

# Moderno programmēšanas valodu praktikums

Dr. sc. ing. Pāvels Rusakovs

Mg. sc. ing. Vladislavs Nazaruks

# Programmēšanas valoda *Java*

## Literatūras saraksts:

1. *Эккель Брюс. Философия Java.*  
Санкт-Петербург, “Питер”, 2001.
2. *Смирнов Николай. Java 2. Учебное пособие.*  
Москва, “Три Л”, 2000.
3. *Ноутон Патрик, Шилдт Герберт. Java 2.*  
Санкт-Петербург, “БХВ-Петербург”, 2006.
4. *Дейтел Х. М., Дейтел П. Дж. Как программировать на Java.*  
Москва, “Бином-Пресс”, 2003.
5. *Хорстманн Кей, Корнелл Гари.*  
*Java 2. Тонкости программирования.*  
Москва, “Вильямс”, 2004.

6. *Шилдт Герберт.*

Полный справочник по Java, 7-е издание.

Москва, “Вильямс”, 2007, 1040 с.

7. *Макконнелл Стив.*

Совершенный код. Мастер-класс.

Microsoft Press, Санкт-Петербург, “Питер”, 2005, 896 с.

## Java valodas pamatprincipi

1. Sākumā *Java* valodu plānoja lietot *sadzīves iekārtās*.
2. Galvenā doma: *pārnesamība starp platformām*.
3. Vēlāk Java iekaroja vietu *vispasaules tīmeklī (WWW)*.
4. C valoda principiāli izmainīja sistēmas programmēšanu; Java valoda – *internetu*.
5. Java *nenodrošina* savietojamību ar C++.
6. Java *līdzīga* C# valodai.
7. Java lieto konsoles lietojumos, sīklietotnēs, grafiskos lietojumos, serversīklietotnēs.

## Elementārā programma

Teksta ziņojuma izvade (fails *Hello.java*)

```
public class Hello {  
    public static void main(String args[]) {  
        System.out.println("Hello, world !");  
    }  
}
```

---

### Komentāri:

1. *System* ir klase.
2. *out* ir objekts.
3. *println()* ir metode.
4. *args[]* ir komandrindas parametru masīvs
5. Failā ir tikai viena *public* – klase.

## Komandrindas parametru apstrāde

```
public class MyParams {  
    public static void main(String args[]) {  
        for(int i=0; i<args.length; i++)  
            System.out.println(  
                (i+1) + ". " + args[i]);  
    }  
}
```

---

## Programmas palaišana

```
javac MyParams.java
```

```
java MyParams "data1.txt" "data2.txt"
```

---

*javac* – kompilators. Rezultāts: *\*.class* faili ar baitkodu.

*java* – Java virtuālās mašīnas (JVM) palaišana.

## *Datu tipu pārveidošanas īpatnības*

Pārveidošana starp “lieliem” un “maziem” tiem:

```
int i = 2;  
long l = i; // pareizi  
  
long l = 2;  
int i = l; // NEPAREIZI !!
```

---

```
long l = 2;  
int i = (int) l; // pareizi
```

---

```
byte b1=5, b2=4, b3;  
  
b3=b1*b2; // NEPAREIZI !!  
  
b3=(byte) (b1*b2); // pareizi
```



Nevar izmantot neinizializētos mainīgos:

```
int x;  
System.out.println(x); // Kļūda
```

---

Ieliktos koda blokos nevar izmantot mainīgos ar *vienu un to pašu vārdu*:

```
int x = 1;  
{  
    int x = 2; // Kļūda  
}
```

---

Nulles pārbaudei **neizmanto** operatoru !

```
int x = 0;  
if (!x) { ... // Kļūda
```

1. Klases “koordinātu punkts” fragments (fails *Demo.java*).  
Sākums.

```
class CoordPoint {  
    private int X;  
    private int Y;  
    public CoordPoint(int X, int Y) {  
        this.X = X;  
        this.Y = Y;  
    }  
    public CoordPoint() {  
        this(1, 2);  
    }  
    public int getX() {  
        return X;  
    }  
}
```

2. Klases “koordinātu punkts” fragments (fails *Demo.java*).  
Beigas.

```
public void setX(int X) {
    this.X = X;
}
```

...

```
public void print() {
    System.out.println("X: " + X +
        ", Y: " + Y);
}
```

```
}
```

Piezīme: perspektīvā *print()* metodes nebūs. Tiks izmantota cita tehnika.

## Komentāri:

1. Nav sekciju **private**, **protected**, **public**.
2. Nav destruktora.
3. Nav iespējas atdalīt interfeisu no realizācijas.
4. No viena konstruktora var izsaukt citu konstrukturu ar **this** palīdzību.
5. Metodēs ar piekļuves modifikatoru **public** ieteicams rakstīt pirmo simbolu apakšējā reģistrā.
6. Bāzes klašu bibliotēka (*pakotne*) *java.lang* ir pieslēgta automātiski.

### 3. Galvenā programma (fails *Demo.java*).

```
public class Demo {  
    public static void main(String args[]) {  
        CoordPoint CP1 = new CoordPoint(),  
        CP2 = new CoordPoint(3, 4);  
        CP1.print();  
        CP2.print();  
    }  
}
```

---

#### Komentāri:

1. Nav *rādītāju*.
2. Visi objekti ir *dinamiskie* objekti un ir izvietoti kaudzē (*heap*).
3. Visi objekti tiks iznīcināti *automātiski*.

## 1. Darbs ar pakotnēm *bez* informācijas importēšanas.

```
java.util.ArrayList AL = new
```

```
    java.util.ArrayList();
```

```
AL.add(new Integer(1)); // var vienkārši ... (1)
```

```
AL.add(new Integer(2));
```

---

## 2. Darbs ar pakotnēm ar *konkrētu* klašu importēšanu.

```
import java.util.ArrayList;
```

```
...
```

```
ArrayList AL = new ArrayList();
```

```
AL.add(new Integer(1));
```

```
AL.add(new Integer(2));
```

### 3. Darbs ar pakotnēm importējot *visu* informāciju.

```
import java.util.*;  
  
...  
ArrayList AL = new ArrayList();  
LinkedList LL = new LinkedList();  
  
AL.add(new Integer(1));  
AL.add(new Integer(2));  
  
LL.add(new Integer(1));  
LL.add(new Integer(2));
```

Programmā var eksistēt vairākas rindiņas **import**.

## Pakotnes veidošana

```
package MyPackages.x;  
public class Rand {  
    private double R = Math.random();  
    public double getR() {  
        return R;  
    }  
}
```

---

### Komentāri:

1. Pakotnes vārds ir *Rand.java*.
2. Pakotnē var būt tikai viena rindiņa **package**....
3. Galvenajā programmā būs rindiņa **import** *MyPackages.x.\**;



## Pakotnes meklēšana

1. Analīzē **CLASSPATH** mainīgā saturu. Lai ir:

`CLASSPATH=.;D:\JAVA`

2. Aizvieto *punktus* pakotnes vārdā ar *slīpām svītrām*:

`MyPackages\x`

3. Pievieno iegūto ceļu *jau eksistējošajiem ceļiem* CLASSPATH mainīgajā. Ir divi varianti:

`MyPackages\x un D:\JAVA\MyPackages\x.`

4. Meklēšana abās mapēs.

## Piekļuves specifikatoru efekts

	<i>private</i>	<i>public</i>	<i>protected</i>	bez specifikatora
klase	+	+	+	+
pakotne	-	+	+	+
cita pakotne	-	+	-	-
apakšklase pakotnē	-	+	+	+
apakšklase citā pakotnē	-	+	+	-

## Destruktora imitācija

Finalizatora kods:

```
class Temp {  
    static int Counter = 0;  
  
    public Temp() {  
        Counter++;  
        System.out.println(Counter);  
    }  
  
    public void finalize() {  
        Counter--;  
        System.out.println("Destroyed !");  
    }  
}
```

Galvenā programma:

```
public class Demo {  
    public static void main(String [] args) {  
        final int N = 10000;  
        for(int i=0; i<N; i++) {  
            new Temp();  
        }  
    }  
}
```

---

**Komentāri:**

1. Tiks izvadīti vairāki ziņojumi “*Destroyed !*”.
2. Skaitītāja *Counter* vērtība nebūs vienāda ar *N* (būs, piemēram, 4249).

## Statiskie bloki *Java* klasē

Blokus bieži izmanto darbā ar objektu masīviem:

```
class ArrInit {
    static final int M = 5;
    static Integer Vec[] = new Integer[M];
    static {
        for(int i=0; i<Vec.length; i++)
            Vec[i] = new Integer(i);
    }
    static {
        for(int i=0; i<Vec.length; i++)
            System.out.print(Vec[i]+" ");
        System.out.println();
    }
}
```

## Programmas fragments:

```
IntArray IA1 = new IntArray();
```

## Rezultāts:

0 1 2 3 4

---

## Komentāri:

1. Var eksistēt *vairāki* statiskie bloki.
2. Statisko bloku kods izpildās *automātiski* pēc objekta izveidošanas programmā, ievērojot bloku deklarēšanas secību.
3. Līdzīgajā stilā var arī izveidot *dinamiskus* blokus.

## Statiskās metodes

1. Var izsaukt tikai citas **static** – metodes.
  2. Var apstrādāt tikai **static** – datus.
  3. Nevar norādīt uz **this** vai **super**.
- 

Informācijas izvades saīsināšana:

```
class P {  
    public static void pr(String S) {  
        System.out.print(S);  
    }  
    public static void prln(String S) {  
        System.out.println(S);  
    }  
}  
...  
P.prln("Result:");
```

## Mantošana *Java* valodā

Visas *Java* klases ir klases *Object* apakšklases.

```
class CoordPoint {...
```

```
class CoordPoint extends Object {...
```

```
class CoordPoint extends java.lang.Object {...
```

Visām rindiņām ir *viens un tāds pats* efekts.

---

Vienas superklases izmantošana:

- Atvieglo *parametru nodošanu*.
- Pavienkāršo *drazu savākšanu*.
- Atļauj *mantot* (un pārdefinēt) dažas metodes.



## Informācijas izvade ar *toString()* metodes palīdzību

C++ valodā var pārlādēt operatoru <<.

```
#include <iostream.h>
class CoordPoint {
    ...
    friend ostream& operator <<
        (ostream& Out, const CoordPoint& CP);
};
```

---

### Komentāri:

1. Pārlādētais operators << ir *draugs*, nevis klases metode.
2. Informāciju izvieto izvades plūsmā *ostream*.

Operatora `<<` *realizācija*:

```
ostream& operator <<
    (ostream& Out, const CoordPoint& CP) {
    Out << "X: " << CP.X << ", Y: " << CP.Y;
    return Out;
}
```

---

Operatora *izmantošana* programmā:

```
CoordPoint CP;
cout << CP;
```

---

**Komentāri:**

1. *cout* ir *ostream* klases objekts.
2. CP ir *statiskais* objekts.

*toString()* metodes pārdefinēšana *Java* valodā:

```
class CoordPoint
    ...
    public String toString() {
        return ("X: " + X + ", Y: " + Y);
    }
}
```

---

*toString()* metodes izmantošana programmā:

```
CoordPoint CP = new CoordPoint();
System.out.println(CP);
```

```
//var arī
```

```
System.out.println(CP.toString());
```

## Objektu vienādības pārbaude ar *equals()* metodi

*equals()* metodes pārdefinēšana *Java* valodā:

```
class CoordPoint
    ...
    public boolean equals(Object O) {
        return ( (X == ((CoordPoint) O).X) &&
                (Y == ((CoordPoint) O).Y) );
    }
}
```

---

Programmas fragments:

```
CoordPoint CP1 = new CoordPoint(),
    CP2 = new CoordPoint();
if (CP1.equals(CP2)) {...
```

## Apakšklases veidošana *Java* valodā

Klase *DisplayPoint*. Sākums:

```
class DisplayPoint extends CoordPoint {  
    private int Color;  
    public DisplayPoint  
        (int X, int Y, int Color) {  
        super(X, Y);  
        this.Color = Color;  
    }  
  
    public void setColor(int Color) {  
        this.Color = Color;  
    }  
  
    public int getColor() {  
        return Color;  
    }  
}
```

Klase *DisplayPoint*. Beigas.

```
public String toString() {  
    return super.toString() +  
        ", Color: " + Color;  
}
```

---

## Komentāri:

1. Apakšklases konstruktorā izsauc superklases konstruktoru ar rezervētā vārda **super** palīdzību.
2. Apakšklases metodē *toString()* izsauc superklases metodi *toString()* ar **super** palīdzību.
3. Konstruktoros “pēc noklusējuma” **super** var nepielietot.

Superklases mainīgais var norādīt uz *apakšklases objektu*:

```
CoordPoint CP = new DisplayPoint();
```

---

Informācijas izvade:

```
System.out.println(CP.getX());
```

```
System.out.println(  
    (DisplayPoint) CP).getColor());
```

---

Nepareizi:

```
System.out.println(CP.getColor()); // Kļūda
```

## Ierobežojumi *Java* valodā

Rezervēto vārdu **final** izmanto ar:

1. *Mainīgajiem (atribūtiem)*. Rezultāts: *konstantes* ekvivalents.

Konstante *funkcijā*:

```
final int N = 5;
```

Atribūts *klasē*:

```
private final static int DefX = 1;
```

---

2. *Metodēm*. Rezultāts: nevar *pārdefinēt* metodes apakšklasēs.

```
public final int getX() { ... }
```

---

3. *Klasēm*. Rezultāts: nevar *mantot* no klases.

```
final class CoordPoint { ... }
```



## Masīvi

Jebkurš masīvs ir *norāde* uz objektu vadāmajā kaudzē.

```
int V[] = {1, 2};  
System.out.println(V);  
// [I@45a877
```

```
Object [] O = new Object[2];  
System.out.println(O);  
// [Ljava.lang.Object;@45a877
```

Piezīme: **neder** C++ varianti ar *masīva izmēra* norādīšanu.

```
int V[2];  
int V[2]= {1, 2};
```

---

*Primitīvu* tipu masīvs satur *elementu vērtības*.

```
System.out.println(V[0] + " " + V[1]); // 1 2
```

Objektu masīvs satur *norādes*.

Norāžu sākotnēja vērtība: `null`.

```
for(int i=0; i<O.length; i++)  
    System.out.print(O[i] + " ");  
// null null
```

---

Masīvu deklarācija:

```
int V1[], V2;    // V1 ir masīvs, V2 - skaitlis  
int [] V1, V2;  // V1, V2 - masīvi
```

---

Abos gadījumos masīvi *nav inicializēti*.

```
System.out.println(V1.length);  
// Kompilācijas kļūda
```

Lai ir masīvs *V*. Masīva inicializēšanas varianti:

1. Masīva izveidošana ar *elementu uzskaitījumu*.

```
int [] V = {1, 2, 3};
```

2. Masīva izveidošana *pēc deklarācijas*.

```
int [] V = new int[3];
```

```
V[0] = 1; V[1] = 2; V[2] = 3;
```

vai

```
int [] V;
```

```
...
```

```
V = new int[3];
```

```
V[0] = 1; V[1] = 2; V[2] = 3;
```

Lai ir koordinātu punktu masīvs:

```
CoordPoint Line[] = new CoordPoint[2];  
System.out.println(Line[0]);    //null  
System.out.println(Line[0].getX());  
//Izņēmums: java.lang.NullPointerException
```

---

Pareizi:

```
CoordPoint Line[] = {  
    new CoordPoint(1, 2),  
    new CoordPoint(3, 6)  
};
```

vai:

```
CoordPoint Line [] = new CoordPoint[2];  
Line[0] = new CoordPoint(1, 2);  
Line[1] = new CoordPoint(3, 6);
```

Masīva *tiešā piešķire*: norādes kopēšana.

```
int [] V1 = {1, 2}, V2 = {3, 4};
```

```
V2 = V1;
```

```
System.out.println(V2[0]); //1
```

```
V1[0] = 10;
```

```
System.out.println(V2[0]); //10
```

Tā pati situācija ir ar *objektu* masīvu.

Rezultāts: “jauns” masīvs *tiks atkarīgs* no “veca” masīva.

Divas norādes saistītas ar vienu un to pašu masīvu.

## Masīva *elementu* kopēšana.

```
System.arraycopy(<masīvs - avots>,  
    <nobīde avotā>,  
    <masīvs-rezultāts>,  
    <nobīde rezultātā>,  
    <elementu daudzums>)
```

---

## Nokopēt visu *masīvu*:

```
System.arraycopy(v1, 0, v2, 0, v1.length);
```

---

## Nokopēt *masīva daļu*:

```
int v1[] = {1, 2, 3, 4};
```

```
int v2[] = new int[4];
```

```
v2[0] = v2[3] = 6;
```

```
System.arraycopy(v1, 0, v2, 1, 2); //v2: 6 1 2 6
```

*Masīva aizpildīšana.*

```
import java.util.*;  
...  
int v[] = new int[4];  
Arrays.fill(v, 5);    // 5 5 5 5
```

---

*Masīva daļas aizpildīšana.*

```
int x[] = new int[5];  
Arrays.fill(x, 1, 4, 3); // 0 3 3 3 0
```

---

*Objektu masīva aizpildīšana.*

```
Arrays.fill(Line, new CoordPoint(0, 0));
```

## Masīva kārtošana.

*Primitīvu masīvs (augošā secība):*

```
int v[] = {4, 3, 2, 1};  
Arrays.sort(v);           // 1, 2, 3, 4
```

---

*Objektu masīvs (dilstošā secība):*

```
Integer v[] = {  
    new Integer(1), new Integer(5),  
    new Integer(3), new Integer(4) };  
  
Arrays.sort(v, Collections.reverseOrder());  
// 5, 4, 3, 1  
  
Arrays.sort(v);  
// 1, 3, 4, 5
```



*Divdimensiju masīva deklarēšana:*

```
int [][] M = { {1, 2, 3}, {4, 5}, {6} };
```

*Cita iespēja iegūt to pašu rezultātu:*

```
int [][] M = new int[3][];  
M[0] = new int[]{1, 2, 3};  
M[1] = new int[]{4, 5};  
M[2] = new int[]{6};
```

*Rezultātu izvade:*

```
for(int i=0; i<M.length; i++) {  
    for(int j=0; j<M[i].length; j++) {  
        System.out.print(M[i][j] + " ");  
    }  
    System.out.println();  
}
```

Java2, versija 5.0: Uzlabotais cikls **for**.

*Viendimensijas masīva apstrāde:*

```
int [] V = {1, 2, 3};  
int Sum = 0;  
for (int Elem: V) {  
    Sum += Elem;  
}                                     // Sum = 6
```

---

*Divdimensijas masīva apstrāde:*

```
int [][] M = { {1, 2, 3}, {4, 5, 6} };  
for (int [] Row: M) {  
    for (int Elem: Row) {  
        Sum += Elem;  
    }  
}                                     // Sum = 21
```

Uzlabotā cikla **for** mainīgo izmanto *tikai lasīšanai*.

```
for (int Elem: V) {  
    Elem += 5; // masīva elementi netiks izmainīti  
}
```

---

Informācijas meklēšana un izeja no cikla.

Atrast *pirmo negatīvo* elementu un iziet no cikla.

```
boolean Found = false;  
for (int Elem: V) {  
    if (Elem < 0) {  
        Found = true;  
        break;  
    }  
}
```

## Refleksija

Informācijas par klases *atribūtiem*, *konstruktoriem* un *metodēm* iegūšana.

Ja attiecīgās klases (*CoordPoint*) nav, tiks ierosināts izņēmums *ClassNotFoundException*.

```
import java.lang.reflect.*;
```

```
...
```

```
try {  
    Class cl = Class.forName("CoordPoint");  
    Constructor [] c = cl.getConstructors();  
    Method [] m = cl.getMethods();  
    Field [] f = cl.getFields();  
    ...  
}
```

## Informācijas izvade:

```
System.out.println("---Constructors:");  
for (int i=0; i<c.length; i++)  
    System.out.println(c[i]);  
  
System.out.println("---Methods:");  
for (int i=0; i<m.length; i++)  
    System.out.println(m[i]);  
  
System.out.println("---Attributes:");  
for (int i=0; i<f.length; i++)  
    System.out.println(f[i]);  
}  
  
catch (ClassNotFoundException e) {  
    System.out.println("ERROR !");  
}
```

## Rezultātu fragments:

---Constructors:

```
public CoordPoint()
```

---Methods:

```
public java.lang.String CoordPoint.toString()
```

```
public int CoordPoint.getX()
```

```
public void CoordPoint.setX(int)
```

```
public final native java.lang.Class
```

```
    java.lang.Object.getClass()
```

```
public boolean
```

```
    java.lang.Object.equals(java.lang.Object)
```

---Attributes:

Informācijas par **private** un **protected** klases locekļiem *nav*.

Informācija par *visiem* klases locekļiem (tajā skaitā par **private** un **protected**):

```
import java.lang.reflect.*;
...

Class cl = Class.forName("CoordPoint");
Constructor [] c = cl.getDeclaredConstructors();
Method [] m = cl.getDeclaredMethods();
Field [] f = cl.getDeclaredFields();
...
```

---

Rezultātu fragments:

```
---Attributes:
protected int CoordPoint.X
protected int CoordPoint.Y
```

Informācija par klasi no *citas pakotnes*.

Pakotnes vārdu izmanto *kā prefiksu*.

```
import java.lang.reflect.*;
...
try {
    Class cl = Class.forName("java.util.Vector");

    Method [] m = cl.getMethods();
    Constructor [] c = cl.getConstructors();
}

catch (ClassNotFoundException e) {
    System.out.println("ERROR !");
}
```



Ir iespēja iegūt informāciju par klasi *bez izņēmumu apstrādes*.

```
Constructor [] c =  
    CoordPoint.class.getDeclaredConstructors();  
  
Method [] m =  
    CoordPoint.class.getDeclaredMethods();  
  
Field [] f =  
    CoordPoint.class.getDeclaredFields();
```

---

## Komentāri:

1. Piemērā izmanto atribūtu *class*.
2. Atribūts *class* ir klases *Class* objekts.

## Dinamiska tipu identifikācija (RTTI)

Var uzzināt, kas ir mainīgajā-konteinērā.

```
CoordPoint CP1 = new CoordPoint();  
CoordPoint CP2 = new DisplayPoint();  
DisplayPoint DP = new DisplayPoint();
```

---

Konteinera satura pārbaude (sākums):

```
System.out.println("CP1 is CoordPoint ?" +  
    (CP1 instanceof CoordPoint));  
System.out.println("CP1 is DisplayPoint ?" +  
    (CP1 instanceof DisplayPoint));  
System.out.println("CP2 is CoordPoint ?" +  
    (CP2 instanceof CoordPoint));  
System.out.println("CP2 is DisplayPoint ?" +  
    (CP2 instanceof DisplayPoint));
```

## Konteinera satura pārbaude (beigas):

```
System.out.println("DP is CoordPoint ?" +  
    (DP instanceof CoordPoint));  
System.out.println("DP is DisplayPoint ?" +  
    (DP instanceof DisplayPoint));
```

---

## Rezultāti:

```
CP1 is CoordPoint ?true  
CP1 is DisplayPoint ?false  
CP2 is CoordPoint ?true  
CP2 is DisplayPoint ?true  
DP is CoordPoint ?true  
DP is DisplayPoint ?true
```

To pašu efektu var sasniegt ar metodes *isInstance()* palīdzību:

```
System.out.println(  
    CoordPoint.class.isInstance(CP1));  
System.out.println(  
    DisplayPoint.class.isInstance(CP1));  
System.out.println(  
    CoordPoint.class.isInstance(CP2));  
System.out.println(  
    DisplayPoint.class.isInstance(CP2));  
System.out.println(  
    CoordPoint.class.isInstance(DP));  
System.out.println(  
    DisplayPoint.class.isInstance(DP));
```

Rezultāti pilnīgi sakrīt ar **instanceof** pielietojšanas rezultātiem.

## Interfeisi

### Iespējamā klases deklarācija

```
class <klase>  
    [extends <superklase>]  
    [implements <1. interfeiss>  
    [, <2. interfeiss>]...  
] {
```

---

Interfeisā var izmantot:

1. *Konstantes* (*final* – atribūtus).
2. Metožu *specifikācijas*.

*Visas* komponentes ir **public** (cits variants nav iespējams).

- ✓ *Visām* interfeisa metodēm jābūt realizētām klasē, kura realizēs uzdoto interfeisu, citādi iegūsim *abstrakto klasi*.
- ✓ Metožu signatūras realizācijas procesā *pilnīgi sakrīt*.
- ✓ Interfeisiem *ir sava hierarhija*, kura nekādā veidā nekrustojas ar klašu hierarhiju.
- ✓ Viens interfeiss var būt realizēts vairāk nekā vienā klasē, pat ja klases nekāda veidā *nav saistītas* ar mantošanas hierarhiju.
- ✓ Interfeisus lietderīgi izmantot, lai *atdalītu metodes (metožu) definīciju* no mantošanas hierarhijas.
- ✓ Interfeisa tuvākais sinonīms – **kontrakts**.

Interfeisa piemērs:

```
interface Account {  
    double TAX = 0.23;  
    double getPrice();  
    double getTotalPrice(int N);  
    void sell(int N);  
}
```

---

**Komentāri:**

1. Interfeisu deklarē ar rezervētā vārda **interface** palīdzību.
2. Piekļuves modifikatoru **public** parasti neraksta.
3. Interfeiss satur vienu konstanti un trīs metožu deklarācijas.

## Interfeisa realizācija. Sākums.

```
class Book implements Account {  
    private String Name;  
    private double Price;  
    private int N;  
  
    public Book(String Name, double Price, int N) {  
        this.Name = Name;  
        this.Price = Price;  
        this.N = N;  
    }  
  
    public double getPrice() {  
        return Price;  
    }  
}
```



Interfeisa realizācija. Beigas.

```
public double getTotalPrice(int N) {  
    return ...; // programmētāja kods  
}  
  
public void sell(int N) {  
    ...           // programmētāja kods  
}  
}
```

---

Objekta veidošana un izmantošana programmā.

```
Book Java = new Book("Valoda Java", 4.30, 100);  
  
System.out.println(Java.getPrice());
```

Interfeisu var izmantot *konstanšu grupēšanai*.

```
interface Week {  
    int MONDAY = 1, TUESDAY = 2, WEDNESDAY = 3,  
        THURSDAY = 4, FRIDAY = 5, SATURDAY = 6,  
        SUNDAY = 7;  
}
```

---

Klases deklarācija:

```
class Plan implements Week {  
    ...  
    // darbs ar MONDAY;  
}
```

Interfeisu ar konstantēm var arī *nerealizēt* klasē.

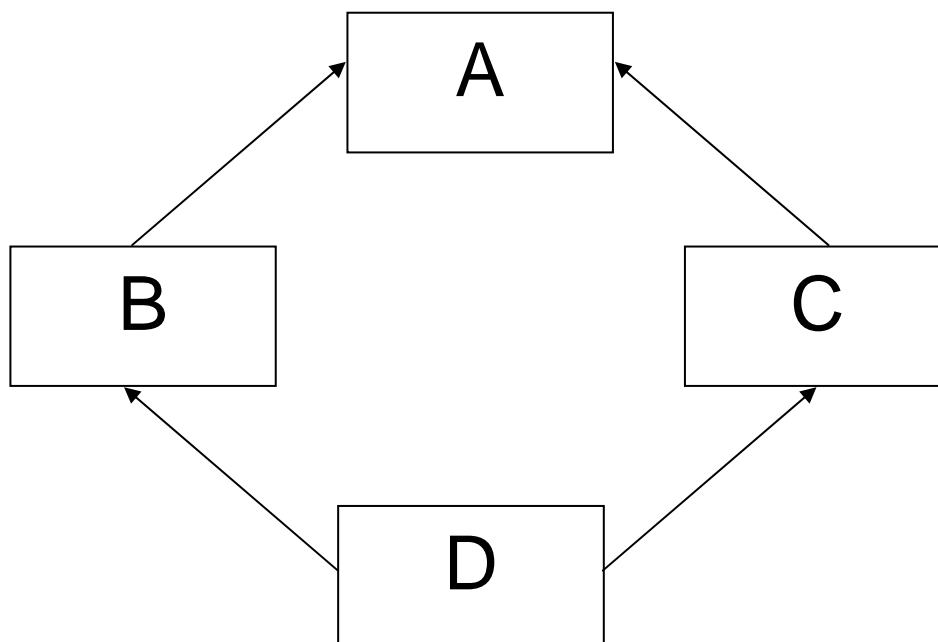
```
class Plan {  
    ...  
    // darbs ar Week.MONDAY;  
}
```

---

Viens interfeiss var mantot *no vairākiem interfeisiem vienlaicīgi* (atšķirībā no klašu mantošanas).

```
interface C extends A, B {  
    ...  
}
```

## Rombveida mantošana



Simetriskā pieceja:

```
interface A {  
}
```

```
interface B extends A {  
}
```

```
interface C extends A {  
}
```

```
class D implements B, C {  
}
```

Asimetriskā pieceja:

```
interface A {  
}
```

```
class B implements A {  
}
```

```
interface C extends A {  
}
```

```
class D extends B implements C {  
}
```

Daudzkāršās mantošanas imitācija

Hidroplāns var *lidot* un *peldēt*.

```
interface Plane {  
    void fly();  
}  
  
interface Ship {  
    void swim();  
}  
  
class HydroPlane implements Plane, Ship {  
    public void fly() {};  
    public void swim() {};  
}
```

Operācijas ar hidroplānu: lidošana, peldēšana, pārvietošana.

```
public class Test {  
    static void flying(Plane P) {  
        System.out.println("Flying !");  
    }  
  
    static void swimming(Ship S) {  
        System.out.println("Swimming !");  
    }  
  
    static void movement(HydroPlane H) {  
        System.out.println("ALL !");  
    }  
    ...  
}
```



Objekta veidošana un operāciju pielietošana.

```
public static void main(String args[]) {  
    HydroPlane HP = new HydroPlane();  
    flying(HP);  
    swimming(HP);  
    movement(HP);  
}  
}
```

---

## Komentāri:

1. Interfeisus var norādīt metožu *parametru sarakstā*.
2. Ja klase realizē interfeisu, un interfeiss ir formālais parametrs, klases objektu var nodot kā faktisko parametru.

Interfeiss *Comparable*.

Interfeisa metode:

```
public int compareTo(Object <params>)
```

Metodes rezultāts: -1, 0, 1.

---

Uzdevums: sakārtot masīvu no objektiem – automašīnām cenu augošajā secībā.

Klases deklarācija un atribūti:

```
class Auto implements Comparable {  
    private String Name;  
    private float Price;
```

## Interfeisa metodes realizācija:

```
public int compareTo(Object O) {  
    float P = ((Auto) O).Price;  
    if (Price < P)  
        return -1;  
    else  
        return (Price > P)?1:0;  
}  
}
```

---

## Objektu masīvs:

```
Auto [] Autos = {  
    new Auto("Ford", 5000),  
    new Auto("Mersedes", 3000),  
    new Auto("Renault", 4000)  
};
```

Masīva kārtošana:

```
Arrays.sort(Autos);
```

---

Interfeisa metodes realizācijas *saīsināšana*:

```
public int compareTo(Object O) {  
    float P = ((Auto) O).Price;  
    return (Price < P) ? -1 : ((Price > P) ? 1 : 0);  
}
```

---

Kārtošana *dilstošajā* secībā:

```
return (Price < P) ? 1 : ((Price > P) ? -1 : 0);
```

Interfeiss *Comparator*.

Interfeisa metode:

```
public int compare (Object o1, Object o2)
```

Metodes rezultāts: -1, 0, 1.

- 
- ✓ Interfeisu realizē *citā klasē*.
  - ✓ Visbiežāk klase satur tikai vienu metodi.
  - ✓ Var sakārtot masīvu pēc *vairākiem* atribūtiem.

Klases Auto fragments (interfeisu *nerealizē*):

```
class Auto {  
    private String Name;  
    private float Price;  
    ...  
}
```

---

Jaunā klase:

```
class MyCompare implements Comparator {  
    public int compare(Object O1, Object O2) {  
        float P1 = ((Auto) O1).getPrice();  
        float P2 = ((Auto) O2).getPrice();  
        return (P1<P2) ? -1 : ((P1>P2) ? 1 : 0);  
    }  
}
```

## Masīva kārtošana:

```
Arrays.sort(Autos, new MyCompare());
```

---

Interfeisu *Comparable* un *Comparator* metodes var arī vienkārši atgriezt *negatīvas un pozitīvas* vērtības (neobligāti -1, 0, 1).

## Teksta rindiņu kārtošana:

```
class Asc implements Comparator {  
    public int compare(Object O1, Object O2) {  
        return ((String) O1).toLowerCase().  
            compareTo(((String) O2).toLowerCase());  
    }  
}
```

## Izņēmumu apstrāde

- ✓ Visi izņēmumi ir *objekti*.
  - ✓ Izņēmumu klases ir *Exception* klases apakšklases.
- 

Rezervētie vārdi:

- try** – kontrolējamais bloks
- catch** – izņēmuma apstrādātājs
- throw** – izņēmuma ierosināšana
- throws** – neapstrādāto izņēmumu deklarācija
- finally** – obligāti izpildāmais bloks



Kontrolējamais bloks un izņēmumu apstrādātājs:

```
final int z=0;  
int x;  
  
try {  
    x = 1/z;  
}  
  
    catch (ArithmeticException e) {  
        System.out.println("Division by zero");  
    }  
}
```

---

Izņēmuma ierosināšana:

```
throw new  
    ArithmeticException("Division by zero");
```

Personiskā izņēmuma *deklarācija*:

```
class DataError extends Exception {  
    protected int ErrorCode;  
  
    public DataError(int Code) {  
        ErrorCode = Code;  
    }  
  
    public DataError() {  
        ErrorCode = 1;  
    }  
  
    public String toString() {  
        return ("Error with code: " +  
            ErrorCode + ".");  
    }  
}
```

Personiskā izņēmuma *ierosināšana* un *apstrāde*:

```
try {  
    throw new DataError(5);  
}  
  
catch (DataError e) {  
    System.out.println(e);  
}
```

---

*Neapstrādātā* izņēmuma deklarācija:

```
public static void f() throws DataError {  
    ...  
    throw new DataError(5);  
}
```

*Obligāti izpildāmais bloks:*

```
try {  
    throw new DataError(5);  
}  
catch(DataError e) {  
    ...  
}  
finally {  
    System.out.println("Done !");  
}
```

---

Ziņojums “*Done !*” tiks izvadīts *vienmēr*.

Visu izņēmumu tveršana:

```
catch(Exception e) {  
}
```

## Izņēmumu *apstrādes hierarhija*

```
class Main extends Exception {}
class Sub extends Main {}
...
try {
    throw new Sub();
}
catch (Sub e) {    // (*)
    System.out.println("Sub !");
}
catch (Main e) {   // (**)
    System.out.println("Main !");
}
```

Ar apstrādātāju (\*) vienmēr iegūsim “Sub !”.

Ar apstrādātāju (\*\*) un bez apstrādātāja (\*) iegūsim “Main !”.

## Klases - *čaulas*

```
Byte B = new Byte((byte)1);  
Short S = new Short((short)1);  
Integer I = new Integer(1);  
Long L = new Long(1);  
  
Float F = new Float(1.0);  
Double D = new Double(1.0);  
  
Character C = new Character('1');  
Boolean BL = new Boolean(true);
```

---

Piezīme: visi skaitļi (*Byte – Double*) ir abstraktās klases *Number* apakšklases.

```
Number N = L;
```

## Darbs ar klasēm – tipu čaulām

```
Integer I = new Integer(5);  
int i = I.intValue();  
  
System.out.println(i);      // 5  
  
Float F = new Float(2.5);  
float f = F.floatValue();  
  
System.out.println(f);      // 2.5
```

---

### Aritmētisko operāciju piemērs.

```
Integer I = new Integer(1), J = new Integer(4);  
Integer K = new Integer(I.intValue() +  
    J.intValue());  
  
System.out.println(K);      // 5
```

## Java2, versija 5.0. Tipu čaulas.

### Boxing (iepakošana):

```
int i = 2;  
Integer I = i + 1;  
i = I;                //i=3
```

---

### Klases Object izmantošana:

```
int i = 2;  
Object O = i + 1;  
i = (Integer)O;       //i=3  
i = (int)O;           //kompilācijas kļūda
```

---

### Aritmētisko operāciju piemērs:

```
Integer i = 1, j = 2;  
i += j;                //i=3
```



## Pakotne *java.util*. Darbs ar klases *Object* objektiem.

### 1. Klase *Vector*.

```
import java.util.Vector;
...
Vector V = new Vector();
V.addElement(new Integer(5)); // [5]

try {
    V.insertElementAt(new Float(3.5), 1);
}
catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Index Error");
}
// [5, 3.5]
```

## 1. Klase *Vector*. Turpinājums.

```
for (int i=0; i<V.size(); i++)  
    System.out.println(i + ". " +  
        V.elementAt(i));  
  
System.out.println("FirstElement: " +  
    V.firstElement()); // 5  
  
System.out.println("LastElement: " +  
    V.lastElement()); // 3.5  
  
Integer I;  
I = (Integer) V.firstElement();  
System.out.println("I: " + I);  
//I: 5
```

## 1. Klase *Vector*. Turpinājums.

```
V.setElementAt(new Float(1.2), 0);  
System.out.println(V); // [1.2, 3.5]
```

```
V.removeElementAt(0);  
System.out.println(V); // [3.5]
```

```
V.removeAllElements();  
System.out.println("Is vector empty ? " +  
    V.isEmpty());  
// Is vector empty ? true
```

## 1. Klase *Vector*. Beigas.

Pēdējās *Java* versijās var izmantot arī “*saīsinātus*” metožu vārdus.

```
V.add(new Integer(5));           // [5]
V.add(1, new Float(3.5));        // [5, 3.5]
System.out.println(V.get(0));    // 5
V.set(0, new Integer(4));        // [4, 3.5]
V.remove(0);
System.out.println(V);           // [3.5]
System.out.println("Is vector empty ? " +
    V.isEmpty());
//Is vector empty ? false
```

## 2. Klase *Stack*.

Klase *Stack* ir klases *Vector* apakšklase.

```
Stack S = new Stack();

S.push(new Float(2.5));
S.push(new Float(3.5));
S.push(new Float(4.5));
System.out.println(S);           // [2.5, 3.5, 4.5]

System.out.println(S.peek());    // 4.5
System.out.println(S);           // [2.5, 3.5, 4.5]

System.out.println(S.pop());     // 4.5
System.out.println(S);           // [2.5, 3.5]

System.out.println(S.firstElement() +
    " " + S.lastElement());      // 2.5 3.5
```

### 3. Klase *ArrayList*.

Populāras metodes:

- ✓ *size()* – saraksta *izmērs*.
- ✓ *add(<objekts>)* – *pievienot* jaunu objektu.
- ✓ *add(<indekss>, <objekts>)* – *ielikt* jaunu objektu.
- ✓ *get(<indekss>)* – *iegūt* saraksta objektu ar norādītu indeksu.
- ✓ *set(<indekss>, <objekts>)* – *izmainīt* objektu.
- ✓ *remove(<indekss>)* – *izslēgt* elementu ar *norādītu indeksu*.
- ✓ *clear()* – *izdzēst visus* elementus.

*Piezīme: tās pašas metodes ir arī klases *LinkedList* metodes.*

*Pamatā ir divu interfeisu metodes: *Collection* un *List*.*

*Piemēram, *size()* ir metode no *Collection*, bet *get()* – no *List*.*

### 3. Klase *ArrayList*. Turpinājums.

```
ArrayList AL = new ArrayList();
AL.add(new Integer(5));
AL.add(1, new CoordPoint());
AL.add(new DisplayPoint());
AL.set(0, new Float(2.5));

for(int i=0; i<AL.size(); i++)
    System.out.println((i+1) + ". " +
        AL.get(i).getClass().getName());

// 1. java.lang.Float
// 2. CoordPoint
// 3. DisplayPoint
```

*Piezīme:* var iegūt klases nosaukumu *bez* pakotnes vārda.

Lai ir: `AL.get(i).getClass().getSimpleName()`

Pirmais rezultāts būs: `Float`.

### 3. Klase *ArrayList*. Darbs ar noteiktas klases objektiem.

```
class CoordPointList {  
    private ArrayList List = new ArrayList();  
    public void add(CoordPoint CP) {  
        List.add(CP);  
    }  
  
    public CoordPoint get(int i) {  
        return (CoordPoint) List.get(i);  
    }  
  
    public int size() {  
        return List.size();  
    }  
}  
...  
CoordPointList CP_List = new CoordPointList();  
CP_List.add(new CoordPoint(1, 2));
```



### 3. Klase *ArrayList*. Iteratora izmantošana.

Populāras operācijas:

- ✓ Iegūt iteratoru no konteinerā (metode *iterator()* ).
- ✓ Iegūt nākošu objektu no secības (metode *next()* ).
- ✓ Pārbaudīt nākošā elementa eksistēšanu (metode *hasNext()* ).

```
Iterator It = AL.iterator();  
while (It.hasNext())  
    System.out.println(It.next());
```

*Piezīme:* to pašu rezultātu var iegūt, ja izmantot *ListIterator*.

```
ListIterator It = AL.listIterator();
```

Tad pēc cikla pabeigšanas var apstrādāt sarakstu *pretējā* virzienā:

```
while (It.hasPrevious())  
    System.out.println(It.previous());
```

Iteratorus var izmantot kolekcijas elementu *modifikācijai*.

*Uzdevums:* pārveidot teksta virknes no formas “x” uz “\_x”.

```
ListIterator It = AL.listIterator();
String Elem;
while (It.hasNext()) {
    Elem = (String) It.next();
    It.set("_" + Elem);
}
```

Saraksta apstrādei var arī izmantot modernizēto ciklu **for**:

```
for (Object Elem : AL) {
    System.out.println(Elem);
}
```

1. Apstrāde notiek tikai *vienā virzienā*.
2. *Nepar* izmainīt elementu vērtības.

### 3. Klase *ArrayList*. Klases noskaņošana (*generic*).

Sarakstā ir tikai *veselie skaitļi*.

```
ArrayList<Integer> AL_Int =  
    new ArrayList<Integer>();  
//ArrayList<int> AL_Int2 =  
    new ArrayList<int>(); Kļūda
```

---

```
AL_Int.add(2);  
AL_Int.add(3);  
AL_Int.add(0, 1);  
//AL_Int.add(2.5); Kļūda
```

---

```
System.out.println(AL_Int);           //[1, 2, 3]  
int IElem    = AL_Int.get(0);         //1  
float FElem  = AL_Int.get(0);         //1.0
```

### 3. Klase *ArrayList*. Iteratora noskaņošana.

```
Iterator<Integer> It = AL_Int.iterator();  
while (It.hasNext())  
    System.out.println(It.next());
```

---

### Superklases un apakšklases objekti noskaņotajā sarakstā

```
ArrayList<CoordPoint> AL_P =  
    new ArrayList<CoordPoint>();  
AL_P.add(new CoordPoint());  
AL_P.add(new DisplayPoint());
```

---

Piezīme: tagad lejupejošā pārveidošana no Object *nav aktuālā*.

```
CoordPoint CP1, CP2;  
CP1 = AL_P.get(0);  
CP2 = (CoordPoint) AL_P.get(0); //nevajag
```

#### 4. Klase *LinkedList*. Steka veidošana.

```
class LStack {  
    private LinkedList LL = new LinkedList();  
    public void push(Object O) {  
        LL.addLast(O);  
    }  
    public Object pop() {  
        return LL.removeLast();  
    }  
    public Object peek() {  
        return LL.getLast();  
    }  
    public String toString() {  
        return LL.toString();  
    }  
}
```

## 5. Interfeiss *List*.

```
List AutoList = new ArrayList();
```

```
List AutoList = new LinkedList();
```

---

Nav iespējams:

```
List AutoList = new List();
```

---

*ArrayList* versus *LinkedList*:

1. Elementa izvēle *ArrayList* objektā vienmēr aizņem apmēram *vienu un to pašu laiku*. *LinkedList* objektā viss ir atkarīgs no elementa pozīcijas.

2. Elementa ielikšana (dzēšana) *ArrayList* objektā notiek *lēnāk*.

Sarakstu kārtšanai izmanto interfeisus *Comparable* un *Comparator*.

## 1. Interfeisa realizācija.

```
class Auto implements Comparable {  
    ...  
    public int compareTo(Object o) {  
        ...  
    }  
}
```

---

## 2. Automašīnu masīvs.

```
ArrayList AL = new ArrayList();  
AL.add(new Auto("Ford", 4500f));  
AL.add(new Auto("Toyota", 3500f));
```

3. Metodes *sort()* izsaukums. Klases *Collections* izmantošana.

```
Collections.sort (AL) ;
```

---

4. Līdzīgajā stilā strādā ar *Comparator*. Lai klasē *MyComp* ir realizēts interfeiss *Comparator*.

```
Collections.sort (AL, new MyComp ( ) ) ;
```

---

5. Ja sarakstā ir citu klašu objekti (ne tikai *Auto*), kārtotāšanas laikā tiks ierosināts izņēmums *java.lang.ClassCastException*.



## 6. Klase *HashSet*.

```
HashSet HS = new HashSet();  
...  
HS.add("Key1");  
HS.add("Key1");  
HS.add("Key2");  
System.out.println(HS); // [Key2, Key1]  
HS.remove("Key1");  
System.out.println(HS); // [Key2]  
HS.remove("Key3");  
if (HS.contains("Key2"))  
    System.out.println("Key2 exists.");
```

---

Var arī izmantot interfeisu *Set*.

```
Set HS = new HashSet();
```

## 7. Interfeiss *Collection* .

Šo interfeisu realizē vairākas klases no pakotnes *java.util* (*ArrayList*, *LinkedList*, *HashSet*, ...).

Objektu radīšana:

```
Collection C_AL = new ArrayList();  
Collection C_LL = new LinkedList();  
Collection C_HS = new HashSet();
```

---

Informācija par izveidotiem objektiem:

```
System.out.println(C_AL.getClass().getName());  
//java.util.ArrayList  
System.out.println(C_LL.getClass().getName());  
//java.util.LinkedList  
System.out.println(C_HS.getClass().getName());  
//java.util.HashSet
```

### Izveidotā saraksta *ArrayList* apstrāde:

```
System.out.println(C_AL.add(1)); //true  
System.out.println(C_AL.add(1)); //true  
System.out.println(C_AL);       //[1, 1]
```

---

### Izveidotā saraksta *HashSet* apstrāde:

```
System.out.println(C_HS.add(1)); //true  
System.out.println(C_HS.add(1)); //false  
System.out.println(C_HS);       //[1]
```

---

### Specifisko metožu lietošana:

```
((LinkedList)C_LL).addFirst(1);  
((LinkedList)C_LL).addLast(2);  
System.out.println(C_LL);       //[1, 2]
```

---

Piezīme: **nav** iespējams

```
C_LL.addFirst(1); // kļūda
```

Dažas citas metodes:

1. *contains(Object)*. Elementa eksistēšanas pārbaude.

```
System.out.println(C_AL.contains(1)); // true  
System.out.println(C_AL.contains(2)); // false
```

---

2. *isEmpty()*. Kolekcijas tukšuma pārbaude.

```
System.out.println(C_AL.isEmpty()); // false
```

---

3. *addAll(Collection)*. Citas kolekcijas pievienošana.

```
C_AL.addAll(AL); // AL = [2, 3]  
System.out.println(C_AL); // [1, 1, 2, 3]
```

---

4. *clear()*. Kolekcijas dzēšana.

```
C_AL.clear();  
System.out.println(C_AL); // []
```

## 8. Klase *TreeSet*.

```
TreeSet TS = new TreeSet();
```

```
TS.add(1);
```

```
TS.add(3);
```

```
TS.add(2);
```

```
TS.add(0);
```

```
System.out.println(TS); // [0, 1, 2, 3]
```

```
System.out.println(TS.first()); // 0
```

```
System.out.println(TS.last()); // 3
```

```
System.out.println(TS.headSet(2)); // [0, 1]
```

```
System.out.println(TS.tailSet(2)); // [2, 3]
```

Var nodrošināt kopas elementu kārtošanu *dilstošajā* secībā.

*TreeSet* konstruktoram nodod parametru: objektu, kurš ir klases ar implementētu interfeisu *Comparator* piemērs.

1. Klase elementu *kārtošanas kontrolei*.

```
class LangsDesc implements Comparator {  
    public int compare (Object O1, Object O2) {  
        String L1 = (String) O1;  
        String L2 = (String) O2;  
        return L2.compareTo (L1) ;  
    }  
}
```

2. Kārtotas kopas veidošana.

```
TreeSet TS_L = new TreeSet ( new LangsDesc () ) ;
```

### 3. Elementu pievienošana kopai.

```
TS_L.add("Ada");  
TS_L.add("C++");  
TS_L.add("Java");
```

### 4. Rezultāts.

```
System.out.println(TS_L); // [Java, C++, Ada]
```

*Piezīme:* pēc noklusējuma ir augošā secība.

Var arī patstāvīgi norādīt augošu secību:

```
return L1.compareTo(L2); // (1)
```

```
return -L2.compareTo(L1); // (2)
```

**Nav iespējams** kārtot kopu uz *HashSet* pamata.

```
HashSet TS_L = new HashSet(new LangsDesc());  
// kļūda: tāda konstruktora nav
```

## 9. Klase *HashMap*. Sākums.

```
HashMap HM = new HashMap();  
...  
HM.put("Java", "Programming Language");  
HM.put("Oracle", "DBMS");  
  
System.out.println(HM);  
// {Oracle=DBMS, Java=Programming Language}  
  
System.out.println("Keys: " + HM.keySet());  
// Keys: [Oracle, Java]  
  
System.out.println("Values: " + HM.values());  
// Values: [DBMS, Programming Language]
```



## 9. Klase *HashMap*. Beigas

```
System.out.println("Size: " + HM.size()); // 2
System.out.println("Oracle: " +
    HM.get("Oracle")); // Oracle: DBMS
System.out.println("MS Access: " +
    HM.get("MS Access"));
// MS Access: null

HM.remove("Oracle");
System.out.println(HM);
// {Java=Programming Language}

HashMap NewHM = new HashMap();
NewHM.put("MS Access", "DBMS");
HM.putAll(NewHM);
System.out.println(HM);
//{Java=Programming Language, MS Access=DBMS}
```

## 10. Klase *TreeMap*.

Iepriekšējā piemērā vārdnīcu var izveidot arī tā:

```
TreeMap TM = new TreeMap();  
  
TM.put("Java", "Programming Language");  
TM.put("Oracle", "DBMS");  
  
System.out.println(TM.firstKey()); // Java  
System.out.println(TM.lastKey());  // Oracle
```

---

Ir arī citas metodes apakšstabulu iegūšanai:

- ✓ *headMap(<toKey>)*
- ✓ *tailMap(<fromKey>)*.
- ✓ *subMap(<fromKey>, <toKey>)*. Informācijas par pēdēju elementu nebūs.

## 11. Klase *StringTokenizer*. Vārdu meklēšana teikumā.

```
String S1 = "Please, test me !", S2 = S1;

String Delim = ", !.";

StringTokenizer ST1 = new StringTokenizer(S1,
    Delim), ST2 = new StringTokenizer(S2, Delim);

int i = 1;
while (ST1.hasMoreTokens())
    System.out.println((i++) + ". " +
        ST1.nextToken());

int N = ST2.countTokens();
for (i=0; i<N; i++)
    System.out.println((i+1) + ". " +
        ST2.nextToken());
```

11. Klase *StringTokenizer*. Meklēšanas rezultāti.

```
1. Please  
2. test  
3. me
```

---

Bibliotēka *java.util* satur arī vairākas citas klases.

Tika aplūkotas klases: *Vector*, *Stack*, *ArrayList*, *LinkedList*, *HashSet*, *TreeSet*, *HashMap*, *TreeMap*, *StringTokenizer*, kā arī interfeiss *List*.

Noskaņojamās (vispārinātās) klases *deklarācija*.

```
class CoordPoint<TX, TY> {  
    private TX X;  
    private TY Y;  
    public CoordPoint(TX X, TY Y) {  
        this.X = X;  
        this.Y = Y;  
    }  
    ...  
}
```

---

Noskaņojamās klases *izmantošana*.

```
CoordPoint<Integer, Integer> CPi =  
    new CoordPoint<Integer, Integer>(1, 2);  
CoordPoint<Double, Double> CPd =  
    new CoordPoint<Double, Double>(1.0, 2.0);
```

Iepriekšējā piemērā var izmantot `raw`-tipu.

```
CoordPoint CP = new CoordPoint("a", "b");
```

*Rezultāts:* nav nodrošināta tipu drošība.

Kompilators visbiežāk brīdina par šo problēmu.

Noskaņošanas gadījumā par kļūdām bieži informē *kompilācijas* laikā.

```
Integer I = CPd.getX(); //kompilācijas kļūda
```

Kļūdas iemesls: nevar pārveidot `Double` uz `Integer`.

`Raw` tipu gadījumā ir orientācija uz klasi `Object`.

Nepieciešama lejupejošā pārveidošana.

```
Integer I = (Integer) CP.getX();
```

*Rezultāts:* izņēmums *izpildes* laikā.

Noskaņojamā interfeisa *izmantošana*:

```
class Auto implements Comparable<Auto> {  
    ...  
    public int compareTo(Auto A) {  
        float P = A.Price;  
        return (Price < P) ? -1 : (Price > P) ? 1 : 0;  
    }  
}
```

---

Objektu masīvs un kārtošana:

```
Auto [] Autos = { new Auto("Ford", 5000), ... };  
...  
Arrays.sort(Autos);
```

## Noskaņošanas *ierobežojumi*

*Uzdevums:* izveidot klasi `Array` *patvaļīga skaitļu* masīva veidošanai un apstrādei.

```
class Array<T extends Number> {
    T [] Arr;
    ...
    T maxElem() {
        T max;
        max = Arr[0];
        for(int i=1; i<Arr.length; i++)
            max = (Arr[i].doubleValue() >
                    max.doubleValue()) ? Arr[i] : max;
        return max;
    }
    ...
}
```



Komentāri:

1. **N**eder tiešā elementu salīdzināšana:

```
max = (Arr[i]>max)?Arr[i]:max;
```

Šajā gadījumā tipam T **nav** definēts operators >.

2. **Nav** iespējama noskaņošana **bez** parametra ierobežojumiem.

```
class Array<T> {  
    ...  
    max = (Arr[i].doubleValue() >  
        max.doubleValue()) ? Arr[i]:max;
```

Metode doubleValue() ir klases Number metode.

3. Tagad **nav** iespējama pilnīgi patvaļīga masīva veidošana.

```
Array<String> A = new Array<String>();
```

Var būt tikai skaitļu masīvi:

```
Array<Integer> A = new Array<Integer>();
```

## Argumenti-šabloni

*Uzdevums:* pārbaudīt masīvu `Array` vienādību.

Nepieciešams salīdzināt *dažādu tipu* masīvus.

```
class Array<T extends Number> {
    ...
    boolean equal (Array<?> P) {
        if (P.Arr != null) {
            ...
        }
    }
}
```

Masīvu veidošana un salīdzināšana:

```
Array<Integer> AI = new Array<Integer>();
Array<Double> AL = new Array<Double>();
if (AI.equal(AL)) ...
```

*Piezīme:* **neder** variants

```
boolean equal (Array<T> P) {
```

Lai masīvā var būt tikai objekti, kuru klase realizē *interfeisu* Ship:

```
class Arr<T extends Ship> { ...
```

Lai masīvā var būt tikai objekti, kuru klase realizē *divus* interfeisus – Ship un Plane:

```
class Arr<T extends Ship & Plane> { ...
```

Lai masīvā var būt tikai objekti, kuru klase *manto* no klases Base un realizē *divus* interfeisus – Ship un Plane:

```
class Sub extends Base implements Ship, Plane {  
    ...  
}
```

```
class Arr<T extends Sub & Ship & Plane> { ...
```

*Piezīme:* klase var būt tikai *pirmais elements* sarakstā.

Objekta veidošana:

```
Arr <Sub> A = new Arr<Sub> ();
```

## Vispārinātas metodes

*Uzdevums: atrast elementa pozīciju noskaņojamajā masīvā.*

```
class GenArrMethods {
    static<T, V extends T> int
        pos(V[] Arr, T Elem) {
            int i=0;
            ...
            return (i != Arr.length) ? i : -1;
        }
}
```

Metodes *izmantošana*:

```
Integer IntV[] = {1, 3, 2};
System.out.println(GenArrMethods.pos(IntV, 3));
```

1. Klase *nav* noskaņota. Noskaņota *tikai metode*.
2. Var vispārināt *jebkuru* metodi (ne tikai statisko).

## Vispārinātie konstruktori nevispārinātajās klasēs

*Uzdevums:* izveidot klasi `CoordPoint` ar **double**-koordinātēm. Objektu veidošanai paredzēt *vienu* vispārinātu konstruktoru.

```
class CoordPoint {
    private double X;
    private double Y;

    <T extends Number> CoordPoint(T X, T Y) {
        this.X = X.doubleValue();
        this.Y = Y.doubleValue();
    }
    ...
}
```

Programmas fragments:

```
CoordPoint CP_I = new CoordPoint(1, 2);
CoordPoint CP_F = new CoordPoint(1.5f, 2.5f);
```

## Darbs ar teksta rindiņām

Teksta rindiņas ir *objekti*. Teksta rindiņu veidošana:

```
String s1, s2, s3, s4;
```

```
s1 = "12"; // "12"
```

```
s2 = new String("12"); // "12"
```

```
s3 = String.valueOf(12); // "12"
```

```
s4 = String.valueOf(12.5); // "12.5"
```

---

Dažas operācijas ar teksta rindiņām.

```
String s = "Hello, user!";
```

```
System.out.println(s.startsWith("Hello, "));  
//true
```

```
System.out.println(s.endsWith("user!"));  
//true
```

Darbs ar objektiem: *norāžu* piešķire.

```
class IntVal {  
    public int Key;  
    public IntVal(int Key) {  
        this.Key=Key;  
    }  
}
```

---

```
IntVal IV1, IV2;  
IV1 = new IntVal(1);  
IV2 = IV1;  
System.out.println(IV1.Key + " " + IV2.Key); //1 1  
IV1.Key = 2;  
System.out.println(IV1.Key + " " + IV2.Key); //2 2
```

Darbs ar teksta rindiņām: *vērtību* piešķire.

```
String S1, S2;
```

```
S1 = "1";
```

```
S2 = S1;
```

```
System.out.println(S1 + " " + S2); //1 1
```

```
S1 = "2";
```

```
System.out.println(S1 + " " + S2); //2 1
```

---

Komentārs: pēc piešķires `S1 = "2"` tika izveidots jauns objekts `"2"`.

Tagad `S2` norāda uz veco objektu `"1"`.



Teksta rindiņu modifikācija. Klase *StringBuffer*.

```
String S = "Good.";
StringBuffer SB = new StringBuffer(S);
SB.insert(4, " morning");
SB.insert(12, ", user");
SB.setCharAt(18, '!');
S = SB.toString();
System.out.println(S);
```

Rezultāts:

Good morning, user!

## Informācijas ievade/izvade (Java 2, versija 5.0)

Klase *Formatter*: informācijas izvade.

```
import java.util.*;
...
Formatter f = new Formatter();
int N = 5;
for(int i=1; i<=N; i++) {
    f.format("i:%1d. i*i:%d\n", i, i*i); // (*)
}
System.out.println(f);
```

---

```
i:1. i*i:1
i:2. i*i:4
i:3. i*i:9
i:4. i*i:16
i:5. i*i:25
```

Informācijas izvade: alternatīvais risinājums.

Rindiņa (\*) no iepriekšējā piemēra:

```
System.out.printf("i:%1d. i*i:%d\n", i, i*i);
```

---

Klases *Formatter* objekts tiks izveidots automātiski.

---

Izvadāmās informācijas formatēšana:

```
double D = 15.675;
```

```
System.out.printf("%f\n%.2f\n%.1f", D, D, D);
```

Rezultāti:

15,675000

15,68

15,7

Informācijas ievade: klase *Scanner*.

Lasīšana no tastatūras: tiks nolasīti trīs vektora elementi.

```
Scanner Sc = new Scanner(System.in);  
  
double [] DVect = new double[3];  
  
for(int i=0; i<DVect.length; i++)  
    DVect[i] = Sc.nextDouble();
```

---

Lasāmo elementu daudzums *nav zināms*:

```
ArrayList<Double> AL = new ArrayList<Double>();  
  
while (Sc.hasNextDouble())  
    AL.add(Sc.nextDouble());  
}
```

Ievadāmās informācijas analīze: atrast *veselu* skaitļu un *reālu* skaitļu daudzumu.

```
Scanner Sc = new Scanner(System.in);  
int iCount=0, dCount=0;  
for (;;) {  
    if (Sc.hasNextInt()) {  
        Sc.nextInt();  
        iCount++;  
    }  
  
    else if (Sc.hasNextDouble()) {  
        Sc.nextDouble();  
        dCount++;  
    }  
  
    else  
        break;  
}
```

## Metožu *parametru vērtību* izmaiņa

Var izmainīt *tikai objektu*.

1. Klase ar vienīgo atribūtu.

```
class MyNum {  
    public int Num;  
}
```

---

2. Metode.

```
public static void change (MyNum X) {  
    X.Num++;  
}
```

### 3. Metodes pielietošana.

```
int x = 6;  
System.out.println(x); // x = 6  
  
MyNum N = new MyNum();  
N.Num = x;  
  
change(N);  
  
x = N.Num;  
System.out.println(x); // x = 7
```

---

Nebūs izmaiņu:

```
public static void changeInt(int X) {  
    X++;  
}
```

Nepalīdzēs arī klases *Integer* izmantošana.

1. Metodes *change()* realizācija.

```
public static void change(Integer X) {  
    X = new Integer(X.intValue() + 1);  
}
```

---

2. Metodes *change()* izsaukums.

```
Integer I = new Integer (6);  
System.out.println(I);    // x = 6
```

```
change (I) ;
```

```
System.out.println(I);    // x = 6
```



## Abstraktās klases

1. Nav iespējams izveidot abstraktās klases *objektu*.
2. Ja klasē ir *vismaz viena* abstraktā metode, visu klasi obligāti deklarē *kā abstrakto*.
3. Abstraktajā klasē var būt arī *neabstraktās* metodes.
4. Mantojot no abstraktās klases un realizējot abstraktās metodes, iegūsim “parasto” klasi.

Lai ir abstraktā klase `ClosedFigure`.

Klasei `ClosedFigure` ir atribūts `Id` (objekta identifikators).

Ir paredzēta abstraktā metode `area()` figūras platības izskaitļošanai.

```

abstract class ClosedFigure {
    private String Id;
    public ClosedFigure(String Id) {
        this.Id = Id;
    }
    abstract public double area();
}

class Square extends ClosedFigure {
    private double a;
    ...
    public double area() {
        return a*a;
    }
}

Square S = new Square("S", 3);
System.out.println(S.area()); //9.0
    
```

Abstraktās klases *garantē*, ka prasāmas metodes tiks *obligāti pārdefinētās apakšklasēs*.

Abstraktās klases *norādes* var izmantot darbā ar apakšklašu objektiem.

```
ClosedFigure CF;  
CF = S;    // jau izveidotais kvadrāts  
System.out.println(CF.area());    // 9.0
```

**Nevar izveidot:**

1. Abstrakto **konstruktoru**.
2. Abstrakto **statisko** metodi.
3. Abstrakto **privāto** metodi.

Operatoru **break** un **continue** paplašināšana:

**break** <iezīme> un **continue** <iezīme>

Uzdevums: atrast *pirmā* negatīvā skaitļa pozīciju divdimensiju masīvā.

```
R = C = -1;
```

```
Rows:
```

```
for (int i=0; i<M.length; i++) {
```

```
    Cols:
```

```
    for (int j=0; j<M[i].length; j++)
```

```
        if (M[i][j] < 0) {
```

```
            R=i;
```

```
            C=j;
```

```
            break Rows;
```

```
        }
```

```
    }
```

Funkcijā var būt *mainīgais* parametru daudzums.

*Uzdevums:* lai funkcijai `sum ( . . . )` ir *vairāki* **int**-parametri.  
Atrast parametru summu.

```
static int sum(int ... v) {  
    int S=0;  
    for (int Elem: v) {  
        S += Elem;  
    }  
    return S;  
}  
  
...  
System.out.println(sum(1, 2, 3)); //6
```

1. Parametru `varargs` apstrādā *kā masīvu*.
2. Parametrs `varargs` var būt *tikai viens*.
3. Parametrs `varargs` vienmēr ir *pēdējais*.

## Java sīklietotņu pamati

Sīklietotnes (*applets*) – nelielie lietojumi Java valodā, kuras izpildās pārlūkprogrammā.

Sīklietotņu superklase ir *Applet*.

```
import java.applet.Applet;
```

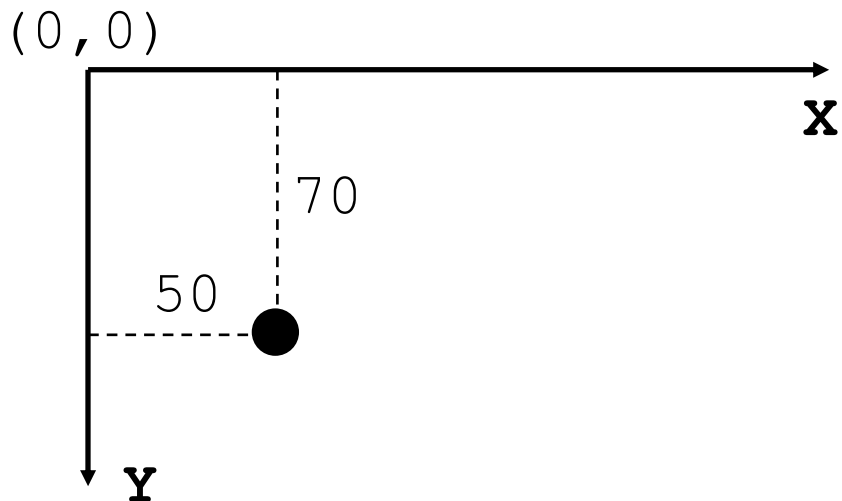
```
import java.awt.Graphics;
```

```
public class Hello extends Applet {  
    public void paint(Graphics g) {  
        g.drawString("Hello from Applet !", 50, 70);  
    }  
}
```

*Applet* nodrošina savstarpējo iedarbību ar pārlūkprogrammu.

*Graphics* realizē piekļuvi pārlūkprogrammas grafiskajai telpai.

## Java koordināšu sistēma (ekrāna pikseļi)



Klasē *Graphics* ir vairākas zīmēšanas metodes, fontu vadības metodes: *drawLine()*, *drawRect()*, utt.

Klase *Graphics* ir abstraktā klase, nevar izveidot *Graphics* objektu.

Nevar izveidot kādu vienu klasi, kas nodrošina grafiskas iespējas visās platformās (*Unix*, *Macintosh*, *Windows*, ...).

Java instalēšanas procesā tiks izveidota *Graphics* apakšklase konkrētajai platformai.

Parastos Java lietojumos var lietot kā komandrindas interfeisu, tā arī grafisko lietotāja interfeisu (GUI).

Sīklietotnēs var izmantot tikai GUI.

Sīklietotnēs nav *main()* metodes.

Sīklietotni var arī palaist bez informācijas importēšanas.

```
public class Hello extends java.applet.Applet {  
    public void paint (java.awt.Graphics g) {  
        g.drawString("Hello from Applet !",  
            50, 70);  
    }  
}
```



Visbiežāk izmanto arī citas klases no divām pakotnēm

```
import java.applet.*;
```

```
import java.awt.*;
```

Sīklietotnes *Hello.java* kompilācijas rezultāts: fails *Hello.class*.

---

Sīklietotnes palaišana: tīmekļa lappuse *Hello.htm*.

```
<html>
```

```
  <body>
```

```
    <applet code="Hello" height="200"
      width="300">
    </applet>
```

```
  </body>
```

```
</html>
```

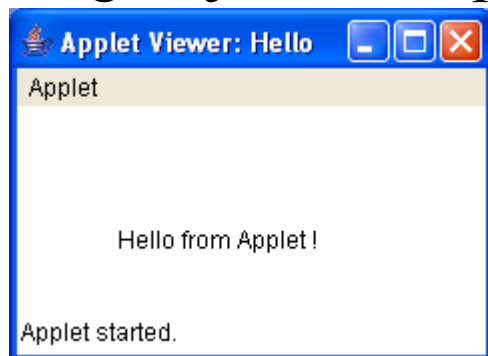
**code** – klases vārds. Var rakstīt: *Hello.class*.

**height**, **width** – sīklietotnes apgabala izmēri

## Fotogrāfija no pārlūkprogrammas *Internet Explorer*

Hello from Applet !

## Fotogrāfija no utilītprogrammas *Applet Viewer*



← appletviewer.exe

*Java* atbalsta pārlūkprogrammā pārbaude

`<applet code="..." ...>`

Your browser doesn't support Java.

`</applet>`

Mūsdienās ieteicams izmanto elementu `<object>`, nevis `<applet>`.

Klasē *Color* definētas konstantes un metodes krāsu vadībai.

1. Statiskie atribūti: **public final static** *Color.x*

a. Sīklietotnes fons (var norādīt metodē *init()* )

```
setBackground(Color.gray);
```

b. Teksta krāsa

```
g.setColor(Color.red);
```

```
g.drawString("Red Color !", 20, 20);
```

2. Konstruktors *Color(int R, int G, int B)*. Vērtības: 0 - 255.

```
Color MyColor = new Color(255, 0, 0);
```

```
g.setColor(MyColor);
```

3. Konstruktors *Color(float R, float G, float B)*. Vērtības: 0.0 – 1.0.

```
Color MyColor = new Color(  
    (float) 1.0, (float) 0.0, (float) 0.0);
```

```
g.setColor(MyColor);
```

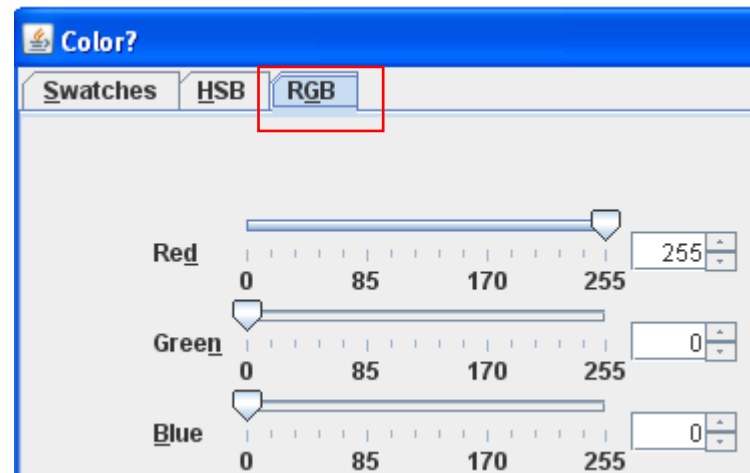
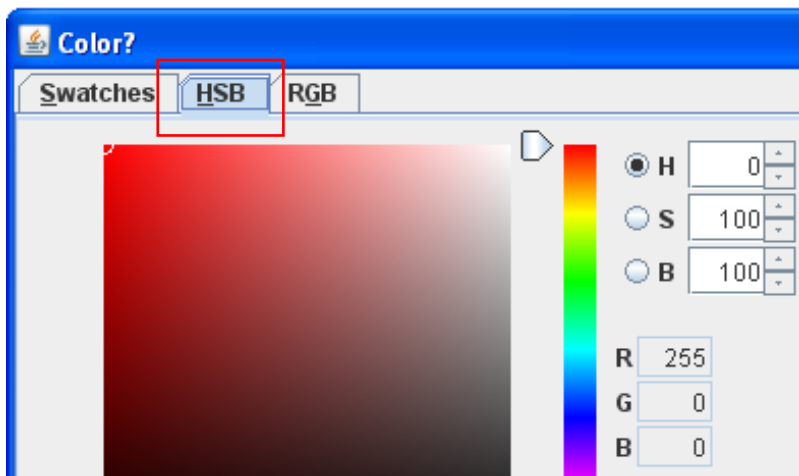
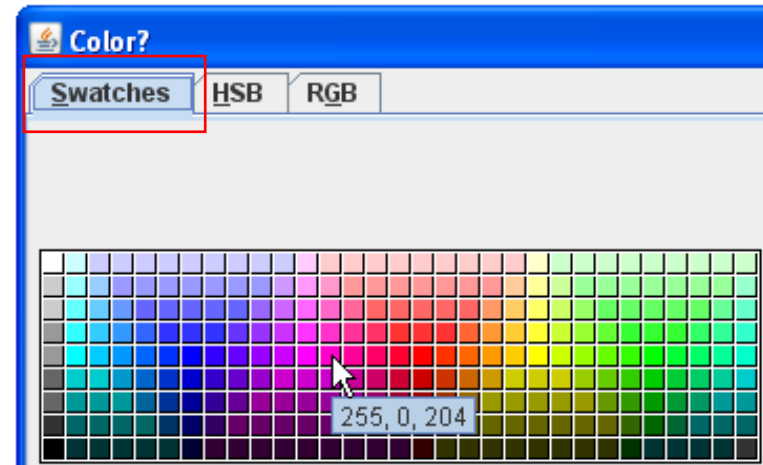
## Krāsas izvēle dialogā

```
import java.applet.*;  
import java.awt.*;
```

```
import javax.swing.*;
```

```
public class Hello extends Applet {  
    public void init() {  
        Color C = Color.red; //kāda inicializācija  
        C = JColorChooser.showDialog(  
            Hello.this, "Color?", C);  
        if (C == null)  
            C = Color.gray;  
        setBackground(C);  
    }  
}
```

## Krāsas izvēle: dialoga fragmenti



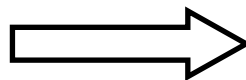
Parametru *nodošana* sīklīetotnei no tīmekļa lappuses.

```
<applet code="..." ...>  
  <param name="Author" value="Juris Strods">  
</applet>
```

Parametru *iegūšana* sīklīetotnē.

```
public class Hello extends Applet {  
    String PAuthor;  
  
    public void init() {  
        PAuthor = getParameter("Author");  
    }  
  
    public void paint(Graphics g) {  
        g.drawString("Hello from Applet !", 50, 70);  
        g.drawString("Author: " + PAuthor, 50, 100);  
    }  
}
```

Rezultāts



Hello from Applet!  
Author: Juris Strods

## Parametru *nodošana*: papildus aspekti

Ieteicams pārbaudīt katra parametra *eksistēšanu* (salīdzināt iegūto no tīmekļa lappuses vērtību ar **null**).

Ja kāda vērtība netika nodota, izmanto iepriekš paredzētas vērtības *pēc noklusēšanas*.

*Uzdevums*: ievadīt teksta ziņojumu. Parametri: teksts, fonts, simbolu izmērs.

Tīmekļa lappuses fragments:

```
<applet code="..." ...>
  <param name="Message" value="Hello, users !">
  <param name="TextFont" value="Courier New">
  <param name="TextSize" value="18">
</applet>
```

## Sīklīetošanas kods

```
import java.applet.*;
import java.awt.*;

public class Mess extends Applet {
    // Atribūti: teksts, fonts, teksta izmērs
    String Msg, Fnt;
    int TxtSz;

    public void init() {
        String Temp;

        //Teksta ziņojuma eksistēšanas pārbaude
        if ( (Msg = getParameter("Message")) == null)
            Msg = "Hello, World !";

        //Fonta eksistēšanas pārbaude
        if ( (Fnt = getParameter("TextFont")) == null)
            Fnt = "Arial";
    }
}
```



```
//Teksta izmēra eksistēšanas pārbaude
//un datu tipa pārveidošana
Temp = getParameter("TextSize");
if (Temp == null)    TxtSz = 12;
    else TxtSz = Integer.parseInt(Temp);

//Fonta veidošana
Font F = new Font(Fnt, Font.BOLD, TxtSz);
setFont(F);
}

//ziņojuma izvade
public void paint(Graphics g) {
    g.drawString(Msg, 100, 100);
}
}
```

Rezultāts  **Hello, users !**

## Sīklīetošanas metodes un dzīves cikls

1. *Konstruktors*. Satur objekta inicializācijas kodu.
2. *init()*. Izpildās pēc konstruktoru izsaukuma. Bieži *aizvieto* konstruktoru.
3. *start()*. Izpildās pēc inicializācijas un pēc atgriešanas no cita Windows loga.
4. *paint(Graphics g)*. Izpildās pēc jebkurām izmaiņām grafiskajā telpā. Visbiežāk izpildāmā metode.
5. *stop()*. Izpildās pēc pārejas citā logā vai loga minimizēšanas.
6. *destroy()*. Destrukta (finalizatora) ekvivalents.

Sīklīetotnes metožu demonstrēšana.

Informāciju izvada *pārlūkprogrammā* un *konsole*.

```
import java.applet.*;
import java.awt.*;

public class Hello extends Applet {
    int constrCount = 0;
    int initCount = 0;
    int startCount = 0;
    int paintCount = 0;
    int stopCount = 0;

    // Konstruktors
    public Hello() {
        System.out.println("Constructor !");
        constrCount++;
    }
}
```

```
// Metode init()  
public void init() {  
    System.out.println("Init !");  
    initCount++;  
}  
  
// Metode start()  
public void start() {  
    System.out.println("Start !");  
    startCount++;  
}  
  
// Metode stop()  
public void stop() {  
    System.out.println("Stop !");  
    stopCount++;  
}
```

```
// Metode paint()
public void paint(Graphics g) {
    System.out.println("Paint !");
    paintCount++;
    g.drawString("Constructor called:" +
        constrCount, 50, 20);
    g.drawString("Init called: " + initCount,
        50, 40);
    g.drawString("Start called: " + startCount,
        50, 60);
    g.drawString("Paint called: " + paintCount,
        50, 80);
    g.drawString("Stop called: " + stopCount,
        50, 100);
}
}
```

## *Statusa rindiņas adresēšana*

```
public void paint(Graphics g) {
    showStatus("Please, wait...");
    ...
}
```

Please, wait...

## *Sīklietotnes konteksta noteikšana un dokumentu ielāde*

```
...
import java.net.*;
```

Metodes `init(...)` saturs:

```
AppletContext ac = getAppletContext();
URL url = getCodeBase();
```

Abu objektu saturs - `toString()` rezultāts:

```
// ac = sun.plugin.viewer.context.IExplorer...
// url = file:/D:/WORK/JAVA
```

```
try {
    ac.showDocument(new URL(url + "info.htm"));
}
catch (MalformedURLException e) {}
```

Jaunu dokumentu var atvērt *citā logā*:

```
ac.showDocument(
    new URL(url + "info.htm"), "_blank");
```

Iespējamās parametra vērtības: `_self`, `_parent`, `_top`, `_blank`.

*Piezīme:* `showDocument()` ir interfeisa `AppletContext` metode.

Var noteikt *tīmekļa lappuses* bāzi:

```
URL url = getDocumentBase();
```

Rezultāts:

```
//file:/D:/WORK/JAVA/Test.html
```

## Notikumu apstrāde

*Uzdevums:* saskaitīt peles klikšķinājumus un izvadīt rezultātu teksta logā.

1. Novecojušā shēma.

```
import java.applet.*;
import java.awt.*;

public class Test extends Applet {
    private int Counter = 0;
    private TextField T = new TextField(" 0 ");

    //pogas un teksta loga izvietošana
    public void init() {
        T.setEditable(false);
        add(new Button("Click me"));
        add(T);
    }
}
```



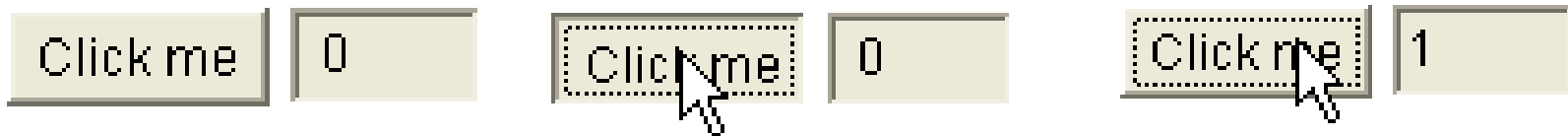
```
public boolean action(Event e, Object arg) {  
    if ( ((Button) e.target).getLabel() ==  
        "Click me") {  
        Counter++;  
        T.setText("" + Counter);  
    }  
    return true;  
}
```

## Kompilatora ziņojums

Note: test.java uses or overrides a deprecated API.

Note: Recompile with -deprecation for details.

## Rezultāti



## 2. Modernā shēma: notikumu noklausīšana

```
import java.applet.*;
```

```
import java.awt.*;
```

```
import java.awt.event.*;
```

```
public class Test extends Applet {
```

```
    private int Counter = 0;
```

```
    private TextField T = new TextField(" 0 ");
```

```
    private Button B = new Button("Click me");
```

```
// Klase notikumu noklausīšanai
```

```
// Tika realizēts interfeiss ActionListener
```

```
class Click implements ActionListener {
```

```
    public void actionPerformed(ActionEvent e) {
```

```
        Counter++;
```

```
        T.setText("" + Counter);
```

```
    }
```

```
}
```

```
// Pogai pievienots notikumu klausītājs
public void init() {
    B.addActionListener(new Click());
    add(B);
    T.setEditable(false);
    add(T);
}
}
```

---

Piezīme: var izveidot *anonīmo* objektu bez klases *Click*

```
public void init() {
    B.addActionListener(
        new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                Counter++;
                T.setText("" + Counter);
            }
        }
    );
}
```

Interfeisa *MouseListener* lietošana: izmaiņas klasē *Click*

```
class Click implements MouseListener {
    public void mouseClicked(MouseEvent e) {
        Counter++;
        T.setText("" + Counter);
    }
    public void mouseExited(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}
    public void mousePressed(MouseEvent e) {}
}

public void init() {
    B.addMouseListener(new Click());
    ...
}
```

Piezīme: bija arī iespējams apstrādāt notikumus *mousePressed()* un *mouseReleased()*.

Ir iespēja strādāt ar speciālām klasēm – adapteriem.  
Tad nevajag “fiktīvi” realizēt nevajadzīgas metodes.

```
public void init() {  
    B.addMouseListener(new MouseAdapter() {  
        public void mouseClicked(MouseEvent e) {  
            Counter++;  
            T.setText("" + Counter);  
        }  
    });  
    add(B);  
    ...  
}
```

Atbilstība starp adapteriem un realizējamajiem interfeisiem:

MouseAdapter → MouseListener

MouseMotionAdapter → MouseMotionListener

KeyAdapter → KeyListener

WindowAdapter → WindowListener

## Swing bibliotēkas komponentu lietošana

TextField -> **J**TextField      Button-> **J**Button

```
import java.applet.*;  
import java.awt.*;  
import java.awt.event.*;  
import javax.swing.*;
```

```
public class Test extends Applet {  
    ...  
    private JTextField JT=new JTextField(" 0 ");  
    private JButton JB=new JButton("Click me");  
    ...  
}
```

Rezultāts

Click me

0

Click me

0

Click me

1

*Uzdevums:* palielināt skaitītāju, ja nospiests '+' un samazināt, ja nospiests '-'.

```
import javax.swing.*;
import java.applet.*;
import java.awt.event.*;

public class Hello extends Applet
    implements KeyListener {

    private int Counter = 0;
    private JLabel JL = new JLabel("Counter: ");
    private JTextField JT = new JTextField
        (" 0 ");

    public void keyPressed(KeyEvent KE) {

        char C = KE.getKeyChar();
        switch (C) {
```

```
    case '+':  
        Counter++;break;  
    case '-':  
        Counter--;break;  
}  
JT.setText("" + Counter);  
}
```

```
public void keyReleased(KeyEvent KE) {}
```

```
public void keyTyped(KeyEvent KE) {}
```

```
public void init() {  
    addKeyListener(this);  
    JT.setEditable(false);  
    add(JL);  
    add(JT);  
}
```

```
}
```



Var mantot no klases *JApplet*, nevis *Applet*.

Tad **obligāti** jāpielieto izvietojšanas menedžeri.

Importēšana no pakotnes *java.applet* nav vajadzīga.

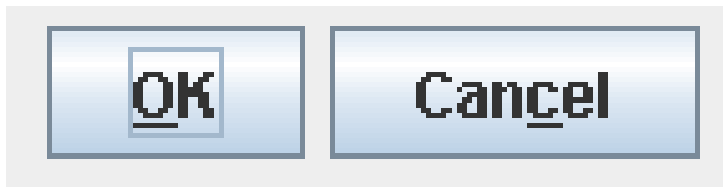
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Hello extends JApplet {
    ...
    public void init() {
        // izvietojšanas menedžera pielietošana
        setLayout(new BorderLayout());
        ...
    }
}
```

## Aktīvo taustiņu veidošana

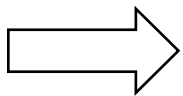
```
JButton OK = new JButton("OK"),  
    Cancel = new JButton("Cancel");  
setLayout(new FlowLayout());  
  
OK.setMnemonic('O');  
add(OK);  
  
Cancel.setMnemonic('c');  
Cancel.setDisplayedMnemonicIndex(3);  
add(Cancel);
```

Pogai *Cancel* pasvītrots simbols ar indeksu 3.



## Elementāras operācijas ar *teksta iezīmēm* un *paneļiem*

Rezultāts



**Bold 20** *Italic 16*

```
JPanel JP = new JPanel();
JLabel JL1 = new JLabel(" Bold 20 "),
JL2 = new JLabel("Italic 16");
public void init() {
    JP.setBackground(Color.red);
    JL1.setFont(new Font("Arial", Font.BOLD, 20));
    JL1.setForeground(Color.yellow);
    JP.add(JL1);
    JL2.setFont(new Font("Courier",
        Font.BOLD + Font.ITALIC, 16));
    JL2.setForeground(Color.black);
    JP.add(JL2);
    add(JP);
}
```

## Pakotne AWT (*Abstract Windowing Toolkit*)

Java valodas izstrādes sākumā – vienīga pakotne grafiskā interfeisa realizēšanai.

AWT komponenti ir *smagie* komponenti.

Tie balstās uz noteiktās platformas logu sistēmu.

Katram AWT komponentam ir partneris (*peer*) – objekts no OS GUI.

Partneris atbild par savstarpēju iedarbību starp komponentu un platformu.

Ta, objektam Java - programmā atbilst objekts Windows vidē.

## Pakotne Swing

Java 2 izmanto *vieglus* (“*atvieglotus*”) komponentus.

Šajā gadījumā nav vajadzības radīt papildus objektus.

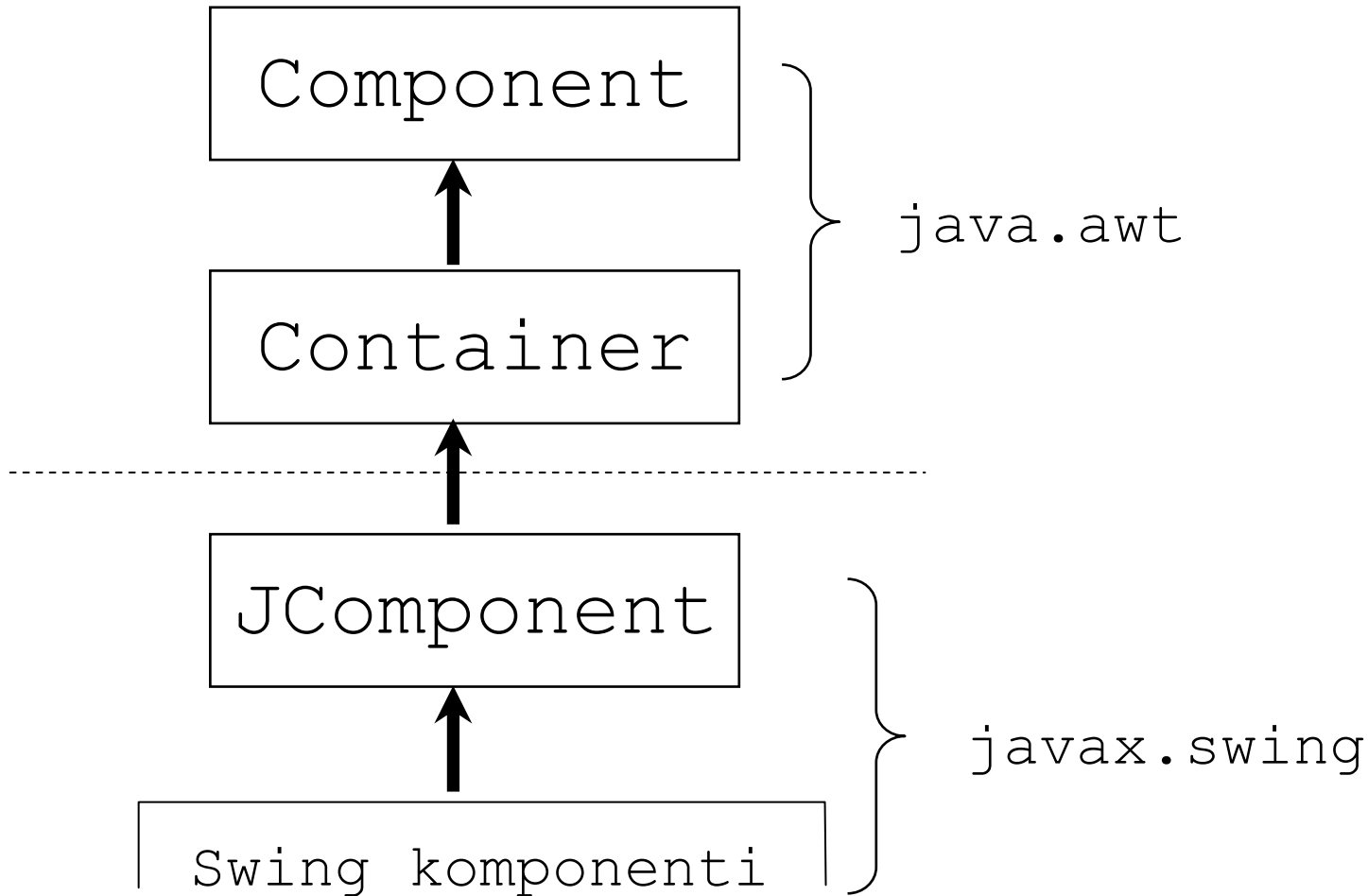
Vieglie komponenti tika izstrādāti firmā Sun, projekta “Swing” ietvaros. Līdz ar to, tos sauc par Swing komponentiem.

Komponentus piegādā pakotnē *javax.swing*.

Vairāki Swing komponenti pilnīgi uzrakstīti Java valodā un funkcionē zem Java vadības (tīrie Java komponenti).

Daži Swing komponenti tomēr ir smagie komponenti (*JApplet*, *JFrame*) .

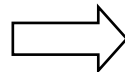
## Swing bibliotēkas radīšana



## Izvēles rūtiņu apstrāde

```
JCheckBox Pascal=new JCheckBox("Pascal", true),
    Cpp=new JCheckBox("C++"),
    Java=new JCheckBox("Java");
JButton Accept=new JButton("Accept");
setLayout(new FlowLayout());
add(Pascal); add(Cpp); add(Java);
add(Accept);
Cpp.setSelected(true);
Accept.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(null,
            "Pascal:" + Pascal.isSelected() +
            "\nC++:" + Cpp.isSelected() +
            "\nJava:" + Java.isSelected());
    }
});
```

Rezultāts



Pascal



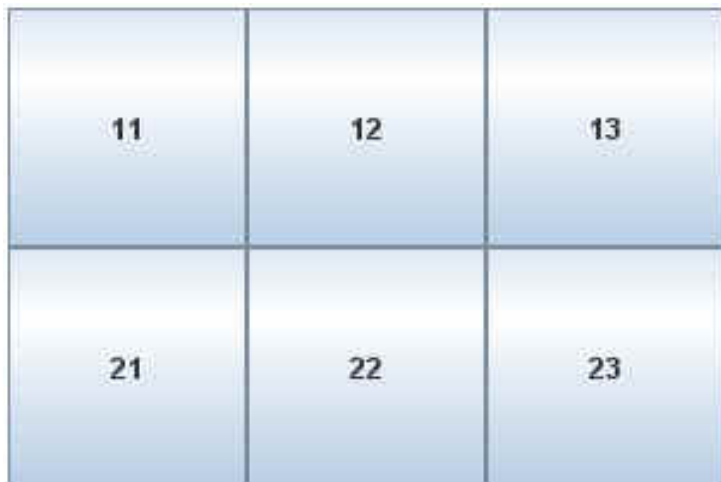
C++



Java

Accept

## Menedžeris *GridLayout*



JButton

```
JB11 = new JButton("11"),
JB12 = new JButton("12"),
JB13 = new JButton("13"),
JB21 = new JButton("21"),
JB22 = new JButton("22"),
JB23 = new JButton("23");
```

```
public void init() {
    setLayout(new GridLayout(2, 3));
    add(JB11); add(JB12); ...; add(JB23);
}
```

Konstruktorā parametri: rindiņu daudzums, kolonu daudzums.

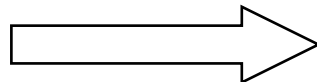
```
setLayout(new GridLayout(2, 3, 20, 40));
```

Divi papildu parametri: horizontālais un vertikālais attālumi starp kontroles elementiem.



## Opciju grupas apstrāde

Ieplānotais rezultāts



The image shows a standard Java Swing window with a title bar. Inside, there is a vertical stack of three radio buttons. The first is labeled 'Pascal', the second 'C++' (which is selected, indicated by a black dot in the center of the circle), and the third 'Java'. To the right of these radio buttons is a rectangular button labeled 'Accept'.

```
// Tika pielietots izvietojšanas menedžeris
// GridLayout (tabula)
```

```
JPanel JP = new JPanel(
    new GridLayout(3, 1, 0, 5));
```

```
JRadioButton
```

```
Pascal = new JRadioButton("Pascal"),
```

```
Cpp = new JRadioButton("C++", true),
```

```
Java    = new JRadioButton("Java");
```

```
ButtonGroup BG = new ButtonGroup();
```

```
JButton Accept = new JButton("Accept");
```

```
// Opciju veidošana un pievienošana
setLayout(new FlowLayout());
BG.add(Pascal); JP.add(Pascal);
BG.add(Cpp); JP.add(Cpp);
BG.add(Java); JP.add(Java);

add(JP); add(Accept);

// Notikuma apstrāde
Accept.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(null,
            "Pascal: " + Pascal.isSelected() +
            "\nC++: " + Cpp.isSelected() +
            "\nJava: " + Java.isSelected()
        );
    }
});
```

Uzdevums: ir trīs izvēles rūtiņas (krāsas *Red*, *Green*, *Blue*).

Uzstādīt sīklietotnes fonu pēc lietotāja izvēles.



Uzdevumā tiks izmantots interfeiss *ItemListener*.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Hello extends Applet {
    private JCheckBox Red, Green, Blue;

    class StateHandler implements ItemListener {
        float R, G, B;
```

```
public void itemStateChanged(ItemEvent e)
{
    if (e.getSource() == Red)
        R = (e.getStateChange() ==
            ItemEvent.SELECTED) ? 1 : 0;

    if (e.getSource() == Green)
        G = (e.getStateChange() ==
            ItemEvent.SELECTED) ? 1 : 0;

    if (e.getSource() == Blue)
        B = (e.getStateChange() ==
            ItemEvent.SELECTED) ? 1 : 0;

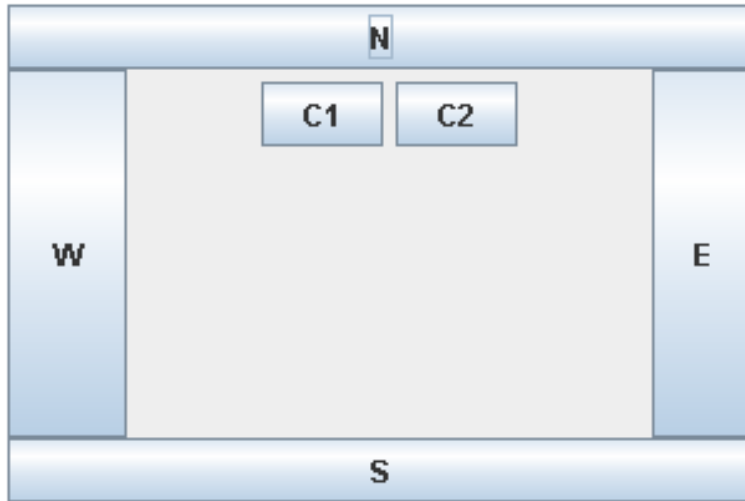
    setBackground(new Color(R, G, B));
}
}
```

```
public void init() {  
    Red = new JCheckBox("Red");  
    Green = new JCheckBox("Green");  
    Blue = new JCheckBox("Blue");  
  
    StateHandler SH = new StateHandler();  
  
    Red.addItemListener(SH); add(Red);  
    Green.addItemListener(SH); add(Green);  
    Blue.addItemListener(SH); add(Blue);  
}  
}
```

Piezīme: metodi *itemStateChanged(ItemEvent e)* var saīsināt.

```
R = (Red.isSelected()) ? 1 : 0;  
G = (Green.isSelected()) ? 1 : 0;  
B = (Blue.isSelected()) ? 1 : 0;  
setBackground(new Color(R, G, B));
```

## Izvietojšanas menedžeris *BorderLayout*



Telpai ir 5 sastāvdaļas

1. *North* – ziemeļi.
2. *South* – dienvidi.
3. *West* – rietumi.
4. *East* – austrumi.
5. *Center* – centrs.

```
JButton JB1 = new JButton("N"),  
        JB2 = new JButton("S"),  
        JB3 = new JButton("W"),  
        JB4 = new JButton("E"),  
        JB5 = new JButton("C1"),  
        JB6 = new JButton("C2");  
JPanel Center = new JPanel();
```

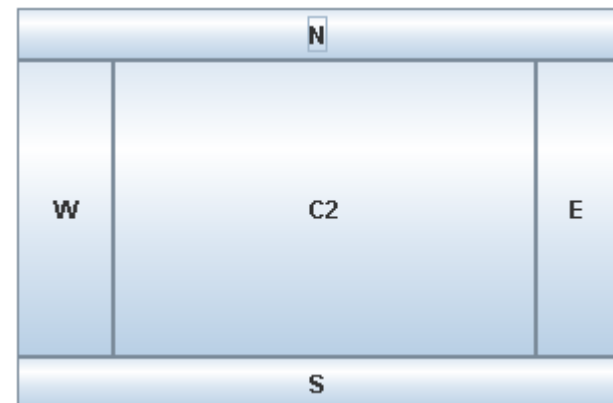
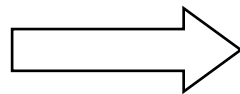
```
setLayout(new BorderLayout());  
add("North", JB1);  
add("South", JB2);  
add("West", JB3);  
add("East", JB4);  
Center.add(JB5);  
Center.add(JB6);  
add("Center", Center);
```

---

Piezīme: centrālais panelis ir nepieciešams.

Citādi iegūsim rezultātu:

```
add("Center", JB5);  
add("Center", JB6);
```



*Uzdevums:* imitēt statusa rindiņu.



```
JLabel JL1 = new JLabel("First"),
    JL2 = new JLabel("Second"),
    JL3 = new JLabel("Third");
JPanel JP = new JPanel();
public void init() {
    setLayout(new BorderLayout());
    setBackground(Color.green);
    JP.setLayout(new GridLayout(1, 3));
    JP.add(JL1); JP.add(JL2); JP.add(JL3);
    add("South", JP);
}
```

Cita iespēja pievienot paneli (statiskās konstantes):

```
add(BorderLayout.SOUTH, JP);
```



## Menedžeris *BoxLayout*

*Uzdevums:* izvietot vadības elementus stabiņā.



```

JButton JB11 = new JButton("11"),
        JB21 = new JButton("21"),
        JB31 = new JButton("31");

JPanel JP = new JPanel();
    
```

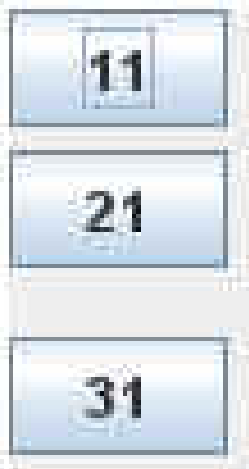
```

public void init() {
    BoxLayout BL = new BoxLayout(JP,
        BoxLayout.Y_AXIS);

    JP.setLayout(BL);
    JP.add(JB11); JP.add(JB21); JP.add(JB31);
    add(JP);
}
    
```

`BoxLayout.X_AXIS`: elementu rindīņa.

## Attālums starp vadības elementiem menedžerī *BoxLayout*



```
JP.add(JB11);
```

```
JP.add(Box.createVerticalStrut(5));
```

```
JP.add(JB21);
```

```
JP.add(Box.createVerticalStrut(15));
```

```
JP.add(JB31);
```



```
BoxLayout BL = new BoxLayout(JP, BoxLayout.X_AXIS);
```

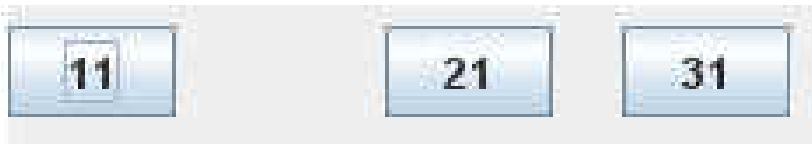
```
JP.add(JB11);
```

```
JP.add(Box.createHorizontalStrut(5));
```

```
JP.add(JB21);
```

```
JP.add(Box.createHorizontalStrut(15));
```

```
JP.add(JB31);
```



```
BoxLayout BL = new BoxLayout(JP,  
    BoxLayout.X_AXIS);  
JP.add(JB11);
```

```
JP.add(Box.createRigidArea(new Dimension(60, 0)));  
JP.add(JB21);
```

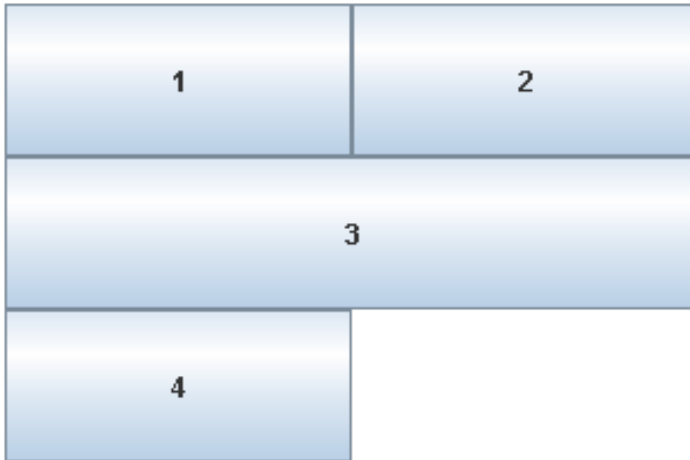
```
JP.add(Box.createRigidArea(new Dimension(20, 0)));  
JP.add(JB31);
```



```
JP.add(JB11);  
JP.add(Box.createHorizontalGlue());  
JP.add(JB21);  
JP.add(Box.createHorizontalGlue());  
JP.add(JB31);
```

## Izvietojšanas menedžera *GridBagLayout* pamati

Ieplānotais rezultāts



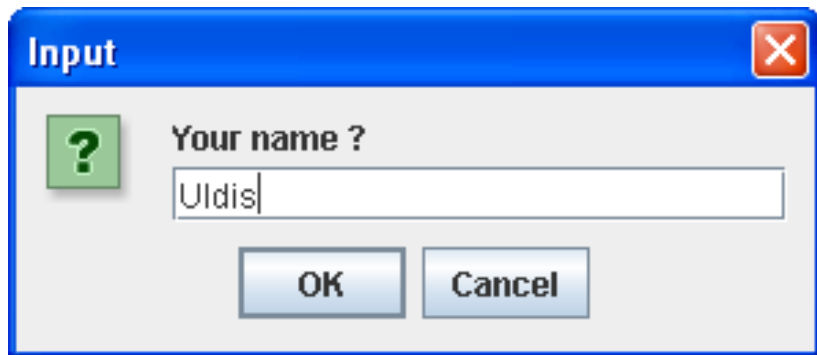
Sīklīktnes fragments

```
//Četru pogu deklarācija. Menedžera izveidošana.  
//JButton JB1 = new JButton("1"), ...  
GridBagLayout GBL = new GridBagLayout();  
GridBagConstraints GBC = new  
    GridBagConstraints();
```

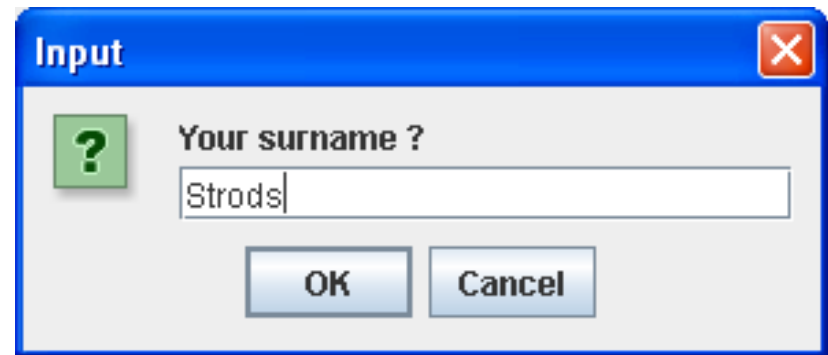
```
//Papildus ierobežojumi
GBC.fill = GridBagConstraints.BOTH;
GBC.weightx = 1;
GBC.weighty = 1;
//Pirmā rindiņa
setLayout(GBL);
GBL.setConstraints(JB1, GBC);
GBC.gridwidth = GridBagConstraints.REMAINDER;
add(JB1);
GBL.setConstraints(JB2, GBC);
add(JB2);
//Otrā rindiņa
GBL.setConstraints(JB3, GBC);
add(JB3);
//Trešā rindiņa
GBC.gridwidth = 1;
GBL.setConstraints(JB4, GBC);
add(JB4);
```

## Grafisko lietojumu izstrāde

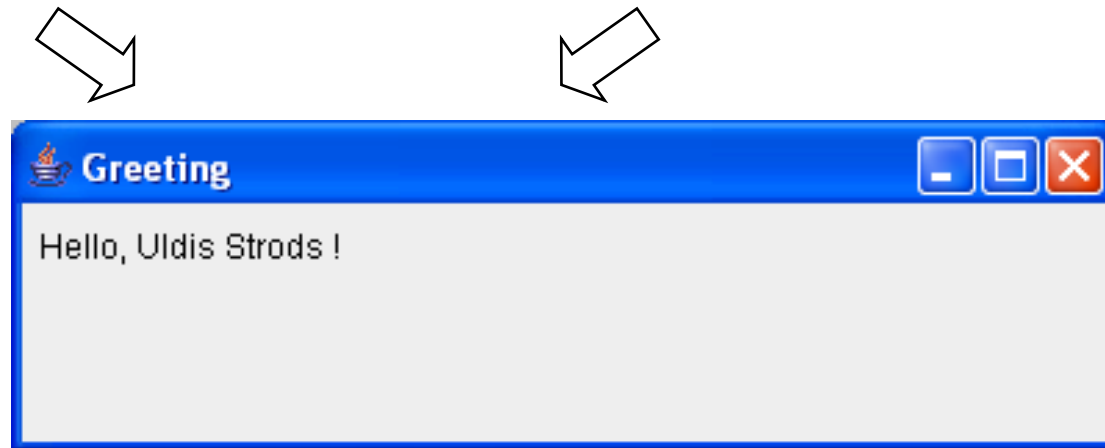
*Uzdevums:* ievadīt lietotāja vārdu un uzvārdu, pēc tam izvadīt šo informāciju ekrānā.



A Windows-style dialog box titled "Input" with a red close button. It contains a green question mark icon, the text "Your name ?", a text input field with "Uldis" entered, and "OK" and "Cancel" buttons at the bottom.



A Windows-style dialog box titled "Input" with a red close button. It contains a green question mark icon, the text "Your surname ?", a text input field with "Strods" entered, and "OK" and "Cancel" buttons at the bottom.



A Windows-style dialog box titled "Greeting" with a blue icon and standard window controls (minimize, maximize, close). It displays the text "Hello, Uldis Strods !" in a large text area.

```
import java.awt.Graphics;
import javax.swing.*;

public class Hello extends JFrame {
    String Name, Surname;

    public Hello() {
        super("Greeting"); // rāmīša virsraksts
        setSize(400, 120); // formas izmērs
        setVisible(true);
        setDefaultCloseOperation(
            JFrame.EXIT_ON_CLOSE);
    }

    public void paint(Graphics g) {
        super.paint(g);
        Name = JOptionPane.showInputDialog(
            "Your name ?");
    }
}
```

```
Surname = JOptionPane.showInputDialog(  
    "Your surname ?");  
g.drawString("Hello, " + Name + " " +  
    Surname + " !", 10, 50);  
}  
  
public static void main(String args[]) {  
    Test T = new Test();  
}  
}
```

Dažas citas *JFrame* vērtības:

DO\_NOTHING\_ON\_CLOSE – ignorēt loga aizvēršanu

HIDE\_ON\_CLOSE – paslēpt logu



*Uzdevums:* saskaitīt peles klikšķinājumus grafiskajā lietojumā.



```
import java.awt.*; // bez pakotnes „applet”
import java.awt.event.*;
import javax.swing.*;

public class Demo extends JFrame {
    int Counter = 0;

    private JTextField JTF =
        new JTextField(" 0 ");

    private JButton JB =
        new JButton("Click me");
```

```
Container c = getContentPane();
```

```
public Demo() {
```

```
    super("Mouse demo");
```

```
    c.setLayout(new FlowLayout());
```

```
    JB.addMouseListener(new MouseAdapter() {  
        public void mouseClicked(MouseEvent e) {  
            Counter++;  
            JTF.setText("" + Counter);  
        }  
    });
```

```
    c.add(JB);
```

```
    c.add(JTF);
```

```
    setSize(400, 120);
```

```
    ...
```

```
}
```

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

```
    ...
```

```
}
```

```
}
```

# Programmēšanas valoda **Python**

## Pielietošanas pamatprincipi

## Literatūras saraksts:

1. <http://www.python.org>
2. *Сузи Роман*. Python.  
Санкт-Петербург, BHV, 2002. 786 lpp.
3. *Лесса Андре*. Python. Руководство разработчика.  
Санкт-Петербург, ООО “ДиаСофтЮП”, 2001. 688 lpp.
4. *Лутц Марк*. Программирование на Python.  
Санкт-Петербург, издательство “Символ-Плюс”, 2002. 1136 lpp.
5. *Иван ван Лейнингем*. Освой самостоятельно Python за 24 часа.  
Вильямс, 2001. 448 lpp.

Valodas izstrādātājs: *Guido van Rossum*.

---

Izmanto:

- ✓ Teksta failu apstrādē.
  - ✓ Web programmēšanā.
  - ✓ Grafikā.
- 

Skripta failu paplašinājums: `*.py`.

## Elementārās programmas

Teksta ziņojuma izvade (fails *Out.py*)

```
print("Hello, World !")
```

Pēc `print` – pāreja uz nākamo rindiņu.

---

*Piezīme:* vecās *Python* versijās (līdz 3.0) var arī neizmantot apaļas iekavas.

```
print "Hello, World !"
```

---

**Nepareizi:**

```
print("Hello, World !")
    print("Hello, World !") # ir nepamatota atkāpe
```

**Pareizi:**

```
print("Hello, World !")
print("Hello, World !")
```

*Python* valodā operatorus *izlīdzina blokos*.

Nav figūriekavu vai “**begin ... end**”.

---

Elementārie aprēķini:

```
from math import *  
  
def CircleArea(r):  
    return pi*r**2  
  
#līdz 3.0 - 'int' automātiski  
r = int(input("Radius: "))  
  
print ("Area: ", CircleArea(r))
```

---

Rezultāts:

```
Radius: 2  
Area: 12.5663706144
```

## Operācijas ar *lieliem skaitļiem*:

```
# vecās versijās ierosina izņēmumu  
# pēc Python 3.0 ir vienīgi pareizs variants  
print(10**10)    #100000000000
```

---

```
print(10**10L)   #100000000000, der līdz Python 3.0
```

---

## Teksta rindiņu *pārnesums*:

```
print("Hello, "\n\nWorld !")    #Hello, World !
```

---

## *Daudzrindu komentāri*:

```
"""\n\nProgramming\nusing Python\n"""
```



Datu tipu pārveidošana: *veselie* skaitļi.

```
import locale                                #līdz 3.0: string
S = "11"
print(int(S) + 1)                            #12
print(locale.atoi(S) + 1)                 #12
print(eval(S) + 1)                          #12
```

---

Datu tipu pārveidošana: *reālie* skaitļi.

```
import locale                                #līdz 3.0: string
S = "11.5"
print(float(S) + 1)                          #12.5
print(locale.atof(S) + 1)                   #12.5
print(eval(S) + 1)                          #12.5
```

Parametru iegūšana no komandrindas. Masīvs *argv*.

```
from sys import *  
print("File name: ", argv[0])  
print("Total parameters: ", len(argv))  
print("Parameters: ")  
for P in argv[1:]:  
    print(P)
```

---

Programmas palaišana:

```
python.exe prog.py  dat1.txt dat2.txt
```

---

Rezultāts:

```
File name:  prog.py  
Total parameters:  3  
Parameters:  
dat1.txt  
dat2.txt
```

## Kontroles struktūras

Cikls **for**:

```
Sum = 0
for i in (1, 2, 3):
    Sum += i
print("Sum: ", Sum)      #Sum: 6
```

---

Funkcija *range()* ciklā **for**:

```
Sum = 0
for i in range(1, 4):
    Sum += i
print("Sum: ", Sum)      #Sum: 6
```

Cikls **while**:

```
i = Sum = 0
while (i<3) :
    i += 1
    Sum += i
print("Sum: ", Sum)      #Sum: 6
```

---

Ciklos izmanto arī mainīgos:

```
Sum=0
# vairāki operandi vienā rindiņā
Start=1;End=4;Step=1;

for i in range(Start, End, Step) :
    Sum += i
print("Sum: ", Sum)      #Sum: 6
```

Ciklos **for** un **while** izmanto arī operatoru **else**.

*Uzdevums:* atrast *pirmo negatīvo* skaitli kortežā (pretējā gadījumā rezultāts būs **1**).

```
L = (4, -1, 3)
for x in L:
    if x < 0:
        break
else:
    x = 1
print(x)    # -1
```

Zars **else** izpildās vienu reizi, ja nebija izpildīts **break**.

Sazarojums **if**: ievadītā skaitļa analīze.

```
Num = int(input("Numeric ?"))
```

```
if Num==0:
```

```
    print("Zero !")
```

```
elif Num<0:
```

```
    print("Negative !")
```

```
else:
```

```
    print("Positive !")
```

---

Sazarojums **if**: ievadīta skaitļa analīze. Diapazona pārbaude.

```
Min=1; Max=5; X=2
```

```
if Min <= X <= Max:
```

```
    print("OK !")
```

```
else:
```

```
    print("ERROR !")
```

## Korteži un saraksti (*lists and tuples*)

1. Sarakstu var *izmainīt*, bet kortežu - nē.
  2. Sarakstos izmanto *kvadrātiekavas*, bet kortežos - *apaļas iekavas*.
  3. Sarakstiem ir *metodes*, bet kortežiem - nav.
- 

Var norādīt:

- ✓ Elementu *pēc indeksa* (no abām malām).
- ✓ Elementus *līdz noteiktām indeksam*.
- ✓ Elementus *pēc noteiktā indeksa*.
- ✓ Elementu *diapazonu* (no ... līdz).

### *Elementu adresēšana:*

```
L= ("C++", "Java", "Perl", "Python")  
print(len(L)) #4  
print(L[0]) #C++  
print(L[-1]) #Python  
print(L[:2]) #('C++', 'Java')  
print(L[1:3]) #('Java', 'Perl')  
print(L[1:]) #('Java', 'Perl', 'Python')
```

---

### *Korteža daļas iegūšana:*

```
ServL = L[2:4]  
print(ServL[0], ServL[1]) #Perl, Python
```



Korteža elements var būt cits kortežs:

```
Triangle = ( (1, 2), (2, 3), (3, 2) )  
print(Triangle[0])      # (1, 2)  
print(Triangle[0][0])   # 1
```

---

Lai ir *saraksts*:

```
L = ["C++", "Java", "Perl", "Python"]
```

Tagad var *izmainīt* saraksta elementu:

```
L[0] = "C"      # ['C', 'Java', 'Perl', 'Python']
```

Elementa *ielikšana*:

```
L[0:0] = ["C++"]  
# ['C++', 'C', 'Java', 'Perl', 'Python']
```

Cita iespēja ielikt elementu (ar *to pašu* rezultātu):

```
L.insert(0, "C++")  
#['C++', 'C', 'Java', 'Perl', 'Python']
```

---

Elementu ielikšana *ar dzēšanu*:

```
L[1:2] = ["JavaScript", "VBScript"]  
#['C++', 'JavaScript', 'VBScript',  
  'Java', 'Perl', 'Python']
```

---

*Vairāku* elementu dzēšana:

```
L[1:3] = ["C"]  
#['C++', 'C', 'Java', 'Perl', 'Python']
```

Elementa *dzēšana* bez ielikšanas:

```
L[0:1] = []    # ['C', 'Java', 'Perl', 'Python']
```

Cita iespēja *izdzēst* elementu bez ielikšanas:

```
del L[0:1]      # vai del L[0]
```

---

Informācija par elementu *tipiem*:

```
L = [1, 2.5, "ab"]  
for El in L:  
    print(type(El), 3*El)
```

Rezultāts:

```
<class 'int'> 3  
<class 'float'> 7.5  
<class 'str'> ababab
```

*Piezīme:* vecās versijās rezultātā būtu **type**, nevis **class**.

Dažas sarakstu *metodes*:

1. *append(<Elem>)*. *Elementa pievienošana.*

```
L = ["C"]  
L.append("C++")           # ['C', 'C++']
```

Cita iespēja:

```
L[len(L) :] = ["C++"]     # ['C', 'C++']
```

---

2. *extend(<List>)*. *Saraksta pievienošana.*

```
L.extend(["C#"])          # ['C', 'C++', 'C#']
```

---

3. *pop([<n>])*. *Elementa izslēgšana.*

```
print(L.pop(2))           # C#  
print(L.pop())            # C++  
print(L)                  # ['C']
```

4. *count* (<*Elem*>) - atrast elementa *Elem* daudzumu.

```
L = ["C", "C++", "C#"]  
print(L.count("C#"))      #1
```

---

5. *remove* (<*Elem*>) - izdzēst *pirmo* elementu *Elem* no saraksta. Ja elementa nav, iznēmums *ValueError*.

```
L = ["C", "C++", "C#"]  
L.remove("C")      # ['C++', 'C#']
```

---

6. *index*(<*Elem*>) - atrast elementa indeksu sarakstā. Ja elementa nav, iznēmums *ValueError*.

```
print(L.index("C#"))      # 1
```

## Saraksta *kārtošana*

```
L = [2, 3, 1]
L.sort()           #[1, 2, 3]
L.reverse()        #[3, 2, 1]
```

---

Funkcija elementu kārtošanai *dilstošajā secībā*.

```
def Desc(a, b):
    if (a<b):
        return 1
    else:
        if (a>b):
            return -1
        return 0
```

```
...
L.sort(Desc)      #[3, 2, 1]
```

*Piezīme:* šis kods **ne**strādā jaunās *Python* versijās (pēc 3.0).

## *Iespējamais risinājums savietojamības nodrošināšanai*

### 1. Klase no *Python* dokumentācijas

```
def CmpToKey(mycmp) :
    'cmp function => key function'
    class K(object):
        def __init__(self, obj, *args):
            self.obj = obj
        def __lt__(self, other):
            return mycmp(self.obj, other.obj) == -1
        def __gt__(self, other):
            return mycmp(self.obj, other.obj) == 1
        def __eq__(self, other):
            return mycmp(self.obj, other.obj) == 0
        def __le__(self, other):
            return mycmp(self.obj, other.obj) != 1
        def __ge__(self, other):
            return mycmp(self.obj, other.obj) != -1
        def __ne__(self, other):
            return mycmp(self.obj, other.obj) != 0
    return K
```

## 2. Kārtošanas funkcijas *lietošana*

```
L.sort(key=CmpToKey(Desc))
```

Elementu kārtošana sarakstā *no kartežiem*.

Uzdevums: kārtot informāciju par mājas lapām *izmēru dilstošajā secībā*.

```
def Size(a, b):
    if a[1]<b[1]:
        return 1
    else:
        if a[1]>b[1]:
            return -1
    return 0
```

...

```
L = [ ("perl.htm", 1000), ("python.htm", 2000) ]
```

```
L.sort(key=CmpToKey(Size))
```

```
# [('python.htm', 2000), ('perl.htm', 1000)]
```



## Vērtību *apmaiņa*

```
a = 1; b = 2
print(a, " ", b)    #1    2

[a, b] = [b, a]      #var arī (a, b) = (b, a)
print(a, " ", b)    #2    1
```

## Cita iespēja:

```
a, b = b, a
```

---

## Formatētā informācijas izvade:

```
L = [1.5, 2.5, 4.5]
for i in range(0, len(L)):
    print("%1d.    => %3.1f" % (i+1, L[i]))

# 1.    => 1.5
# 2.    => 2.5
# 3.    => 4.5
```

## Saraksta *kopēšana*

```
L1 = ["C++"]
```

```
L2 = L1
```

```
print(L2) #['C++']
```

---

```
L1[0] = "Java"
```

```
print(L2) #['Java']
```

Rezultāts: ir *norāžu piešķire*.

---

## Informācijas kopēšana:

```
L2 = L1[:]
```

Pirms un pēc L1 modificēšanas iegūsim vienu un to pašu rezultātu:

```
print(L2) #['C++']
```

## Masīvi

```
from array import *  
Arr = array("i", [4, 5])
```

Parametri: masīva *elementu tips* (mūsu gadījumā integer) un *vērtību saraksts*.

---

Nav iespējams:

```
Arr[0] = 2.5 #kļūda  
Arr[0] = "a" #kļūda
```

---

Dažas operācijas:

```
Arr[0:0] = array("i", [2, 3]) #[2, 3, 4, 5]  
Arr.insert(0, 1)                #[1, 2, 3, 4, 5]  
Arr.append(6)                    #[1, 2, 3, 4, 5, 6]  
del Arr                          #izņēums NameError
```

## Vārdnīcas

Vārdnīcas deklarācija:

```
Soft = {"Unix": "OS", "Oracle": "DBMS",  
        "Windows": "OS"}
```

```
print(Soft)  
#{'Oracle': 'DBMS', 'Windows': 'OS',  
  'Unix': 'OS'}
```

---

Katru vārdnīcas elementu var norādīt *neatkarīgi*:

```
Soft = {}
```

```
Soft["Unix"] = "OS"
```

```
Soft["Oracle"] = "DBMS"
```

```
...
```

Sākumā *obligāti* deklarē tukšo vārdnīcu.

Elementa *vērtības* iegūšana (ja atslēgas nav, tiks ierosināts izņēmums *KeyError*).

```
print (Soft["Unix"]) #OS
```

---

Elementa vērtības iegūšana *ar atslēgas eksistēšanas pārbaudi*:

```
c = Soft.get("Unix")
```

```
if c==None:
```

```
    <elementa nav>
```

Var izmantot operatoru **not**:

```
if not(c): #ja "true", tad: if c:
```

---

Elementa *atslēgas* eksistēšanas pārbaude:

```
if Soft.has_key("Oracle"): #līdz Python 3.0
```

```
if "Oracle" in Soft: #pēc Python 3.0
```

Atslēgu *tipi*:

- ✓ Teksta rindiņa (vispopulārākais)
  - ✓ Veselie skaitli
  - ✓ Korteži
  - ✓ Reālie skaitļi
- 

Par atslēgām **nevar** būt:

- ✓ Saraksti
- ✓ Vārdnīcas

Elementu *daudzums* vārdnīcā:

```
print(len(Soft)) #3
```

---

*Kortežs* – atslēga:

```
Employees = {("Strods", "Juris"): "Programmer",
              ("Celms", "Ivars"): "Operator"}

print(Employees[("Strods", "Juris")]) #Programmer
print(Employees[("Celms", "Ivars")]) #Operator
```

---

*Atslēgu* saraksts:

```
print(Soft.keys())
#dict_keys(['Oracle', 'Windows', 'Unix'])
```

---

*Vērtību* saraksts:

```
print(Soft.values())
#dict_values(['DBMS', 'OS', 'OS'])
```

*Piezīme: elementu adresēšanai atslēgu un vērtību sarakstos nepieciešama papildu pārveidošana.*

```
Keys=Soft.keys()  
print(Keys[0]) #izņēmums TypeError  
# 'dict_keys' object does not support indexing
```

*Līdz Python 2.5 papildu pārveidošana nebūtu nepieciešama.*

*Python 3.0 pārveidošana uz sarakstu:*

```
Keys=list(Soft.keys())  
print(Keys[0]) #Oracle
```

*Python 3.0 pārveidošana uz kortežu:*

```
Keys=tuple(Soft.keys())  
print(Keys[0]) #Oracle
```



Vārdnīcas *elementu izvade*:

```
for Key in Soft.keys():  
    print(Key, " ", Soft[Key])
```

---

Saraksta *items()* izmantošana elementu izvadei:

```
for (Key, Value) in Soft.items():  
    print(Key, " ", Value)
```

---

Vārdnīcas elementa *dzēšana*:

```
del Soft['Unix']
```

---

Vārdnīcas *attīrīšana*:

```
Soft.clear()
```

---

Vārdnīcu *apvienošana*:

```
Soft1.update(Soft2)
```

Vārdnīca no sarakstiem:

```
Network = {"WS1":["Windows", "Linux"],  
           "WS2":["DOS", "Windows"]}  
print(Network)  
# {'WS1': ['Windows', 'Linux'],  
  'WS2': ['DOS', 'Windows']}
```

---

Informācija par *pirmā* datora *pirmo* operētājsistēmu:

```
print(Network["WS1"][0])    # Windows
```

---

*Atslēgu un vērtību izvade:*

```
for Key in Network.keys():  
    print(Key, ":", end="")  
    for Attrib in Network[Key]:  
        print(Attrib, " ", end="")  
    print()
```

## Iegūtie rezultāti

```
# WS1 :Windows  Linux  
# WS2 :DOS    Windows
```

---

*Piezīme – vecās Python versijās būtu iespējams:*

```
print (Key, ":", )  
  
...  
print (Attrib, " ", )
```

Komata simbols nodrošinātu informācijas izvadi *tajā pašā rindiņā*.

---

Tagad sintaktiskās kļūdas nav, efekta arī.

Pēc noklusējuma:

```
end="\n"
```

Lai datoram *WS1* ir trešā operētājsistēma *DOS*:

```
Network["WS1"].append("DOS")  
  
# {'WS1': ['Windows', 'Linux', 'DOS'], 'WS2':  
# ['DOS', 'Windows']}
```

---

Ielikt operētājsistēmu saraksta sākumā (divi varianti):

```
Network["WS1"][0:0] = ["DOS"]  
  
Network["WS1"].insert(0, "DOS")  
# {'WS1': ['DOS', 'Windows', 'Linux'], 'WS2':  
# ['DOS', 'Windows']}
```

---

Operētājsistēmu daudzums uz darba stacijas *WS1*:

```
print(len(Network["WS1"]))
```

Vārdnīca no vārdnīcām: programmatūras kategorijas un prioritātes.

```
Soft = {  
    "Operating Systems": {  
        1:"Windows", 2:"Unix",  
        3:"Novell NetWare", 4:"Solaris"  
    },  
    "Object-Oriented Languages": {  
        1:"Java", 2:"C++", 3:"Ada"  
    },  
    "Web Technologies": {  
        1:"DHTML", 2:"Python", 3:"ASP", 4:"PHP"  
    }  
}
```

## Informācijas izvade:

```
for Key1 in Soft.keys():
    print(Key1, "\n", end="")
    for Key2 in Soft[Key1].keys():
        print(Soft[Key1][Key2], " ", end="")
    print()
```

---

## Rezultāti:

```
# WEB Technologies
#     DHTML Python ASP PHP

# Object-Oriented Languages
#     Java C++ Ada

# Operating Systems
#     Windows Unix Novell NetWare Solaris
```

## Izņēmumu apstrāde

Dalīšana ar nulli

```
try:  
    s=1/0  
except ZeroDivisionError:  
    print("Error !")    #Error !
```

---

Papildinformācija par izņēmumu

```
try:  
    s=1/0  
except ZeroDivisionError as X:  
    print("Error: ", X)  
# Error:  int division or modulo by zero
```

---

*Piezīme – vecās Python versijās būtu:*

```
except ZeroDivisionError, X:
```

## Indeksa kļūda (izņēmums *IndexError*)

```
L = ["C++", "Java"]  
try:  
    print(L[2])  
except Exception:  
    print("Exception !")  
print("Good Bye !")  
# Exception !  
# Good Bye !
```

Piezīme – vecās *Python* versijās *Exception* vietā būtu *StandardError*.

---

Gandrīz visi izņēmumi ir *Exception* izņēmuma apakšklases.

Bija arī iespējams:

```
except BaseException:
```



Izņēmumu apstrādātāji un *visu izņēmumu* tveršana

```
try:
    print(L[2])
except ArithmeticError:
    print("ArithmeticError")
except:
    print("Unknown Error")
# Unknown Error
```

---

Izņēmuma *ierosināšana*

```
try:
    raise Exception("MyError")
except:
    print("Error")
# Error
```

Vienā blokā var apstrādāt *vairākus* izņēmumus:

```
try:
    ...
except (IndexError, ArithmeticError):
    print("Error.")
```

---

Lai izņēmums *netika* ierosināts. Var apstrādāt šo situāciju operatorā **else**.

```
try:
    ...
except ArithmeticError:
    print("ArithmeticError.")
except:
    print("Error.")
else:
    print("No problems.")
```

Izņēmumu apstrādātāji un *universālā* reakcija

```
try:
    ...
except:
    print("Error !")
finally:
    print("Done !")
```

---

Vārdnīcas ērti izmantot lietotāja izvēles apstrādei

```
def Create():
    ...
def Update():
    ...
Choice = {"1":Create, "2":Update}

print("1 - Create\n2 - Update")
C = input()

try:
    Choice[C]()
except:
    print("Error !")
```

## Izņēmumu hierarhija

```
L = ["C"]
try:
    print(L[1])
except IndexError:
    print("IndexError")          #1
except BaseException:
    print("BaseException")      #2
except Exception:
    print("Exception")          #3
```

---

*Jebkurš no blokiem (1), (2) vai (3) var būt pirmais bloks.  
Rezultāti: *IndexError*, *BaseException*, *Exception* attiecīgi.*

---

Bloks “pēc noklusēšanas” var būt *tikai pēdējais*.

**except:**

## Failu apstrāde

Fails *data.txt*:

```
1  
2a  
33  
44.4
```

---

Faila *atvēršana* un *informācijas nolasīšana*:

```
f = open("data.txt", "rt")  
List = f.readlines()  
print(List)
```

---

*Vienlaicīga* atvēršana un nolasīšana:

```
List = open("data.txt").readlines()
```

## *Veselu skaitļu summēšana:*

```
f = open("data.txt", "rt");  
Sum=0  
  
for Elem in f.readlines():  
    try:  
        Sum = Sum + int(Elem)  
    except:  
        print("Type error !")  
  
print("Total SUM: ", Sum)  
f.close()
```

---

## Rezultāts:

```
# Type error !  
# Type error !  
# Total SUM: 34
```

Metodes *open()* parametri līdzīgi funkcijas *fopen()* parametriem C (C++) valodās.

**r** - Read, **w** - Write, **a** - Append,  
**b** - Binary, **t** - Text, ...

---

Lai summēšanas operators izskatās tā:

```
Sum = Sum + float(Elem)
```

Programmas izpildes rezultāts:

```
# Type error !  
# Total SUM: 78.4
```

---

Papildu funkcijas: modulis OS.

```
import os  
if os.access("dati.txt", os.F_OK) :  
    print("Exists !")  
print(os.path.abspath("dati.txt"))  
# C:\TEST\dati.txt
```

Fails var arī neeksistēt – runa ir par aktuālo katalogu.

## Lielu failu apstrāde

```
f = open("data.txt", "rt");
Sum = 0
while 1:                #Python 3.0: var 'while True'
    S = f.readline();

    if not S:
        print("END OF File")
        break

    try:
        Sum = Sum + int(S)
    except:
        print("Type error !")

print("Total SUM: ", Sum)
f.close()
```

Rezultāti sakrīt ar *iepriekšējiem* rezultātiem.



Lai failā *data.txt* ir teksta rindiņas:

C++

Java

Python

---

Informācija nolasīta uz sarakstu:

```
List = f.readlines();  
print(List)  
# ['C++\n', 'Java\n', 'Python\n']
```

---

Uzdevums: *atmest* pēdējo simbolu \n.

```
for i in range(0, len(List)):  
    List[i] = List[i][:-1]  
print(List)  
# ['C++', 'Java', 'Python']
```

## Teksta rindiņas *Python* valodā

Teksta rindiņas *garums*:

```
S = "Python Language"  
print(len(S))           # 15
```

---

Teksta rindiņas *fragmenti*:

```
S = "Python Language"  
  
L = S[7:]           # Language  
  
L = S[7:15]         # Language  
  
L = S[7:26]         # Language  
  
L = S[0:6]          # Python
```

Teksta rindiņas var saturēt “*Escape*-secības”.

```
S = "Python\nLanguage"
```

Izvadot informāciju ekrānā, iegūsim pāreju uz nākamo rindiņu.

Teksta rindiņas garums:

```
print(len(S)) #15
```

Var atteikties no “*Escape*-secību” interpretācijas.

Tādiem nolūkiem izmanto “raw strings”.

Attiecīgo rindiņu prefikss ir ‘r’ (vai ‘R’).

```
S = r"Python\nLanguage"
```

```
print(S) #Python\nLanguage
```

```
print(len(S)) #16
```

Arī tādu rindiņu gadījumā *pēdējais* simbols **ne**var būt “\”.

```
S = r"PythonLanguage\" #klūda
```

Eksistējošo teksta rindiņu *nevar izmainīt*:

```
S="Jva"
```

```
S[1:1]="a" # izņēums TypeError
```

---

*Pareizas darbības:*

```
S1 = S[:1]
```

```
S2 = S[1:]
```

```
S = S1 + "a" + S2 # Java
```

Vai:

```
S = S[:1] + 'a' + S[1:] # Java
```

---

Simbolu apstrāde ciklā:

```
S = "Python"; D = ""
```

```
for C in S:
```

```
    D += C
```

```
print(D) # Python
```

Teksta rindiņas iegūšana no citiem datu tiem:

```
X = 10
```

1. Funkcija *str()*.

```
S = str(X)      #10
```

2. Funkcija *repr()* – representation.

```
S = repr(X)     #10
```

3. Formatētā izvade.

```
S = "%s" % X    #10
```

4. Simbols „tilde” (tikai līdz *Python* 3.0).

```
S = `X`         #10
```

---

Piezīme: **n**eder Java paņēmiens:

```
S = "" + X      #TypeError
```

Formatēta izvade:

```
print("%s %d" % (X, X))  #10 10
```

## *Dažas funkcijas teksta rindiņu apstrādei*

```
S = "Java2"
print(S.upper())      #JAVA2
print(S)              #Java2

S = S.upper()
print(S)              #JAVA2

print(S.lower())      #java2
print(S.capitalize()) #Java2

S=" Java "
print(S.replace(" ", "*")) #*Java*

print("Month:{ }. Day:{ }.".format("January", 1))
#Month:January. Day:1.
```

## Teksta *formatēšana*: papildu jautājumi.

```
Counter = 1
```

```
print ("%2d" % Counter)          #1
```

```
print ("%2d" % (Counter) )      #1
```

```
print ("%2d" % (Counter,) )     #1
```

Otrajā un trešajā gadījumā tika izmantots *kortežs*.

Visos gadījumos operators `'%'` ir teksta rindiņas formatēšanas operators (*interpolācijas operators*).

Formatēšanā var lietot arī *vārdnīcu*:

```
print ("% (m) s % (d) 2d" % {"m": "January", "d": 1} )
```

```
print ("% (m) s % (d) 2d" % {"d": 1, "m": "January"} )
```

Rezultāts:

```
January 1
```

## Apakšprogrammas

*Nepārskaitļa (pārskaitļa) pārbaude:*

```
def Odd (Num) :  
    if Num%2 :  
        return 1  
    else:  
        return 0
```

Programmas fragments:

```
if Odd (1) :  
    print ("Odd")  
else:  
    print ("Even")
```



Saraksta apstrāde un parametru *vērtības pēc noklusēšanas*:

```
def PrList(L, N=0):
    i=0
    for Elem in L[N:]:
        i+=1
        print(str(i)+". ", Elem)
```

Apakšprogrammas izsaukumi:

PrList(['a', 'b', 'c'])	PrList(['a', 'b', 'c'], 1)
-------------------------	----------------------------

Rezultāts:

```
1.  a
2.  b
3.  c
```

Rezultāts:

```
1.  b
2.  c
```

Parametri – *nemaināmie* tipi, tiks veidota *parametra kopija*:

- ✓ Skaitļi
- ✓ Teksta rindiņas
- ✓ Korteži

Parametri – *maināmie* tipi, tiks izmainīts *oriģināls*:

- ✓ Saraksti
- ✓ Vārdnīcas

Mēģinājums izmainīt *parametru* - *skaitļi*:

```
def Inc (X) :
```

```
    X=X+1
```

```
...
```

```
A = 1
```

```
Inc (A) # 1
```

---

Saraksts no *viena* elementa:

```
def Inc (X) :
```

```
    X[0]=X[0]+1
```

```
...
```

```
A = [1]
```

```
Inc (A) # [2]
```

Lai apakšprogramma `CheckString(...)` atrodas *citā failā* (piemēram, `service.py`).

```
...
def CheckString(S):
    print("Done.")
...
```

Failā ar apakšprogrammām nav *nekādu* papildus virsrakstu.

Apakšprogrammas lietošana *galvenajā* programmā:

```
from service import *
...
CheckString(URL)
```

Tiks izveidots papildus fails ar *bināro* kodu: `service.pyc`.

Var izmantot *citū* oriģinālās funkcijas vārdu:

```
...
from service import CheckString as ChkS
...
ChkS(URL)
```

## Speciālās funkcijas

1. Funkcija *map()*.

```
<List> = map(<function>,  
            <sequence>  
            [, <sequence>...])
```

---

Funkcijas izmantošanas piemēri.

a. Saraksta elementu tipa *pārveidošana* ( $char \rightarrow int$ ).

```
L = ['1', '2']
```

```
print(list(L))    # ['1', '2']
```

```
L = map(int, L)
```

```
print(list(L))    #[1, 2]
```

## b. Divu sarakstu summēšana.

```
def Sum(x, y):
    x = x + y
    return x

L1 = [1, 2]
L2 = [3, 4]

L = map(Sum, L1, L2)

print(list(L))  #[4, 6]
```

---

Līdz *Python 3.0* pārveidošana uz sarakstu *nav vajadzīga*

```
print(L)      # <map object at 0x013CA0F0>
```

---

Var izmainīt jau eksistējošu sarakstu:

```
L1 = map(Sum, L1, L2)  #[4, 6]
```

---

L1 un L2 var būt *korteži*.

## 2. Funkcija *lambda()*.

`lambda` [`<argumentu saraksts>`] :  
izteiksme

---

### a. Saraksta elementu tipa *pārveidošana* ( $\text{char} \rightarrow \text{int}$ ).

```
L = ['1', '2']
```

```
L = map(lambda x: int(x), L) #[1, 2]
```

---

### b. Saraksta *elementu palielināšana* par 1.

```
L = [1, 2, 3, 4]
```

```
L = map(lambda x: x+1, L) #[2, 3, 4, 5]
```

---

### c. Divu sarakstu *summēšana*.

```
L1 = [1, 2]; L2 = [3, 4]
```

```
L1 = map(lambda x, y: x + y, L1, L2)
```

```
print(list(L1))          #[4, 6]
```

### 3. Funkcija *reduce()*.

```
<Value> = reduce(<function>,
                 <sequence>
                 [, <initial>])
```

---

Atrast visu saraksta elementu *reizinājumu*.

*Piezīme:* pēc *Python 3.0* nepieciešama importēšana:

```
from functools import *
```

```
L = [1, 2, 3, 4]
```

```
print(reduce(lambda x, y: x*y, L))      #24
```

vai

```
print(reduce(lambda x, y: x*y, L, 1))  #24
```



## Secību kārtošana: *Python 3.1*

```
L = [2, 3, 1]           #skaitļu saraksts
L.sort(reverse=True)    #[3, 2, 1]
L.sort(reverse=False)   #[1, 2, 3]
```

---

```
#kortežu saraksts
L = [ ("c.htm", 2000), ("java.htm", 1000) ]
```

---

Kārtošanas parametrs: tīmekļa lappuses *izmērs*.

```
L.sort(key=lambda x: x[1])
# [('java.htm', 1000), ('c.htm', 2000)]
```

---

```
L.sort(key=lambda x: -x[1])
# [('c.htm', 2000), ('java.htm', 1000)]
```

---

```
L.sort(key=lambda x: -x[1], reverse=True)
# [('java.htm', 1000), ('c.htm', 2000)]
```

## Kopu radīšana

Kopā *nav dublikātu*.

```
L=set([1, 2, 3, 2])  
print(L)    # {1, 2, 3}
```

---

Kopā *nav indeksu*.

```
print(L[0]) #izņēmums TypeError
```

---

Ir iespējama pārveidošana uz sarakstu:

```
print(list(L)[0]) # 1
```

---

Kopu *šķērsojums*:

```
print(L.intersection([1, 6, 3])) # {1, 3}
```

---

Kopu *apvienošana*:

```
print(L.union([4])) # {1, 2, 3, 4}
```

## Deku radīšana

Pakotne *collections* (Python 3.1.)

```
from collections import *  
  
L = deque(["C++", "C#"])  
print(L[0], " ", L[1])    # C++    C#  
print(L)                  # deque(['C++', 'C#'])
```

---

## Elementu pievienošana no abām malām

```
L.append("Java")  
L.appendleft("C")  
print(L)    # deque(['C', 'C++', 'C#', 'Java'])
```

---

**Nav** iespējams ielikt elementu:

```
L[0:0] = ["Pascal"] # iznēmums TypeError
```

## Objekts *Counter* (Python 3.1)

```
C = Counter({'*':4, '+':5})  
print(C)    # Counter({'+': 5, '*': 4})  
print(C['*']) # 4
```

---

## Elementu saraksta iegūšana

```
print(list(C.elements()))  
['+', '+', '+', '+', '+', '*', '*', '*', '*']
```

---

## Saraksta radīšana ciklā

```
C=Counter()  
for Lang in ("C", "C++", "C++", "Java"):  
    C[Lang]+=1  
print(C) #Counter({'C++':2, 'C':1, 'Java':1})
```

## Secību pārveidošana

Pārveidot *sarakstu no sarakstiem* uz *vārdnīcu*

```
L = [ ["WS1", "Windows"], ["WS2", "Unix"] ]
D = dict(L) # {'WS1': 'Windows', 'WS2': 'Unix'}
```

---

## Kortežu saraksta iegūšana

```
WS = ["WS1", "WS2"] # atslēgu saraksts
OS = ["Windows", "Unix"] # vērtību saraksts
L = zip(WS, OS)
```

---

```
print(L) # <zip object at 0x013C9CB0>
```

```
print(list(L))
#[ ('WS1', 'Windows'), ('WS2', 'Unix')]
```

---

```
D = dict(L) # {'WS1': 'Windows', 'WS2': 'Unix'}
```

## Klases un objekti

Visas klases manto no `object`. Elementārās *klase* un *apakšklase*:

```
class Super:                                #superklase
    pass
class Sub(Super):                          #apakšklase
    pass

SupObj = Super()                           #objekta radīšana
SubObj = Sub()                             #objekta radīšana
-----
print(list(map(type,
    [Super, Sub, SupObj, SubObj])))
# [<class 'type'>, <class 'type'>, <class
'__main__.Super'>, <class '__main__.Sub'>]

print(isinstance(SubObj, Super)) #True
print(isinstance(SubObj, Sub))   #True
```

Klase “koordinātu punkts” *bez* informācijas izvades metodes:

```
class CPoint:
    def __init__ (self, x=0, y=0):
        self.x = x
        self.y = y
        print ("Object created.")
    def __del__(self):
        print ("Object destroyed.")
    def GetX(self):
        return self.x
    def SetX(self, x):
        self.x = x
    def GetY(self):
        return self.y
    def SetY(self, y):
        self.y = y
```

## Komentāri:

1. `__init()` ir konstruktors.
2. `__del()` ir destruktors.
3. `self` ir norāde uz pašu objektu.
4. Visas speciālas metodes prasa, lai pirmais parametrs būtu *klases eksemplārs*.
5. Metodes izsaukuma procesā pirmais formālais parametrs *vienmēr tiks aizstāts* ar izsaucošā objekta vārdu.
6. Faktisko parametru daudzums *vienmēr mazāk* par formālo parametru daudzumu.
7. Informācijas izvadei vēlams *pārdefinēt* metodi `__str()`.

```
CP = CPoint()  
print(CP)  
<__main__.CPoint object at 0x013CADB0>
```



Lai konstruktorā **neizmanto** parametru *self*.

```
class CPoint:
    def __init__ (self, Px=0, Py=0) :
        x = Px
        y = Py
    ...
```

---

*Rezultāts:* *x* un *y* ir *lokālie* mainīgie, kuri pieejami tikai attiecīgajā funkcijā.

```
CP = CPoint(1, 2)
print (CP.x)
```

```
AttributeError:
'CPoint' object has no attribute 'x'
```

*Piezīme:* problēma nav saistīta ar iekapsulēšanu.

Visi atribūti iepriekšējā (“pareizajā”) piemēra ir “public”.

Konstruktoru var realizēt arī tā:

```
def __init__(q, x=0, y=0):  
    q.x = x  
    q.y = y
```

---

Objektu radīšana un iznīcināšana:

```
CP1 = CPoint(1, 4)  
CP2 = CPoint()
```

```
del CP1
```

```
del CP2
```

Ja pamēģināt izdzēst objektu atkārtoti, notiks izņēmums  
*NameError*.

```
del CP1
```

```
del CP1
```

```
#NameError: name 'CP1' is not defined
```

Ja neizsaukt destrukturu, objekts tiks iznīcināts *pēc skripta pabeigšanas*.

Python *pats iznīcina* nevajadzīgus objektus un destruktora nevajag atbrīvot atminu.

**Nav ieteicams** veidot destruktorus bez nepieciešamības.

Objekti ar metodi `__del__` **ne**var būt iznīcināti, izmantojot automātisku atmiņas atbrīvošanu.

---

Ar objektu vienmēr saistīti daži atribūti.

`__class__`: objekta klase.

`__module__`; objekta modulis.

```
print(CP1.__class__)    #<class '__main__.CPoint'>
print(CP1.__module__)  #__main__
```

## Iekapsulēšanas principi

```
CP = CPoint(1, 4)
print(CP.x, CP.GetX())
del CP
print("Program finished.")
#Object created.
#1 1
#Object destroyed.
#Program finished.
```

---

Informācijas slēpšana: darbs ar `__x` un `__y`.

```
def __init__(self, x=0, y=0):
    self.__x = x
    self.__y = y

...
print(CP.__x)      # AttributeError
```

Informācijas izvade: `__str__()` metodes pārdefinēšana.

```
class CPoint:
    ...
    def __str__(self):
        return "X: " + str(self.GetX()) +
            ", Y: " + str(self.GetY())
    ...
CP = CPoint()
print(str(CP))    # X: 0, Y: 0
print(CP)         # X: 0, Y: 0
```

---

Var arī pārdefinēt `__repr__()` metodi:

```
def __repr__(self):
    return "CPoint (%d, %d)" % (self.x, self.y)
    ...
print(repr(CP))  # CPoint (0, 0)
print(`CP`)     # Tikai vecās versijās CPoint (0, 0)
```

*Statisko metožu veidošana:*

```
class Out:
    @staticmethod
    def Format(Param) :
        ...
```

Tādās metodēs *nav* pirmā parametra.

*Statisko metožu lietošana no klases un objekta:*

```
Out.Format(URL)          # (1)
Out().Format(URL)        # (2)
```

Bez @staticmethod rindiņā (2) notiktu *izpildes kļūda*.

*Alternatīvais risinājums:* @staticmethod **neizmanto** klasē.

*Galvenās programmas fragments:*

```
Out.Format = staticmethod(Out.Format)
Out.Format(URL)
```

Ir iespējams izmantot citu `staticmethod` formu:

```
class Out:
    def Format(Param) :
        print (Param)
    Format = staticmethod(Format)
```

*Galvenajā* programmā var izmantot rindiņas (1) un (2).

Eksistē arī *klases* metodes.

Tad norāda *papildus* parametru “parastu” metožu stilā.

```
class Out:
    @classmethod
    def Format(self, Param) :
        print (Param)
```

*Galvenajā* programmā arī var izmantot rindiņas (1) un (2).

## *Klases atribūtu veidošana*

```

class CPoint:
    Status = True

CP1=CPoint()
CP2=CPoint()

print(CPoint.Status) #True
print(CP1.Status)    #True
print(CP2.Status)    #True

CP1.__class__.Status = False

print(CP1.Status)    #False
print(CP2.Status)    #False

Ja būtu:
CP1.Status = False    #False
#CP2.Status = True
    
```



## Atribūtu *adresēšanas* un *pievienošanas* kontrole

Lai ir klase, kas apraksta *datoru*. Atribūti *nav reglamentēti*.

```
class Computer: #nav konstruktora
    def __getattr__(self, Attr):
        print("Attribute '" + Attr + "' IS missing.")
        return False      #vai izņēmuma ierosināšana
    def __setattr__(self, Attr, Val):
        print("Attribute '" + Attr + "' WAS missing.")
        self.__dict__[Attr] = Val      #pievienot
```

### Programmas fragments:

```
C = Computer()
print(C.OS)      #Attribute 'OS' IS missing.      False
C.OS = "MS Windows" #Attribute 'OS' WAS missing.
print(C.OS)      #MS Windows
```

## Atribūta *eksistēšanas* kontrole

Lai klasē ir metode `__setattr__()`, bet nav metodes `__getattr__()`.

```
C = Computer()
print(hasattr(C, "OS"))    #False
C.OS = "MS Windows"      #Attribute 'OS' WAS missing.
print(hasattr(C, "OS"))    #True
```

Ja klasē būtu metode `__getattr__()`, abos gadījumos būtu rezultāts `True`.

Metode `hasattr()` savā darbā izsauc metodi `__getattr__()`.

Izņēmuma nav, ir situācijas apstrāde. Līdz ar to: `True`.

## Atribūtu dzēšana

Var izdzēst koordinātpunkta atribūtu:

```
CP = CPoint(2, 3)
print(CP)      # (*)
del CP.x
```

Tagad nav iespējams izpildīt rindiņu (\*).

Tiks izsaukta metode `__str__()`, bet atribūts `x` vairs neeksistē.

Rezultāts: izņēmums *AttributeError*.

```
AttributeError: 'CPoint' object has no
attribute 'x'
```

Ir elementāra iespēja aizliegt *visu* atribūtu dzēšanu:

```
def __delattr__(self, a):
    pass
```

Dzēšot atribūtu, nebūs nekāda efekta (pat izņēmuma).

## Īpašību veidošana

```
class CPoint:
    def __init__(self, Px, Py):
        self.__x = Px
        ...
    @property
    def x(self):
        return self.__x
    @x.setter
    def x(self, Px):
        self.__x = Px
    @x.deleter
    def x(self):
        del self.__x
```

*Piezīme:* var nenodrošināt visas iespējas.

Piemēram, var atteikties no *atribūta dzēšanas*.

## *Galvenās programmas fragments*

```
C = CPoint(1, 1)
print(C.x) #1
C.x = 2
print(C.x) #2
del C.x
print(C.x) #izpildes kļūda
```

## *Alternatīvais risinājums:*

```
class CPoint:
    ...
    def GetX(self):
        return self.__x
    ...
    x = property(GetX, SetX, DelX)
```

## *Galvenajā programmā izmaiņu nav.*

## Operatoru *pārlāde*

Lai ir klase *Vector*. Klases pamatā ir saraksts.  
Atrast divu vektoru summu, izmantojot *pārlādēto* operatoru.

```
class Vector:
    def __init__(self):
        self.V = []
    def AddElem(self, Elem):
        self.V.append(Elem)
    def __str__(self):
        return str(self.V)
    def __add__(self, other):
        NewVector = Vector()
        NewVector.V = list(map(lambda x, y: x+y,
                                self.V, other.V))
        return NewVector
```

## Objektu radīšana un elementu pievienošana

```
V1 = Vector()
V1.AddElem(2) # [2]

V2 = Vector()
V2.AddElem(3) # [3]
-----
V3 = V1+V2      # pārlāde
-----
print(V1)       # [2]
print(V2)       # [3]
print(V3)       # [5]
-----
```

*Piezīme:* līdzīgajā stilā var pārlādēt dažas citas metodes.

```
def __sub__(self, other):
    ...
    NewVector.V = list(map(lambda x, y: x-y,
                           self.V, other.V))
```

Lai ir nepieciešams reizināt vektoru ar skaitli.

```
class Vector:
    ...
    def __mul__(self, Num):
        NewVector = Vector()
        NewVector.V = list(map(lambda x: x*Num,
                                self.V))
        return NewVector
```

---

```
V1 = Vector()
V1.AddElem(3)

V2 = V1*2
print(V2) # [6]

V2 = 2*V1
```

```
TypeError: unsupported operand type(s) for *:
'int' and 'Vector'
```



**Neder variants:**

```
def __mul__(Num, self):  
    ...
```

TypeError: unsupported operand type(s) for \*: 'int' and 'Vector'

---

**Pareizs risinājums:**

```
class Vector:  
    ...  
    def __rmul__(self, Num):  
        return self*Num
```

---

```
V2 = V1*2  
print(V2) # [6]  
  
V2 = 2*V1  
print(V2) # [6]
```

Lai vektoru reizina ar *teksta rindiņu*:

```
V2 = V1*"2"
```

```
V2 = "2"*V1
```

Rezultāts sakrīt:

```
['222']
```

Var arī pārlādēt *saīsināto* operatoru (piemēram, \*=)

```
class Vector:
```

```
    ...
```

```
    def __imul__(self, Num):
```

```
        self.V = list(map(lambda x: x*Num,
                           self.V))
```

```
        return self
```

```
V1 *= 2
```

```
print(V1) # [6]
```

*Piezīme:* operators *mul* iekļauj sevī *imul* iespējas.

## Operatoru *pārlāde* (turpinājums)

Piemērs: punkta attālums no koordināšu centra.

```
class CPoint:
    ...
    def __r__(self):
        return self.x**2 + self.y**2

    def __lt__(self, CP):
        return self.__r__() < CP.__r__()
```

---

```
CP1 = CPoint(4, 2)
CP2 = CPoint(2, 5)
```

---

Programmā ir fragments:

```
if (CP1<CP2):
    print("CP1 is less.")
else:
    print("CP1 is greater.")
#CP1 is less.
```

Metode `__bool__(self)` atgriež loģisku objekta būtību (*True* vai *False*).

```
def __bool__(self):  
    return self.x != 0 or self.y != 0
```

*Piezīme:* vecās metodēs tā ir metode `__nonzero__`.

---

Lai programmā ir objekta pārbaudes fragments:

```
if (CP):  
    print("True")  
else:  
    print("False")
```

Var arī:

```
print(bool(CP))
```

---

Objekta CP radīšana:

```
CP = CPoint(1, 0) #rezultāts - True  
CP = CPoint(0, 0) #rezultāts - False
```

Mantošana: *displeja punkts*.

```
class DPoint(CPoint):
    def __init__(self, x=0, y=0, color=0):
        CPoint.__init__(self, x, y)
        self.color=color

    def GetColor(self):
        return self.color

    def SetColor(self, color):
        self.color = color

    def __str__(self):
        return CPoint.__str__(self) + \
            ", Color: " + str(self.color)

...
DP = DPoint(1)
print(DP)           # X: 1, Y: 0, Color: 0
```

Agregācija: lauzta līnija.

```
class BrokenLine:
    def __init__(self):
        self.Points = []

    def Add(self, P):
        self.Points.append(P)

    def __str__(self):
        S = ""
        for El in self.Points:
            S = S + str(El) + "\n"
        return S

...
BL = BrokenLine()
BL.Add(CPoint(1, 2))
BL.Add(CPoint(2, 3))
print(BL)
```

## Papildu iespēja: kompleksa skaitļi

Tādi skaitļi sastāv no divām daļām:

```
c = 2 + 3j
```

```
c = (2 + 3j)    # tāds pats efekts
```

```
print(c)         # (2+3j)abos gadījumos
```

---

Lai ir divi skaitļi:

```
c1 = 1 + 2j
```

```
c2 = 4 + 3j
```

Elementāras operācijas:

```
print(c1 + c2)    # (5+5j)
```

```
print(c1 - c2)    # (-3-1j)
```

```
print(c1 * c2)
```

```
# (-2+11j)
```

```
# (a + bj) * (c + dj) = (ac-bd, ad+bc)
```

```
print(c1 / c2)    # (0.4+0.2j)
```

## GUI programmēšanas pamati

Iezīmes izvade grafiskajā logā:

```
from tkinter import * # Agrāk: Tkinter
w = Label(None, text="Hello, World!")
w.pack()
w.mainloop()
```



*tkinter* ir pārnesama GUI bibliotēka ar atvērtu kodu.

Runa ir par *neformālo* standartu GUI izstrādei Python valodā.

*tkinter* pamatā ir bibliotēka *Tk*.

*Tk* ir *open source* standarts; to izmanto valodas *Perl* un *Tcl*.



*tkinter* vienmēr piegādā kopā ar *Python*.

Stingri ņemot, *tkinter* ir interfeiss: *Python* valoda → *Tk* bibliotēka.

*Python* / *tkinter* programmas ir notikumu vadāmas programmas.

Sākumā veido formas un reģistrē notikumu apstrādātājus, pēc tam nav nekādu darbību – notiek notikumu gaidīšana.

Mūsu piemērā iezīme *Label* tika izvietota formā pateicoties metodei *mainloop()*.

Galīgais *mainloop()* lietošanas rezultāts: gaidīšanas stāvoklis.

Notiek tastatūras un peles notikumu atsekošana.

Metode *mainloop()* bez *pack()* rāda tukšo logu.

Metode *pack()* bez *mainloop()* neko nerāda.

Minimāli iespējama programma:

```
from tkinter import *  
Label(None, text="Hello, World!").pack()  
mainloop()
```

Metode *pack()* izsauc ģeometrijas menedžeri (packer).

Pēc noklusēšanas grafiskais elements saistīts ar augšējo malu.

Alternatīvais menedžeris: *grid()*.

Grafiskie elementi tiks izvietoti konteinerā, tabulas veidā.

Retāk lieto menedžeri *placer()*.

# C++ valodas jaunās iespējas

## STL bibliotēka (Standard Template Library)

Izstrādātā firmā *Hewlett Packard*.

Atļauj pielietot *vispārīgo programmēšanu* plaši izplatītos algoritmos.

Bibliotēkas sastāvdaļas:

- ✓ *Konteineri* (rindas, vektori).
- ✓ *Algoritmi* (elementu meklēšana, kārtošana).
- ✓ *Iteratori* (pārmeklēšanas rīki).

Konteineru klasifikācija:

- ✓ *Secīgie* konteineri (vektori, saraksti).
- ✓ *Asociatīvie* konteineri (kopas).

1. *Konteiners* ir datu glabāšanas organizēšanas veids.  
Konteiners satur dažas metodes specifisko uzdevumu izpildei.
2. *Algoritms* ir procedūra, kas apstrādā konteinerus.  
Algoritms pēc savas būtības ir *neatkarīgā* funkcija.  
Algoritmus var izmantot masīvos un pat personiskajos konteineros.
3. *Iterators* ir rādītāju koncepcijas vispārinājums.  
Iteratori inkrementēšanas rezultātā secīgi norāda uz konteintera elementiem.  
Iteratori saista algoritmus ar konteineriem.  
Iteratorus uztver kā STL atslēgas daļu (datora *kopnes* analogs).

**STL** bibliotēkas izmantošanas piemērs. Masīva apstrāde (sākums).

```
#include <conio>
#include <iostream>
#include <list>
#include <numeric>
```

```
using namespace std;
```

```
void PrintList(const list<int>& L) {
    list<int>::const_iterator El;

    for (El = L.begin(); El != L.end(); El++)
        cout << *El << " ";
    cout << endl;
}
```

**STL** bibliotēkas izmantošanas piemērs.

Masīva apstrāde (turpinājums).

```
void main(void) {
    const int N = 4;
    list<int> List;

    int Dati[N] = {1, -2, 3, -4};

    for(int i=0; i<N; i++)
        List.push_front(Dati[i]);

    PrintList(List);    // -4 3 -2 1

    List.sort();
    PrintList(List);    // -4 -2 1 3
```

## STL bibliotēkas izmantošanas piemērs.

### Masīva apstrāde (beigas).

```

int Elem = 4;
int Counter = 0.0;
count(List.begin(), List.end(), Elem,
      Counter);
cout << "Elementu " << Elem <<
      " daudzums: " << Counter << endl; // 1

List.reverse();

PrintList(List);
cout << "Elementu summa: " <<
      accumulate(List.begin(), List.end(), 0.0);
// -2
}

```



Var izmantot arī *cit*us iteratorus.

Uzdevums: apstrādāt saraksta elementus *pretējā* secībā.

```
list<int>::const_reverse_iterator El;
for (El = L.rbegin(); El != L.rend(); El++)
    cout << *El << " "; // 1 -2 3 -4
```

Var adresēt *pirmo* un *pēdējo* saraksta elementu:

```
cout << L.front() << " " << L.back(); // -4 1
```

*Elementu daudzums* sarakstā:

```
cout << List.size(); // 4
```

*Piezīme*: ne visas metodes ir universālās.

Saraksta gadījumā *nav* iespējama adresēšana *pēc indeksa*.

Indeksēšanas `[]` nepieciešamības gadījumā lieto `vector<T>`.

## Vektora veidošanas un apstrādes piemērs.

```
#include <vector>
...
using namespace std;
...
const int N = 4;
int V[N] = {10, 20, 30, 20};

vector<int> Vect(N+1);

for(int i=0; i<N; i++) //indeksēšana
    Vect[i] = V[i]; //10 20 30 20
```

### Elementu *dublikātu* dzēšana:

```
// dublikāti jābūt blakus - kārtošana
sort(Vect.begin(), Vect.end());

// saspiešanas operācija
unique(Vect.begin(), Vect.end()); //10 20 30
```

Elementa meklēšana *parastajā* masīvā:

```
#include <algorithm>
...
using namespace std;
...
const int N=3;
int V[N] = {30, 10, 20};
...
int *P;

P = find(V, V+N, 10);
cout << "Position:" << P-V; //1

P = find(V, V+N, 100);
cout << "Position:" << P-V; //3
```

Ja meklējamā elementa nav, rezultāts ir vienāds ar N.

## Dinamiskā tipu identifikācija (RTTI).

Izmanto operatoru **typeid**.

1. **typeid**(<izteiksme>).
2. **typeid**(<tips>).

Rezultāts: norāde uz struktūras *typeinfo* objektu.

Visbiežāk strādā ar **public** locekļu *name()*.

Programmā nepieciešama rindiņa:

```
#include <typeinfo>
```

Darbam ar operatoru **typeid** klasē jābūt *virtuālās funkcijas*.

```
class CoordPoint {  
    public:  
        virtual void Print() {  
            cout << "COORD Point." << endl;  
        }  
};  
  
class DisplayPoint : public CoordPoint {  
    public:  
        virtual void Print() {  
            cout << "DISPLAY Point." << endl;  
        }  
};
```

Programmas fragments: informācijas *par objektiem* izvade.

```
CoordPoint *CP1 = new CoordPoint();  
DisplayPoint *DP1 = new DisplayPoint();  
CoordPoint *CP2 = new DisplayPoint();  
  
(*CP1).Print(); // COORD Point.  
(*DP1).Print(); // DISPLAY Point.  
(*CP2).Print(); // DISPLAY Point.
```

---

**Komentārs:** lai *Print()* metode *nav* virtuālā. Rezultāti:

```
// COORD Point.  
// DISPLAY Point.  
// COORD Point.
```

Programmas fragments: **typeid** darbā ar *objektiem*.

```
cout << typeid(*CP1).name() << endl;
// CoordPoint
cout << typeid(*DP1).name() << endl;
// DisplayPoint
cout << typeid(*CP2).name() << endl;
// DisplayPoint
```

---

**Komentārs:** lai *Print()* metode *nav* virtuālā. Rezultāti:

```
// CoordPoint
// DisplayPoint
// CoordPoint
```

Programmas fragments: **typeid** darbā ar *mainīgajiem*.

```
cout << typeid(CP1).name() << endl;
// CoordPoint *
cout << typeid(DP1).name() << endl;
// DisplayPoint *
cout << typeid(CP2).name() << endl;
// CoordPoint *
```

---

**Komentārs:** lai *Print()* metode *nav* virtuālā.

Rezultāti *pilnīgi sakrīt*:

```
// CoordPoint *
// DisplayPoint *
// CoordPoint *
```



Operatoru **typeid** var izmantot darbā ar *primitīviem tipi*.

```
double D, *PD;
cout << typeid(D).name() << endl;    // double
cout << typeid(*PD).name() << endl;  // double
cout << typeid(PD).name() << endl;   // double *
```

---

Dinamisko tipu identifikāciju ieteicams izmantot, ja nav iespējams noteikt objektu tipu *kompilācijas laikā* vai ar *virtuālo funkciju* palīdzību. Tas nodrošina:

- ✓ Drošuma paaugstināšanu.
- ✓ Labāko koda efektivitāti.

## Vārdu telpas

Izmanto, lai nepieļautu *vārdu konfliktu*.

Ir trīs sintaksiskās formas:

1. <vārdu telpa> : :<elements>;
2. **using** <vārdu telpa> : :<elements>;
3. **using namespace** <vārdu telpa>;

Vārdu telpu izmantošana programmā.

```
namespace A {  
    int X;  
    int Y;  
}
```

```
namespace B {  
    int X;  
    int Z;  
}
```

**Komentārs:** abās vārdu telpās eksistē *neatkarīgie mainīgie* ar vārdu *X*.

## Programmas fragments:

```
using B::X;
```

```
A::X = 1;
```

```
X = 2;
```

```
cout << "A::X -> " << A::X << ", B::X -> " <<  
    B::X << endl; // A::X -> 1, B::X -> 2
```

```
using namespace B;
```

```
Z = 3;
```

```
using namespace A;
```

```
X = 4;
```

```
cout << "A::X -> " << A::X << ", B::X -> " <<  
    B::X << endl; // A::X -> 1, B::X -> 4
```

*Piezīme:* B::X=4, tā kā sākumā bija **using** B::X;

Var eksistēt *vairākas* vārdu telpas ar *vienu un to pašu* vārdu.  
Rezultāts: viena *apvienotā* vārdu telpa.

```
namespace A {  
    int X;  
    int Y;  
}
```

```
namespace A {  
    int Z;  
}
```

...

```
using namespace A;  
X = 1; Y = 2; Z = 3;
```

## Ierobežojumu atcelšana C++ valodā

Vārds **mutable** atļauj ignorēt **const** – ierobežojumu.

```
class Auto {
    private:
        char Id[20];
        mutable int NextCheckingYear;
    public:
        Auto(char* P_Id, int P_Year) :
            NextCheckingYear(P_Year) {
            strcpy(Id, P_Id);
        }
        void Certify(int Interval) const {
            NextCheckingYear += Interval;
        }
};
```

## Galvenās programmas fragments:

```
Auto Ford("1111", 1999);  
Ford.Print();           // ID: 1111, Next Year: 1999  
Ford.Certify(5);  
Ford.Print();           // ID: 1111, Next Year: 2004
```

---

Var izmainīt arī *konstanšu vērtības*.

```
const Auto Toyota("1112", 1999);  
Toyota.Print();         // ID: 1112, Next Year: 1999  
Toyota.Certify(4);  
Toyota.Print();         // ID: 1112, Next Year: 2003
```

Pārveidošana **const\_cast** atļauj neizmantot **mutable**.

```
class Auto {
    private:
        char Id[20];
        int NextCheckingYear;
    public:
        ...
        void Certify(int Interval) const {
            Auto* const Local =
                const_cast<Auto* const>(this);

            Local->NextCheckingYear += Interval;
        }
};
```

Abi iepriekšējie programmas fragmenti dod *to pašu rezultātu*.



## Konstruktoru izsaukumu ierobežojumi

```
class CoordPoint {  
    private:  
        int X, Y;  
  
    public:  
        CoordPoint(int Px = 1, int Py = 2) :  
            X(Px), Y(Py) {};  
};  
...  
CoordPoint CP = 3;  
  
// Viss pareizi. Rezultāts: X=3, Y=2.
```

Aizliegt automātiskās konstruktora pārveidošanas:

```
explicit CoordPoint(int Px = 1, int Py = 2) :
```

*Ir iespējami* konstruktora izsaukumi:

```
CoordPoint CP1;  
CoordPoint CP2 = CoordPoint(4);  
CoordPoint CP3(4);  
CoordPoint CP4(4, 5);
```

*Nav iespējama* tiešā piešķire:

```
CoordPoint CP = 3;
```

---

Datu tipu pārveidošana: *vecais* stils.

```
char C = '0';  
C = (char) (C + 1.0);
```

---

Datu tipu pārveidošana: *jaunais* (*ieteicamais*) stils.

```
char C = '0';  
C = static_cast<char>(C + 1.0);
```

## Dinamiskas pārveidošanas

**dynamic\_cast**<tips> (<rādītājs vai norāde>)

Operatoru **dynamic\_cast** var lietot, ja ir mantošana un klasēs ir virtuālās funkcijas (runa ir par *polimorfiskām* klasēm).

```
class CoordPoint {
    ...
    virtual void Print() {}
};
```

```
class DisplayPoint : public CoordPoint {
    ...
};
```

Piezīme: nākamajā piemērā bez vārda **virtual** notiktu kompilācijas kļūda.

## Displeja punkta pārbaude

```
CoordPoint *CP = new DisplayPoint(); // (*)
DisplayPoint *DP =
    dynamic_cast<DisplayPoint*>(CP);
if (DP == 0)
    cout << "Incorrect conversion.";
```

Rezultāts: ziņojums netiks izvadīts.

Lai rindiņa (\*) izskatās tā:

```
CoordPoint *CP = new CoordPoint(); // (*)
```

Rezultāts:

```
Incorrect conversion.
```

Cita iespēja pārbaudīt rādītāju:

```
if (DP == NULL)
```

## Perl pamati

Practical Extraction and Report Language.

Failu paplašinājums: *\*.pl*.

Autors: *Larry Wall*. Izstrādes gads: *1986*.

Reģistrjutīgā programmēšanas valoda.

---

Valodas elementi:

**\$S** – skalārs.

**@L** – masīvs (saraksts).

**%H** – heš-struktūra (vārdnīca).

## Perl vs Python

Saraksti:

```
@L = ("C++", "Java", "Perl", "Python");  
print "@L";      #C++ Java Perl Python
```

---

Informācijas izvade **for** ciklā :

```
for ($i=0; $i<@L; $i++) {  
    print "$L[$i] "  #C++ Java Perl Python  
}
```

---

Komentāri:

1. Perl ir *kontekstatkarīgā* programmēšanas valoda:

```
$Len = @L;      # $Len = 4  
@LCopy = @L;    #@LCopy = (C++ Java Perl Python)
```

2. Saraksta @L elementi ir skalāri \$L[\$i] .

*Piezīme:* var arī @L[\$i] .

Informācijas izvade ciklā “**foreach**”:

```
for $E1 (@L) {  
    print "$E1 " # C++ Java Perl Python  
}
```

```
foreach $E1 (@L) {  
    print "$E1 " # C++ Java Perl Python  
}
```

-----  
Vērtību apmaiņa:

```
($A, $B) = (2, 3);  
print "$A $B\n";          # 2 3
```

```
($A, $B) = ($B, $A);  
print "$A $B\n";          # 3 2
```

Vārdnīcas:

```
%Soft = ("Unix" => "OS", "Oracle" => "DBMS",  
         "Windows" => "OS");
```

```
print join(", ", %Soft);
```

```
# Windows, OS, Unix, OS, Oracle, DBMS
```

---

Par sadalītājiem var būt *tikai komati*:

```
%Soft = ("Unix", "OS", "Oracle", "DBMS",  
         "Windows", "OS");
```

---

Katru vārdnīcas elementu var norādīt *neatkarīgi*:

```
$Soft{"Unix"} = "OS";
```

```
$Soft{"Oracle"} = "DBMS";
```

```
$Soft{"Windows"} = "OS";
```

Sākumā *neobligāti* deklarēt tukšo vārdnīcu.



Vārdnīcas izvade (atslēgu saraksta iegūšana):

```
for $Key (keys %Soft) {  
    print "$Key => $Soft{$Key}\n";  
}
```

---

Vārdnīcas izvade (vārdnīcas elementu apstrāde):

```
while ( ($Key, $Value) = each(%Soft) ) {  
    print "$Key => $Value\n";  
}
```

---

Vērtību saraksts: `values(%Soft)` .

---

Vārdnīcas elementa *dzēšana*:

```
delete $Soft{"Unix"};
```

Ja atkārtot operāciju, kļūdas (izņēmuma) nebūs. Rezultāta arī.

Saraksts no sarakstiem:

```
@Soft = (  
    ["PERL", "Python"], ["C", "C++", "Java"]  
);
```

---

Konkrētas programmēšanas valodas adresēšana:

```
print "$Soft[0][1]"; #Python
```

---

Informācijas izvade:

```
for $Row (@Soft) {  
    for $Elem (@{$Row}) {  
        print "$Elem ";  
    }  
    print "\n";  
}
```

Katru saraksta elementu var norādīt neatkarīgi:

```
$Soft[0] = ["PERL", "Python"];  
$Soft[1] = ["C", "C++", "Java"];
```

---

Elementu izvade ekrānā:

```
for ($i = 0; $i < @Soft; $i++) {  
    for ($j = 0; $j < @{$Soft[$i]}; $j++) {  
        print "$Soft[$i][$j] ";  
    }  
    print "\n";  
}  
  
#PERL Python  
#C C++ Java
```

## Vārdnīca no vārdnīcām:

```
%Soft = (  
  "Operating Systems" => {  
    1=>"Windows", 2=>"Unix",  
    3=>"Novell NetWare", 4=>"Solaris"  
  },  
  "Object-Oriented Languages" => {  
    1=>"Java", 2=>"C++", 3=>"Ada"  
  },  
  "Web Technologies" => {  
    1=>"DHTML", 2=>"Python",  
    3=>"ASP", 4=>"PHP"  
  }  
);
```

Informācijas izvade:

```
for $Frst (keys %Soft) {  
    print "$Frst\n";  
    for $Scnd (keys %{ $Soft{ $Frst} }) {  
        print "$Scnd:", " $Soft{ $Frst}{ $Scnd} ";  
    }  
    print "\n";  
}
```

---

Rezultāti:

Object-Oriented Languages

1:Java 3:Ada 2:C++

Operating Systems

4:Solaris 1:Windows 3:Novell NetWare

2:Unix

Web Technologies

4:PHP 1:DHTML 3:ASP 2:Python

## Vārdnīca no sarakstiem:

```
%Soft = (  
    "Operating Systems" => [  
        "Windows", "Unix",  
        "Novell NetWare", "Solaris" ],  
    "Object-Oriented Languages" => [  
        "Java", "C++", "Ada"  
    ],  
    "Web Technologies" => [  
        "DHTML", "Python", "ASP", "PHP"  
    ]  
);
```

---

## Elementa adresēšana:

```
print $Soft{"Web Technologies"}[0]; #DHTML
```

## Informācijas izvade:

```
for $Key (keys %Soft) {  
    print "$Key\n";  
    for $Elem (@{$Soft{$Key}}) {  
        print "$Elem ";  
    }  
    print "\n";  
}
```

---

## Rezultāti:

Object-Oriented Languages

Java C++ Ada

Operating Systems

Windows Unix Novell NetWare Solaris

WEB Technologies

DHTML Python ASP PHP

Lasīšana no faila uz sarakstu:

```
open (DATA, "<data.txt");  
@Data = <DATA>;  
close (DATA);
```

---

Atmest pēdējo simbolu "\n" :

```
chop @Data;
```

---

Rindiņu lasīšana **while** ciklā:

```
while (defined($String = <DATA>) ) {  
    print $String;  
}
```

---

Rindiņu lasīšana: pavienkāršotais kods.

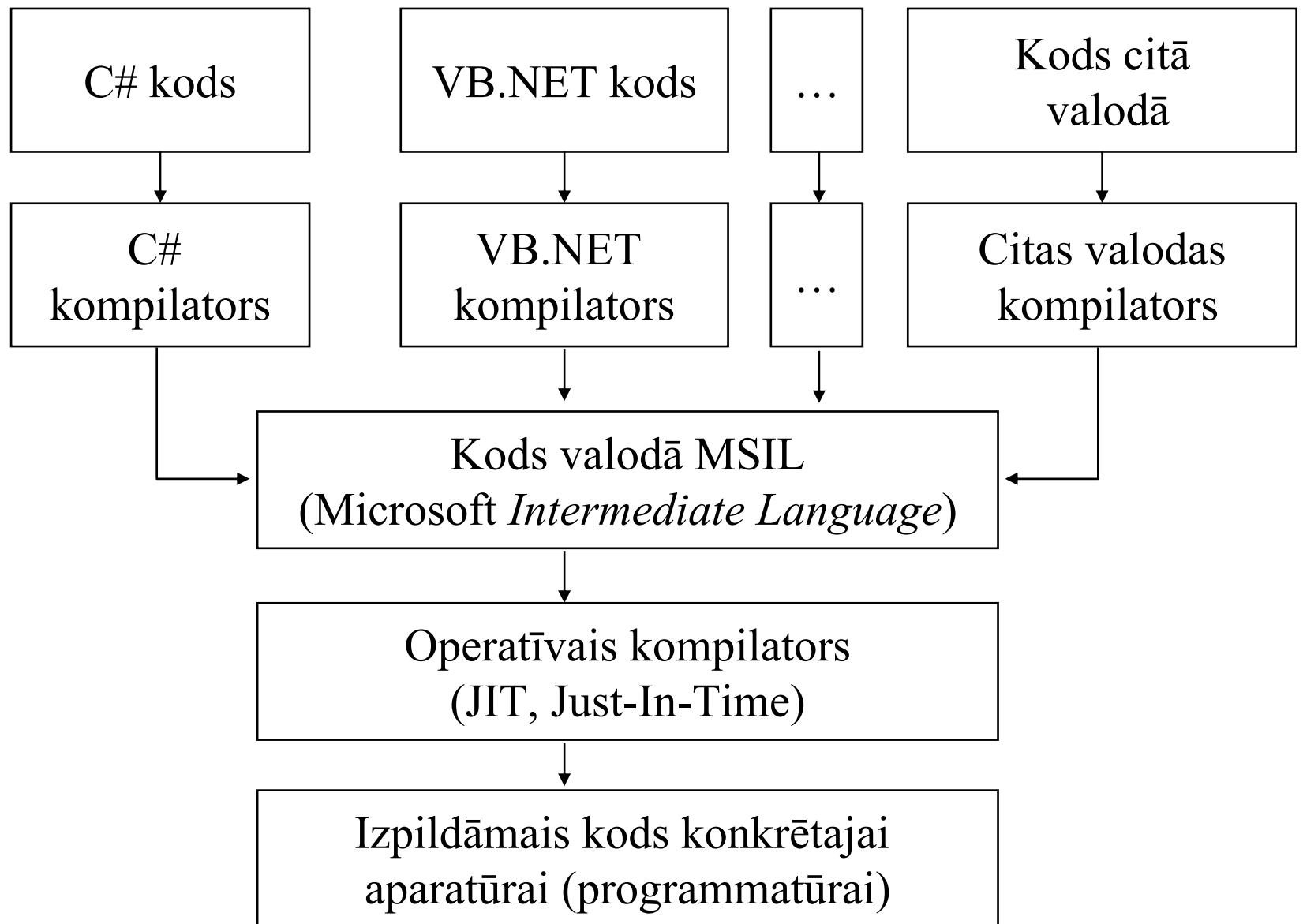
```
while (<DATA>) {  
    print;  
}
```



# Programmēšanas valodas **C#** pamatprincipi

## Literatūras saraksts:

1. *Троелсен Эндрю*. С# и платформа .NET.  
Санкт-Петербург, издательский дом “Питер”, 2004. 796 lpp.
2. *Шилдт Герберт*. С# 2.0. Серия «Полное руководство».  
Москва, издательство «ЭКОМ Паблишерз», 2007. 976 lpp.
3. *Нейгел Кристиан, Ивѐн Билл и др.* С# 2005 для профессионалов.  
Москва, издательский дом “Вильямс”, 2006. 1376 lpp.
4. *Дейтел Харви, Дейтел Пол и др.* С#.  
Санкт-Петербург, издательство “БХВ-Петербург”, 2006. 1056 lpp.
5. *Балена Франческо, Димауро Джузеппе*. Современная практика программирования на Microsoft Visual Basic и Visual C#.  
Москва, “Русская редакция”, 2006. 640 lpp.



## .NET sastāvdaļas

### Bāzes klašu bibliotēka (Base Class Library)

Piekluve  
datiem

GUI

Pavedienu  
kontrolē

XML/SOAP

...



### Common Language Runtime (CLR)

Common Type System  
(CTS)

Common Language Specification  
(CLS)

**CLR** – zema līmeņa darbs: atmiņas kontrole, valodu savstarpēja iedarbība, ...

**CTS** – visu datu tipu aprakstīšana, informācija par to pārstāvēšanu *metadatu* veidā.

**CLS** – kopīgo datu tipu apakškopa. Šos tipus var izmantot visās .NET valodās.

---

Tipu nesavietojamības problēma:

System.Int16: **short** (C#) un **Short** (VB.NET)

System.UInt16: **ushort** (C#) un **neeksistē** (VB.NET)

Valodas **C#** vēsture:

Versija	Specifikācija	<i>Microsoft</i> kompilators
	(12/1998)	
1.0	12/2001	01/2002
2.0	12/2002	11/2005
3.0	06/2005	11/2006
4.0	06/2006	04/2010

Valoda **C#** ir specificēta starptautiskajos standartos ECMA-334 un ISO/IEC 23270.

Etalonkompilators ir *Microsoft Visual C#*.

Valoda **C++** ir valodas **C** objektorientētā uzbūve.

Valoda **Java** ir valodas **C++** „attīrītā” versija.

Valoda **C#** ir valodas **Java** „attīrītā” versija.

---

Elementāra programma C# valodā:

```
using System;
class Hello {
    static void Main(string [] args) {
        Console.WriteLine("Hello, World !");
    }
}
```

---

Ieejas punkts: `Main(...)`, nevis `main(...)`

Komandrindas parametri: `string [] args`, nevis  
`string args []`

## Testpiemēra pamatjēdzieni

1. `System` – vārdu telpa.
2. `Console` – klase.
3. `WriteLine()` – klases `Console` statiskā metode.
4. **`static`** – vienīguma modifikators.
5. **`string`** [] `args` – komandrindu parametru masīvs
6. `String` vai **`string`** – parametru-objektu klase

---

```
Console.WriteLine(typeof(String));  
    //System.String
```

```
Console.WriteLine(typeof(string));  
    //System.String
```



Metodei `Main()` var būt arī citas formas.

Tas atšķir C# no Java.

---

```
//piekļuves modifikators public
public static void Main(string [] args) {

//nav parametru saraksta
static void Main() {

//"int" skaitļa atgriešana (neder "short")
static int Main(string[] args) {
    ...
    return 1;
}

//citi piekļuves modifikatori
private static void Main(string [] args) {...
protected static void Main(string [] args) {...
```

Rezervēto vārdu **using** var neizmantot.

Tad vārdu telpa ir *prefikss*.

```
class Hello {
    static void Main(string [] args) {
        System.Console.WriteLine("Hello, World !");
        System.Console.ReadLine(); //aizture
    }
}
```

---

Komandrindas parametru saraksta apstrāde:

```
for(int i=0; i<args.Length; i++) {
    Console.WriteLine("{0}. {1}", (i+1), args[i]);
}
```

{ i } – parametra numurs

Programmas vārds var būt *patvaļīgs*.

Sistēma meklēs metodi *Main()* visās klasēs.

---

```
using System;
class First {
    static void Main() {
        Console.WriteLine("First"); //rezultāts
    }
}
class Second {
    static void Info() {
        Console.WriteLine("Second");
    }
}
```

---

Ja apmainīt *Main()* un *Info()* vārdus, rezultāts būs:  
*Second*.

## C# programmas kods pēc noklusēšanas (*Microsoft Visual Studio 2005*)

---

```
using System;
using System.Collections.Generic;
using System.Text;
namespace ConsoleApplication1 {
    class Program {
        static void Main(string[] args) {
        }
    }
}
```

---

Informācija par vidi: klase *Environment*.

Environment.MachineName //LAB-1

Environment.SystemDirectory

//D:\WINDOWS\system32

## Datu tipi

### 1. *Value-based. Strukturālie* tipi.

Atmiņu izdala *stekā*. Piešķir elementa *kopiju*.

Skaitliskie datu tipi (**int**, **float**, ...), uzskaitījumi (**enum**), struktūras (**struct**).

### 2. *Reference-based. Norāžu* tipi.

Atmiņu izdala vadāmajā *kaudzē*. Piešķir elementa *norādi*.

Klases (**class**), interfeisi (**interface**).

---

## Tipi un pseidonīmi

```
short i=2;  
i++; //3
```

```
Int16 i=2;  
i++; //3
```

## Formāli:

1. gadījumā *i* ir tipa *short* piemērs.
2. gadījumā *i* ir klases *Int16* objekts.

**Faktiski:** *short* ir klases *Int16* pseidonīms.

---

Ir iespējama tiešā piešķire:

```
short i1 = 2;  
Int16 i2 = i1;
```

Informācija par *short* un *Int16*:

```
Console.WriteLine(typeof(Int16));  
// System.Int16  
Console.WriteLine(typeof(short));  
// System.Int16
```

**Nav** konstruktora ar vienu parametru:

```
Int16 i = new Int16(3); //klūda
```

Ir tikai konstruktors “pēc noklusēšanas”.

```
Int16 S16 = new Int16();
```

Pārveidošanu piemērs

```
int i1 = 2; // (1)
```

```
object O = i1; // (2)
```

```
Int32 i2 = (int) O; // (3)
```

Komentāri:

1. Rindīnā (2) notiek *iepakošana*. Struktūras tipa objektu pārveido norādes objektā.
2. Rindīnā (3) izpildās pretēja operācija – *izpakošana*.
3. Ja rindīnā (3) tiks norādīts tips **byte**, **short** vai daži citi tipi, notiks izņēmums *System.InvalidCastException*.

## Primitīvie datu tipi (*veselie skaitļi*)

1. **sbyte** (1 baits, zīmju skaitlis). Klase *SByte*.
2. **byte** (1 baits, bezzīmju skaitlis). Klase *Byte*.
3. **short** (2 baiti, zīmju skaitlis). Klase *Int16*.
4. **ushort** (2 baiti, bezzīmju skaitlis). Klase *UInt16*.
5. **int** (4 baiti, zīmju skaitlis). Klase *Int32*.
6. **uint** (4 baiti, bezzīmju skaitlis). Klase *UInt32*.
7. **long** (8 baiti, zīmju skaitlis). Klase *Int64*.
8. **ulong** (8 baiti, bezzīmju skaitlis). Klase *UInt64*.



## Primitīvie datu tipi (*turpinājums*)

9. **float** (4 baiti). Klase *Single*.

10. **double** (8 baiti). Klase *Double*.

11. **decimal** (16 baiti). Klase *Decimal*.

12. **bool** (true, false). Struktūra *Boolean*.

13. **char** Struktūra *Char*.

---

Nav iespējama piešķire:

```
sbyte sb = 1 + 127; // kompilācijas kļūda
```

Pārpildes sekas:

```
sbyte sb = 127;  
sb++; // -128
```

## Datu tipu pārveidošana

Klases *Convert* izmantošana:

```
string S;
int Num;
S = Console.ReadLine();
Num = Convert.ToInt32(S); // (*)
Console.WriteLine("Square: {0}.", Num*Num);
```

Rindīnai (\*) var būt arī citas realizācijas:

```
Num = int.Parse(S);
Num = Int32.Parse(S);
```

---

```
Num = byte.Parse(S); // pareizi
Num = long.Parse(S); // kļūda
```

Dažas citas *Convert* pārveidošanas:

```
string    S = "12";
ushort   us = Convert.ToUInt16(S);
uint      ui = Convert.ToUInt32(S);
ulong     ul = Convert.ToUInt64(S);
```

Teksta rindas iegūšana

```
uint N = 12;
string S;
...
// 12 students.
S = N + " students.";
S = N.ToString() + " students.";
S = Convert.ToString(N) + " students.";
S = string.Format("{0}", N) + " students.";
```

## Informācijas izvade

```
int One=1, Two=2;
Console.WriteLine("{0} {1}", One, Two); // 1 2
Console.WriteLine("{1} {0}", One, Two); // 2 1
```

---

## Objekti un struktūru piemēri

Lai ir divas rindiņas:

```
CoordPoint CP1 = new CoordPoint(4, 5);
CoordPoint CP2 = CP1;
```

1. CP1 un CP2 ir *struktūras*. Rezultāts: patstāvīga *kopija*. CP1 izmaiņas nebūs saistītas ar CP2.
2. CP1 un CP2 ir *objekti*. Rezultāts: *norāde*. CP1 izmaiņas būs saistītas ar CP2.

## Elementāra klase “koordinātpunkts” (sākums)

```
class CoordPoint {
    private const int DefX = 2; //konstante
    private const int DefY = 3; //konstante
    private int x;
    private int y;
    public CoordPoint(int x, int y)      {
        this.x = x;
        this.y = y;
    }

    //konstruktors pēc noklusēšanas
    public CoordPoint():this(DefX, DefY){}

    //destruktors
    ~CoordPoint() {
        Console.WriteLine("Done !");
    }
}
```

## Elementāra klase “koordinātpunkts” (beigas)

```

public int GetX() {
    return x;
}
public int GetY() {
    return y;
}
public void SetX(int x) {
    this.x = x;
}
public void SetY(int y) {
    this.y = y;
}
public override string ToString() {
    return "X: " + x + ", Y: " + y;
}
}
    
```

## Darbs ar objektiem galvenajā programmā

```
class Demo {  
    static void Main(string[] args)    {  
        CoordPoint CP1 = new CoordPoint();  
        CoordPoint CP2 = new CoordPoint(5, 6);  
        Console.WriteLine(CP1); // X: 2, Y: 3  
        Console.WriteLine(CP2); // X: 5, Y: 6  
        ...  
    }  
}
```

---

### Piezīmes:

1. Atribūtus var apvienot vienā rindiņā.

```
private const int DefX=2, DefY=3;  
private int x, y;
```

2. Galvenajā programmā var atdalīt objektu *izveidošanu* no norāžu *deklarēšanas*.

```
CoordPoint CP1, CP2;    // norādes  
/* objektu izveidošana */  
CP1 = new CoordPoint();  
CP2 = new CoordPoint(5, 6);
```

3. Konstruktors *CoordPoint()* izsauc konstruktoru *CoordPoint(int, int)*, izmantojot **this**.

4. Desrutors nav obligāts.

Objekti tiks iznīcināti pēc programmas pabeigšanas.  
Pirms destruktora nevar izmantot **private**,  
**protected**, **public**.

5. Pirms konstantēm aizliegts izmantot modifikatoru **static**.



6. Metode *ToString()* bija pārdefinēta. Obligāti jāizmanto rezervētais vārds **override**, citādi izpildīsies metode *ToString()* no klases *Object*.

---

Var *patstāvīgi* nodrošināt objektu iznīcināšanu:

```
CP1 = null;  
CP2 = null;  
GC.Collect(); // ziņojumi "Done !"  
GC.WaitForPendingFinalizers();
```

Piezīme:

Objektus *CP1* un *CP2* nav iespējams izmantot pēc vērtības **null** piešķires.

Citādi notiks izņēmums *System.NullReferenceException*.

## Daļējās (**partial**) klases

Klases daļas var atrasties *vairākos* failos.

Tādas daļas var kompilēt *neatkarīgi*.

Klase *CoordPoint*: informācija par abscisu  $x$  un konstruktori.

```
partial class CoordPoint {  
    private const int DefX = 2;  
    private int x;  
    public CoordPoint(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    ...  
}
```

Klase *CoordPoint*: informācija par ordinātu *y*.

```
partial class CoordPoint {  
    private const int DefY = 3;  
    private int y;  
    public int GetY() {  
        return y;  
    }  
    public void SetY(int y) {  
        this.y = y;  
    }  
}
```

---

Nekādu citu izmaiņu galvenajā programmā *nebūs*.

Objekti tiks izveidoti *parastajā* stilā.

## Atribūtu vērtību pastāvīgums

Atribūtu-konstanti (**const**) inicializē tikai *deklarācijas laikā*.

Atribūtu “tikai lasīšanai” (**readonly**) inicializē tikai *deklarācijas laikā* vai *konstruktorā*.

```
class Auto {
    private readonly string Firm;
    ...
    public Auto(string Firm) {
        this.Firm = Firm;
    }
    // Metode setAuto(string Firm) nav iespējama.
    // Notiks kompilācijas kļūda
    ...
}
```

Var arī deklarēt atribūtu kā konstanti:

```
private readonly string Firm = "Toyota";
```

## Konstanšu klases

Dažreiz lietderīgi apvienot vairākas konstantes vienā klasē.

```
class Physics {  
    public const double G = 9.8;  
}
```

## Konstanšu izmantošana

```
Console.WriteLine(Physics.G);
```

---

*Problēma:* vēlams aizliegt veidot klases *Physics* objektus.

```
Physics P = new Physics();
```

Ir trīs iespējas atrisināt problēmu.

1. Klase ar **private** konstruktoru pēc noklusēšanas.

```
class Physics {  
    ...  
    private Physics() {}  
}
```

2. Deklarēt klasi kā abstrakto (**abstract**).

```
abstract class Physics {  
    ...  
}
```

3. Deklarēt klasi kā statisko (**static**).

```
static class Physics {  
    ...  
}
```

Šajā gadījumā nav iespējams pat deklarēt norādi.

```
Physics P; //kompilācijas kļūda
```

## Operatoru pārlāde

Uzdevums: noteikt attālumu starp diviem punktiem uz plaknes ar pārlādētā operatora „-” palīdzību.

```
class CoordPoint {
    public static double operator -
        (CoordPoint P1, CoordPoint P2) {
        return Math.Sqrt(Math.Pow(P2.x - P1.x, 2) +
            Math.Pow(P2.y - P1.y, 2) );
    }
}
```

Piezīmes:

1. Operators ir *statiskais*.
2. Operatoram ir *divi parametri* (tas atšķir C# no C++).
3. *Math* ir klase ar matemātiskām konstantēm un metodēm.

## Konstruktoru veidi

1. *Objektu* konstruktori. Pielietoti iepriekšējos piemēros.
2. *Klašu* konstruktori (**static** konstruktori).

```
class Test {  
    public static int Counter;  
    static Test() {  
        Counter = 0;  
        Console.WriteLine("Class constructor !");  
    }  
    public Test() {  
        Console.WriteLine("Object constructor !");  
    }  
}
```



## Programmas fragments

```
Test T1 = new Test(), T2 = new Test();  
Console.WriteLine(Test.Counter);
```

## Programmas rezultāti

```
Class constructor !  
Object constructor !  
Object constructor !  
0
```

### Piezīmes:

1. Statiskā konstruktora izpildei nepieciešams vismaz *viens* objekts.
2. Statiskais konstruktors izpildās tikai *vienu reizi* — uzreiz pirms [pirmā] objektu konstruktora.
3. Statiskajā konstruktorā nevar izmantot piekļuves modifikatoru.
4. Statiskajam konstruktoram jābūt bez parametriem.

## Līdzīga iespēja valodā Java:

```
class Test {  
    static {  
        System.out.println("Static block");  
    }  
    Test() {  
        System.out.println("Object constructor");  
    }  
}
```

## Interfeisu būtība

C# interfeisos var būt tikai *metožu deklarācijas*.

Atšķirībā no Java, interfeisā **nevar** deklarēt *konstantes*.

```
interface IPhysics {  
    double G=9.83; //klūda  
}
```

C# valodā ieteicams sākt interfeisu vārdu ar burtu **I** (*ungāru notācija*).

Objektu iznīcināšana *bez destruktora*

```
class CoordPoint : IDisposable {  
    ...  
    public void Dispose() { // IDisposable metode  
        GC.SuppressFinalize(this);  
        Console.WriteLine("Destroyed !");  
    }  
    ...  
}
```

## Īpašības

Var atteikties no *Get()* un *Set()* metodēm un izmantot *get/set* īpašības (vai kādu vienu no īpašībām).

```
class CoordPoint {
    private int x;
    private int y;
    ...
    public int X {
        get {return x;}
        set {x = value;}
    }
    public int Y {
        get {return y;}
        set {y = value;}
    }
}
```

## Programmas fragments

```
CoordPoint CP = new CoordPoint(1, 2);  
Console.WriteLine(CP);    // X: 1, Y: 2  
Console.WriteLine(CP.X);  // 1 (get)  
CP.X = 0;                 // set  
Console.WriteLine(CP.X);  // 0
```

*get* īpašības piemērs:

```
public int X {  
    get {  
        return x;  
    }  
}
```

Īpašībās var pārbaudīt datu pareizību.  
Piemēram, var ierosināt *izņēmumus*.

## Struktūras

Visas struktūras ir tipa *System.ValueType* piemēri.

```
struct CoordPoint : IDisposable {
    private int x, y;
    public CoordPoint(int x, int y)      {
        this.x = x;
        this.y = y;
    }
    public void Dispose() {
        Console.WriteLine("Done !");
    }
    ... // viss, kā bija
}
```

Struktūru piemēru izveidošana:

```
CoordPoint CP1 = new CoordPoint(); // X: 0, Y: 0
CoordPoint CP2 = new CoordPoint(5, 6);
// X: 5, Y: 6
```

## Struktūru īpatnības

1. **Nav** destruktoru. Destruktori var būt *tikai klasēs*.
  2. **Nav** iespējams realizēt konstruktoru *pēc noklusēšanas*.
  3. **Nav** iespējams inicializēt atribūtus *deklarācijas laikā*.
  4. **Nav** iespējama mantošana, pat no klases **object**.
- struct** CoordPoint : **object** { // kļūda
5. Var realizēt interfeisus, piemēram, *IDisposable*.
  6. Var pārdefinēt dažas metodes: piemēram, *ToString()*.

```
struct CoordPoint {
    ...
    public override string ToString() {
        return "X:" + x + ", Y:" + y;
    }
}
```

7. Var izveidot gan ar operatoru `new`, gan bez tā (šajā gadījumā objektu varēs lietot tikai kad būs inicializēti visi atribūti).

## Uzskaitāmie tipi

Visi uzskaitāmie tipi manto no klases *System.Enum*.

Lai ir trīs krāsas:

```
enum Colors {Red, Green, Blue};
```

Informācijas izvade:

```
for(Colors C=Colors.Red; C<=Colors.Blue; C++) {  
    Console.WriteLine(C);  
}
```

Cita iespēja iegūt to pašu rezultātu – datu tipu pārveidošana:

```
for(int C=0; C<=2; C++) {  
    Console.WriteLine((Colors) C);  
}
```



Uzskaitāmā tipa vērtību vai arī iegūt no teksta rindas:

```
Colors C;  
string S = "Red";  
C = (Colors) Enum.Parse(typeof(Colors),  
    S, true);
```

*Parse()* metodes parametri:

1. Uzskaitāmais tips.
2. Teksta rinda.
3. Simbolu reģistra ignorēšana.

Lai ir koda fragments:

```
string S = "red";  
C = (Colors) Enum.Parse(typeof(Colors),  
    S, false);
```

Rezultāts: izņēmums *System.ArgumentException*.

## Vārdu telpas

Lai vārdu telpa RTU satur trīs klases:

```
namespace RTU {  
    class Human{ }  
    class Student{ }  
    class Teacher{ }  
}
```

Darbs ar klasēm programmā:

1. Vārdu telpu izmanto *kā prefiksu* konkrētos gadījumos.

```
RTU.Human H = new RTU.Human();
```

2. Pieslēdz *visu* vārdu telpu.

```
using RTU;
```

```
...
```

```
Human H = new Human();
```

Instrukciju **using** izmanto pirms vārdu telpas deklarēšanas.

### 3. Izveido *pseidonīmu*.

Lai ir vēl viena vārdu telpa: LU. Klašu vārdi pilnīgi vai daļēji sakrīt.

```
namespace LU {
    class Human{ }
    class Student{ }
    class Teacher{ }
}
```

Abas vārdu telpas ir pieslēgtas:

```
using RTU;
using LU;
```

Programmā ir rindiņa:

```
Human H = new Human(); //kļūda
```

**Problēma:** nav skaidrs, *par kuru* vārdu telpu ir runa.

## Problēmas risinājumi:

a. Vārdu telpa – prefikss (princips jau bija parādīts).

```
RTU.Human RH = new RTU.Human();  
LU.Human LH  = new LU.Human();
```

b. Var izveidot vārdu telpas *pseidonīmu*.

```
using RTU_Human = RTU.Human;  
using LU_Human  = LU.Human;
```

Izveidotu pseidonīmu izmantošana programmā:

```
RTU_Human RH = new RTU_Human();  
LU_Human LH  = new LU_Human();
```

## Mantošana

Visas C# klases ir klases **object** (*Object*) apakšklases.

```
using System;
class Hello : object {
    ...
}
```

Ir iespējama tikai *vienkāršā* mantošana.

C# atļauj imitēt *daudzkāršo* mantošanu ar interfeisu palīdzību.

Uzdevums: izveidot klases *CoordPoint* apakšklasi *DisplayPoint* displeja punkta aprakstīšanai.

Izmaiņas klasē *CoordPoint*:

```
...
protected int x, y; // bija private
```

## Klase *DisplayPoint*

Jauns atribūts, konstruktori, destruktors.

```
class DisplayPoint : CoordPoint {  
    protected int Color;  
    protected const int DefColor = 4;  
    public DisplayPoint(int x, int y,  
        int Color) : base(x, y) {  
        this.Color = Color;  
    }  
    public DisplayPoint() : base() {  
        this.Color = DefColor;  
    }  
    ~DisplayPoint() {  
        Console.WriteLine("Done Subclass!");  
    }  
}
```

## Klase *DisplayPoint* (turpinājums)

Piekluves metode, modifikators, informācijas atgriešanas metode.

```
public int GetColor() {
    return Color;
}
public void SetColor(int Color) {
    this.Color = Color;
}
public override string ToString() {
    return base.ToString() + ", Color: " +
        Color;
}
}
```

Piezīme:

Ar **base ()** palīdzību izsauc superklases konstruktorus un metodes.

Izmantot apakšklasē superklases konstruktoru nav obligāti (tas **atšķir** C# no Java), bet izdevīgi.

---

Var aizliegt *turpmāko mantošanu* no klases

```
sealed class DisplayPoint : CoordPoint {
```

Tajā pašā stilā var aizliegt *metožu pārdefinēšanu*

```
sealed public int GetX() {  
    return x;  
}
```



## Metožu pārdefinēšana

Lai klasēs *CoordPoint* un *DisplayPoint* ir metode *Draw()*, kura nodrošina punkta grafisko izvadi.

```
class CoordPoint {  
    public void Draw() {  
        Console.WriteLine("CoordPoint !");  
    }  
}  
  
class DisplayPoint : CoordPoint {  
    public new void Draw() {  
        Console.WriteLine("DisplayPoint !");  
    }  
}
```

## Piezīmes:

1. Apakšklasē **nevar** izmantot vārdu **override** – superklases metode *Draw()* nav virtuālā (**virtual**), abstraktā (**abstract**) vai pārdefinētā (**override**).
  2. Apakšklasē bez vārda **new** būtu tāds pats efekts. Tikai būtu kompilatora brīdinājums (jābūt **new**).
- 

## Programmas fragments:

```
CoordPoint CP1 = new CoordPoint();  
CoordPoint CP2 = new DisplayPoint();  
DisplayPoint DP = new DisplayPoint();
```

## Rezultāti:

```
CP1.Draw();    // CoordPoint !  
CP2.Draw();    // CoordPoint !  
DP.Draw();     // DisplayPoint !
```

Lai metode *Draw()* klasē *CoordPoint* ir virtuālā:

```
class CoordPoint {
    public virtual void Draw() {
        ...
    }
```

a. Apakšklases metodi *Draw()* apraksta ar **new** palīdzību. Pēc noklusēšanas vai **virtual** – tāds pats efekts, tikai ar kompilatora brīdinājumu.

```
class DisplayPoint : CoordPoint {
    public new void Draw() {
        ...
    }
```

Rezultāti pilnīgi sakrīt ar iepriekšējiem rezultātiem.

```
CP1.Draw(); // CoordPoint !
CP2.Draw(); // CoordPoint !
DP.Draw();  // DisplayPoint !
```

b. Apakšklases metodi *Draw()* apraksta ar **override** palīdzību.

```
class DisplayPoint : CoordPoint {
    public override void Draw() {
        ...
    }
```

Rezultāti:

```
CP1.Draw(); // CoordPoint !
CP2.Draw(); // DisplayPoint !
DP.Draw();  // DisplayPoint !
```

Trīs rezervētie vārdi: **sealed**, **virtual**, **override** ir *mantošanas modifikatori*.

Piezīme: pārdefinējamo metodi var arī deklarēt kā **abstract**. Tas notiek *abstraktajās* klasēs.

## Masīvi

Masīvi ir klases *System.Array* objekti.

Viendimensiju masīva deklarācija:

```
int [] V = {1, 2, 3};
```

Var deklarēt masīvu arī tā:

```
int [] V = new int [] {1, 2, 3};
```

Masīva izvade uz ekrānu

```
for(int i=0; i<V.Length; i++)  
    Console.WriteLine("{0} ", V[i]);    // 1 2 3
```

Elementu daudzums *pirmajā dimensijā*:

```
for(int i=0; i<V.GetLength(0); i++)  
    Console.WriteLine("{0} ", V[i]);    // 1 2 3
```

Var izmantot darbā *augšējo* un *apakšējo* indeksus:

```
for (int i=V.GetLowerBound(0);
     i<=V.GetUpperBound(0); i++)
    Console.WriteLine("{0} ", V[i]); // 1 2 3
```

Indeksu vērtības:

```
Console.WriteLine("{0} {1}",
    V.GetLowerBound(0), V.GetUpperBound(0)); // 0 2
```

---

Var atdalīt masīva deklarāciju no izveidošanas.

1. Masīva *deklarācija*.

```
int [] V;
```

2. Masīva *izveidošana*.

```
V = new int [3];
```

```
V[0] = 1;
```

```
...
```

Lai ir divi divdimensiju masīvi.

```
int [] V1 = {1, 2, 3};
int [] V2 = new int[V1.Length];
```

*Norāžu piešķire (ir saite ar oriģinālu)*

```
V2 = V1;      //V2: 1 2 3
V1[2]= 4;     //V2: 1 2 4
```

*Masīva kopēšana (nav saites ar oriģinālu)*

```
V1.CopyTo(V2, 0); //V2: 1 2 3
V1[2]= 4;         //V2: 1 2 3
```

Lai ir kods:

```
int []V2 = new int[V1.Length+2];
V1.CopyTo(V2, 2); //V2: 0 0 1 2 3
```

## Divdimensiju masīvi

### *Taisnstūrainie masīvi*

```
int [,] M = { {1, 2, 3}, {4, 5, 6} };
```

### **Nav** iespējams

```
int [,] M = { {1, 2, 3}, {4, 5, 6, 7} };
```

### Elementa adresēšana

```
Console.WriteLine(M[0, 1]); //2
```

### **Nav** iespējams

```
Console.WriteLine(M[0][1]); //kļūda
```

### Elementu daudzums masīvā (visās dimensijās)

```
Console.WriteLine(M.Length); // 6
```

### Fiksētā izmēra pārbaude

```
Console.WriteLine(M.IsFixedSize); // True
```



*Dimensiju daudzums:*

```
Console.WriteLine(M.Rank); // 2
```

*Rindu daudzums:*

```
Console.WriteLine(M.GetLength(0)); // 2
```

*Kolonu daudzums:*

```
Console.WriteLine(M.GetLength(1)); // 3
```

Elementu daudzums *katrā dimensijā:*

```
for(int i=0; i<M.Rank; i++)  
    Console.WriteLine(M.GetLength(i));
```

Elementu izvade uz ekrānu

```
for(int i = 0; i<M.GetLength(0); i++) {  
    for(int j = 0; j<M.GetLength(1); j++)  
        Console.Write(" {0}", M[i,j]);  
    Console.WriteLine();  
}
```

*Robainie masīvi*

```
int [][] M = new int[3][];  
M[0] = new int [] {1, 2};  
M[1] = new int [] {3, 4, 5};  
M[2] = new int [] {6};
```

*Robainu masīvu izvade uz ekrānu*

```
for(int i=0; i<M.Length; i++) {  
    for(int j=0; j<M[i].Length; j++)  
        Console.Write("{0} ", M[i][j]);  
    Console.WriteLine();  
}
```

**Rezultāti**

```
1 2  
3 4 5  
6
```

## Masīvu kārtošana

Interfeiss *Comparable*

Java valodas analogs: interfeiss *Comparable*.

Interfeisa deklarācija

```
interface Comparable {  
    int compareTo(object O) ;  
}
```

Programmētāja pienākums: realizēt sakārtojamo objektu klasē metodi *compareTo(object O)*.

Metode *compareTo()* salīdzina kaimiņelementus un atgriež  $\{-1, 0, 1\}$  (vispārīgajā gadījumā: {negatīvs skaitlis, 0, pozitīvs skaitlis}).

## 1. Lai ir klase *Auto*

```
class Auto : IComparable {  
    private String Name;  
    private float Price;  
    ...  
    public int CompareTo(object O) {  
        float P = ((Auto) O).Price;  
        return (Price < P) ? -1 : ((Price > P) ? 1 : 0);  
    }  
}
```

## 2. Automašīnu masīvs

```
Auto [] Autos = {  
    new Auto("Ford", 5000),  
    new Auto("Mersedes", 3000),  
    new Auto("Renault", 4000)  
};
```

### 3. Objektu kārtošana

```
Array.Sort(Autos);
```

### 4. Masīva elementu izvade

```
foreach (Auto C in Autos) {  
    Console.WriteLine(C);  
}
```

---

Interfeiss *IComparer*

Java valodas analogs: interfeiss *Comparator*.

Interfeiss atrodas vārdu telpā *System.Collections*.

```
interface IComparer {  
    int Compare(object O1, object O2);  
}
```

1. Lai ir klase *Auto* bez interfeisa realizācijas

```
using System.Collections;
...
class Auto {
    private String Name;
    private float Price;
    ...
    public float GetPrice() {
        return Price;
    }
}
```

2. Papildu klase *AutoPriceASC*

```
class AutoPriceASC : IComparer {
    public int Compare(object O1, object O2) {
        float P1 = ((Auto) O1).GetPrice();
        float P2 = ((Auto) O2).GetPrice();
        return (P1<P2)?-1:((P1>P2)?1:0);
    }
}
```

3. Objektu kārtošana. Tiks izveidots jauns objekts

```
Array.Sort(Autos, new AutoPriceASC());
```

4. Masīva elementu izvade

```
foreach (Auto C in Autos) {  
    Console.WriteLine(C);  
}
```

Risinājuma trūkums: lietotājs domās par citas klases objektu izveidošanu. Klasē *Auto* var realizēt papildu metodi:

```
class Auto {  
    ...  
    public static IComparer AutoPriceASC() {  
        return (IComparer)new AutoPriceASC();  
    }  
}  
...  
Array.Sort(Autos, Auto.AutoPriceASC());
```

## Parametru nodošana

*Uzdevums:* jāatrod veselu skaitļu summa. Skaitļu daudzums nav ierobežots.

Ir divi risinājumi.

1. Nodot kā parametru veselu skaitļu *masīvu*.

```
class C {
    static int Sum1 (int [] L) {
        int Sum = 0;
        foreach (int El in L) {
            Sum += El;
        }
        return Sum;
    }
}
```



## 2. Parametru *sarakstā* izmantot rezervēto vārdu **params**.

```
class C {
    static int Sum2 (params int [] L) {
        int Sum = 0;
        foreach (int E1 in L) {
            Sum += E1;
        }
        return Sum;
    }
}
```

Metožu *Sum1()* un *Sum2()* izmantošana programmā

```
/* Rezultāts abos gadījumos: 6 */
Console.WriteLine(C.Sum1(new int [] {1, 2, 3}));
Console.WriteLine(C.Sum2(1, 2, 3));
```

1. Parametru masīvs ar modifikatoru **params** var būt *tikai viens*.
  2. Tas ir *pēdējais* parametrs sarakstā.
- 

*Uzdevums:* atrast elementa ieiešanas daudzumu masīvā.

```
static int Count(int X, params int[] V) {
    int C = 0;
    foreach(int Elem in V)
        if (Elem == X)
            C++;
    return C;
}
```

Metodes izsaukums

```
Console.WriteLine("Total: {0}.",
    Count(3, 1, 3, 2, 3)); //2
```

## Parametru inicializācija funkcijā

Uzdevums: metode atgriež

1. Masīva elementu *summu* (ar **return** palīdzību).
2. Elementu *vidējo aritmētisko* papildus parametrā.

```
static int SumAvg(int [] L, out float Avg) {
    int Sum = 0;
    foreach(int El in L)
        Sum += El;
    Avg = (float) Sum/L.Length;
    return Sum;
}
```

Rezervētais vārds **out** nodrošina *obligātu* parametra inicializāciju.

Vārdu **out** izmanto arī *metodes izsaukumā*.

## Metodes izsaukums

```
int [] Arr = {1, 2, 3, 4};  
int S;  
float Av;  
S = SumAvg(Arr, out Av); // out!
```

## Piezīmes:

1. Ja “izmest” vārdu **out** no abām rindiņām, notiks kompilācijas kļūda – mainīgais *Av* nav inicializēts pirms metodes izsaukuma.
2. Ja “izmest” vārdu **out** un inicializēt *Av* pirms metodes izsaukuma (lai būs 0), kļūdas nebūs, bet *Av* vērtība neizmainīsies.

## Parametru nodošana pēc norādes

Uzdevums: *apmainīt* parametru vērtības.

```
static void Swap(ref int a, ref int b) {  
    int c;  
    c = a;  
    a = b;  
    b = c;  
}
```

### Metodes izsaukums

```
int x = 2, y = 3;  
Swap(ref x, ref y);    // x=3, y=2
```

Metodes izsaukumā *obligāti* norāda **ref**, citādi notiks kompilācijas kļūda.

## Izņēmumu apstrāde

Visi izņēmumi ir klases *Exception* objekti.

Rezervētie vārdi:

1. **try** – kontrolējamais bloks.
2. **catch** – izņēmuma apstrādātājs.
3. **throw** – izņēmuma ierosināšana.
4. **finally** – vienmēr izpildāmais bloks.

Izņēmumu tveršana:

```
catch (Exception) { ... }
```

Var norādīt izņēmuma objektu un izvadīt informāciju par to.

```
catch (Exception e) {  
    Console.WriteLine(e);  
}
```

Izņēmumu apstrādes piemērs.

```
int [] v = new int[2];
try {
    v[2] = 2;
}
catch (IndexOutOfRangeException) {
    Console.WriteLine("Index out of range.");
}
catch (Exception) {
    Console.WriteLine("Unknown Error.");
}
```

Rezultāts: ziņojums *Index out of range*.

---

Var iegūt informāciju no izņēmuma objekta:

```
catch (IndexOutOfRangeException e) {
    Console.WriteLine("Error: " + e.Message);
}
```

//Error: Index was outside the bounds...

Izņēmumu var izveidot patstāvīgi

```
class MyError : Exception {  
    private int Code = 1;  
    public MyError(int Code) {  
        this.Code = Code;  
    }  
    public override string ToString() {  
        return "Error: code " + Code + ".";  
    }  
}
```

Izņēmuma ierosināšana un apstrāde

```
try {  
    throw new MyError(2);  
}  
  
catch(MyError e) {  
    Console.WriteLine(e); // Error: code 2.  
}
```



Neapstrādājamo izņēmumu var *nedeklarēt*.

Tas atšķir *C#* no *Java*, kur tādos gadījumos izmanto **throws**.

Bloka **finally** izmantošana

```
try {  
    ...  
}  
    ...  
    finally {  
        Console.WriteLine("Done.");  
    }
```

Jebkuru iepriekš nezināmo izņēmumu var apstrādāt arī tā:

```
catch {  
    Console.WriteLine("Unknown Error !");  
}
```

Nebija izmantota klase *Exception*.

*Uzdevums:* pārbaudīt datu tipu pārveidošanas pareizību.

```
short Sh = 128;  
sbyte Sb = 0;  
try {  
    checked {  
        Sb = (sbyte) Sh;  
    }  
}  
catch (OverflowException) {  
    Console.WriteLine("Overflow.");  
}  
Console.WriteLine(Sb); //0
```

Kontrolējamo bloku var saīsināt:

```
try {  
    Sb = checked ((sbyte) Sh);  
}
```

## Refleksija, dinamiskā tipu identifikācija (RTTI)

*Uzdevums:* pārbaudīt superklases mainīgā saturu.

```
CoordPoint CP1 = new CoordPoint(),  
    CP2 = new DisplayPoint();  
DisplayPoint DP = new DisplayPoint();
```

Pārbaude ar operatora *is* palīdzību:

```
... CP1 is CoordPoint           // True  
... CP1 is DisplayPoint        // False  
... CP2 is CoordPoint          // True  
... CP2 is DisplayPoint        // True  
... DP is CoordPoint           // True  
... DP is DisplayPoint         // True
```

Metodes *GetType()* izmantošana:

```
Type t = CP1.GetType(); // CoordPoint  
      t = CP2.GetType(); // DisplayPoint  
      t = DP.GetType();  // DisplayPoint
```

Operatora *typeof* izmantošana:

```
Type t = typeof(CoordPoint)    // CoordPoint  
      t = typeof(DisplayPoint) // DisplayPoint
```

### Tipu pārveidošana

```
DP = (DisplayPoint) CP2;  
DP = CP2 as DisplayPoint;
```

Piezīme: Ja pārveidošana nav iespējama, tad pirmajā gadījumā tiks izraisīts izņēmums *System.InvalidCastException*, bet otrajā gadījumā mainīgajam DP vienkārši tiks piešķirta vērtība *null*.

## Informācijas par metodēm izvade:

```
using System.Reflection;
```

```
...
```

```
Type t = CP1.GetType();
```

```
MethodInfo[] mi = t.GetMethods();
```

```
foreach (MethodInfo m in mi) {
```

```
    Console.WriteLine("{0} {1}", m.ReturnType.Name, m.Name);
```

```
    ParameterInfo[] pi = m.GetParameters();
```

```
    if (pi.Length > 0) {
```

```
        Console.WriteLine("(");
```

```
        foreach (ParameterInfo p in pi)
```

```
            Console.WriteLine("{0} {1}, ", p.ParameterType.Name, p.Name);
```

```
        Console.WriteLine("\b\b)\n");
```

```
    }
```

```
    else
```

```
        Console.WriteLine("()\n");
```

```
}
```

## Programmas izvade:

```
Int32 get_X()  
Void set_X(Int32 value)  
Int32 get_Y()  
Void set_Y(Int32 value)  
Int32 GetX()  
Int32 GetY()  
Void SetX(Int32 x)  
Void SetY(Int32 y)  
String ToString()  
Double op_Subtraction(CoordPoint P1, CoordPoint P2)  
Boolean Equals(Object obj)  
Int32 GetHashCode()  
Type GetType()
```

Piezīme: Eksistē arī metode *GetConstructors()*, kas atgriež informāciju par objekta konstruktoriem.

## Vispārinājumi (*generics*)

### Vispārinātie tipi

```
class CoordPoint<T> {
    private T x;
    private T y;
    public CoordPoint(T x, T y) {
        this.x = x;
        this.y = y;
    }
    public T X {
        get { return x; }
        set { x = value; }
    }
    ...
}
```

Kods galvenajā programmā:

```
CoordPoint<int> C1 = new CoordPoint<int>(10, 20);  
CoordPoint<float> C2 =  
    new CoordPoint<float>(10.5f, 20.5f);  
  
Console.WriteLine("{0}; {1}", C1.X, C1.Y);  
Console.WriteLine("{0}; {1}", C2.X, C2.Y);  
  
Console.WriteLine(C1.GetType());
```

Programmas izvade:

```
(10; 20)  
(10,5; 20,5)  
ConsoleApplication1.CoordPoint`1[System.Int32]
```



Piezīmes:

- 1) *CoordPoint<T>* ir *nekonkretizētais vispārinātais tips*.
- 2) *CoordPoint<int>* ir *konkretizētais vispārinātais tips*.
- 3) *int* ir vispārinātā tipa *arguments*.
- 4) Ir iespējami vispārinātie tipi ar vairākiem argumentiem:

```
class C <T1, T2> { ... }
```

- 5) Ir iespējams ierobežot tipa argumentus:

```
// bāzes klases / interfeisa ierobežojums:
```

```
class C <T1, T2> where T1: A { ... }
```

```
class C <T1, T2> where T1: IB { ... }
```

```
class C <T1, T2> where T1: T2 { ... }
```

```
// konstruktora bez parametriem ierobežojums:
```

```
class C <T1, T2> where T1: new() { ... }
```

5) Ir iespējams ierobežot tipa argumentus (*turp.*):

```
// vērtību / norāžu tipa ierobežojums:  
class C <T1, T2> where T1: struct { ... }  
class C <T1, T2> where T1: class { ... }
```

6) Ir iespējams vienlaicīgi definēt vairākus ierobežojumus:

```
class C <T1, T2>  
    where T1: class, T2, IComparable, new()  
    { ... }
```

```
class C <T1, T2>  
    where T1: IComparable, IEnumerable  
    where T2: struct  
    { ... }
```

## Vispārinātās metodes

```
class Calc {
    public static T getMin<T> (T s1, T s2)
        where T: IComparable<T> {
        return s1.CompareTo(s2) <= 0 ? s1 : s2;
    }
}
```

Kods galvenajā programmā:

```
Console.WriteLine( Calc.getMin<int>(10, 5) );
    // 5
Console.WriteLine( Calc.getMin(10, 5) );
    // 5
Console.WriteLine( Calc.getMin(10, 5.5f) );
    // 5,5
```

## Vispārināto tipu hierarhija

1) Gan bāzes klase, gan atvāsinātā klase ir vispārinātas:

```
class CoordPoint<T> { ... }
```

```
class DisplayPoint<T> : CoordPoint<T> { ... }
```

```
Console.WriteLine( typeof(CoordPoint<>) );
```

```
// ConsoleApplication1.CoordPoint`1[T]
```

```
Console.WriteLine( typeof(CoordPoint<int>) );
```

```
// ConsoleApplication1.CoordPoint`1[System.Int32]
```

```
Console.WriteLine( typeof(DisplayPoint<>) );
```

```
// ConsoleApplication1.DisplayPoint`1[T]
```

```
Console.WriteLine( typeof(DisplayPoint<int>) );
```

```
// ConsoleApplication1.DisplayPoint`1[System.Int32]
```

2) Bāzes klase nav vispārināta, atvāsinātā klase ir vispārināta:

```
class CoordPoint { ... }
```

```
class DisplayPoint<T> : CoordPoint { ... }
```

```
Console.WriteLine( typeof(CoordPoint) );
```

```
// ConsoleApplication1.CoordPoint
```

```
Console.WriteLine( typeof(DisplayPoint<>) );
```

```
// ConsoleApplication1.DisplayPoint`1[T]
```

```
Console.WriteLine( typeof(DisplayPoint<int>) );
```

```
// ConsoleApplication1.DisplayPoint`1[System.Int32]
```

## Datu konteineri

Nenoskaņojamie konteineri atrodas vārdu telpā *System.Collections*:  
**using** System.Collections;

### Klase *Queue* (rinda)

```
Queue Q = new Queue();  
Q.Enqueue("C++"); // [C++]  
Q.Enqueue("Java"); // [C++, Java]
```

### Elementu apstrāde

```
foreach (object El in Q)  
    Console.WriteLine(El); // C++ Java
```

### Pirmā elementa *lasīšana* un *izslēgšana*

```
Console.WriteLine(Q.Peek()); // C++  
Console.WriteLine(Q.Dequeue()); // C++  
Console.WriteLine(Q.Dequeue()); // Java
```

## Klase *ArrayList* (dinamiskais saraksts)

```
ArrayList AL = new ArrayList();
```

### Elementu *pievienošana*:

```
AL.Add("C++"); // C++  
AL.Add("Java"); // C++, Java
```

### Elementu *ielikšana*:

```
AL.Insert(1, "C#"); // C++, C#, Java
```

### Elementu *indeksēšana*:

```
Console.WriteLine(AL[1]); // C#
```

### Elementu *daudzuma noteikšana*:

```
Console.WriteLine(AL.Count); // 3
```

Elementu grupas *pievienošana*:

```
string [] Dyn = {"Python", "Ruby"};  
AL.AddRange(Dyn); // C++, C#, Java, Python, Ruby
```

Elementu grupas *ielikšana* (iepriekšējā piemēra vietā):

```
AL.InsertRange(0, Dyn);  
// Python, Ruby, C++, C#, Java
```

Citas *ArrayList* metodes:

```
int i = AL.IndexOf("C#");  
AL.Reverse();  
AL.Remove("C#");  
AL.RemoveAt(2);  
AL.Clear();
```



## Klase *Hashtable* (vārdnīca)

```
Hashtable HT = new Hashtable();
```

### Elementu *pievienošana*:

```
HT.Add("ASP", "Active Server Pages");  
HT["JSP"] = "JavaServer Pages";
```

### Vārdnīcas izvade uz ekrānu:

```
foreach (DictionaryEntry DE in HT) {  
    Console.WriteLine(DE.Key + " -> " + DE.Value);  
}
```

### Elementu *indeksēšana*:

```
Console.WriteLine(HT["JSP"]); //JavaServer Pages
```

### Atslēgas/vērtības *pārbaude*:

```
bool b = HT.ContainsKey("ASP"); // vai .Contains  
bool b = HT.ContainsValue("Java Platform");
```

## Citi vārdu telpas *System.Collections* datu konteineri (nenoskaņojamie):

BitArray – bitu masīvs

Stack – steks

SortedList – sakārtota pēc atslēgām vārdnīca;  
piekļuve elementiem arī caur indeksēšanu

## Klase *LinkedList<T>* (dinamiskais saraksts)

```
using System.Collections.Generic;
```

```
LinkedList<string> LL =  
    new LinkedList<string>();
```

### Elementu *pievienošana*:

```
LL.AddFirst("C++");    // C++  
LL.AddLast("Java");    // C++, Java
```

## Klase *ListDictionary* (vārdnīca)

```
using System.Collections.Specialized;  
ListDictionary LD = new ListDictionary();
```

### Elementu *pievienošana*:

```
LD.Add("ASP", "Active Server Pages");  
LD["JSP"] = "JavaServer Pages";
```

### Piezīmes:

- 1) Klase *Hashtable* ir bazēta uz jaucējsummu tabulas, un ir efektīva darbā ar lieliem datu (pāru atslēga-vērtība) apjomiem; klase *ListDictionary* ir bazēta uz vienkāršsaistītā saraksta, un ir efektīva darbā ar nelieliem datu apjomiem (10 vai mazāk elementu).
- 2) Eksistē klase *HybridDictionary*, kas, atkarībā no vārdnīcas elementu skaita, automātiski izvēlas piemērotāku konstrukciju datu glabāšanai: *Hashtable* vai *ListDictionary*.

## Klase *Dictionary*<TKey,TValue> (vārdnīca)

```
using System.Collections.Generic;
```

```
Dictionary<string, string> D = new  
    Dictionary<string, string>();
```

### Elementu *pievienošana*:

```
D.Add("ASP", "Active Server Pages");  
D["JSP"] = "JavaServer Pages";
```

### Elementu *pārlase*:

```
foreach (KeyValuePair<string, string> KVP in D)  
    Console.WriteLine( KVP.Key + " -> " +  
        KVP.Value );
```

### Atslēgas/vērtības *pārbaude*:

```
bool b = D.ContainsKey("ASP"); // neder Contains  
bool b = D.ContainsValue("Java Platform");
```

## Nedrošais kods

Uzdevums: uzzināt tipa **int** izmēru (baitos).

```
unsafe {  
    Console.WriteLine(sizeof(int)); // 4  
    Console.WriteLine(sizeof(decimal)); // 16  
}
```

Piezīme: ir papildus projekta parametrs

Project -> Properties -> Build ->  
Allow Unsafe Code: True

*Rādītāju* izmantošana nedrošajā kodā

```
int X = 2;  
unsafe {  
    int*P = &X;  
    Console.WriteLine(*P); // 2  
}
```

*Uzdevums:* apmainīt mainīgo vērtības.

Funkcijas *deklarācija*

```
static unsafe void Swap (int*A, int*B) {  
    int C;  
    C = *A;  
    *A = *B;  
    *B = C;  
}
```

Funkcijas *izsaukums*

```
int A = 1, B = 2; // A=1, B=2  
unsafe {  
    Swap (&A, &B); // A=2, B=1  
}
```

Piezīme: funkciju obligāti izsauc **unsafe** blokā.  
Citādi notiks kompilācijas kļūda.

## Teksta rindu apstrādes īpatnības

Teksta rindu vērtības var izmantot **switch** operatorā:

```
string S = "C#";
switch (S) {
    case "C":
        Console.WriteLine("C"); break;
    case "C++":
        Console.WriteLine("C++"); break;
    case "C#":
        Console.WriteLine("C#"); break;
}
```

*Piezīmes:*

1. Obligāti jāpielieto operators **break** (citādi notiks kompilācijas kļūda).
2. Par iezīmēm var būt tikai konstantes (kā valodās *C* un *C++*).

Teksta rindās var izmantot slīpas līnijas, ja pielietot simbolu @.

```
string S1 = "C:\\WORK";
string S2 = @"C:\\WORK";
Console.WriteLine(S1 + " " + S2);
//C:\WORK C:\WORK
```

Informācijas lasīšana no tastatūras

```
Console.WriteLine("Path:");
string SPath = Console.ReadLine();
Console.WriteLine(SPath); //C:\\WORK
```

Operācijas ar klases *StringBuilder* objektiem

```
using System.Text;
...
StringBuilder SB = new StringBuilder(S);
SB.Insert(0, "C ");
S = SB.ToString();
Console.WriteLine(S); //C C++ Java C#
```



## Izvades formatēšana

```
Console.WriteLine("{0:#.##}", 123456.789);  
123456,79
```

```
Console.WriteLine("{0:# #.##}", 123456.789);  
12345 6,79
```

```
Console.WriteLine("{0,15}", 123456.789);  
123456,789
```

```
Console.WriteLine("{0,-15:F2}!", 123456.789);  
123456,79      !
```

## ***Nullable tipi***

*Nullable*-tipa deklarācija:

```
int? i;  
Nullable<int> i; // alternatīva deklarācija
```

Vērtības piešķiršana:

```
i = null;  
i = 5;
```

Pārbaude uz *null* vērtību:

```
if (i == null) ...  
if (!i.HasValue) ...
```

Vērtības noteikšana:

```
Console.WriteLine(i);  
Console.WriteLine(i.Value);  
// iespējams izņēmums, ja i == null
```

Piemērs:

```
static void Describe(Nullable<int> i) {
    Console.WriteLine(i == null ? "null" : "not null");
    Console.WriteLine(i.HasValue);
    try { Console.WriteLine(i.Value); }
    catch (Exception e) {
        Console.WriteLine(e.GetType()); }
}

...
int? i;

i = null; Describe(i);
// null
// False
// System.InvalidOperationException

i = 5; Describe(i);
// not null
// True
// 5
```

## Piezīmes:

- 1) Par *nullable*-tipu var būt jebkurš *vērtību* tips.
- 2) Darbā ar *nullable*-tipiem ir ērti izmantot operatoru `??`:  
 $(i \ ?? \ m)$  atgriež  $i$ , ja  $i \neq null$ , pretējā gadījumā —  $m$ .
- 3) Pārveidojot *nullable*-tipus uz parastajiem tipiem, notiek griešanās pie īpašības *Value*:

```
int? i = null;
int j = (int)i; // InvalidOperationException

i = 5;
j = (int)i; // OK: j == 5
```

## Delegāti

Delegāts ir droša norāde uz metodi.

Delegāta deklarācija (ārpus metodēm):

```
delegate int D(int i, int j);
```

Delegāta izmantošana:

```
static int Add(int i, int j) { return i + j; }  
static int Sub(int a, int b) { return a - b; }  
...
```

```
D d = new D(Add);  
Console.WriteLine(d(1, 2)); // 3
```

```
d = new D(Sub);  
Console.WriteLine(d(1, 2)); // -1
```

## Anonīmās metodes

Anonīmā metode bez parametriem, bez atgriežamā tipa:

```
delegate void DoSomething();
...
DoSomething deleg = delegate {
    int ii = 5; Console.WriteLine(++i+ii); };
i = 5;
deleg(); // 11
deleg(); // 12
```

Anonīmā metode ar parametriem, ar atgriežamo tipu:

```
delegate int ReturnSomething(int i);
...
ReturnSomething ret =
    delegate(int y) { return y + 1; };
Console.WriteLine(ret(5)); // 6
```

## Lambda-izteiksmes

Iepriekšējās anonīmās metodes saīsinātais pieraksts:

```
delegate int ReturnSomething(int i);
```

```
...
```

```
ReturnSomething ret = z => z + 1;
```

```
Console.WriteLine(ret(5)); // 6
```

Lambda-izteiksmes ar vairākiem parametriem:

```
delegate int D(int i, int j);
```

```
...
```

```
D d = (x, y) => x + y;
```

```
Console.WriteLine(d(1,2)); // 3
```

```
d = (int x, int y) => x + y;
```

```
Console.WriteLine(d(1,2)); // 3
```

## Notikumi

```

delegate void DoSomething();
class A {
    public event DoSomething E;
    public void SignalE() {
        if (E != null) E();
    }
}

...
A a = new A();
DoSomething eventHandler1 = () =>
    Console.WriteLine("Event handled (1). ");
DoSomething eventHandler2 = () =>
    Console.WriteLine("Event handled (2). ");
a.E += eventHandler1; a.E += eventHandler2;
for (int n = 0; n < 2; ++n)
    a.SignalE();
// Event handled (1). Event handled (2).
// Event handled (1). Event handled (2).
    
```



## Netieši tipizēti mainīgie

```
var v = "abc";  
string s = v; // iv == "abc"  
v = 5; // Kompilācijas kļūda: Cannot implicitly  
       convert type 'int' to 'string'
```

## Objekta īpašību inicializācija

```
class C {  
    public int a, b, c;  
    public override string ToString() {  
        return String.Format  
            ("a = {0}, b = {1}, c = {2}", a, b, c);  
    }  
}  
...  
C c = new C { a = 1, b = 2, c = 3 };  
Console.WriteLine(c);  
// a = 1, b = 2, c = 3
```

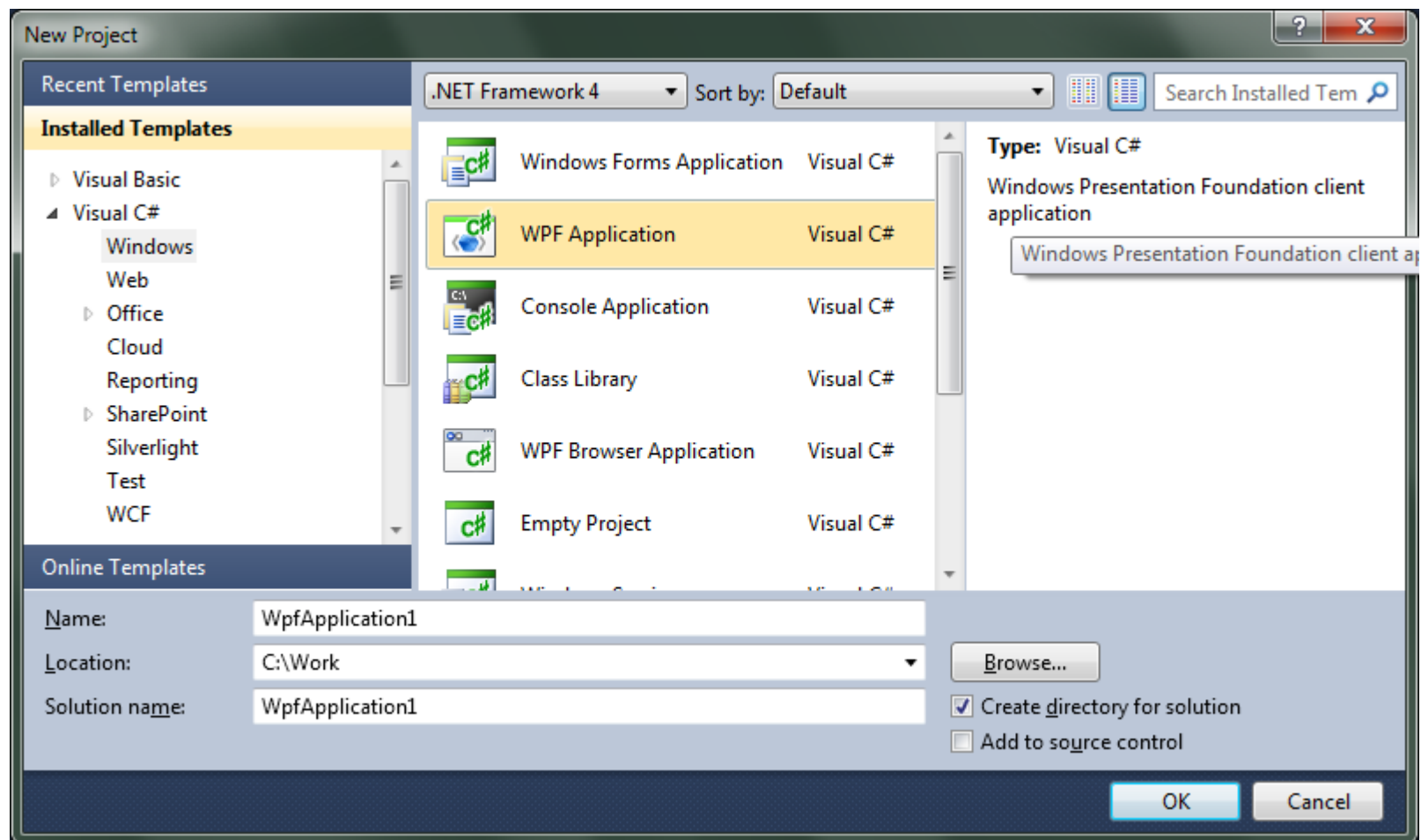
## Windows Presentation Foundation (WPF)

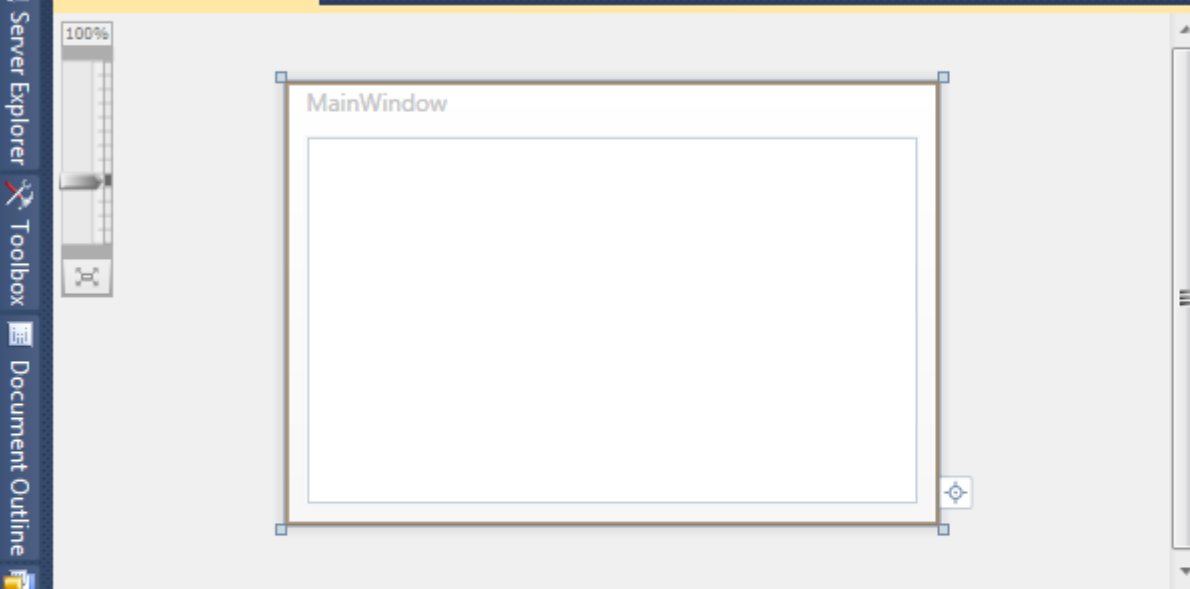
WPF ir platformas Microsoft .NET Framework (sākot no versijas 3.0) sastāvā esošā grafiskā apakšsistēma.

WPF paredz grafiskās lietotāja saskarnes (GUI) atdalīšanu no programmas loģikas.

WPF kā pamatu GUI aprakstam izmanto valodu XAML (Extensible Application Markup Language).

WPF lietojums var tikt palaists gan darbvirsmā, gan tīmekļa pārlūkprogrammā.



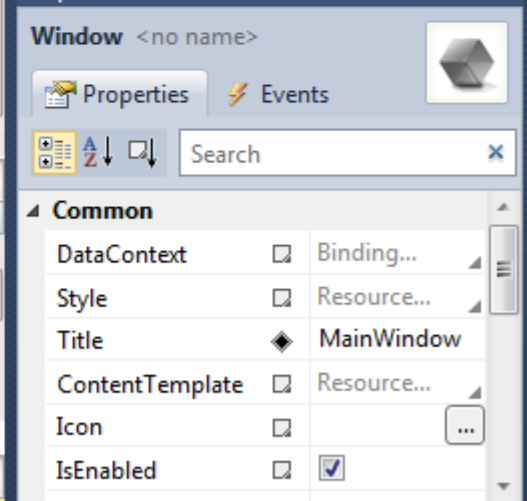
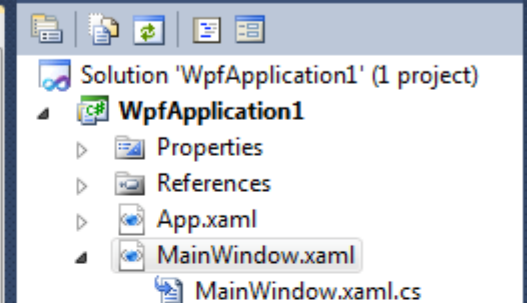


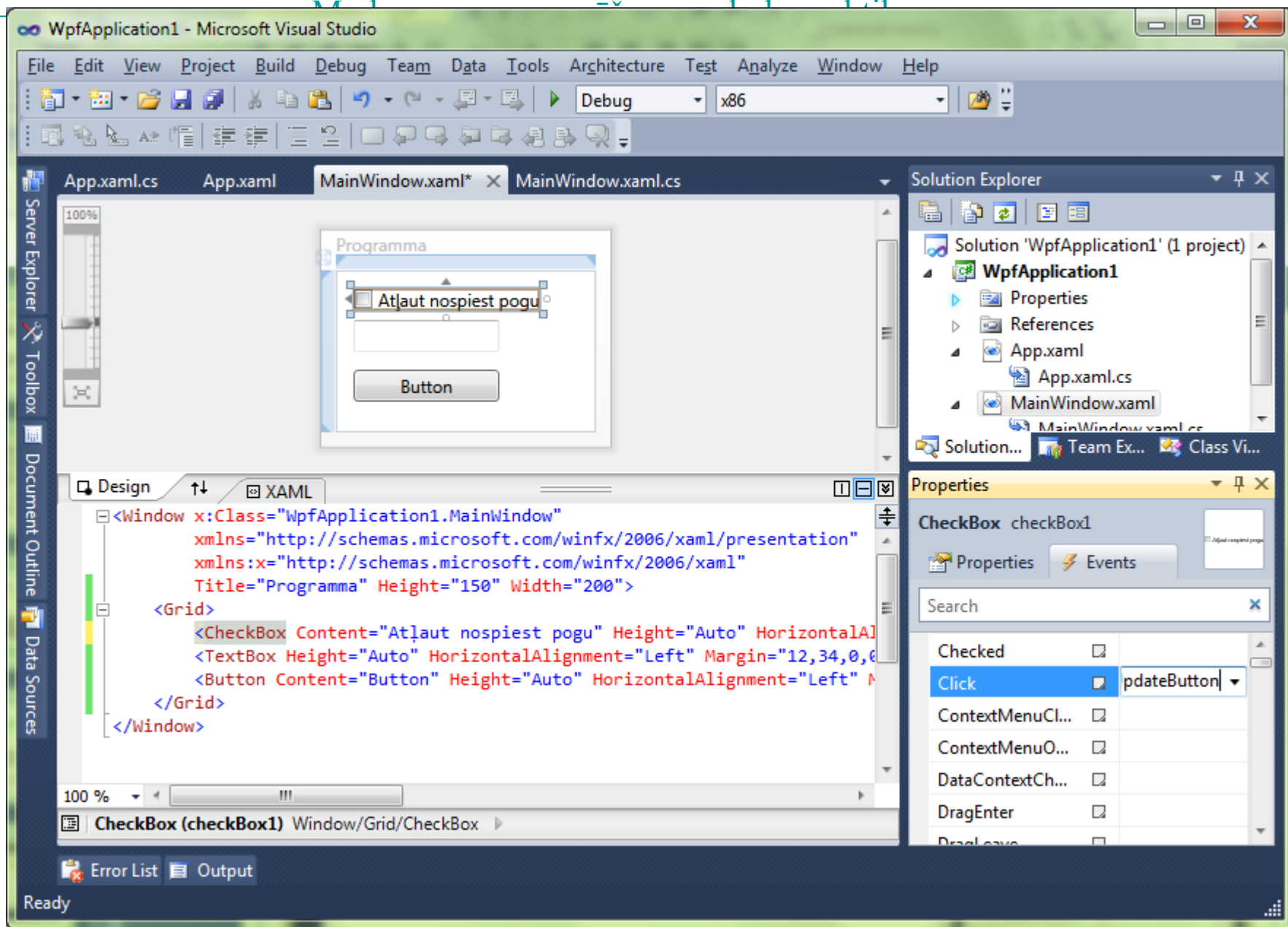
Design XAML

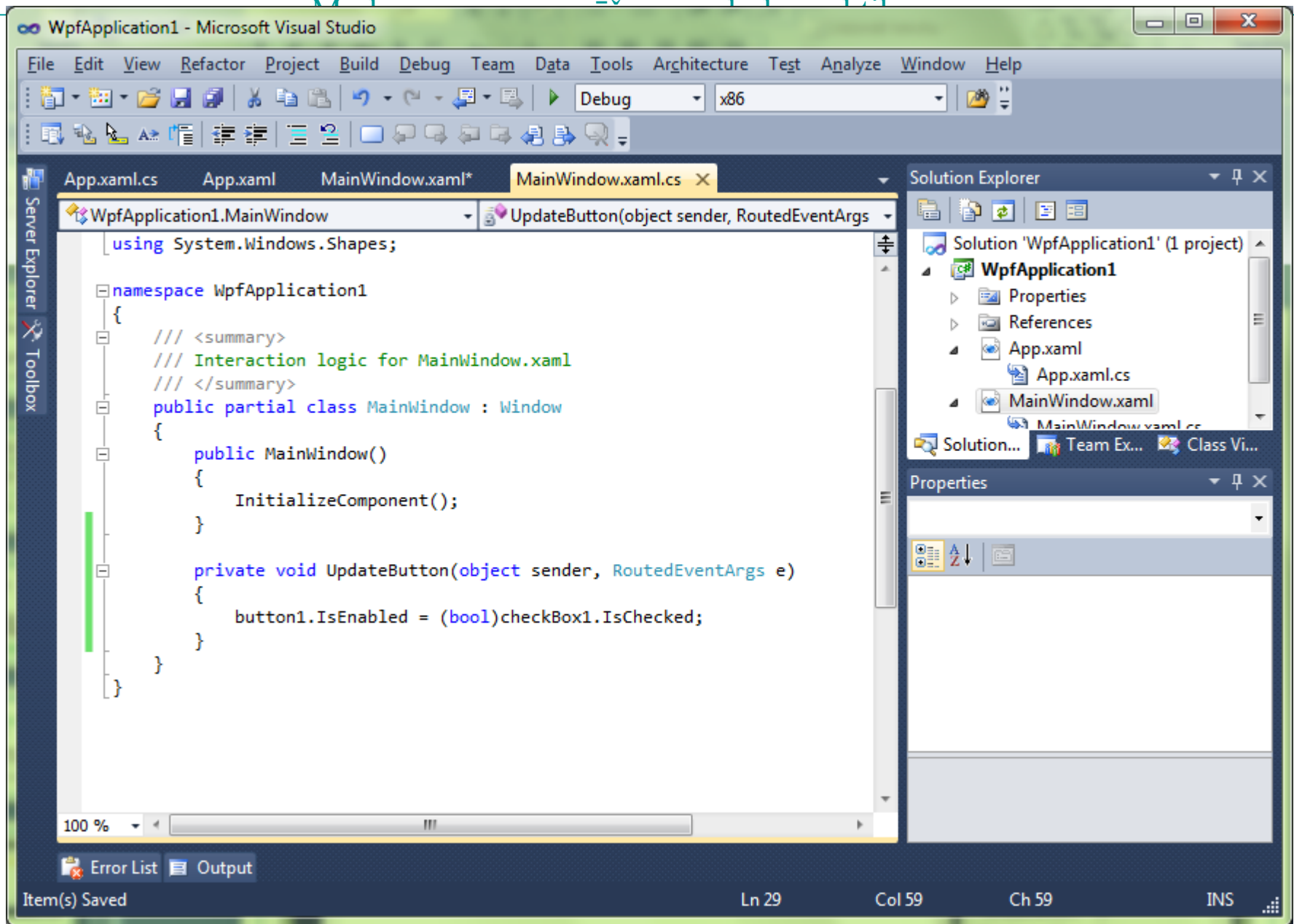
```
<Window x:Class="WpfApplication1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="222" Width="327">
    <Grid>
    </Grid>
</Window>
```

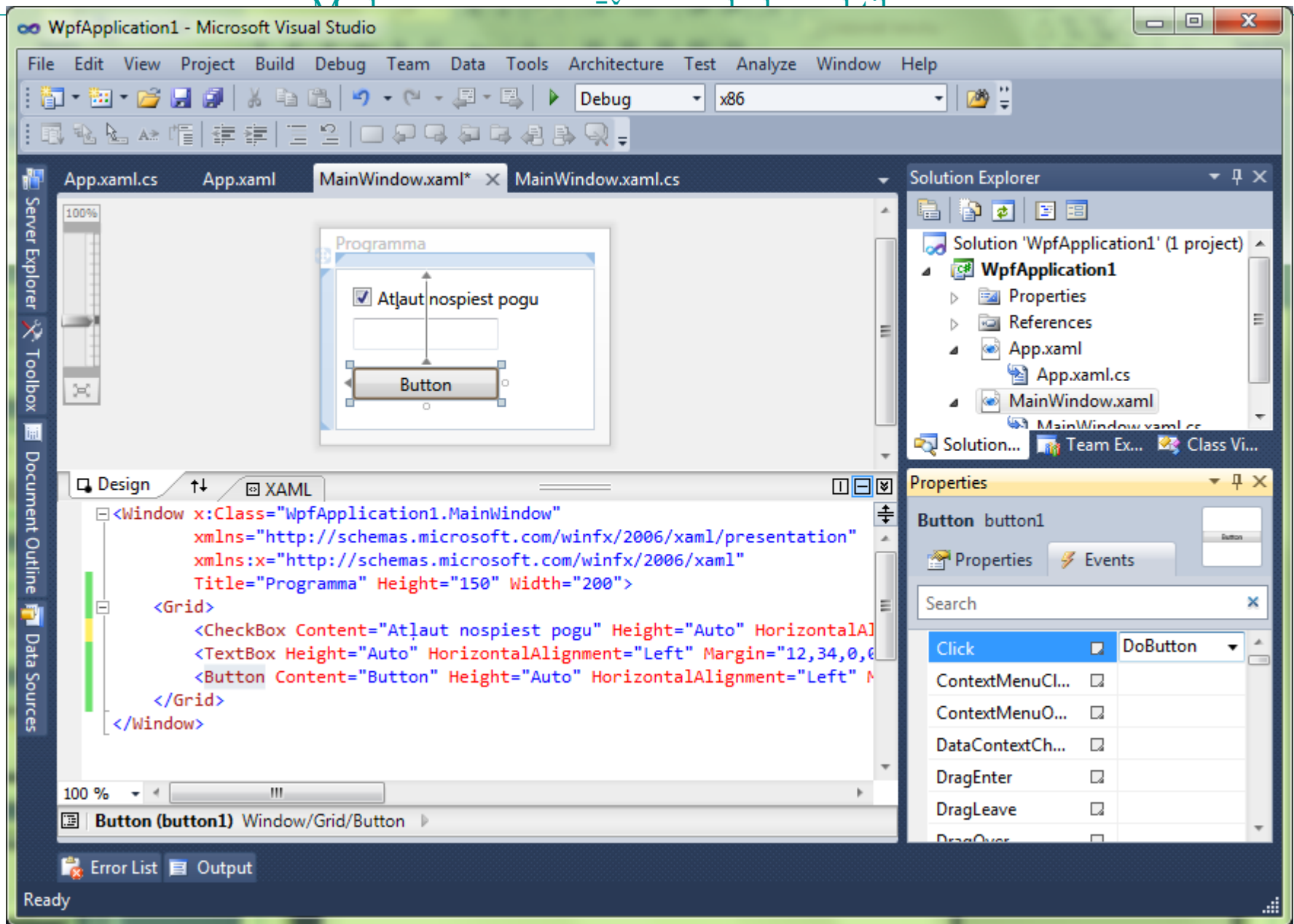
100 %

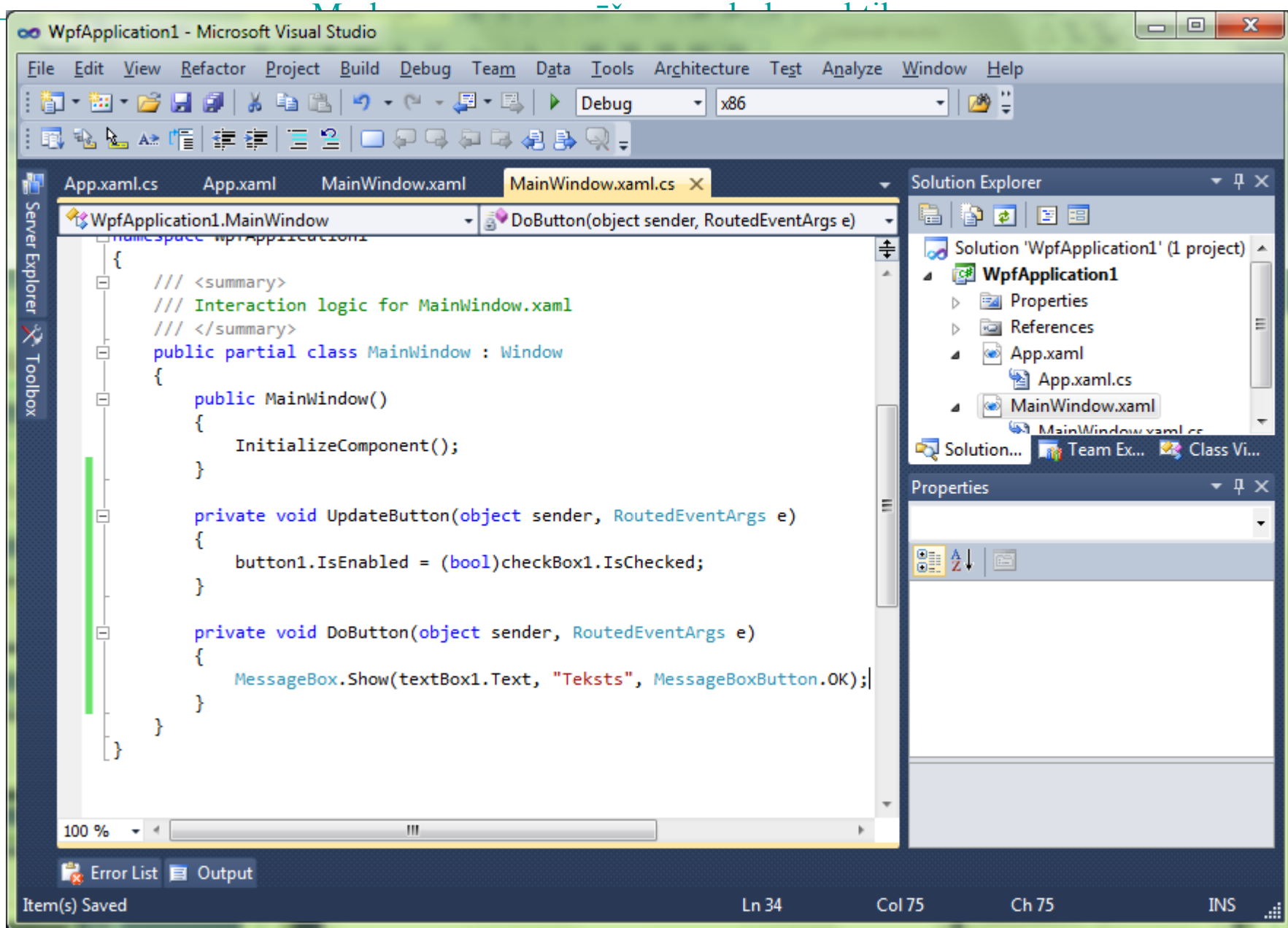
Window Window













```
<Window x:Class="WpfApplication1.MainWindow"
  xmlns=http://schemas.microsoft.com/wfx/2006/xaml/presentation
  xmlns:x="http://schemas.microsoft.com/wfx/2006/xaml"
  Title="Programma" Height="150" Width="200">

  <Grid>

    <CheckBox Content="Atļaut nospiest pogu" Height="Auto"
      HorizontalAlignment="Left" Margin="12, 12, 12, 0" Name="checkBox1"
      VerticalAlignment="Top" Width="Auto" Click="UpdateButton"
      IsChecked="True" />

    <TextBox Height="Auto" HorizontalAlignment="Left" Margin="12, 34, 0, 0"
      Name="textBox1" VerticalAlignment="Top" Width="100"
      Text="(teksts)" />

    <Button Content="Button" Height="Auto" HorizontalAlignment="Left"
      Margin="12, 68, 0, 12" Name="button1" VerticalAlignment="Top"
      Width="100" Click="DoButton" />

  </Grid>

</Window>
```

