C# (čita se si-šarp) takođe se može smatrati izvedenim iz istog C/C++ korjena kao što se Java barem po sintaksi može smatrati dijelom istog nasljeđa. Na početku životnog vijeka C# on je veoma podsjećao na Java-u ali kako godine idu dolazi do postepene divergencije i evolucije ova dva jezika u različitim pravcima. Kao i programski jezik Java i C# je nastao na inicijativu kompanije, u ovom slučaju Microsofta. On nije nastao u tolikoj mjeri kao potreba da se izađe u susret radu sa različitim, često distribuiranim, platformama kao što je to slučaj kod Java-e već više kao odgovor na praktične probleme koji su se javili programerima u baratanju sa Windows operativnim sistemom čiji je vlasnik i kreator kompanija Microsoft. Naime, pokazalo se da nije jednostavno prenositi programski kod i izvršavati programe između različitih verzija Windows operativnog sistema, ponekad i na istoj verziju u zavisnost od dodataka koji jesu ili nisu instalirani. Ovo je počelo da predstavlja ogroman problem za programere i morao se tražiti novi sistematski pristup da bi ovo riješilo. Dodatni izazov predstavljao je zahtjev programera da mogu da koriste kod napisan na različitim programskim jezicima kako bi se bolje koristila programerska zaostavština i ubrzao razvoj softvera. Stoga je Microsoft razvio softverski okvir (na engleskom framework) koji se naziva .NET (čita se dot-net) a koji u suštini nije ništa drugo nego sistem klasa nazvan FCL (Framework Class Library) uz obezbjeđenje jezičke interoperabilnosti. Ova programska-jezička interoperabilnost je obezbijeđena preko jezika posrednika (Intermediate Language). Programer u suštini ne bi trebalo da brine o načinu kako se operacije interoperabilnosti vrše i kako do toga dolazi i ima jedan razlog više da se suštinski prepusti pisanju programa sa smanjenim razmišljanjem o infrastrukturi koja je ovome namjenjena. Dakle, .NET okvir je samo još jedan infrastrukturni softverski dodatak ili sloj koji omogućuje pisanje koda bez potrebe da se previše bavimo načinom organizacije samog računarskog sistema.

Prije nego objasnimo neke od osnovnih karakteristika programskog jezika C# na isti način kako smo već obradili programski jezik Java vrijedi se pozabaviti očiglednom činjenicom da je C# manje popularan u ovom trenutku od Java-e. Po našem mišljenju razlog je koncentracija C# na probleme koji su se pojavili u Microsoft Windows okruženju i mali prodor koji je operativni sistem Windows napravio na mobilnim platformama. Pored navedenog postoji i određeni otpor prema Windows-u i Microsoft proizvodima zbog njihove visoke komercijalizacije. Međutim, ovo nipošto ne znači da je C# loš programski jezik i da ga treba preskakati u svom programerskoj edukaciji. C# je veoma pogodan da se koristi u Visual Studio programskom okruženju koje se danas može u mnogim varijantama koristiti besplatno na svom računaru ili na mrežnim resursima. Riječ je o odličnom, stabilnom programskom okruženju sa kojim se bez obzira da li ga koristite ili ne treba upoznati svaki programer. Drugi razlog zbog kojega je potrebno da se programeri familijarizuju sa C# i .NET okvirom je činjenica da je Microsoft što direktno a što ohrabrujući druge napravio ogroman napor da se velika programerska zaostavština sa prevede u obliku biblioteka i dodataka za C#. Neko će reći da ovo postoji i za druge programske jezike i kompajlere. Djelimično se možemo složiti sa ovom konstatacijom, međutim količina softvera koja je prevedena za C# i .NET kao i ujednačeni kvalitet istoga nije karakterističan za druge softverske proizvode. Opšte je poznata činjenica da u mnogim kompajlerima i za programske jezike postoje biblioteke i rješenja za pojedine namjene na nivou remek djela ali da za druge stvari ne postoje rješenja ili je njihov kvalitet loš ili sporan. Posebno ovo važi kod sistema otvorenog koda kod kojega programerska zajednica doprinosi narastanju sistema. Kod C# i .NET okvira dostupni dodaci, bilioteke, funkcije, klase su sveobuhvatni (nema gotovo oblasti koja nije pokrivena) i ujednačenog (industrijskog) kvaliteta.

Pređimo sada na osnovne karakteristike ovog programskog jezika. Baš kao i kod programskog jezika Java i ovdje ne postoji kod van klase uključujući i glavni program **Main**() mora biti implementiran unutar klase. C# dozvoljava rad sa pokazivačima ali samo u dijelu koda koji je označen kao nebezbjedan. Slično kao i kod Java-e u programskom jeziku C# nema potrebe da se vrši dealokacija memorije jer to obavlja program za prikupljanje otpada (Garbage Collector). Dakle, u ova dva programska jezika, osim u rijetkim slučajevima, nije potrebno vršiti dinamičku dealokaciju memorije već će ona biti urađena bez kontrole programera i korisnika programa. Riječ je o odličnom konceptu i mana C++ je što ovaj concept do trenutka pisanja ove knjige nije u njemu implementiran. C# je kao i Java znatno strožije tipiziran programski jezik ne dozvoljavajući neke implicitne konverzije na koje smo navikli. Svaka promjenljiva u programskom jeziku C# koja nije inicijalizovana ne može se koristiti stoga je po pravilu deklaracija promjenljive praćena njenom inicijalizacijom:

```
int i=1;
```

Vidimo da su ključne riječi uključujući nazive tipova podataka slične kao u C-u. Ovo važi i za operatore kao i za naredbe za kontrolu toka programa. U C# možemo deklarisati nizove na isti način kao u C-u ali su ovo implicitno klasni tipovi podataka
koji pripadaju klasi **System.Array**. U programskom jeziku C# jedan broj podataka je vrijednosti (kao u C-u) dok su složeniji – klasni tipovi podataka referencni, odnosno ukazuju na vrijednost u memoriji. Kod prvih primjena operatora dodjele vrijednosti je uobičajena a kod drugoga u slučaju da pridružimo jednoj promjenljivoj drugi objekat istoga tipa ne stvara se novi memorijski objekat već ove dvije promjenljive ukazuju na istu memorijsku lokaciju (druga promjenljiva je referenca – drugo ime za isti memorijski objekat). Sada možemo na jednom jednostavnom primjeru ilustrovati neke razlike vezane za sintaksu C#, naredbe za kontrolu toka programa i rad sa nizovima u odnosu na C/C++:

```
class ForeachPrimjer {
        static void Main(string[] args) {
                int [] a = new int[] { 0, 1, 2, 3, 5, 8, 13 };
                        foreach (int i in a) {
                                System.Console.WriteLine(i);
                        }
                }
```

}

Ovdje vidimo da smo unutar klase morali da deklarišemo glavni program **Main** sa pripadajućim argumentom. Zatim deklarišemo niz i uz alokaciju memorije za njega vršimo inicijalizaciju. Naredba **foreach** danas postoji u gotovo svim programskim jezicima i podrazumijeva baš ono što piše, iteriranje za svaki element koji se nalazi u nekoj kolekciji (u ovom slučaju nizu **a**). Potom se vrši štampanje za što se u ovom slučaju koristi konzolna metoda **WriteLine**. Već smo rekli da **a** nije prosti niz već je instanca klase pa to znači da između ostalog postoje metodi koji su asocirani sa ovim tipom podataka kao na primjer **a.Length()** koji nam daje dužinu niza.

U prethodnom primjeru vidjeli smo deklaraciju klase koja je slična kao u C++-u. Postoje privatni, javni i zaštićeni metodi. Međutim, pored modifikatora **private**, **public** i **protected** postoje još **static** (za čitavu klase a ne za objekte te klase), **virtual** (ovdje je to metod koji se može redefinisati u izvedenoj klasi), **abstract** (ono što predstavlja čistu virtuelnu funkciju u C++), **override** (naglašava nadjačavanje nasljeđenog virtuelnog ili čistog virtuelnog metoda), **sealed** (naglašava nadjačavanje nasljeđenog virtuelnog ili čistog virtuelnog metoda ali i činjenicu da se ovaj metod ne može dalje zasjenjivati u izvedenim klasama), **new** (naglašava da je ovo nova klasa koja nema veze sa onom koja je eventualno definisana u roditeljskoj klasi), **internal** (metod kojem se može pristupiti samo iz posmatrane programske cjeline - sklopa), **extern** (metod koji je implementiran eksterno odnosno u drugom programskom jeziku). Kako i kod Java-e i ovdje se realizacija metoda obavlja u samoj klasi jer ne postoji kod van klase. Interesantna mogućnost u programskom jeziku C# je da funkcije imaju više od jednog rezultata. Naime, kao i u C/C++ i ovdje funkcija vraća direktno putem **return**-a samo jedan rezultat. Međutim,. U C/C++ smo više rezultata funkcije dobijali tako što smo podatak prosljeđivali funkciji putem reference ili pokazivača. Umjesto toga podatak koji treba da predstavlja dodatni rezultat funkcije u listi argumenata funkcije ima modifikator **out** što naglašava da promjene toga podatka imaju efekat na stvarni argument u pozivajućem metodu.

Nasljeđivanje je u C# slično kao kod Java-e odnosno dozvoljeno je samo prosto nasljeđivanje kada jedna klasa nasljeđuje tačno jednu odnosnu klasu:

```
class IzvedenaKlasa : OsnovnaKlasea{
        // podaci i funkcije izvedene klase
}
```

Ovdje je uočljivo da ne postoji privatno i javno nasljeđivanje već da je sva nasljeđivanje istoga tipa. Posmatrajmo jedan primjer nasljeđivanja:

```
class OsnovnaKlasa{
        public int a;
        public double f(){
                return a;
        }
}
class IzvedenaKlasa : OsnovnkaKlasa{
        public double g(){
        return base.f()*0.5;
        }
}
```

U ovom primjeru vidimo da funkcija **g** iz izvedene klase poziva funkciju **f** iz osnovne klase što mora uraditi navođem naziva osnovne klase. Umjesto toga koristi se ključna riječ **base** koja predstavlja osnovnu kako bi se obavila ista funkcionalnost. Dakle u C# pozivanje metoda iz same klase se obavlja navođenjem po imenu dok je za pozivanje metoda iz drugih klasa (u ovom slučaju i iz osnovne klase) potrebno navesti ime klase u kojoj se navedeni metod nalazi. Naravno, u pitanju je prosta ilustracija metoda koji se mogao obaviti jednostavnije sa **return a*0.5;**.

U C# postoje interfejsi koji predstavljaju ugovor koje neka klasa garantuje a istovremeno predstavljaju i mogućnost da se prevaziđu nedostaci u smislu višestrukog nasljeđivanje te ujedno se donekle pojednostavljuje i unificira rad sa metodima koji potiču iz različitih klasa pa čak i različitih programskih jezika ako za time postoji potreba. Posmatrajmo deklaraciju jednog interfejsa:

```
public interface ID {
        void D ();
}
class NekaKlasa : ID {
        public void D() {
                //implementacija metoda
        }
}
```

Klasa može naslijediti više interfejsa čime se donekle prevazilazi izostanak višestrukog nasljeđivanja. Sintaksa je veoma slična prethodnoj odnosno sintaksi višestrukog nasljeđivanja u C++:

```
class Klasa : Interfejs1, Interfejs2 {
// tijelo klase
}
```

Interesantan koncept u C#-u su svojstva. Svojstva su u mnogo čemu slična promjenljivima samo što je stvarna realizacija svojstva opisana realizacijom u kojoj se pojavljuju ključne riječi **get** (daje vrijednost svojstva) i **set** (postavlja vrijednost svojstva). Svojstvo može da ima obije ove ključne riječi odnosno može da ima samo jednu. Svojstvo ima sljedeću sintaksu:

```
tip_rezultata ime_svojstva {
set {
// tijelo set-a
}
get { // tijelo get naredbe
}
}
```

Polje **set** mora imati jednu povratnu vrijednost koja je istog tipa kao što je podatak član na koji su odnosi, dok **get** ima podrazumijevanu vrijednost **value** istog tipa kao što je svojstvo na koje se odnosi. Na ovaj način se može izbjegavati suštinski pozivanje inspektorskih i mutatorskih funkcija za pristup podacima članovima ali primjena **set** i **get** nije na ovo ograničena. Svojstva se mogu deklarisati da su apstraktna (**abstract**) i virtuelna (**virtual**) baš kao i metodi. Ilustrujmo sintaksu vezanu za svojstva na jednom primjeru:

```
class Program {
public static void Main() {
        BaseClass obj = new BaseClass();
        Obj.Starost = 96;
        Console.WriteLine("Ima "+obj.Starost+" godina");
        }
}
class BaseClass {
        public int Starost{
        get{
                return godine;}
        set{
                godine = value;}
```

```
        }
}
```

U okviru klase **BaseClass** deklarisana je privatna clelobrojna promjenljiva **godine**. Pomenutoj promjenljivoj može se pristupiti samo iz klase u kojoj je deklarisana. Međutim, iskoristili smo svojstvo **Starost** za pristup van klase. Ukoliko svojstvu u programu dodijelimo vrijednost aktivna je naredba **set**, a kad od njega zahtijevamo da vrati vrijednost aktivna je naredba **get**.

Programski jezik C# dozvoljava preklapanje konstruktora što se može demonstrirati sljedećim primjerom:

```
class Neka_klasa {
private int value;
public Neka_klasa() {
        value = 0; }
public Neka_klasa(int value) {
        this.value = value; }
}
```

Ako se u nekoj klasi deklariše objekat ove klase podrazumijeva se da poziv podrazumijevanog konstruktora, dok ako se inicijalizuje kao:

```
Neka_klasa nk(100);
```

biva pozvan odgovarajući konstruktor sa jednim argumentum.

Kod konstruktorske funkcije postoje svi koncepti koje smo upoznali u C++, uključujući inicijalizaciju podobjektom osnovne klase objekata izvedene klase. Što se tiče destrukcije programski jezik C# posjeduje sakupljač otpada koji povremeno pobriše objekte koji se više ne koriste u programu. Ako nam je potrebno da dealociramo objekat iz memorije u momentu kada mi to hoćemo možemo da deklarišemo sopstveni destruktor putem koga sami kontrolišemo dalokaciju promjenljivih. Ipak to se u programskom jeziku C# radi veoma rijetko zbog postojanja sistema za sakupljanje otpada.

Još jedna karakteristika po kojoj je C# sličniji programskom jeziku C++ nego Java je preklapanje operatora koje je u C# dozvoljeno. Mogu se preklopiti sljedeći opeatori: aritmetički binarni (+, *, /, -, %) i unarni (+, -, ++, --), operator na bitovima i to binarni (&, |, ^, <<, >>) i unarni (!, ~, **true**, **false**) i logičkih operacija (==, !=, <=, <, >). Posmatrajmo kako bi se u C# realizovao dobro poznat problem klase kompleksnih brojeva sa nekoliko konstruktorskih funkcija (obratite pažnju na primjenu operator – objekta **this**):

```
class Complex{
        private double real, imag;
        public Complex(double real, double imag) {
                this.real = real;
                this.imag = imag; }
        public Complex() : this(0.0, 0.0) { }
        public Complex(Complex comObj) {
                this.real = comObj.real;
                this.imag = comObj.imag; }
        public static Complex operator+ (Complex x, Complex y) {
                Complex rez = new Complex(x);
                rez.real += y.real;
                rez.imag += y.imag;
                return rez; }
```

```
        public static Complex operator+(Complex x, double y) {
                Complex rez = new Complex(x);
                rez.real+=y;
                return rez; }
}
```

Indekseri su u C# specijalan tip preklapanja operatora **[]** namjenjen za pristupanje podacima članovima klase kao nizovima:

```
struct Point {
public double x,y,z;
        public double this [int i] {
                get {
                switch(i) {
                        case 0: return x;
                        case 1: return y;
                        case 2: return z;
                        default:
                        throw new IndexOutOfRangeExeption ("van granica "); }
                }
                set {
                        case 0: x = value;
                        case 1: y = value;
                        case 2: z = value;
                        default:
                        throw new IndexOutOfRangeExeption ("van granica "); }
        }
}
```

Ključna riječ/modifikator **readonly** naglašava da je podatak namjenjen samo za čitanje. Programski jezik C# posjeduje strukture (ključna riječ **struct**) koje su po svemu slične klasama sa svega nekoliko razlika: klase su memorijski zahtjevnije i čuvaju se na hipu, dok se strukture čuvaju na steku; klase su referencni tipovi podataka, dok su strukture vrijednosni tipovi; nemamo mogućnost nasleđivanja kod struktura; kod struktura se ne može mijenjati bezparametarski konstruktor i preporuka je gdje god je moguće, u cilju povećanja performansi programa, koristiti strukture.

Konverzija podataka u programskom jeziku C# je moguća na sličan kao programskom jeziku C++ sa time što C# prepoznaje mogućnost da se na različiti način implementiraju konvertori koji rade kao eksplicitni (**explicit**) odnosno implicitni (**implicit**). Zaglavlja ovih metoda su:

```
public static explicit operator @(Tip v) { //tijelo funkcije
}
public static implicit operator @(Tip v) { //tijelo funkcije
}
```

Programski jezik C#, slično kao i programski jezik Java, posjeduje predefinisanu hijerarhiju klasa za obradu izuzetaka koje u sve izvedene iz klase **Object.Exception**. Izuzeci se bacaju putem **throw** iskaza:

```
throw new konstruktor_klase_ciji_je_tip_izuzetak;
```

Izuzeci se mogu obrađivati kao u C++ putem **catch** blokova:

```
catch (tip e){//obrada izuzetka
}
```

dok se **catch** blok bez argumenata koristi da obradi svaki tip izuzetka:

```
catch{  //obrada izuzetka
}
```

Ovaj blok se koristi za obradu nepredviđenih izuzetaka i stavlja se kao i sličan blok **catch(...)** u C++-u na kraju liste blokova za obradu izuzetaka. U **finally** bloku čiste se svi resursi i preuzimaju akcije koje treba izvršiti nakon blokova **try** i **catch**. Ovaj blok se izvršava bez obzira da li se desio izuzetak, ili ne. I ovdje kao u programskom jeziku C++ dozvoljeno je ugnježdavanje **try** blokova.

1.5.3 Programski jezik Python

Programski jezik Python se dosta razlikuje u odnosu na prethodno opisane koji dijele mnoštvo zajedničkih karakteristika a prije svega sintaksu koja vodi porijeklo iz programskog jezika C. Ovaj programski jezik je kreirao Guido van Rossum krajem 1980-tih godina. Ovaj programski jezik je prvobitno namjenjene za pisanje skript fajlova pod Unix operativnim sistemom. Naziv je dobio po neobičnoj britanskoj humorističkoj seriji „Leteći cirkus Monti Pajtona“ čime se na neki naglašava da je programski jezik Python namjenjen za „zabavno“ programiranje koje podržava veći broj programerskih paradigmi.

Bez želje da detaljnije ulazimo u sintaksu i mogućnosti ovog programskog jezika navodimo samo nekoliko karakteristika ovog programskog jezika. Vrijedi pomenuti nekoliko elemenata sintakse ovog programskog jezika koji su čest izvor fascinacije ali teško da su od suštinskog značaja, osim što ponekad omogućavaju kompaktnije pisanje koda. Naime, u Python-u blokovi naredbi nisu odvojeni vitičastim zagradama kao u programskim jezicima koje smo do sada učili niti na primjer sa ključnom riječju **end** kao što je to u MATLAB-u. Naime, blokovi naredbi u Pythonu su naglašeni uvlačenjem programskih linija. Dok je ovo opciono i preporučljuivo da se radi u drugim programskim jezicima a poštovano je i u ovoj knjizi u Python-u je to obaveza.

Selekcija u Python-u se obavlja putem naredbe **if**, na kraju uslovnog iskaza treba da stoji dvotačka a blok naredbi je zatim uvučen u odnosu na poziciju **if**-a. U slučaju da ima još uslovnih iskaza ovdje postoji razlika u odnosnu na druge učene programske jezike jer se koristi naredba **elif** (ponovo sa dvotačkom na kraju) dok je za blok koji se izvršava kada nijedna od uslovnih naredbi nije ispunjena iskorišćena ključna riječ **else**:

```
if uslov1:
        blok1
        blok1
elif uslov2:
        blok2
        blok2
elif uslov3:
        blok3
else:
        blok4
```

Obično se uvlačenje vrši za četiti bjeline međutim preporučuje se editor koji je prilagođen Python-u i koji sam podešava način i veličinu ovog uvlačenja. U slučaju ugnježdivanja drugog uslova unutar prvoga pomjeranje bloka se vrši za dodatna (četiri) mjesta.

Brojački ciklus u Python-u se obavlja naredbom **for** koja već može da dovede do niza zabuna:

```
for x in range(0,3):
        print x
else:
        print 'Kraj petlje'
```

Dok je **x** u granicama (uočite ključnu riječ **in**) od **0** do **3** (naredba **range** liči na formiranje vektora putem operatora dvotačka u MATLAB-u) nalazimo se u ciklusu u kojm se vrši štampanje brojeva kada završimo u **else** dijelu štampamo obavještenje. Napominjemo da **else** blok koji se izvršava samo jednom na kraju ciklusa nije obavezan već je naveden da bi ilustrovao ovu mogućnost u Python-u. Ono što može da promakne je objektna orjentisanost kod Python-a jer naredba **print** u ovom slučaju štampa dva različita objekta, u jednom slučaju cijeli broj a u drugom string bez posebnog podešavanja. Kada smo kod objektne orjentacije u Python-u, koja se često zanemaruje u poplavi drugih noviteta i vizuelno različitih programskih koncepata, pogledajmo sljedeći primjer:

```
skup = ['zdravo', 11, 'A']
for x in skup:
        print x
```

Dakle, činjenica da je **skup** kolekcija podataka različitog tipa ne predstavlja poteškoću da predmetni kod funkcioniše te da **for** blok radi sa podacima različitog tipa (u ovom slučaju sa stringom, cijelim brojem i karakterom). Moramo znati da su svi ugrađeni tipovi podataka u Python-u objekti i da se ponekad nad njima mogu vršiti naredbe na način kako se to radi sa objektima klasnog tipa u C++:

```
>>> x=101
>>> x.bit_length()
7
```

U ovom primjeru **x** je cijeli broj kojem je dodijeljenja neka vrijednost a nad ovim objektom pozvali smo funkciju koja određuje koliko je dužina ovog broja u binarnoj reprezentaciji ($\mathbf{1100101_2}$) odnosno **7** bita. Naredbu smo pozvali u Python-ovom komandnom interpreteru pa odatle se pojavljuje komandni prompt koji ovdje ima oblik tri znaka veće (**>>>**). Naravno većinu naredbi u Python-u nećemo pozivati (niti ćemo moći) za ugrađene tipove na ovaj način već ćemo to raditi kako je uobičajeno i u drugom programskim jezicima, npr. **abs(x)**. Python je veoma bogat kolekcijama koje mogu imati istorodne ali i raznorodne tipove podataka. Indeksiranje može biti veoma raznoliko, moćno i ponekad zbunjujuće. Pogledajmo sljedeće primjere:

```
>>> R=[1, 2, 6, 5, 8, 11]
>>> R[0]
1
>>> R[0:3]
[1, 2, 6]
>>> R[-1]
11
>>> R[-2]
8
```

Prva činjenica koja nije iznenađujuća po sebi je da Python ideksira od nule. Drugo što ovdje uočavamo je da možemo primjenjivati operator **:** radi višestrukog indeksiranja slično kako je takav operator primjenjivan u MATLAB-u. Međutim, ovdje **R[0:3]** ne obuhvata element sa indeksom **3** što je razlika u odnosu na MATLAB a to dalje znači da se primjena ovakvog operatora mora vršiti tek nakon što se malo detaljnije upoznate sa Python-om i njegovom sintasom. Konačno, i vjerovatno kao najveće iznenađenje, Python dozvoljava negativne indekse tako da indeks **-1** korespondira posljednjem elementu u nizu a **-2** pretposljenjem itd. Sve ovo nam je bilo potrebno da prikažemo

jedan veoma jednostavan primjer koji ukazuje na objektno-orjentisanost kod Pythona. Pogledajmo sljedeću prostu naredbu:

```
>>> "kuca poso".split()[-1].upper()
'POSO'
```

Primjer pokazuje do kojega je nivoa objektna orjentacija dostignuta u Python-u po pravilu koje se može formulisati kao „sve je objekat“. Prvo imamo string ne kao promjenljivu već kao konstantu koja se sastoji od dvije riječi. Naredbu **split()** primjenjujemo direktno na ovu konstnatu što nije uobičajeno u drugim programskim jezicima, odnosno tretiramo string kao objekat. Predmetna naredba dijeli string u riječi (rezultat su dvije riječi). Zatim, ponovo bez potrebe da pridružimo međurezultat nekoj promjenljivoj tražimo posljednji string (indeks **[-1]**) a zatim nad tim objektom primjenjujemo naredbu **upper** koja mala slova pretvara u velika. Svi ovi elementi ukazuju na savršenstvo u objektnoj orjentaciji Python-a koja nije karakteristična ni za hibridne jezike kakav je C++ koji su nastali nadograđujući OO koncepte nad postojećim programskim jezicima, ali ni kod jezika koji su od početka OO dizajnirani.

Prije nego uvedemo koncept klase od kojega smo zapravo u C++-u i počeli da pričamo o objektnoj orjentaciji daćemo nekoliko osnovnih informacija o funkcijama u Python-u. Funkcije se u Python-u uvode sa ključnom riječju **def** praćenom nazivom funkcije, a u zagradama se nalazi lista argumenata funkcije praćena dvotačkom. Kod liste argumenata ne daju se njihovi tipovi niti se igdje mora navoditi tip rezultata koji funkcija vraća. Pogledajmo prostu funkciju koja štampa Fibonačijeve brojeve koji su manji od *n*:

```
def fib(n):
    a, b = 0, 1
    while a < n:
        print (a)
        a, b = b, a+b
```

Funkcija je vrijedna tumačenja pošto u jednoj liniji možemo obaviti dvije dodjele čime se kod može dodatno smanjiti (Python je programski jezik poznat po izuzetnoj kompaktnosti koda, mada kao i u svim sličnim situacijama, programeri ponekad pretjeruju na uštrb jasnosti a ponekad i ispravnosti programa). Vrijedi reći da funkcije u Python-u nude bogatstvo koncepata i mogućnosti od podrazumijevanih argumenata, preklapanje i drugih koncepata koji su u učenim programskim jezicima djelovali veoma neobično dočim se u Python-u veoma često koriste a pored njih postoji i mnogo drugih egzotičnih opcija. Napomenimo da funkcija koja je napisana u jednom Python fajlu (fajlovi sa ekstenzijom .py) se veoma lako može koristiti u drugim fajlovima njenim importovanjem. Postoje i druge opcije za preuzimanje funkcija i formiranje biblioteka ali o njima ne možemo detaljnije diskutovati.

Sada možemo da damo nekoliko osnovnih informacija vezanih za klase u Pythonu. Bez želje da se dublje pozabavimo ovom problematikom daćemo samo dva primjera. Međutim, vrijedi istaći da mnogi smatraju da je potreba za klasama u Python-u manja nego u druim programskim jezicima pošto su već implementirani napredni tipovi podataka kakvi su liste, torke (tuples) rječnici, skupovi, itd. koji su već objektno-orjentisani i veoma moćni. Međutim, OO programski jezik nemoguće je zamisliti bez mogućnosti da programer definiše klasu. U prvom primjeru posmatrajmo jednostavnu klasu osoba;

```
class Osoba:
    def __init__(self,ime,starost,plata=0,posao=None):
        self.ime=ime
        self.starost=starost
        self.plata=plata
        self.posao=posao
```

```
        def Prezime(self):
            return self.ime.split()[-1]
        def Povisica(self,procenat):
            self.plata*=(1+procenat)
if __name__=='__main__':
   Marko=Osoba('Marko Markovic',42,400,'magacioner')
   Nikola=Osoba('Nikola Nikolic',45,550,'prodavac')
   print(Marko.ime,Nikola.ime)
   print(Marko.Prezime())
   Nikola.Povisica(0.1)
   print(Nikola.plata)
```

U primjeru deklarišemo klasu **Osoba** sa četiri podatka člana. Funkcija **__init__** predstavlja vrstu konstruktora čija je namjena da inicijalizuje predmetnu klasu. Vidimo da ova funkcija ima i dva podrazumijevana podatka člana (pretpostavljamo da osoba ne mora da ima posao niti platu). Promjenljiva **self** u Python-u glumi pokazivač **this** iz C++-a i ovdje se mora navoditi jer kada bi se izostavila zbog izostavljanja deklaracija u Python-u ne bi se moglo protumačiti da li je to podatak član ili je u pitanju nova promjenljiva koja se koristi ad-hok. Vidimo i dvije funkcije koje definišemo unutar klase od kojih jedna daje prezime a druga daje povišicu osobi. Testiranje programa smo obavili u istom fajlu što nije obaveza već se može obaviti i na drugom mjestu (Python je po tome fleksibilniji od Java i C#). Dio koda koji slijedi iza **if __name__=='__main__':** se izvršava samo ako se fajl sa definicijom klase pozove kao glavna skripta dok ako se koristi samo definicija klase ovaj dio koda se nebi izvršavao.

Zbog skučenosti prostora ali i želje da se samo ovlaš upoznamo sa objektno-orjentisanim konceptima iz Python-a posmatrajmo još samo koncept nasljeđivanja:

```
from osoba import Osoba
class Menadzer (Osoba):
        def Povisica(self,procenat,bonus=0.1):
            self.plata*=(1.0+procenat+bonus)
```

Prvo što smo demonstrirali je naredba **import** koju u Python-u često koristimo da bi u nekom fajlu omogućili dostupnost drugih djelova koda, biblioteka, funkcija, klasa ,itd. Nasljeđivanje se obavlja prostim navođenjem klase iz koje nasljeđujemo osobine, u ovom slučaju klasa **Menadzer** nasljeđuje sve osobine koje ima klasa **Osoba** uz samo jednu uočljivu izmjenu a to je da menadžerim imaju plate koje se obračunavaju na poseban način (sa dodatnim bonusom). Nasljeđivanje u Pythonu može da bude višestruko čime ponovo pokazuje svoju fleksibilnost.

Nadamo se da ste na ovaj način upoznati sa nekim osnovnim konceptima programskog jezika Python. Premda kompaktna sintaksa i mnoštvo modula koji omogućavaju lak pristup mnogim konceptima često su u prvom planu, bitno je istaći da je Python primarno OO programski jezik u kojem je sve objekat (na primjer čak i funkcije su objekti) što otvara veoma često ogroman prostor za rad. Toplo preporučujemo (samo)edukaciju u ovom sjajnom programskom jeziku koji će direktno, ili preko jezika i koncepata koji ga kloniraju, veoma dugo uticati na svijet programiranja.

1.5.4 Klase u MATLAB-u

Svi koji su radili u MATLAB-u zasigurno su veoma rano uočili njegovu objektnu orjentaciju odnosno fleksibilnost u radu. Ista funkcija se izvršavala različito u zavisnosti od broja i tipa argumenata sa kojima se funkcija pozivala. Premda objektna orjentacija nije bila vrlo vjerovatno ideja u prvim verzijama MATLAB-a interesantno je kako je težnja za fleksibilnošću programskog sistema

dovela do istih rješenja kao što je objektna orjentacija. Međutim, ovo ne bi trebalo da nas čudi kada imamo na umu istorijat i razloge razvoja ovog koncepta.

Ono što korisnici rijetko koriste u MATLAB-u je mogućnost da se naprave klase. Prije nego objasnimo osnovnu, veoma rudimentarnu sintaksu rada sa klasama vrijedi reći da su svi podaci u MATLAB-u klasnog tipa. Naime, ako je neki podatak cijeli broj MATLAB mu dodjeljuje neku cjeloborjnu klasu, npr. **int16** tako da su operacije nad ovim tipom podataka prilagođene cijelim brojevima. Ovo se dalje koristi prilikom pozivanja određenih metoda ili operacija (npr. kod digitalne slike prije nego je pretvorimo u neki format nije dozvoljeno sabiranje). Ovdje ćemo ukratko objasniti osnovu sintaksu klase u MATLAB-u a zatim i osnove klasinog interfejsa. Definicija klase u MATLAB-u klase ima sljedeći oblik:

```
classdef mojaklasa
        properties %osobine
        %redom osobine klase
        end     %kraj definicije atrubuta-osobina klase
        methods
        %spisak i definicije metoda klase
        end     %kraj metoda – funkcija klase
end     %kraj čitave definicije klase
```

Ovdje su ključne riječi **classdef**, **properties**, **methods** i **end** dok su **%** oznake za komentar koji traje do kraja reda. Ovakva definicija klase se smiješta u fajl koji nosi naziv **mojaklasa.m** koji se mora nalaziti u putanji foldera po kojima se vrši pretraga za funkcijama i klasama. Unutar klase se nalaze osobine (atributi ili podaci članovi) **properties** koji se deklarišu navođenjem imena osobina praćenim karakterom **;** na primjer:

```
a;
b;
c;
```

Neke od promjenljivih mogu imati podrazumijevane vrijednost na primjer:

```
d=2;
```

Jedan ili nekoliko metoda koji se nalaze u sekciji **methods** mogu da nose isto ime kao klasa:

```
function obj=mojaklasa(a,b,c,d)
%definicija objekta klase
end
```

Ovakav metod predstavlja konstruktor koji definiše što će se dešavati sa objektom prilikom njegovog nastanka (inicijalizacije). Na primjer mogući konstruktor za klasu koja ima četiri podatka člana može da bude:

```
function obj=mojaklasa(a,b,c,d)
if(nargin > 0)
obj.a = a;
obj.b = b;
obj.c  = c;
obj.d = d;
end
```

U ovom slučaju objekat klase se inicijalizuje na vrijednosti podataka koje su objektu proslijeđene kao argumenti funkcije. Zgodna osobina MATLAB-a je da funkcije mogu imati

proizvoljan broj argumenata te da se sa **nargin**, **nargout**, **varargin** i **varargout** (ovi argumenti daju koliko ima ulaznih i izlaznih argumenata –rezultata funkcije) može manipulisati sa ponašanjem datih funkcija. Ovim se izbjegava u najvećoj mjeri potreba da se preklapaju funkcije što je osobina C++ i drugih sličnih OO programskih jezika (jezika koji barataju sa klasama). Drugi metodi koji su inherentni klasama mogu da rade različite obrade podataka klasnog tipa. Tako na primjer metod implementiran u tijelo klase:

```
function obj = fudbal(obj,d)
if d>obj.b
obj.a=obj.a+1;
end
```

U predmetnoj funkciji u zavisnosti od vrijednosti argumenta može da se promjene promljenljive članice objekta za koji se funkcija poziva. Novi objekat predmetne klase poziva se pozivom konstruktora:

```
x=mojaklasa(2,3,4,5);
```

dok se funkcija **fudbal** poziva putem operatora tačka kao **x.fudbal(30);**. Prethodna definicija osobina klase dopušta potpuni pristup podacima članovima tako je dozvoljeno pročitati sadržaj podataka članova ili ih izmjeniti:

```
G=x.a;
x.b=20;
```

Međutim, kao u svim OO metodima postoji mogućnost kontrole pristupu podacima članovima klase (tzv. prava pristupa). Ovo se postiže tako što se eventualno može uvesti više **properties** sekcija u kojima se mogu definisati različita prava pristupa za pojedine atribute (osobine):

```
  properties(GetAccess = 'public', SetAccess = 'private')
     % osobine koja se mogu spolja čitati dok se ne mogu mijenjati
  end
  properties(GetAccess = 'private', SetAccess = 'private')
     % osobine koje se ne vide spolja niti im se mogu mijenjati vrijednosti
  end
```

Primjer jednog privatnog metoda je:

```
methods(Access = private)
   function S = FS(obj)
     sec = obj.min*60 + obj.sat*60*60 + obj.dan*24*60*60;
   end
   function TF = isValid(obj)
      TF = obj.min >= 0 && obj.min <= 60;
   end
 end
```

Postoje kao u gotovo svi poznatijim objektno orjentisanim programskim jezicima tri nivoa vidljivosti odnosno prava pristupa: **'private'**, **'public'** i **'protected'**. U MATLAB-u se razlikuju prava vezana za čitanje osobina (postavljaju se sa **GetAccess**) i prava vezana za postavljanje vrijednosti osobina (postavljaju se sa **SetAccess**). U MATLAB-u postoje i neki drugi atributi osobina koji ponekad koriste osim eventualno **'Constant'**, **'Hidden'** i **'Static'**. Kao konstante osobine se označavaju one koje se ne mogu promjeniti kao što je dato u narednom primjeru:

```
  properties(Constant = true)
```

```
    DANA =  365;
    MJESECI = 12;
    GODINA  = 52;
  end
```

Moguće je definisati sakrivene osobine koje se ne vide kada se prikazuje objekat. Sve ostale osobine za objekte neke klase se prikazuju naredbom **properties** koja inače prikazuje osobine-atribute objekta odnosno klase:

```
properties(x)
```

Metodi su funkcije definisane u klasi koje rade sa podacima članovima objekata te klase. Pretpostavimo da imamo metod **dodajA()** koja treba da uveća osobinu **a** za argument **p** navedene klase:

```
function obj=dodajA(obj,p)
        obj.a=p+obj.a;
end
```

Ako je **x** objekat klase onda se predmetni metod može pozvati kao **x1=x.dodajA(3);** odnosno **x1=dodajA(x,3)**. Prvi način pozivanja je prirodniji za ljude koji su familijarni sa C++ i Java-om a treba inače znati da u oba slučaja metod ima dva argumenta (**nargin=2**) od kojih je u prvom slučaju prvi argument objekat **x** za koji je funkcija pozvana. Jedna neobična stvar kod MATLAB-a je činjenica da je neophodno da se objekat ažurira jer kod MATLAB-a ne postoji poziv po reference već samo poziv po vrijednosti pa se objekat ne može ažurirati na alternativan način. Dakle, ako se objekat ne vrati kao rezultat funkcije onda neće ni biti ažuriran.

MATLAB posjeduje i statičke metode koji su prodruženi čitavoj klasi a ne pojedinom objektu te klase. Ovakvi metodi se deklarišu na sljedeći način:

```
methods(Static=true)
        function nazivfunkcije(argumenti)
                %tijelo funkcije
        end
end
```

Ovakvi metodi se pozivaju tako što se navede naziv klase odvojen tačkom pa naziv metoda:

```
klasa.nazivfunkcije()
```

Metodi mogu biti i skriveni što se najavljuje kao prethodno sa **methods(Hidden=true)**. Smisao ove opcije je da sakrije metod od mogućnosti da bude izlistan nekom od metoda za prikazivanje spiska metoda kao što su **methods()** ili **methodsview()**.

Klase mogu da pozivaju sve funkcije nečlanice u MATLAB-u (eksterne funkcije) kako je to već do sada bilo uobičajeno. Fajl koji predstavlja klasu može da se snimi u posebnom folderu koji nosi ime kao i sama klasa sa tom razlikom da mu prethodi karakter **@**. U tom slučaju sve funkcije koje su kreirane kao funkcijski fajlovi u tom folderu, a pod uslovom da je naziv funkcije isti kao i naziv funkcijskog fajla, tretiraju se kao metodi članovi date klase sa tom razlikom da su podrazumijevano javni (ne mogu biti statički, skriveni, zaštićeni, itd). Ipak mogu se učiniti privatnim tako što se u datom folderu (folderu sa definicijom klase) doda subfolder **private** i u njega smjeste metodi koji su tada podrazumijevano privatni. Naravno niko nam ne brani da metode napišemo i u fajlu u kojem definišemo klasu ali su tada predmetni metodi vidljivi samo iz definicije klase. MATLAB-ov sistem za pozivanje metoda u potpunosti zadovoljava koncept polimorfizma. Funkciju

sa istim nazivom možemo implementirati u razlilitim klasama i ako je pozovemo sa **fun(a)** biće izvršena ona koja korespondira objektu **a** koji je argument funkcije.

MATLAB podržava i nasljeđivanje. Sintaksa je sljedećeg oblika u fajlu koji predstavja definiciju klase treba naglasiti da je klasa izvedena iz neke druge klase:

classdef izvedenaklasa < osnovnaklasa

Dakle, operator < se koristi da bi naglasio da je u pitanje izvođenje iz neke osnovne klase. Nasljeđivanje u MATLAB-u posjeduje sve elemente i izazove kakvo nasljeđivanje posjeduje i u drugim OO programskim jezicima. Na primjer MATLAB dozvoljava da izvedena klasa naslijedi metode i atribute više osnovnih klasa:

classdef IK < OK1 & OK2 & OK3

Ako nam se ne sviđa neki metod iz osnovne klase mi možemo napraviti metod u izvedenoj klasi koji nosi isto ime i koji bi se u tom slučaju pozivao za objekte izvedene klase. Ovo se u terminilogiji OO programiranja naziva zasjenjivanjem ili preklapanjem (pojmovi sličnog dejstva ali različitog porijekla). Međutim, ovo zasjenjivanje ne znači da se tom metodu ne može pristupiti iz osnovne klase (ako mu je odgovarajuće pravo pristupa) sa sljedećom sintaksom:

funkcija@osnovnaklasa(argumenti)

Postoji mogućnost da želimo da zabranimo nasljeđivanje klase ili preklapanje pojedinih njenim metoda ili osovbima. Ovo se radio postavljanjem osobine **Sealed** na **true** u definiciji osnovne klase.

```
classdef(Sealed = true) myclass          %klasa koja se ne može naslijediti
methods(Sealed = true)                   %metodi koji se ne mogu nadjačati
properties(Sealed = true)
```

Ukažimo još na mogućnost postojanja apstraktvnih metoda, odnosno metoda koji nemaju realizaciju u datoj klasi, koje smu u C++ nazivali čistim virtuelnim funkcijama. Ovakvi metodi se smiještaju u sekciju sa atributom **Abstract** kao što je:

```
methods(Abstract = true)
        function A = PrvaVirt(obj,a);
        function B= DrugaVirt (obj,b,d);
end
```

Naravno klasa koja ima ovakve metode je apstraktna (u C++ terminologiji) pa ne mogu deklarisati objekti ove klase već ista služi amo za nasljeđivanje.

Konačno, pokažimo i MATLAB-ov rudimentarni sistem za obradu izuzetka. Svima je poznato da prilikom grešaka u programu dolazi do njegovog prekida što u nekim situacijama može da bude neprijatno. Stoga je kreiran (često zanemareni) jednostavni sistem za obadu situacija kada se greške pojavljuju u MATLAB programima i funkcijama. Sintaksa koncepta je:

```
try
        naredbe1
catch
        naredbe2
end
```

Izvršavaju se **naredbe1**. U slučaju da se ni u jednoj od njih ne dogodi pogreška neće biti izvršene **naredbe2**. Ako se u nekod od naredbi u dijelu **naredbe1** dogodi greška (može se postaviti sa MATLAB naredbom **error**) izvršiće se **naredbe2**. Blokovi **try...catch...end** se mogu dalje ugnježdavati. Pogledajmo sljedeći primjer:

```
clear
x=1:10;
y=x(1:9).^2;
plot(x,y)
??? Error using ==> plot
Vectors must be the same lengths.
```

Riječ je o izuzetno jednostavnom primjeru u kojem smo pokušali da nacrtamo funkcionalnu zavisnost vektora **y** i od vektora **x**. Zbog nesaglasnosti dužina vektora došlo je do greške i prekinuto je izvršavanje programa o čemu smo dobili i odgovarajuću poruku o grešci. Međutim, sljedeći blok naredbi funkcioniše:

```
clear
x=1:10;
y=x(1:9).^2;
try
        plot(x,y)
catch
        y=[y,zeros(1,length(x)-length(y))];
        plot(x,y)
end
```

Dakle u naredbi **plot(x,y)** se dešava greška pa se izvršava **catch** dio. Uočite da recimo za razliku od C++ **catch** nema argumenata, dakle sa jednim catch blokom moramo da obradimo sve izuzetke koji se eventualno pojave. Za vjerovati je da će sistem obrade izuzetaka u narednim verzijama MATLAB-a napredovati.

1.5.5 Objective C

Programski jezici C++ i Python su suštinski proizvod napora iz više izvora, sa jedne strane sa centralizovanim komitetom za standardizuju C++ a sa druge strane ekipe koja je okupljena oko tvorca Pythona. Dakle, ne može se reći da postoji kompanija koja je tvorac jednog ili drugoga pa da direktno ili indirektno kontroliše razvoj ovih programskih jezika. To je i razlog što je sintaksa C/C++ uticala na razvoj brojnih drugih jezika i programskih sistema a u novije vrijeme to se dešava i sa Pythonom (da pomenemo ovdje samo alat za matematička izračunavanja koji se naziva SAGE). Sa druge strane MATLAB je komercijalni alat kojega kontroliše kompanija i koji je kloniran u nekim nekomercijalnim programskim sistemima iza kojih stoji zajednica koja radi sa softverima otvorenog koda koji u ovom slučaju kreiraju programske sisteme u cilju smanjivanja troškova koji za MATLAB mogu biti znatni. Tek nešto povoljnija je situacija sa Java-om odnosno C#-om. Danas posrednu kontrolu nad razvojem Java ima kompanija Oracle (prethodno Sun Microsystems) dok je iza C#-a i odnosnog .NET okvira nalazi kompanija Microsoft. Java je u svakom slučaju otvorenija (mada u posljednje vrijeme brzina izdavanja novih verzija je znatno je sporija nego kako je to do sada navikla zajednica koja programira u ovom programskom jeziku) dok je C# donekle a odnosni .NET okvir gotovo u potpunosti, vezani za izazove koji postoje u Microsoft-ovim operativnim sistemima.

Programski jezik Objective-C ima dugu istoriju razvoja u kompanijama od ranih 1980-tih godina. Posebno važan događaj je bilo preuzimanje ovog programskog jezika od strane korporacije Apple. Stoga će zasigurno budućnost ovog programskog jezika zavisiti, barem u neposrednoj

budućnosti, od komercijalnih uspjeha i-rješnja ove kompanije (operativnog sistema iOS, ali i njihovih proizvoa iPad-ova, iPhone-ova itd.) te od aplikativnih interfejsa Cocoa i Cocoa Touch.

Iz imena slijedi da je u pitanju još jedan od klonova programskog jezika C. Stoga se svi kodovi pisani u programskom jeziku C mogu izvršavati i u programskim prevodiocima za programski jezik Objective-C. Međutim, OO a i neki drugi koncepti u Objective-C nemaju takvu predistoriju već vode porijeklo iz programskog jezika Smalltalk. Posjetimo se da je koncepte iz ovog programskog jezika Bjarne Stroustrup ugradio u +C i da je tako nastao C++ ali da je prilagodio sintaksu C-u i nije je preuzeo iz Smalltalk-a.

Porijeklo Objective-C iz dva izvora suštinski znači da sintaksa se može pojaviti u tri oblika: postoje elementi koji se pojavljuju samo u obliku i na način iz preuzet iz programskog jezika C, postoje elementi koji se mogu pojaviti u obliku sintakse oba programska jezika, konačno oni za nas najvažniji OO koncepti kao što je klasa su preuzeti iz Smalltalk-a.

Bez želje da izučavamo sintaktu Objective-C (odnosno Smalltalk-a) ovdje dajemo samo rudimentarno osnovne elemente interfejs klase kao osnovnog OO koncepta u ovim programskim jezicima. Za klasu je neophodno da se naprave dva bloka od kojih jedan predstavlja interfejs klase (**@interface**) odnosno suštinski definiciju atributa i metoda dok drugi predstavlja implementaciju klase (**@implementation**). Dakle, slično se može raditi i u C++ samo se kod Objective-C-a ovi blokovi najavljuju. Uočimo korišćenje oznake **@** za blokove koji moraju biti završeni sa **@end**. Vjerovatno je jasno da je porijeklo ovakvih blokova u Smalltalk-u. Sve klase u Objective-C jeziku su izvedene iz klase **NSObject** što se najavljuje u liniji:

@interface NasaKlasa:NSObject

Moguće je dalje nasljeđivanje ali samo prosto pošto ovaj programski jezik ne podržava višestruko nasljeđivanje. Od ostalih za ovaj kratki pregled bitnih činjenica vrijedi pomenuti da deklaracija funkcija u Objective-C je prilično neobična u odnosu na druge programske jezike. Naime, u ovom programskom jeziku prilikom navođenja argumenata, argumenti se navodi pojedinačno sa tzv. povezivajućim iskazom između (*Joining Statement*). Srećom upotreba ovakvih deklaracija se brzo usvoji mada je činjenica da je praktično nije podržana u drugim programski jezicima dosta ukazuje o njenoj jasnoći. Dobra vijest za inžinjere je da je zbog utemeljenja Objective-C u programskom jeziku Ce u potpunosti zadržano prisustvo i primjenjivost pokazivača.

1.5.6 Visual Basic

Programski jezik VisualBasic je nastao početkom 90-tih godina. Dugo je smatran za jezik koji ima najbržu krivu učenja zbog dostupnosti mnoštva biblioteka i vizuelnih gradivnih programskih elemenata. Derivat je programskog jezika Basic mada sa njim ne dijeli previše sličnosti posebno u upotrebljivosti. I ovaj programski jezik je ukorjenjen u komercijalnom sektoru pošto je proizvod korporacije Microsoft i podrška je mnogim njihovim programskim paketima (*Visual Basic for Application*). Danas je podrška i programskim paketima nekih drugih firmi. Dosta je žuči proliveno vezano za razvoj ovog programskog jezika, povremene greške u verzijama ali i na nedovoljnu podršku OO programiranju. Dobar dio ovih problema je dosta umanjen razvojem .NET okvira i podrškom koju ovaj programski jezik ima ili može da pruži u tom okruženju. Ipak je činjenica da mali broj programera koji rade u Visual Basic-u koristi i kreira sopstvene klase već uglavnom koristi ono sa čime raspolaže.

Postoji i atraktivnost većine Visual Basic okruženja koja omogućavaju da se klasa kreira kroz programsku podršku koja se naziva *wizard* (čarobnjak). Međutim, ovo se pokazalo dosta nepraktično pa programeri obično sami kreiraju klase.

Klasa se obično formira tako što se u opciama za izbor tipa fajlova koji se kreira odabere opcija *class module*. Zatim se upisuju osobine (podaci članovi) kao:

```
Public x As Integer
Private z As Integer
```

Vidimo da se javnost i privatnost naglašava ključnim riječima **Public** i **Private** dok se tip podataka navodi na kraju deklaracije kao **As** praćeno nazivom tipa (ovdje započinje sa velikim slovom). Metodi se isto navode u samom modulu. Ovdje dajemo primjer:

```
Public Sub uvecajX()
        Me.x = Me.x + 1
        If Me.x > 100 Then
                Me.x = 100
        End If
End Sub
```

Cilj ove funkcije je da uveća **x** za **1** ali da ga limitira na **100**. Uočavamo da funkcije započinju sa ključnom riječju **Sub** prethođenom sa vidljivošću ovog metoda. Dalje, uočavamo ključnu riječ **Me** što se ovdje mora navoditi i označava tekući objekat klase za koji je metod pozvan odnosno pandan pokazivaču **this** iz C++-a. Uočavamo i sintaksne elemente kao što su selekcija i kraj funkcije koje imaju unekoliko različiti oblik od većine učenih programskih jezika.

Visual Basic podržava prosto nasljeđivanje ali daje progrmeru manji komfor nego što je navikao u drugim programskim jezicima. Slično je i sa ostalim konceptima OO programiranja.