Grafikus alkalmazások
készítése Vala programozási nyelven

Készítette:

Meskó Balázs

2011

Tartalomjegyzék

1. Bevezetés

A Vala egy új programozási nyelv, amely segítségével modern programozási technikákat alkalmazhatunk, olyan programok készítésére, melyek a GNOME környezet függvénykönyvtáraira épülnek, nevezetesen a GLib-re és a GObject-re.

A platform már régóta egy teljes programozási környezetet biztosított, olyan funkciókkal mint a dinamikus típusrendszer, és a segített memória-kezelés. A Vala nyelv előtt, ha programot szerettünk volna írni a lehetőségeink az alábbiak voltak:

- natív C API használata
  - gyakran feleslegesen alacsony szintű részletekkel kell törődni
- magas szintű nyelv, virtuális géppel
  - ilyen a Python és a Mono C#
- illesztőfelület használata C++-hoz

A Vala nyelv célja ezen lehetőségek hiányosságainak leküzdése volt.

1.1. Miért éppen Vala?

A Vala nyelv rendkívül különböző a fent említett módszerektől, mivel a Vala fordítóprogram C forráskódra fordít, és az egyetlen függőségei a GNOME függvénykönyvtárak. Ezekből következnek az alábbi tulajdonságai:

- Általánosságban a Vala programok hasonló sebességgel futnak, mint a C nyelvű programok. A kód fejlesztése és karbantartása viszont nagyságrendekkel egyszerűbb.
- A Vala programok pontosan ugyanazt képesek elvégezni, mint a C programok – se többet, se kevesebbet. A Vala szerkezetek C szerkezetekké kerülnek átalakításra, viszont ezek gyakran olyanok, melyeket kézzel túl bonyolult vagy időigényes lenne megírni.

Így tehát a Vala programozási nyelv legfőbb tulajdonságai:

- modern: rendkívül sok, más objektum-orientált nyelvekből is ismerős funkció
- kevés függőség: gyakorlatilag csak a GNOME platform (GLib, GObject) szükséges
- konvenció-követés: szorosan illeszkedik a GNOME konvencióihoz
- C nyelvre fordítás: ez kissé megnöveli a fordítási időt – erre van megoldás, valamint kompatibilitást biztosít C függvénykönyvtárak használatához (ez fordítva is igaz, C-ben is használhatunk Vala könyvtárakat)

A jegyzet szempontjából pontosan a GNOME platformba épülés volt a legfontosabb szempont, ugyanis a GNOME eszközkészletei rendkívül jó választások kezdő fejesztőknek, köszönhetően az egyszerűségüknek és viszonylag jó portabilitásuknak.

1.2. Kinek szól ez a jegyzet?

A jegyzet elsődlegesen olyan embereknek szól, akik tisztában vannak a programozás, és különös tekintettel az objektum-orientált programozás alapjaival. A jegyzet nem megy bele mélyen a főbb programozási gyakorlatokba, inkább arról szól, hogy a Vala nyelvben ezek hogyan kerülnek megvalósításra.

A grafikus programok készítéséről szóló rész pedig a GTK+ eszközkészlet használatát mutatja be, Vala nyelvben és a Glade grafikus felület tervező program segítségével. Ez olyan programozóknak szól inkább, akik még nem nagyon foglalkoztak grafikus felületek készítésével. Sajnos egyelőre ez fedi az egyetemen vagy főiskolán frissen végzett informatikusok nagy részét.

1.3. A jegyzet felépítése

A jegyzet több nagy fejezetből áll. Legelőször a 2. fejezetben bemutatunk egy lehetséges (ajánlott) egyszerű fejlesztőkörnyezetet, valamint ezek kialakítását, a szükséges programok, csomagok telepítését.

A következő három fejezetben a Vala nyelv kerül bemutatásra. A bemutatás stílusa inkább lista-szerű, nem időzünk sokat el egyetlen résznél sem. A 3. fejezetben a nyelv alapjait mutatjuk be, az adattípusokat, -szerkezetek és az operátorokat. A 4. fejezetben az objektum-orientált programozással kapcsolatos lehetőségekkel foglalkozunk. Végül az 5. fejezet pedig a nyelv fejlettebb funkciót ecseteli.

A 6. fejezetben röviden bemutatjuk a GNOME platformot. Ez azért hasznos, mert kicsit átfogóbb képet kapunk a később használt függvénykönyvtárakról. Ezután a 7. fejezetben megírjuk első GTK+-t használó Vala programunkat. Majd bemutatjuk a rendkívül hasznos Glade felülettervező alkalmazást és használatát a 8. fejezetben.

2. Fejlesztőkörnyezet kialakítása

2.1. Csomagok feltelepítése

Először is telepítsük fel a Vala fordítóprogramját, amely az Ubuntu 11.04 verziójában a **valac-0.12** nevű csomag, valamint a nyelv dokumentációját is, amely a **vala-0.12-doc** névre hallgat. Ezután akár már el is kezdhetnénk dolgozni, de célszerű megkönnyíteni a dolgunkat pár segédcsomag használatával.

Telepítsük a **make** csomagot, ugyanis a *Makefile*-ok használata jelentősen megkönnyíti a projektek kezelését, amint elkezdenek szaporodni a fájlok, valamint a további eszközökhöz szükség is van rá.

A *gedit* szövegszerkesztőhöz létezik két hasznos beépülő modul a Vala nyelvhez, amelyeket telepítve egy egészen jól használható, egyszerű IDE-t kapunk. Telepítsük a **gedit-valencia-plugin** és a **gedit-valatoys-plugin** csomagokat.

Ezután ezeket még aktiválni kell a *gedit*-ben a SZERKESZTÉS→BEÁLLÍTÁSOK menüpont BŐVÍTMÉNYEK lapfülén. Ha ezután bekapcsoljuk az ALSÓ ABLAKTÁBLA és OLDALSÓ ABLAKTÁBLA opciókat a NÉZET menüben, akkor végeztünk is a beállításokkal.

2.2. Próbáljuk ki!

Nincs más hátra mint, hogy kipróbáljunk a fejlesztőkörnyezetünket. Ehhez próbáljuk ki a *Teszt* projektet, amely a dokumentáció mellett található a **Teszt.zip** állományban.

A projekt nyilván rendkívül egyszerű. Adott két fájl.

Main.vala:

```
class Main : GLib.Object {
    public static int main(string[] args) {
        stdout.printf("Helló világ!");
        return 0;
   }

}
```

Makefile:

```
PROGRAM = build/Main

build:
        mkdir build;
        valac Main.vala;
        mv Main build/Main

clean:
        rm -rf build
```

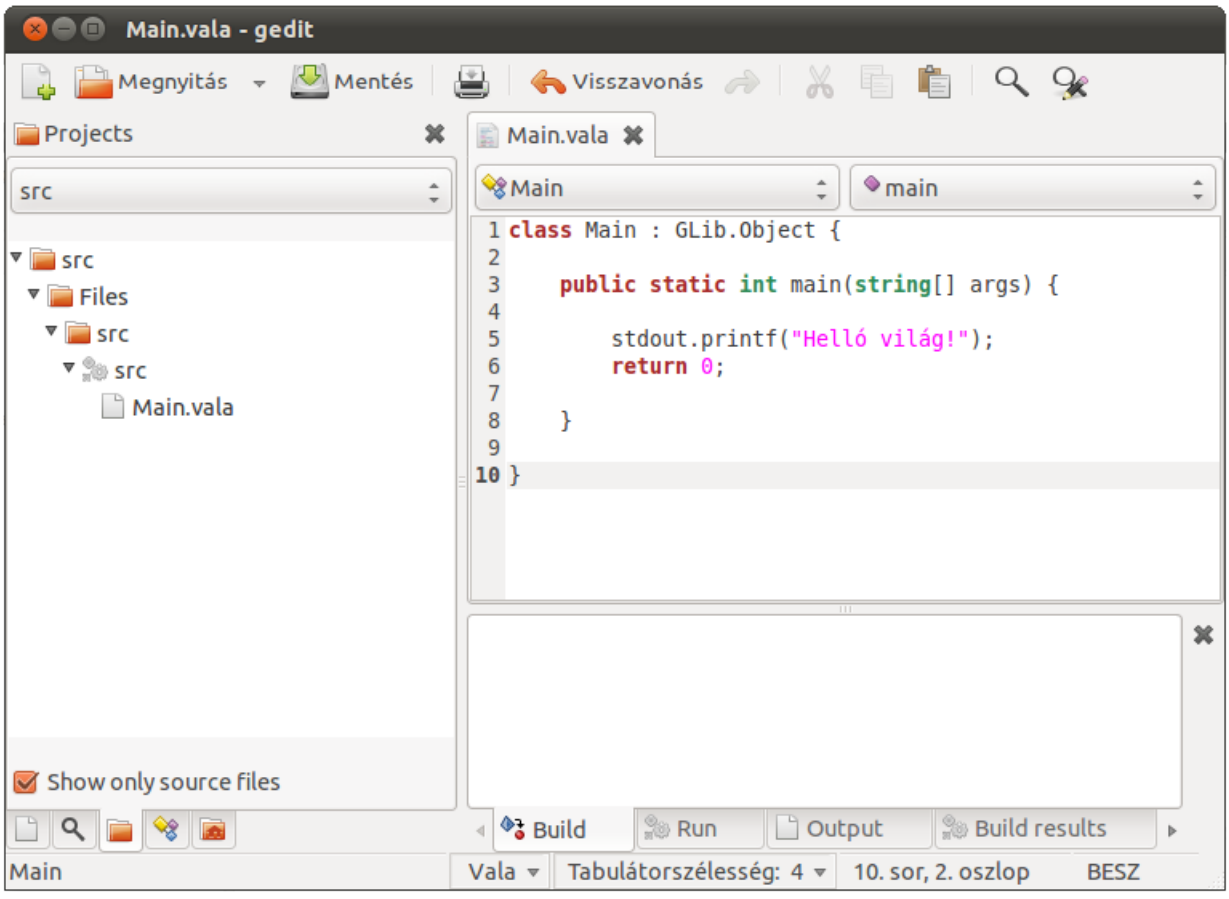
Gyorsan fussunk végig a programon! Létrehozzuk a `Main` osztályt, amely a `GLib.Object` osztály leszármazottja. A Vala nyelv érdekessége, hogy a nyelvi lehetőségek egy részét csak akkor használhatjuk, ha az osztályunk a `Glib.Object` leszármazottja.

Ezután következik a *main* metódusunk. Ha osztályon belül írunk main metódust írunk, akkor a `static` kulcsszó használata kötelező. A hozzáférés módosító viszont lehet `private` és `public` is, nem számít. A visszatérési érték típusa lehet `int` és `void` is, azonban void esetén implicit módon `int`-re cserélődik, és mindig 0-t ad vissza a metódus. A metódus string tömb paramétere opcionális, a program argumentumait tartalmazza.

A következő sor a klasszikus „Helló világ!” üzenet. Jelen esetben az `stdout` objektum a GLib névtér eleme, és ezen objektum *printf()* metódusát használjuk kiíratásra a konzolra. Mivel a metódus visszatérési értéke `int` típusú, ezért 0-val térünk vissza, jelezve a szabályos végrehajtást.

Ezután nézzük meg a Makefile-unkat is. Az első sorban tároljuk a program nevét, ez majd a Valencia pluginnak fontos, ugyanis ez alapján indítja el a bináris programot. A **build** konfiguráció tartalmazza a fordítást. Létrehozzuk a build mappát, a valac-cal lefordítjuk a programot, majd az így kapot binárist áthelyezzük a build mappába. A **clean** konfiguráció nemes egyszerűséggel végérvényesen törli a build mappát.

Bontsuk ki a **Teszt.zip**-et egy mappába, és nyissuk meg a Main.vala fájlt. Ekkor a következő képernyő fogad minket.



A *Makefile*-ok kezelése automatikus, a PROJEKT menü alatti opciók automatikus hívják meg a Makefile-ban szereplő konfigurációkat. A fordítás a **build** célpontot, a tisztítás a **clean** célpontot, a futtatás pedig a PROGRAM változóban megadott fájlt indítja el. Próbáljuk is ki ezeket! Fordítsuk le a programot a PROJEKT→FORDÍTÁS opciót választva, majd PROJEKT→FUTTATÁS.

Ekkor az alsó ablaktábla Futtatás fülén láthatjuk a „Helló világ!” üzenetet – automatikusan odavált a program.

3. Ismerkedés a nyelvvel

A Vala nyelv szintaxisa erősen hasonlít a C# programozási nyelvre, így azok akik használtak már a C nyelv családjába tartozó – pl. C++, Java, C# – nyelvet, azok hamar meg tudják tanulni.

3.1. Adattípusok

A Vala nyelvben kétfajta adattípus létezik – érték típus, és referencia típus. Ebben a fejezetben ezekről lesz szó, valamint a tömbökről szóló rész is saját fejezetet kapott.

3.1.1. Érték típusok

Az érték típusok olyan típusokat jelölnek, amelyeket a procedurális nyelvekből jól ismerhetünk.

Típusnév	Kulcsszó	Megjegyzés
Bájt	char, uchar	neve történelmi okokból ez
Karakter	unichar	32 bites Unicode karakter
Egész	int, uint	
	long, ulong	
	short, ushort	
	[u]int8/16/32/64	garantált méretű egész
Lebegőpontos	float, double	
Logikai	bool	*true* vagy *false*
Összetétel	struct	
Felsorolás	enum	egészekkel reprezentált
Karakterlánc	string	UTF-8 kódolású és állandó

Sajátosságok:

- A legtöbb típusnak különböző méreteik lehetnek, különböző platformokon – kivéve a garantált méretű egészeket. A méret ellenőrzésére a `sizeof` operátor használható:

```
ulong nbytes = sizeof(int32);  // nbytes értéke 4 lesz (= 32 bit)
```

- Az adattípusok méretének meghatározására használható a `.MIN` és `.MAX` adattag. Például: `int.MIN`, `long.MAX`
- A Vala egyik érdekessége az ún. „verbatim string”. Ez egy olyan speciális karakterlánc, amelyben a speciális, és vezérlőkarakterek nem értékelődnek ki. Három idézőjellel hozható létre. Az ilyen karakterláncokba úgy írhatunk sortörést, hogy a `\n` vezérlőkarakter helyett egyszerűen ENTER-t ütünk. Például:

```
string verbatim = """Ez egy úgynevezett "verbatim string".
Nem dolgozza fel a speciális karaktereket, mint a \n, \t, \\ stb.
Ezen kívül szerepelhet benne idézőjel, és többsoros is lehet.""";
```

- A karakterláncok használhatók sablonként (template) is. Erre szolgál a „@” prefix:

```
int a = 6, b = 7;
string s = @"$a * $b = $(a * b)";  // => "6 * 7 = 42"
```

- Két karakterlánc összehasonlítására használhatjuk a == és != operátorokat, mivel a Java nyelvvel ellentétben itt nem referenciák összehasonlítása történik.

- A karakterláncoknak egyszerűen vehetjük a részeit a `[kezdet:vég]` operátor segítségével. Fontos megjegyezni hogy a karakterek számozása 0-ról indul. Ha negatív értékeket adunk meg, az annyit tesz, hogy a karakterlánc végéről számolunk. **Viszont ne felejtsük el, hogy UTF-8 kódolással dolgozunk, ezért nem minden karakter egyforma széles.** A következő példában is az „ó” és „á” karakterek szélessége 2.

```
string greeting = "Helló világ!";
string s1 = greeting[7:13];        // => "világ"
string s2 = greeting[-11:-8];      // => "ló"
```

- Karakterláncoknál még érdemes megemlíteni az `in` operátort, amely azt vizsgálja, hogy egy karakterlánc szerepel-e egy másikban. Például:

```
if ("krumpli" in "paprikás krumpli")
```

- A legtöbb elemi adattípus rendelkezik egyszerű feldolgozó utasításokkal, pár példa:

```
bool b = bool.parse("false");           // => false
int i = int.parse("-52");               // => -52
double d = double.parse("6.67428E-11"); // => 6.67428E-11
string s1 = true.to_string();           // => "true"
string s2 = 21.to_string();             // => "21"
```

3.1.2. Tömbök

Tömböket úgy adhatunk meg, hogy megadunk egy típust, melyet a `[]` operátor követ. Definiálni pedig a `new` operátorral tudjuk. Például:

```
int[] a = new int[10];
```

A tömb hosszát a tömb `length` adatttagjából kaphatjuk meg:

```
int count = a.length;
```

Fontos még megjegyezni, hogy ha referencia típussal hozunk létre tömböt, akkor nem jönnek létre objektumok, csak maga a tömb.

A tömbök a karakterláncokhoz hasonlóan szeletelhetőek. A szeletelés során új tömb keletkezik, a változtatások amelyet a szeletelt tömbön végzünk nem hat ki az eredetire.

```
int[] b = { 2, 4, 6, 8 };
int[] c = b[1:3];      // => { 4, 6 }
```

Többdimenziós tömböket is létrehozhatunk, ehhez csak annyit kell tennünk, hogy a `[]` operátor helyett `[,]`-t, `[,,]`-t stb. írunk.

```
int[,] d = new int[3,4];
int[,] e = {{2, 4, 6, 8},
            {3, 5, 7, 9},
            {1, 3, 5, 7}};
```

Ebben az esetben a tömb hosszát tartalmazó `length` adattag maga is egy tömb lesz.

```
int r = d.length[0];       // => "3"
int c = d.length[1];       // => "4"
```

Fontos: A szeletelés többdimenziós tömbök esetén nem működik.

A tömböket dinamikusan is bővíthetjük a += operátor segítségével. Ennek az a feltétele, hogy lokális vagy privát hozzáférésű legyen a tömb.

```
int[] e = {};
e += 12;
e += 5;
e += 37;
```

Emellett át is méretezhetjük a tömböt a *resize()* metódussal. Az eredeti tartalom megmarad, feltéve ha elfér az átméretezés után.

```
int[] a = new int[5];
a.resize(12);
```

Egyébként létrehozhatunk fix méretű tömböket is. Ezeket később nem méretezhetjük át.

```
int f[10];      // nem kell 'new ...'
```

Végül fontos azt is megjegyezni, hogy a Vala nem ellenőrzi a tömbhatárok túllépését, így ha nagyobb biztonságot szeretnénk használjunk bonyolultabb adattípusokat, mint például ArrayList-et.

3.1.3. Referencia típusok

A referencia típusok közé tartozik minden olyan típus, mely osztályként van deklarálva, akár a `GLib.Object` leszármazottja, akár nem. Egy példa osztály:

```
class Track : GLib.Object {              // öröklés a 'GLib.Object'-ből
    public double mass;                  // egy publikus adattag
    public double name { get; set; }     // egy publikus tulajdonság
    private bool terminated = false;     // egy privát adattag
    public void terminate() {            // egy publikus metódus
        terminated = true;
    }
}
```

Az osztályokról több információ található az objektum orientált programozással foglalkozó fejezetben.

3.1.4. Egyebek

A típusoknál még célszerű megemlíteni az ún. típus kikövetkeztetés funkciót. Ez annyit tesz, hogy egy lokális változót a típus megadása helyett `var` kulcsszóval definiálhatjuk. Ez csakis akkor használható, ha a deklaráció és a definíció egyetlen sorban szerepel.

```
var p = new Person();      // Person p = new Person();
var s = "hello";           // string s = "hello";
var l = new List<int>();   // List<int> l = new List<int>();
var i = 10;                // int i = 10;
```

Elsőre tényleg nem tűnik túl hasznosnak, de generikus típusoknál azért jól jöhet:

```
MyFoo<string, MyBar<string, int>> foo =
                              new MyFoo<string, MyBar<string, int>>();
```

Itt már rögtön kényelmesebb azt írni, hogy

```
var foo = new MyFoo<string, MyBar<string, int>>();
```

Elnevezési konvenciók:

A Vala programokban a változókat kisbetűvel kezdjük, a konstansokat pedig végig nagybetűvel adjuk meg, és a szóközök helyett _ karaktert használunk.

3.2. Operátorok

Mivel az operátorok nagyrészt megegyeznek a legtöbb C-szerű programozási nyelvben megszokottal, ezért csak egy összesítő táblázatot adunk meg:

Operátor	Jelentés
=	értékadás
+ − ∗ /	négy alapművelet
	+: karakterlánc összefűzés
%	maradékképzés, modulus
+ = − = ∗ = / = %=	aritmetika+értékadás
++ −−	inkrementáció, dekrementáció
| ∧ & ~	bitenkénti vagy, kizáró vagy, és, tagadás
| = ∧ = & = ~ =	bitenkénti művelet + értékadás
≪ ≫	bitenkénti eltolás
≪= ≫=	bitenkénti eltolás megadott paraméterrel
==	egyenlőség-vizsgálat
< > >= <= ! =	relációs vizsgálatok
! && ||	logikai műveletek: nem, és, vagy
? :	ternáris operátor: feltétel ? igaz : hamis
??	„null érték eldöntő” operátor
in	tartalmazás operátor

A legtöbb szimbólum ismerős lehet bárki számára, mivel szinte minden programozási nyelvben megtalálhatóak. Az utolsó két operátor viszont nem túl elterjedt, így ezért erre külön kitérünk.

Gyakorlatilag az `a ?? b` művelet ekvivalens a `a != null ? a : b` utasítással. Ez az operátor akkor lehet hasznos, ha egy alapértelmezett értéket szeretnénk megadni ha egy referencia értéke *null*:

```
stdout.printf("Helló %s!\n", name ?? "Ismeretlen");
```

Az `in` operátor használatát már említettük a karakterláncoknál, azonban nem csak ott alkalmazható. Az in operátor működik tömbökön, gyűjteményeken és minden olyan típuson, mely tartalmaz *contains()* metódust.

3.3. Vezérlési szerkezetek

Mivel ezek a szerkezetek rendkívül egyszerűek, és hasonlóak a más programozási nyelvekben megszokottakhoz, ezért csak felsorolás szintjén kerülnek említésre.

3.3.1. Elöltesztelő ciklusok

```
    while (a > b) {
        a--;
    }

for (int i = 0; i < 5; i++) {
    stdout.printf("i=%d\n",i);
}

foreach (int a in array) {
    stdout.printf("i=%d\n",a);
}
```

Erről a ciklusról még lesz szó, de elöljáróban annyit, hogy ez egy `for` ciklus változat, amely esetenként érthetőbb szintaxist eredményez.

3.3.2. Hátultesztelő ciklus

```
do {
    a--;
} while (a < b);
```

Mind a 4 négy ciklusban használható a `break` és `continue` parancs. Előbbi azonnal kilép a ciklusból, míg utóbbi a következő iterációra ugrik.

3.3.3. Elágazások:

```
if (a > 0) {
    stdout.printf("a nagyobb, mint 0\n");
} else if (a < 0) {
    stdout.printf("a kisebb, mint 0\n");
} else {
    stdout.printf("a pontosan 0\n");
}

switch (a) {
case 1:
    stdout.printf("egy\n");
    break;
case 2:
case 3:
    stdout.printf("kettő vagy három\n");
    break;
default:
    stdout.printf("valami más\n");
    break;
}
```

3.4. Metódusok

A Vala-ban minden függvényt metódusnak hívunk, függetlenül attól, hogy egy osztályban szerepel vagy sem.

```
int method_name(int arg1, Object arg2) {
    return 1;
}
```

A metódusok elnevezési konvenciója az, hogy minden metódusnevet csupa kis betűvel írunk, és a szavakat _ karakterrel választjuk el. Ez például Java programozók számára furcsa lehet, de a GObject függvénykönyvtár ezt használja.

Metódus túlterhelés:
A Vala nyelvben nincs lehetőség a metódusok túlterhelésére. Ennek oka az, hogy a Vala-ban írt függvénykönyvtárakat úgy készítik, hogy C programozók is használhassák.

3.5. Megbízottak

A megbízott (delegate) egy normál metódust reprezentál. Arra használható, hogy kódrészleteket úgy kezeljünk, mint egy objektumot. Bármely olyan metódus, amelynek a paraméterszignatúrája megegyezik a megbízottéval, értékül adható egy ilyen változónak, vagy metódus argumentumként használható. Az alábbi példán keresztül jobban megérthető:

```
delegate void DelegateType(int a);

void f1(int a) {
    stdout.printf("%d\n", a);
}

void f2(DelegateType d, int a) {
    d(a);       // "Megbízott" meghívása
}

void main() {
    f2(f1, 5);  // Metódus, mint megbízott argumentum
}
```

3.6. Névtelen metódusok

A Vala nyelvben névtelen metódusokat – más néven lambda kifejezéseket – a => operátorral adhatunk meg. A paraméterlista az operátor bal oldalán szerepel, a metódustörzs pedig a jobb oldalán. Egy példa:

```
(a) => { stdout.printf("%d\n", a); }
```

Persze önmagában egy ilyen kifejezésnek nem sok értelme van. Akkor lehet hasznos, ha egy megbízott típusú változóhoz rendeljük, vagy paraméterként adjuk át egy másik metódusnak.

Névtelen metódus megbízott változóhoz rendelése:

```
delegate void PrintIntFunc(int a);

void main() {
    PrintIntFunc p1 = (a) => { stdout.printf("%d\n", a); };
    p1(10);
}
```

Használata metódus argumentumként:

```
delegate int Comparator(int a, int b);

void my_sorting_algorithm(int[] data, Comparator compare) {
    // ... 'compare' metódust valahol meghívjuk ...
}

void main() {
    int[] data = { 3, 9, 2, 7, 5 };
    // Névtelem metódus mint második paraméter:
    my_sorting_algorithm(data, (a, b) => {
        if (a < b) return -1;
        if (a > b) return 1;
        return 0;
    });
}
```

Egyébként a névtelen metódusokban is használhatjuk a hatókör lokális változóit.

3.7. Névterek

```
namespace NameSpaceName {
    // ...
}
```

A kapcsos zárójelek között minden a névtérbe kerül, így ezekre kívülről így is kell hivatkozni. A külső eléréseknél teljes nevekre kell hivatkozni, vagy használhatjuk a

```
using NameSpaceName;
```

parancsot. Így a névtér elemeire a névtér megadása nélkül hivatkozhatunk. Például, ha beimportáljuk a Gtk névteret a `using Gtk;` paranccsal, akkor a `Gtk.Window` osztályra egyszerűen `Window`-ként hivatkozhatunk. Azonban a `Gtk.Object` osztályra nem hivatkozhatunk ily módon, mivel a neve ütközik a `GLib.Object` osztállyal – mivel a Vala programok implicit módon tartalmazzák a `using GLib;` utasítást.

Ha nem definiálunk saját névteret, akkor egy globális, névtelen névtérbe kerülnek az elemek. Ilyenkor ha névütközés lép fel, akkor a `global::` kulcsszóval oldhatjuk fel.

A névterek egymásba is ágyazhatóak. Ez megtehető a `namespace` deklarációk közvetlen egymásba ágyazazásával, vagy implicit módon: `namespace NameSpace1.NameSpace2` alakban. Implicit névtérmegadás történhet osztálydefinícióban is: `class NameSpace.ClassName`.

3.8. Struktúrák

A struktúrák összetett érték típusok. A Vala struktúrák – korlátozott módon – tartalmazhatnak metódusokat, valamint az adattagok hozzáférési osztályának megadása kötelező – `private` vagy `public`.

```
struct Color {
    public double red;
    public double green;
    public double blue;
}
```

A struktúra típusú változók megadása némileg eltér a többi változóétól:

```
Color c1 = Color();
Color c2 = { 0.5, 0.5, 1.0 };
Color c3 = Color() {
    red = 0.5,
    green = 0.5,
    blue = 1.0
};
```

A struktúrák a veremben tárolódnak, és értékadáskor másolódnak.

3.9. Osztályok

Az osztályok referencia típusok. A struktúrákkal ellentétben a verem helyett a kupacon tárolódnak. Osztályokat az alábbi módon definiálhatunk:

```
class ClassName : SuperClassName, InterfaceName {
    // osztálytörzs
}
```

Az osztályok leszármazhatnak más osztályokból, és interfészeket implementálhatnak. Csak egyetlen ősosztály megadása lehetséges, viszont tetszőleges számú interfészt implementálhatunk. Az osztályokról bővebben az objektum-orientált programozással foglalkozó, 4. fejezetben lesz szó.

3.10. Interfészek

Az interfészek nem példányosítható típusok. Ahhoz hogy használni tudjuk őket, szükség van egy osztályra, mely implementálja az interfészt. A Vala interfészek erősebbek, mint a Java, illetve C# interfészek, ugyanis használhatóak ún. mixinként. Ez azt jelenti, hogy az interfészekben lehetnek implementált metódusok is. Interfészeket az alábbi módon definiálhatunk:

```
interface InterfaceName : SuperInterfaceName {
    // interfésztörzs
}
```

3.11. Kód attribútumok

A kód attribútumok a Vala fordítóprogramnak szóló információk arról, hogy hogyan kell értelmezni az adott forráskód-részleteket. Gyakorlatilag ezek az utasítások megváltoztatják a Vala forráskódból generált C nyelvű programot.

Az attribútomok megadása az `[AttributeName]` vagy az `[AttributeName(param1 = value1, param2 = param2, ...)]` kóddal történik. Ez gyakorlatilag megfelel a C# attribútumainak, és a Java annotációinak. Egy pár példa attribútumot is megadunk:

```
[SimpleType]
```

Ez az attribútum arra szolgál, hogy egy struktúra érték szerint adódjon át.

```
[Import]
```

Az attribútum azt jelzi, hogy az adott metódus implementációja más fájlban található.

```
[CCode]
```

Ez az attribútum a C forráskód generálást szabályozza.

4. Objektum-orientált programozás

Most, hogy már tisztáztuk a Vala nyelv szintaxisát és a nyelvi elemeket, rátérünk a nyelv objektum-orientált lehetőségeire. Mint szinte minden mai nyelvben, ahhoz hogy saját objektumokkal dolgozzunk, ahhoz osztálydefiníciót készítünk.

4.1. Osztálydefiníció

Az osztálydefiníció megadja, hogy az ilyen típusú objektumok milyen típusú adatokat tartalmaznak, milyen referenciákat tartalmazhatnak, és milyen metódusai vannak. A definíció során megadhatunk egy ősosztályt is, melynek alosztálya lesz az új osztály. Emellett egy osztály tetszőleges számú interfészt is implementálhat. A Vala osztályoknak is lehetnek statikus adattagjaik, metódusaik. Ezeket a `static` kulcsszóval jelöljük. A statikus elemek az osztály egészére érvényesek, nem kötődnek objektumhoz, és az osztály példányosítsa nélkül is elérhetőek. Példa egy osztálydefinícióra:

```
public class TestClass : GLib.Object {

    /* Adattagok */
    public int first_data = 0;
    private int second_data;

    /* Konstruktor */
    public TestClass() {
        this.second_data = 5;
    }

    /* Metódus */
    public int method_1() {
        stdout.printf("private data: %d", this.second_data);
        return this.second_data;
    }
}
```

Itt még fontos szót ejteni a Vala hozzáférési minősítőiről. A Vala nyelv négy ilyen minősítővel rendelkezik, melyek az alábbiak:

`public`	Nincsenek megkötések
`private`	Csak az osztályban illetve struktúrában elérhető
`protected`	Csak az osztályban, és a leszármazottaiban elérhető
`internal`	Csak a csomagon belüli osztályokban érhető el

Ha nem adunk meg explicit módon minősítést, akkor az alapértelmezés a `private` minősítés.

4.2. Konstruktorok

A Vala nyelv két kissé eltérő stílusú konstruktorokat használ, az első a C#/Java stílusú konstruktor, a másik pedig a GObject stílusú. Először az elsőről lesz szó.

Fontos megjegyezni, hogy mivel a Vala nyelv nem támogatja a metódus-túlterhelést, így a konstruktort sem lehet túlterhelni. Éppen ezért némileg szokatlan módon a Vala nyelvben *nevesített konstruktorok* vannak.

```
public class Button : Object {

    public Button() {
    }

    public Button.with_label(string label) {
    }

    public Button.from_stock(string stock_id) {
    }
}
```

A példányosítás analóg módon:

```
new Button();
new Button.with_label("Click me");
new Button.from_stock(Gtk.STOCK_OK);
```

A konstruktorok láncolhatóak is a `this()` és a `this.name_extension()` hívásokkal:

```
public class Point : Object {
    public double x;
    public double y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public Point.rectangular(double x, double y) {
        this(x, y);
    }

    public Point.polar(double radius, double angle) {
        this.rectangular(radius*Math.cos(angle), radius*Math.sin(angle));
    }
}

void main() {
    var p1 = new Point.rectangular(5.7, 1.2);
    var p2 = new Point.polar(5.7, 1.2);
}
```

4.3. Destruktor

Bár a Vala elvégzi helyettünk a memória-kezelést, szükség lehet destruktor írására. Például ha kézi memória-kezelést szeretnénk megvalósítani mutatókkal (erről később), vagy ha más erőforrásokat kell felszabadítanunk. A szintaktika megegyezik a C# és C++ nyelvekkel.

```
class Demo : Object {
    ~Demo() {
        stdout.printf("destruktor");
    }
}
```

4.4. Szignálok

A szignálok a Vala nyelv azon funkciói közé tartozik, melyekhez szükséges az, hogy a `GLib` függvénykönyvtár `Object` osztályának egy leszármazott alosztályát használjuk.

Egy szignált egy osztály tagjaként definiálhatunk, majd szignálkezelőket ezután a szignál `connect()` metódusával adhatunk hozzá – a szignálkezelők száma tetszőleges. A következő példában definiálunk egy szignált, majd egy szignálkezelő lambda kifejezést adunk hozzá:

```
public class Test : GLib.Object {

    public signal void sig_1(int a);

    public static int main(string[] args) {
        Test t1 = new Test();

        t1.sig_1.connect((t, a) => {
            stdout.printf("%d\n", a);
        });

        t1.sig_1(5);

        return 0;
    }
}
```

Most tekintsük meg bővebben, hogy mit csinál pontosan ez a forráskód. Létrehozzuk a `Test` osztályt a `GLib.Object` leszármazottjaként. Az osztály első tagja a `sig_1` szignál, amelynek egy egész típusú paramétere van.

A osztály `main` metódusában példányosítjuk az osztályt a `t1` objektumba. Ezután az objektumban lévő szignálhoz hozzáadunk egy kezelő metódust. Ez a metódus egy lambda kifejezés, amelynek két bemenő paramétere van. Az első paraméter a `t`, ez tartalmazza azt az objektumot, amely kibocsátja a szignált. A második paraméter pedig a szignál definícióban megadott egész típusú változó.

Ezután közvetlenül meghívjuk a szignált, mint ha egy metódus lenne. Ekkor automatikusan lefut a szignálkezelő metódus.

Fontos megjegyezni, hogy a szignálok hozzáférése csak `public` módosítójú lehet. Emellett a `[Signal]` attribútummal megváltoztathatjuk a működését.

4.5. Tulajdonságok

Az objektum-orientált programozás egyik alapelve az *adatrejtés*. Ennek egy megoldása az, hogy az adattagokat mindig `private` elérésűvé tesszük, majd ehhez kezelő metódusokat írunk (getter és setter). Ez Java stílusban így nézne ki:

```
class Person : Object {
    private int age = 32;

    public int get_age() {
        return this.age;
    }

    public void set_age(int age) {
        this.age = age;
    }
}
```

Ez teljesen szabályos, viszont sokszor túl bőbeszédű. Erre szolgálnak a Vala tulajdonságai.

```
class Person : Object {
    private int _age = 32;  // aláhúzás prefix a névütközés ellen

    /* Tulajdonság */
    public int age {
        get { return _age; }
        set { _age = value; }
    }
}
```

Ez egy tömörebb megfogalmazás, azonban ha a `get()` és `set()` metódusok tényleg csak az értékadást, illetve értékkinyerést szolgálják, akkor még tömörebben leírhatjuk.

```
class Person : Object {
    /* Tulajdonság: getter, setter, alapértelmezett érték */
    public int age { get; set; default = 32; }
}
```

Lehetőség van csak olvasható tulajdonság megadására is. Ehhez vagy a

```
public int age { get; private set; default = 32; }
```

kódot használjuk, vagy az első megadásban kihagyjuk a `set` sort.

A tulajdonságok neve mellett megadhatunk nekik egy rövid *(nick)* és hosszabb leírást *(blurb)* is. Erre a `[Description]` attribútum szolgál.

```
[Description(nick = "életkor", blurb = "A személy életkora")]
public int age { get; set; default = 32; }
```

Ezek további információkat szolgáltathatnak más segédprogramoknak, mint például az általunk használt Glade grafikus felület készítő programnak.

Minden olyan osztály, mely a `GLib.Object` leszármazottja, rendelkezik egy `notify` szignállal. A szignál mindig kiadásra kerül, ha egy tulajdonság megváltozik.

```
obj.notify.connect((s, p) => {
    stdout.printf("A '%s' tulajdonság megváltozott!\n", p.name);
});
```

Ily módon mindig kiírjuk a konzolra, ha megváltozott a tulajdonság. Az `s` paraméter a szignál forrása, a `p` paraméter pedig egy `ParamSpec` típusú struktúra. Ha csak egy tulajdonság értékváltozásait szeretnénk figyelni, akkor azt alábbi módon tehetjük meg:

```
obj.notify["age"].connect((s, p) => {
    stdout.printf("age - megváltozott\n");
});
```

Ebben az esetben meg kell jegyezni, hogy a változókban szereplő aláhúzás karaktereket kötőjelekre kell cserélni a GObject kódolási konvenciói szerint.

További lehetőségünk az a változások figyelésénél, hogy letiltjuk a szignál kibocsátását. Erre szolgál a `[CCode(notify = false)]` attribútum:

```
public class MyObject : Object {
    [CCode(notify = false)]
    // a szignál NEM bocsátódik ki
    public int without_notification { get; set; }
    // a szignál kibocsátásra kerül a változtatás során
    public int with_notification { get; set; }
}
```

4.6. Öröklődés

A Vala nyelvben minden osztály egy vagy nulla osztály alosztálya. De a gyakorlatban inkább minden osztálynak van ősosztálya, mert a Java nyelvvel ellentétben nincs implicit öröklődés.

Ha egy osztály öröklődik egy másik osztályból, akkor minden egyes művelet mely értelmezett az ősosztály példányain, értelmezett lesz az alosztály példányain is. A következő példa két osztályt tartalmaz, melyek közül az egyik a másik alosztálya:

```
public class Super : Object {
    public virtual int my_method (int x, int y) { /* ... */ }
    public void another_method () { /* ... */ }
}

public class Sub : Super {
    public override int my_method (int x, int y) {
        base.my_method (x, y);
        // ...
    }
}
```

Ilyenkor a `Sub` osztály rendelkezik a `Super` osztály `another_method()` metódusával, valamint felülírja a `my_method` metódust. A Java nyelvvel összehasonlítva az a különbség, hogy a Vala nyelvben implicit módon nem virtuálisak a metódusok – azaz nem írhatóak felül. Így a metódusok csak akkor írhatóak felül, ha explicit módon virtuálisnak jelöljük őket. A felülírás során pedig kötelező az `override` kulcsszó.

A `base` kulcsszó az ősosztályt jelöli, mely az ősosztály tagjainak elérésére használható, valamint az ősosztály konstruktorát is meghívhatjuk vele:

```
class SubClass : SuperClass {
    public SubClass() {
        base(10);
    }
}
```

4.7. Absztakt osztályok

A metódusokra alkalmazható további módosító az `abstract` kulcsszó. Ez azt jelenti, hogy a metódus implementációja nem szerepel az osztályban. Ha egyetlen metódust is absztraktnak jelölünk ki, akkor magát az osztályt így kell jelölnünk. Ez azért van, hogy elkerüljük az osztály véletlen példányosítását.

```
public abstract class Animal : Object {
    public void eat() {
        stdout.printf("*chomp chomp*\n");
    }

    public abstract void say_hello();
}

public class Tiger : Animal {
    public override void say_hello() {
        stdout.printf("*roar*\n");
    }
}

public class Duck : Animal {
    public override void say_hello() {
        stdout.printf("*quack*\n");
    }
}
```

Az absztrakt metódusok felülírásakor is kötelező a `override` kulcsszó használata. Egyébként az `abstract` minősítő a tulajdonságokra is alkalmazható.

4.8. Interfészek / Mixinek

A Vala osztályok tetszőleges számú interfészt implementálhatnak. Minden interfész egy típus, hasonló mint egy osztály, azonban nem példányosítható. Az interfészek implementációjának folyamata megegyezik az absztrakt osztályból örökléssel.

Egy egyszerű interfész-definíció:

```
public interface ITest : GLib.Object {
    public abstract int data_1 { get; set; }
    public abstract void method_1();
}
```

Ez a kód az `ITest` interfészt hozza létre, melynek követelménye a `GLib.Object` osztályból öröklés. A `data_1` egy tulajdonság, melynek `get()` és `set()` metódusainak kell lennie. A `method_1` pedig egy paraméter és visszatérési érték nélküli publikus metódus.

Az interfész legminimálisabb implementációja így néz ki:

```
public class Test1 : GLib.Object, ITest {
    public int data_1 { get; set; }
    public void method_1() {
    }
}
```

Fontos különbség az absztrakt osztályokkal szemben, hogy a metódusokat nem írjuk felül.

A Vala interfészek nem örökölhetnek más interfészekből, de megadhatunk előfeltételeket. Ez azt jelenti, hogy az interfészt csak akkor implementálhatjuk, ha az előfeltételeit is implementáljuk. Lássunk egy példát:

```
public interface List : Collection {
}
```

A `List` interfész előkövetelménye a `Collection` interfész. Így ha implementálni szeretnénk akkor ezt kell tennünk:

```
public class ListClass : GLib.Object, Collection, List {
}
```

A Vala interfészek jelölhetnek osztályokat is előkövetelménynek. A leggyakoribb előkövetelmény a `GLib.Object` osztály.

A másik fontos különbség a Vala és a C#/Java interfészek között az, hogy a Vala nyelvben lehet implementáció az interfész-definícióban. Ezért is kötelező az `abstract` kulcsszó az implementáció nélküli metódusokban, tulajdonságokban. Ha implementációt írunk egy interfészbe, akkor azt *mixinnek* nevezzük.

4.9. Polimorfizmus

A polimorfizmus azt jelenti, hogy egy objektumot úgy tudunk használni, mint ha az több lenne, mint egyetlen objektum. Ennek egy lehetőségére már láttunk példát, nevezetesen, hogy egy alosztályt kezelhetünk úgy, mint annak ősosztályát, vagy annak egy interfészét.

4.10. Metódus elrejtés

A `new` módosító használatával elfedhetjük az öröklött metódust egy ugyanolyan nevű metódussal. Az új metódus paraméter-szignatúrája különbözhet a régitől. Fontos, a metódus elrejtés nem felülírás, mivel a metódus rejtésnek nincs polimorf viselkedése.

```
class Foo : Object {
    public void my_method() { }
}

class Bar : Foo {
    public new void my_method() { }
}
```

A metódus rejtés mellett is elérhetjük az ősosztály metódusát, ehhez csak dinamikus konverzióra van szükség:

```
void main() {
    var bar = new Bar();
    bar.my_method();
    (bar as Foo).my_method();
}
```

Ekkor a leszármazott osztályt dinamikusan az ősosztályba konvertáljuk, így elérhetjük az elfedett metódust.

4.11. Futásidejű típus információk

A Vala osztályok futásidőben regisztrálódnak és minden példány típus információkat hordoz, és ezeket dinamikusan ellenőrizhetjük az `is` kulcsszóval.

```
bool b = object is SomeTypeName;
```

A `sizeof()` operátor segítségével a típus információkat közvetlenül is használhatjuk. Itt például egy új példányt állítunk elő az `Object.new()` metódussal:

```
Type type = typeof(Foo);
Foo foo = (Foo) Object.new(type);
```

A kérdés az, hogy itt most milyen konstruktor fut le. Nos ez a 4.14 fejezetben ismertetett `construct` blokk.

4.12. Dinamikus típuskonverzió

A dinamikus konverzióra az `as` posztfix operátor szolgál. A konverzió érvényességének ellenőrzése futásidőben történik, ha illegális konverzió történik, akkor automatikusan `null` értéket vesz fel. Azonban ez csak akkor működik, ha mindkét objektum egy osztály példánya.

Példa a konverzióra:

```
Button gomb = widget as Button;
```

Ha valamilyen oknál fogva a `widget` objektum nem a `Button` osztály – vagy annak alosztályainak – példánya, vagy nem implementálja a `Button` interfészt, akkor a `gomb` objektum a `null` értéket veszi fel.

4.13. Generikus típusok

A generikus típusok szerepe az, hogy egy osztály egy példányát egy adott típusra, vagy típushalmazra szűkíthetünk. Ezt a megkötést általában arra használjuk, hogy az objektumban tárolt adat a megadott típusú legyen.

A Vala nyelvben a generikus típusok kezelése futásidőben történik. Amikor létrehozunk egy osztályt, amely leszűkíthető, akkor egyetlen osztály jön létre, és az egyes példányokat egyesével szabjuk testre. Ez pont az ellentéte a C++ működésének, ott ugyanis minden egyes lehetséges típushoz új osztály jön létre. E tekintetben a Vala a Java rendszerét követi.

Ennek több következménye van. Először is, a statikus adattagok az osztály egészére vonatkoznak generikus típusok esetén is. Másodszor a generikus osztály leszármazottjai is generikusak, tehát alkalmazhatóak rá az ősosztály megkötései.

A következő forráskód egy minimális generikus osztály:

```
public class Wrapper<G> : GLib.Object {
    private G data;

    public void set_data(G data) {
        this.data = data;
    }

    public G get_data() {
        return this.data;
    }
}
```

Ezt a `Wrapper` osztályt kötelező leszűkíteni, ahhoz hogy példányosítsuk – ebben az esetben a `G` jelöli a leszűkítés típusát, és az egyes példányok egy `G` típusú `data` objektumot tartalmaznak. Itt megjegyezzük, hogy azért szerepelnek Java stílusú getter és setter metódusok, mert a generikus típusoknál egyelőre nem működnek a tulajdonságok.

Az osztály példányosítása során arra van szükség, hogy egy típust válasszunk. Mint például a következő példában a `string` típust:

```
var wrapper = new Wrapper<string>();
wrapper.set_data("test");
var data = wrapper.get_data();
```

4.14. GObject stílusú konstruktor

Mint már arról szó volt, a Vala egy alternatív konstrukciós sémát is támogat. Ez közelebb áll a GObject rendszer működéséhez. A személyes preferencia leginkább attól függ, hogy a programozó milyen rendszerhez szokott.

Ez a GObject stílusú séma új szintaktikai elemeket is bevezet. Ezek a *konstrukciós tulajdonságok*, egy különleges `Object(...)` hívás, és a `construct` blokk. Lássuk hogy is működik:

```
public class Person : Object {

    /* Konstrukciós tulajdonságok */
    public string name { get; construct; }
    public int age { get; construct set; }

    public Person(string name) {
        Object(name: name);
    }

    public Person.with_age(string name, int years) {
        Object(name: name, age: years);
    }

    construct {
        // bármi más feladat elvégzése
        stdout.printf("Üdvözöljük %s\n", this.name);
    }
}
```

A GObject stílusú konstrukciós séma esetén minden konstrukciós metódus csak egyetlen `Object(...)` metódushívást tartalmaz, a konstrukciós tulajdonságok beállítására. A hívás változó hosszúságú argumentumokat tartalmaz, `tulajdonság:érték` formában. Ezeket a konstrukciós tulajdonságokat jelölni kell a `construct` kulcsszóval. Ezen tulajdonságok értéke akkor lesz beállítva, ha minden `construct` blokk meghívódik az osztály-hierarchiában.

A `construct` blokk garantáltan meghívódik minden esetben, amikor az osztálynak létrejön egy példánya, még akkor is, ha egy alosztály példányáról van szó. A blokknak sem paraméterei, sem visszatérési értéke nincs. A blokkon belül más metódusok és adattagok szabadon használhatóak.

Az alábbi lista a tulajdonságok jellemző változatait tartalmazza, beleértve a konstrukciós tulajdonságokat is:

```
public int a { get; private set; }    // Olvasás
public int b { private get; set; }    // Írás
public int c { get; set; }            // Olvasás / Írás
public int d { get; set construct; }  // Olvasás / Írás / Konstrukció
public int e { get; construct; }      // Olvasás / Írás konstrukciónál
```

Néha akkor szeretnénk műveleteket végezni, amikor magát az osztályt hozza létre a GObject folyamat. Ezt a Java nyelv esetén statikus inicializációs blokknak hívjuk. A Vala nyelvben a következő a megadása:

```
static construct {
    ...
}
```

Ez a kódrészlét pontosan akkor fut le, amikor a rendszer regisztrálja az osztályt, mint típust.

5. Fejlett lehetőségek

5.1. Kiértékelések

A kiértékelések segítségével a programozó futásidőben ellenőrizhet feltevéseket. A szintaktika `assert(feltétel)` formájú. Ha egy feltevés tévesnek bizonyul, akkor a program egy megfelelő üzenettel megszakad. Pár példa GLib kiértékelés:

`assert(bool expr)`	Hamis kifejezés esetén hiba
`assert_not_reached()`	Ha lefut mindig hibát ad
`return_if_fail(bool expr)`	Hamis kifejezés esetén visszatérés
`return_if_reached()`	Ha lefut, a metódus visszatér
`warn_if_fail(bool expr)`	Hamis kifejezés esetén figyelmeztetés
`warn_if_reached()`	Ha lefut, figyelmeztetést ad

A programozóban ez előhozhatja azt a késztetést, hogy null érték ellenőrzése használjuk ezeket metódusokban. De erre nincs szükség, a Vala ezt implicit módon elvégzi minden olyan paraméter esetén, amely nincs nullázhatónak jelölve a `?` operátorral.

```
void method_name(Foo foo, Bar bar) {
    /* Nem szükséges:
    return_if_fail(foo != null);
    return_if_fail(bar != null);
    */
}
```

5.2. Elő- és utófeltételek

A Vala nyelv korlátozott módon támogatja az ún. „contract programming” paradigmát. Ez gyakorlatilag arról szól, hogy a komponenseink – osztályok és metódusok – működéséről állításokat fogalmazunk meg, és ezeket futásidőben kiértékeljük. A Vala esetében a paradigma a metódusokban jelenik meg, elő- és utófeltételek formájában.

```
double method_name(int x, double d)
        requires (x > 0 && x < 10)
        requires (d >= 0.0 && d <= 1.0)
        ensures (result >= 0.0 && result <= 10.0)
{
    return d * x;
}
```

A `result` változó egy speciális változó, mely a visszatérési értéket reprezentálja.

5.3. Hibakezelés

A GLib függvénykönyvtár rendelkezik egy alrendszerrel, mely a futásidejű hibákat kezeli, ez a GError. A Vala nyelv ezt a modern programozási nyelvekből ismert formára fordítja, de az implementációs részletek jelentősen különböznek. Ez a fajta hibakezelés a váratlan kivételek kezelésére alkalmas, ha várható hibákat szeretnénk kezelni, nem ez rá a legalkalmasabb eszköz. Erre az előző két fejezetben ismertetett eszközök hatásosabbak.

A Vala kivételeket kezelni kell! Viszont ha elmarad a hibakezelés a fordítóprogram csak figyelmeztetést ad, és nem állítja meg a fordítást. A Vala kivételek használata az alábbi lépésekből áll:

1. Metódus deklaráció, amely dobhat kivételt:

```
void my_method() throws IOError {
    // ...
}
```

2. Kivétel dobása a metódusban:

```
if (hiba) {
    throw new IOError.FILE_NOT_FOUND("A fájl nem található.");
}
```

3. A hiba elkapása a metódushívásnál:

```
try {
    my_method();
} catch (IOError e) {
    stdout.printf("Hiba: %s\n", e.message);
}
```

4. Hibakód ellenőrzése

```
IOChannel channel;
try {
    channel = new IOChannel.file("/tmp/my_lock", "w");
catch (FileError e) {
    if(e is FileError.EXIST) {
        throw e;
    }
    GLib.error("", e.message);
}
```

A Vala nyelvben a kivételeknek három komponensük van. Az első a hibadomén, a példában ez az `IOError` volt. A második a hibakód, mely a doménben található, a példában ilyen volt a `FILE_NOT_FOUND` és az `EXIST`, az utolsó pedig a hibaüzenet.

Mivel a Vala kivételkezelési rendszere a GLib-en alapszik. Így ehhez igazodik a nyelv. A hibadoméneket nekünk kell deklarálnunk az alábbi módon:

```
errordomain IOError {
    FILE_NOT_FOUND,
    EXIST
}
```

Amikor az eldobott kivételt el szeretnénk kapni, akkor megadjuk az elkapni kívánt domént. A hibából ezután kinyerhetjük a hibakódot, és az üzenetet. Ha több doménből szeretnénk hibákat elkapni, akkor egyszerűen több `catch` blokkot kell megadnunk.

A Vala nyelv támogatja a Java nyelvből ismerős `finally` blokkot is. A `finally` blokk minden esetben lefut, akkor is, ha nem kaptunk el kivételt. Ez például arra használható, hogy a `try` blokkban lefoglalt erőforrásokat felszabadítsuk.

A következő teljes példa jól bemutatja a Vala nyelv hibakezelését:

```
public errordomain ErrorType1 {
    CODE_1A
}

public errordomain ErrorType2 {
    CODE_2A,
    CODE_2B
}

public class Test : GLib.Object {
    public static void thrower() throws ErrorType1, ErrorType2 {
        throw new ErrorType1.CODE_1A("Hiba");
    }

    public static void catcher() throws ErrorType2 {
        try {
            thrower();
        } catch (ErrorType1 e) {
            // ErrorType1 lekezelése
        } finally {
            // Takarítás
        }
    }

    public static int main(string[] args) {
        try {
            catcher();
        } catch (ErrorType2 e) {
            // ErrorType2 lekezelése
            if (e is ErrorType2.CODE_2B) {
                // CODE_2B kezelése
            }
        }
        return 0;
    }
}
```

5.4. Paraméter irányok

A Vala metódusoknak nulla vagy pozitív számú argumentuma van. A metódushívás során az alapértelmezett viselkedés a következő:

- Minden érték típus másolódik a metódusba a végrehajtás során
- Minden referencia típusra mutató referencia átadásra kerül

Ez a viselkedés a `ref` és a `out` módosítókkal megváltoztatható:

`out` a hívó oldalon	a változó lehet inicializálatlan, és ekkor számíthatunk arra, hogy a metódusban inicializálódik
`out` a meghívott oldalon	a paramétert inicializálatlannak tekintjük, ezért nekünk kell inicializálni
`ref` a hívó oldalon	az átadott változónak inicializáltnak kell lennie, és a a metódus meg is változtathatja a változót
`ref` a meghívott oldalon	a paraméter inicializáltnak tekintett, és megváltoztatható a metódus belsejében

A módosítókat egyszerűen a paraméter típusa elé írhatjuk:

```
void method_1(int a, out int b, ref int c) {
    // ...
}
void method_2(Object o, out Object p, ref Object q) {
    // ...
}
```

Ezeket a metódusokat az alábbi módon használhatjuk:

```
int a = 1;
int b;
int c = 3;
method_1(a, out b, ref c);

Object o = new Object();
Object p;
Object q = new Object();
method_2(o, out p, ref q);
```

A változók kezelése az alábbi:

- az `a` változó érték típusú, és érték szerint adódik át, tehát a metóduson belüli módosítások nem láthatóak a metóduson kívülről
- a `b` változó is érték típusú, de mivel `out` módosítójú ezért mutató szerint adódik át, tehát a metóduson belüli módosítások a metóduson kívül is megmaradnak
- a `c` változót ugyanúgy kezeljük mint a `b`-t, csak a jelzett szándékban különbözik – eleve inicializált változó átírása
- az `o` változó referencia típusú, a metóduson belül megváltoztatható, azonban, ha újradefiniáljuk, akkor a változás nem marad meg
- a `p` is referencia típusú, de `out` módosítójú ezért újra is definiálhatjuk a metódusban – ha ezt mégsem tennénk, akkor `null` értéket vesz fel a metódus végén
- a `q` változót ugyanúgy kezeljük, mint a `p`-t, azonban itt dönthetünk úgy, hogy nem változtatjuk meg a metódusban

Egy példa megvalósítás a `method_1` metódusra:

```
void method_1(int a, out int b, ref int c) {
    b = a + c;
    c = 3;
}
```

Ha a `b` paraméter `out` típusú, akkor a Vala garantálja, hogy a `b` nem lest `null` értékű. Ezért aztán a metódushívás során a második argumentum biztonságosan lehet `null`.

5.5. Gyűjtemények

A *Gee* függvénykönyvtár egy olyan Vala függvénykönyvtár, mely megvalósítja a gyűjteményeket a Vala nyelvben. A Gee különböző interfészek és típusok halmaza, amelyek a gyűjteményeket más és más módon implementálják. A Gee-t külön kell telepítenünk.

A legfontosabb gyűjtemény típusok:

- Lista: rendezett gyűjtemény, számszerű indexeléssel
- Halmaz: rendezetlen gyűjtemény, különböző elemekkel
- Leképezés: rendezetlen gyűjtemény, tetszőleges típusú indexeléssel

Minden lista és halmaz implementálja a `Collection` interfészt, és minden leképezés implementálja a `Map` interfészt. Ezen felül a listák még implementálják a `List` interfészt, a halmazok pedig a `Set` interfészt.

Ezeknek az interfészeknek nem csak az a szerepük, hogy a hasonló gyűjtemények felcserélhetőek legyenek, hanem az is, hogy saját gyűjteményosztályokat hozhassunk létre.

Ami még közös a gyűjteményekben az az, hogy mindegyik implementálja az `Iterable` interfészt. Ebből az következik, hogy metódusaikkal tudunk léptetni bennük, és az is hogy a `foreach` ciklust alkalmazhatjuk rajtuk.

Minden osztály és interfész alkalmazza a generikus típusokat, tehát minden egyes gyűjteménynél meg kell adni a típusát, vagy a típusok halmazát. Ez biztosítja azt, hogy egy adott típus betehető legyen a gyűjteménybe, és azt is, hogy megfelelő típust nyerjünk ki belőle.

Pár fontos gyűjtemény osztály:

- `ArrayList<G>`
  - Implementálja: `Iterable<G>`, `Collection<G>`, `List<G>`
  - Ez a típus nagyon gyors adatelérést biztosít, azonban a feltöltése lassú lehet – főleg ha nem a végére akarunk beszúrni, vagy ha betelt a háttérben lévő tömb
- `HashMap<K,V>`
  - Implementálja: `Iterable<Entry<K,V>>`, `Map<K,V>`
  - Ez a típus egy 1:1 leképezés `K` és `V` típusú elemek között, a leképezés egy hasítófüggvény segítségével működik, amely egy értéket számít ki minden kulcshoz
  - Opcionálisan megadhatunk saját hasítófüggvényt, és összehasonlító metódust:

    ```
    var map = new Gee.HashMap<Foo, Object>(foo_hash, foo_equal);
    ```

    Karakterláncok, és egész típusú változók esetén ez automatikus, csak akkor kell megadnunk, ha meg szeretnénk változtatni az alapértelmezést.
- `HashSet<G>`
  - Implementálja: `Iterable<G>`, `Collection<G>`, `Set<G>`
  - Ez a típus `G` típusú elemek halmaza, ahol a többszöres elemeket hasítófüggvény segítségével szűrjük ki
  - A hasítófüggvény, és az összehasonlító metódus itt is megadható

Fontos még megjegyezni az ún. *csak olvasható nézetet*. Ez gyakorlatilag egy segédosztály, mely minden gyűjteményből származtatható. A segédosztály az eredetivel megegyezik, azonban nem módosítható, és nem nyerhetjük belőle vissza a gyűjteményt. A csak olvasható nézet kérésére használjuk a `read_only_view` tulajdonságot.

5.6. Metódusok szintaktikai támogatással

A Vala nyelv felismer egyes metódusokat és mintákat, és szintaktikai támogatást biztosít hozzájuk. Például, ha egy típus rendelkezik `contains()` metódussal, akkor az ilyen típusú objektumokon használható az `in` operátor. A következő táblázatokban a `T`, `T1`, `T2` és `T3` típusok csak példák:

Indexelés	
`T2 get(T1 index)`	index hozzáférés: `obj[index]`
`void set(T1 index, T2 item)`	index értékadás: `obj[index] = item`
Indexelés több indexszel	
`T3 get(T1 index1, T2 index2)`	index hozzáférés: `obj[index1, index2]`
`void set(T1 index1, T2 index2, T3 item)`	index értékadás: `obj[index1, index2] = item`
Egyebek	
`T slice(long start, long end)`	Szeletelés: `obj[start:end]`
`bool contains(T needle)`	`in` operátor: `bool b = needle in obj`
`string to_string()`	használható karakterlánc sablonokban: `@"$obj"`
`Iterator iterator()`	Iterálható a `foreach` operátorral

Az `Iterator` típusnak bármilyen nevet adhatunk. A helyes működéséhez arra van szükség, hogy az alábbi két metódusból egyet implementáljunk:

`bool next()` `T get()`	Sztenderd iterációs protokoll: léptetés amíg `next()` nem `false`. Az aktuális elemet a `get()` metódussal kérhetjük le.
`T? next_value()`	Alternatív iterációs protokoll: ha az iterátor objektumnak van `next_value()` metódusa, amely visszatérési értéke nullázható, akkor ezzel a metódussal léptethetünk amíg nem kapunk `null`-t.

A következő példa ezeknek egy részét szemlélteti:

```
public class EvenNumbers {
    public int get(int index) {
        return index * 2;
    }

    public bool contains(int i) {
        return i % 2 == 0;
    }

    public string to_string() {
        return "[This object enumerates even numbers]";
    }

    public Iterator iterator() {
        return new Iterator(this);
    }
```

```
    public class Iterator {
        private int index;
        private EvenNumbers even;

        public Iterator(EvenNumbers even) {
            this.even = even;
        }

        public bool next() {
            return true;
        }

        public int get() {
            this.index++;
            return this.even[this.index - 1];
        }
    }
}

void main() {
    var even = new EvenNumbers();
    stdout.printf("%d\n", even[5]);    // get()
    if (4 in even) {                  // contains()
        stdout.printf(@"$even\n");    // to_string()
    }
    foreach (int i in even) {         // iterator()
        stdout.printf("%d\n", i);
        if (i == 20) break;
    }
}
```

5.7. Többszálú programozás

5.7.1. Szálak a Vala nyelvben

A Vala programoknak több végrehajtási száluk lehet, így több dolgot végezhetnek egyszerre. Az hogy ez pontosan hogyan történik nem tartozik a Vala hatáskörébe – a szálak futhatnak egy vagy több processzoron, a környezettől függően.

A szálakat a Vala nyelvben nem fordítás időben definiáljuk, hanem a futásidőben kérjük az operációs rendszert arra, hogy egy kódrészlet új szálban fusson. Ez a GLib könyvtár `Thread` osztály statikus metódusaival történik. Egy egyszerű példa:

```
void* thread_func() {
    stdout.printf("Thread running.\n");
    return null;
}
```

```
int main(string[] args) {
    if (!Thread.supported()) {
        stderr.printf("Cannot run without threads.\n");
        return 1;
    }

    try {
        Thread.create(thread_func, false);
    } catch (ThreadError e) {
        return 1;
    }

    return 0;
}
```

Ez a rövid program azt kéri, hogy létrejöjjön egy új szál, és végrehajtódjon. A futtatandó kód a `thread_func()` metódus.

Fontos megjegyezni, hogy ha szálakat szeretnénk használni, akkor a `--thread` kapcsolót kell használnunk a fordítóprogramban. Ez fordítás közben ellenőrzi, hogy van-e száltámogatás, és ettól függően dönt, hogy miként fordítson. Ez a parancs csatolja a szükséges könyvtárakat, valamint biztosítja, hogy a szálak alrendszere elinduljon, amennyiben ez lehetséges.

A program a kapcsoló használatával már hibamentesen lefordul, és a futtatás során sem kapunk szegmentációs hibát. De még így sem úgy működik, ahogy várnánk, ugyanis a program véget ér, amint a főszál befejeződik. Tehát nem biztos, hogy a mellékszál ténylegesen lefut.

Lehetőség van arra, hogy egy szál azt mondja a rendszernek, hogy pillanatnyilag nem akar futni, és ezért azt javasolja, hogy egy másik szál fusson. Erre szolgál a `yield()` statikus metódus. Ha a `Thread.yield()` sort hozzáírjuk a `main()` metódushoz, akkor a főszál megáll és futni hagyja a mellékszálat, ha az futásra kész. Ez általában működni fog, de nem garantált!

Ahhoz hogy megvárjunk egy szál végrehajtását a `join()` metódust használhatjuk. Emellett ez arra is jó, hogy a mellékszál visszatérési értékét megkapjuk. A metódus használata:

```
try {
    unowned Thread thread = Thread.create(thread_func, true);
    thread.join();
} catch (ThreadError e) {
    return 1;
}
```

Ezúttal amikor létrehozzuk a szálat, megadunk egy második paramétert is – `true` értékkel. Ez „illeszthetőnek” jelöli a szálat. Megjegyezzük, hogy a szálkészítés visszatérési értéke egy tulajdonos nélküli referencia – ezekről később. Ezzel a referenciával tudjuk összeilleszteni a szálakat. Ez a megoldás garantálja a program elvárt működését.

5.7.2. Erőforrás kezelés

Amikor több végrehajtási szál fut egyszerre, akkor előfordulhat, hogy adatokhoz egyszerre akarunk hozzáférni. Ez versenyhelyzethez vezethet, amelynek kimenetele a rendszertől függ.

Ahhoz hogy ezt a helyzetet kezelni tudjuk, használhatjuk a `lock` kulcsszót, amely biztosítja, hogy egy kódrészlét futása során a többi szál ne szakíthassa félbe. Egy példa:

```
public class Test : GLib.Object {

    private int a { get; set; }

    public void action_1() {
        lock (a) {
            int tmp = a;
            tmp++;
            a = tmp;
        }
    }

    public void action_2() {
        lock (a) {
            int tmp = a;
            tmp--;
            a = tmp;
        }
    }
}
```

Az osztály két metódust definiál, melyek mindketten az a tulajdonságot változtatják meg. Ha nem lennének zárolások, akkor a metódusok összeakadhatnának, és az a értéke gyakorlatilag véletlen lehet.

A Vala nyelvben a zárolások csak egy objektumon belül, és csak adattagokra vonatkozhatnak. Ez elsőre komoly korlátozásnak tűnik, de a gyakorlatban más használatra nagyon nincs is szükség.

5.8. Főciklus

A GLib könyvtár tartalmaz egy alrendszert, mely egy eseményhurkot valósít meg. Ez az alrendszer a `MainLoop` és a hozzá kapcsolódó osztályok. Az alrendszer célja az, hogy úgy írhassunk programokat, hogy eseményekre várunk, és válaszolunk rájuk, ahelyett hogy folyamatosan feltételeket vizsgálunk. Ezt a modellt alkalmazza a GTK+ grafikus eszközkészlet – amelyet mi is használni fogunk. A GTK+ képes várni anélkül, hogy kódot futtatna, addig amíg nem történik felhasználói interakció.

A következő program létrehoz, és elindít egy főciklust, és aztán eseményforrást csatol hozzá. Ebben az esetben a forrás egy egyszerű időzítő, amely a megadott metódust 2 másodpercenként futtatja. A metódus egyébként le is állítja a főciklust, így csak egyetlen egyszer fog lefutni.

```
void main() {

    var loop = new MainLoop();
    var time = new TimeoutSource(2000);

    time.set_callback(() => {
        stdout.printf("Time!\n");
        loop.quit();
        return false;
    });

    time.attach(loop.get_context());

    loop.run();
}
```

Ha használjuk a GTK+ függvénykönyvtárat, akkor a főciklus automatikusan létrejön, és elindul amint meghívjuk a `Gtk.main()` metódust. Ez jelzi azt a pontot, ahol a program futásra kész, és elkezdhet eseményeket fogadni a felhasználótól – vagy máshonnan.

Ez a GTK+ példa ekvivalens az előzővel:

```
void main(string[] args) {

    Gtk.init(ref args);
    var time = new TimeoutSource(2000);

    time.set_callback(() => {
        stdout.printf("Time!\n");
        Gtk.main_quit();
        return false;
    });

    time.attach(null);

    Gtk.main();
}
```

Grafikus alkalmazásoknál gyakori követelmény, hogy kódokat futtassunk amint lehetséges, de anélkül hogy zavarnánk a felhasználót. Ehhez használjuk az `IdleSource` osztály példányait. Ezek elküldik az eseményeket a főciklushoz, és azt kérik, hogy kezeljék őket, ha nincs épp fontosabb esemény.

Bővebb információ a főciklusokról – és sok másról – a GLib és a GTK+ dokumentációkban található.

5.9. Aszinkron metódusok

Az aszinkron metódusok használatával olyan metódusokat írhatunk, amelyek nem kerülnek blokkolt állapotba. A Vala nyelvben ez a GIO függvénykönyvtár segítségével van megvalósítva, de rendelkezik saját szintaktikai támogatással.

5.9.1. Szintaktika és példa

Aszinkron metódusokat az `async` kulcsszóval definiálhatunk. Az aszinkron metódusokat szinkron metódusokból a `async_method_name.begin()` metódussal hívhatjuk meg. A `begin()` metódusnak tetszőleges számú argumentama lehet, amelyek közül az elsők a bemenő paraméterek, az utolsó pedig egy `AsyncReadyCallback` típusú visszahívási metódus. Ebben a metódusban kötelező meghívni az `end()` metódust, hogy felszabadítsuk az erőforrásokat.

Az `end()` metódus is tetszőleges számú argumentummal rendelkezhet, amelyek a kimenő paraméterek, emellett visszaadja az szinkron műveletsor eredményét, és kivételeket dobhat.

Az aszinkron metódusokból más aszinkron metódusokból a `yield` kulcsszó segítségével hívhatunk meg. Ez felfüggeszti az aktuális aszinkron metódust, addig amíg az új aszinkron metódus fut.

Mindez a háttérben visszahívási metódusokkal működik az `AsyncResult` osztály segítségével. Az aszinkron metódusok függenek a GIO függvénykönytártól, ezért a programot `--pkg gio-2.0` kapcsolóval kell lefordítanunk. Egy példaprogram:

```
async void list_dir() {
    var dir = File.new_for_path (Environment.get_home_dir());
    try {
        var e = yield
            dir.enumerate_children_async(FILE_ATTRIBUTE_STANDARD_NAME,
                                         0, Priority.DEFAULT, null);
        while (true) {
            var files = yield
                e.next_files_async(10, Priority.DEFAULT, null);
            if (files == null) {
                break;
            }
            foreach (var info in files) {
                print("%s\n", info.get_name());
            }
        }
    } catch (Error err) {
        warning("Error: %s\n", err.message);
    }
}

void main() {
    list_dir.begin();
    new MainLoop().run();
}
```

A `list_dir()` metódus normális esetben nem kerülne blokkolt állapotba. Ezért a metódus belsejében az `enumerate_children_async()` és a `next_files_async()` aszinkron metó-

dusokat a `yield` kulcsszóval hívjuk meg. A `list_dir()` futása akkor folytatódik, ha a többi metódus visszatér.

5.9.2. Saját aszinkron metódusok

Az előző példában GIO metódusokkal szemléltettük, hogy hogyan használjuk a `begin()` metódust és a `yield` kulcsszót. Azonban arra is lehetőség van, hogy saját aszinkron metódusokat írjunk.

```
class Test : GLib.Object {
    public async string test_string(string s, out string t) {
        assert(s == "hello");
        Idle.add(test_string.callback);
        yield;
        t = "world";
        return "vala";
    }
}

async void run(Test test) {
    string t, u;
    u = yield test.test_string("hello", out t);
    print("%s %s\n", u, t);
    main_loop.quit();
}

MainLoop main_loop;

void main() {
    var test = new Test();

    run.begin(test);

    main_loop = new MainLoop();
    main_loop.run();
}
```

A `.callback` hívást arra használjuk, hogy implicit módon egy `_finish` metódust regisztráljunk a metódushoz. Ezt használjuk az egyedül álló `yield` utasításban.

```
// visszahívási metódus hozzáadása
Idle.add(async_method.callback);
yield;
// az eredmény visszaadása
return a_result_of_some_type;
```

A `yield` utasítás után az eredményt kiszámíthatjuk és visszaadhatjuk. Még egy példa:

```
MainLoop main_loop;
```

```
class Test : Object {
    private async int launch_wait() {
        Idle.add(launch_wait.callback);
        yield;
        int i;
        for (i = 0; i < int.MAX/2; i++) {
            continue;
        }
        return i;
    }

    public async void wait() {
        print("Wait\n");
        var i = yield launch_wait();
        message(@"$i");
        print("Wait done\n");
        main_loop.quit();
    }

    public async void nowait() {
        print ("No wait\n");
        print ("No wait done\n");
    }
}

void main() {
    var test = new Test();

    test.wait.begin();
    test.nowait.begin();

    main_loop = new MainLoop();
    main_loop.run();
}
```

Az `end()` metódus egy szintaktikai formája a `*_finish` metódusnak. Bemenő paramétere egy `AsyncResult` típus, visszatérési értéke a tényleges végeredmény, vagy kivételt dob – ha a metódus ilyet tesz.

```
async_method.end(result)
```

5.10. Gyenge referenciák

A Vala nyelv memória-kezelése a referencia számláláson alapszik. Amikor egy objektumot egy változóhoz rendelünk, akkor a belső referencia számlálót 1-gyel növeljük. Amikor egy változó kikerül a hatókörből, akkor pedig a számlálót 1-gyel csökkentjük. Ha a referencia számláló 0 lesz, akkor az objektum felszabadításra kerül.

Azonban lehetséges, hogy referencia hurok alakul ki. Például ha egy faszerkezetben a szülő és a gyerek elemek kölcsönösen referenciát tartalmaznak egymásra. Ilyen példa még a kétszeresen láncolt lista is.

Ezekben az esetekben az objektumok „életben maradnak” akkor is ha már nincs rájuk szükség. Ennek az oka a referencia hurok. Ezekben az esetekben rendkívül hasznos a gyenge referencia. A gyenge referenciák nem befolyásolják a referencia számlálást, így megszüntetik a hurkokat. Példa:

```
class Node {
    public weak Node prev;
    public Node next;
}
```

5.11. Tulajdonjog

5.11.1. Tulajdonos nélküli referenciák

A tulajdonos nélküli referenciák nem kerülnek feljegyzésre az objektumban. Így lehetséges az, hogy eltávolítsuk az objektumot amikor az logikailag szükséges volna – függetlenül attól, hogy vannak-e még referenciák rá vagy sem. A tulajdonos nélküli referenciák használatának általános módja az, hogy egy metódus ilyen referenciát ad vissza:

```
class Test {
    private Object o;

    public unowned Object get_unowned_ref() {
        this.o = new Object();
        return this.o;
    }
}
```

Amikor a visszatérési értéket tárolni szeretnénk, akkor is szükséges annak a jelölése, hogy tulajdonos nélküli a referencia:

```
unowned Object o = get_unowned_ref();
```

Ez kissé talán komplikáltnak tűnik, de jó oka van annak, hogy így működik.

- Ha az `o` `Object` nem lenne az osztály adattagja, akkor a `get_unowned_ref()` metódus érvénytelen referenciát adna visszatérési értéknek.
- Ha a visszatérési érték nem lenne tulajdonos nélkülinek jelölve, akkor a tulajdonlás a hívó kódra szállna. A hívó kód viszont tulajdonos nélküli referenciát vár, amelyet nem birtokolhat.

5.11.2. Tulajdonságok

A normál metódusokkal ellentétben a tulajdonságok mindig tulajdonos nélküli referenciát adnak vissza. Ez azt jelenti, hogy a `get()` metódus nem adhat vissza új objektumokat. Ennek az az oka, hogy egy `get()` hívás nem változtathatja meg a referencia számlálót.

Ezért az alábbi fordítási hibát okoz:

```
public Object property {
    get {
        return new Object();
    }
}
```

ahogyan ez is:

```
public string property {
    get {
        return getter_method();
    }
}

public string getter_method() {
    return "some text";
}
```

Viszont ez már hibátlan:

```
public string property {
    get {
        return getter_method();
    }
}

public unowned string getter_method() {
    return "some text";
}
```

5.11.3. Tulajdonjog átadása

Az `owned` kulcsszó arra szolgál, hogy a tulajdonjogot átadjuk.

- Mint prefix paraméter azt jelenti, hogy a tulajdonjog az aktuális kontextusé lesz.
- Mint típuskonverziós operátor arra használható, hogy elkerüljük a referencia számlálást nem használó osztályok duplikációját – ez általában egyébként lehetetlen volna. Erre példa:

```
Foo foo = (owned) bar;
```

Ekkor a `bar` változó értéke `null` lesz, és a `foo` örökli a referenciát és a tulajdonjogot.

5.12. Változó hosszúságú argumentumlista

A Vala támogatja a C stílusú változó hosszúságú argumentumlistákat. A deklarációjuk három ponttal történik a szignatúrában. Az argumentumlistához minimum egy fix argumentum szükséges.

```
void method_with_varargs(int x, ...) {
    var l = va_list();
    string s = l.arg();
    int i = l.arg();
    stdout.printf("%s: %d\n", s, i);
}
```

Ebben a példában a kötelező argumentum az `x` egész típusú változó. Az argumentumlistát a `va_list()` metódussal kérhetjük le. Ezután egymás után kinyerhetjük az elemeket a `arg<T>()` generikus metódussal. Ha a kontextusból egyértelműen kiderül a pontos típus, akkor nem kötelező típust megadni – mint az előző példában.

A következő metódus tetszőleges számú karakterlánc és dupla pontosságú lebegőpontos változópárokat dolgoz fel:

```
void method_with_varargs(int fixed, ...) {
    var l = va_list();
    while (true) {
        string? key = l.arg();
        if (key == null) {
            break;  // end of the list
        }
        double val = l.arg();
        stdout.printf("%s: %g\n", key, val);
    }
}

void main() {
    method_with_varargs(42, "foo", 0.75, "bar", 0.25, "baz", 0.32);
}
```

Védelmi mechanizmusként `null` érték vizsgálat is szerepel a metódusban. A Vala mindig implicit módon egy `null` értéket ad át egy változó hosszúságú argumentumlista végén.

Ezeknek a nagy hiányossága az, hogy nem típusbiztos, azaz a fordítóprogram nem tudja eldönteni, hogy helyes-e a metódushívás. Ezért csak nagy odafigyeléssel szabad ezt a lehetőséget használni.

Gyakori minta ennél a szerkezetnél az, hogy karakterlánc - érték párokat várunk, legjellemzőbb módon GObject tulajdonság - érték párokat. Ebben az esetben a `tulajdonság : érték` szintaxist használhatjuk. Egy példa:

```
actor.animate (AnimationMode.EASE_OUT_BOUNCE, 3000, x: 100.0,
               y: 200.0, rotation_angle_z: 500.0, opacity: 0);
```

5.13. Mutatók

A mutatók a Vala nyelvben a kézi memória-kezelés eszközei. Alapesetben ha létrehozunk egy példányt, akkor kapunk egy referenciát. A Vala elintézi az objektum törlését, ha a rá mutató referenciák elfogynak. Azáltal hogy ehelyett egy mutatót kérünk egy referencia helyett átvállaljuk a felelősséget a rendszertől. Így nekünk kell elvégezni a törlést is, ezáltal nagyobb az irányításunk az objektum felett.

Ez a funkcionalitás ma már nem feltétlenül szükséges, hiszen a számítógépek elég gyorsak ahhoz, hogy a referencia számítást elbírják, és a memória-használat kismértékű növekedése sem igazán lényeges. A fő okok amiért mégis szükség lehet rá:

- Ha egy programrészt direkt optimalizálni akarunk.
- Ha egy külső függvénykönyvtárat használunk, amely nem támogatja a referencia számlálást – nagy valószínűséggel nem GObject alapú.

Mutató létrehozása egy objektumhoz:

```
Object* o = new Object();
```

Ahhoz hogy elérjük az adattagokat és metódusokat, használjuk a C/C++ nyelvből ismert *követi* (->) operátort:

```
o->method_1();
o->data_1;
```

A memória felszabadításhoz használjuk a `delete` operátort:

```
delete o;
```

A Vala támogatja a C-ből ismert *címe* (&) és az *indirekció* (*) operátorokat is:

```
int i = 42;
int* i_ptr = &i;
int j = *i_ptr;
```

A viselkedés kicsit különböző a referencia típusoknál:

```
Foo f = new Foo();
Foo* f_ptr = f;     // címe
Foo g = f_ptr;      // indirekció

unowned Foo f_weak = f;  // ekivalens a második sorral
```

A referencia típusú mutatók használata megegyezik a tulajdonos nélküli referenciákkal.

5.14. Nem Object alapú osztályok

Azon osztályok, melyek nem az `Object` osztály leszármazottjai, speciálisak. Közvetlenül a GLib típusrendszeréből származnak, így sokkal könnyebb súlyúak. Mivel a GLib sokkal alacsonyabb szintű, mint a GObject, ezért a köztük lévő kötés során használják. A könnyűsúlyú tulajdonsága miatt pedig a Vala fordítóprogramban is erősen használják. Ezek használatáról bővebben most nem lesz szó.

5.15. D-Bus integráció

A D-Bus rendszer egy egyszerű processzek közti kommunikációt (IPC-t) megvalósító rendszer Unix-szerű rendszerekre – Windows port is létezik. A rendszer szorosan beépül a Vala nyelvbe, így igen könnyű használni.

Ahhoz hogy egy saját osztályunkat D-Bus szolgáltatásként exportáljuk, ahhoz csak használni kell a [DBus] attribútumot, és regisztrálni kell a példányt a helyi D-Bus munkamenetez.

```
[DBus(name = "org.example.DemoService")]
public class DemoService : Object {
    // Privát adattag, nem exportálódik a D-Bus-on keresztül
    int counter;

    // Publikus adattag, nem exportálódik a D-Bus-on keresztül
    public int status;

    // Publikus tulajdonság, exportálódik
    public int something { get; set; }

    /* Publikus szignál, exportálódik. Kibocsátható a szerver oldalon
       és kapcsolódhatunk hozzá a kliens oldalon. */
    public signal void sig1();

    // Public metódus, exportálódik
    public void some_method() {
        counter++;
        stdout.printf("heureka! counter = %d\n", counter);
        sig1();  // emit signal
    }

    /* Publikus metódus, exportálódik és megmutatja a küldőt,
       aki meghívta a metódust (az nem exportálódott) */
    public void some_method_sender(string message, GLib.BusName sender) {
        counter++;
        stdout.printf("counter = %d, '%s' message from sender %s\n",
                      counter, message, sender);
    }
}
```

A szolgáltatás regisztrációja, és a főciklus elindítása:

```
void on_bus_aquired (DBusConnection conn) {
    try {
        // szolgáltatás indítása és regisztrációja, mint D-Bus objektum
        var service = new DemoService();
        conn.register_object ("/org/example/demo", service);
    } catch (IOError e) {
        stderr.printf ("Could not register service: %s\n", e.message);
    }
}
```

```
void main () {
    // a szolgáltatás nevének regisztrációja a munkamenetben
    Bus.own_name (BusType.SESSION, "org.example.DemoService", // név
                  BusNameOwnerFlags.NONE, // jelzők
                  // visszahívási metódus, regisztráció sikere esetén
                  on_bus_aquired,
                  // visszahívás a névregisztráció sikere esetén
                  () => {},
                  // visszaívás a név elvesztése esetén
                  () => stderr.printf ("Could not aquire name\n"));

    // főciklus indítása
    new MainLoop().run();
}
```

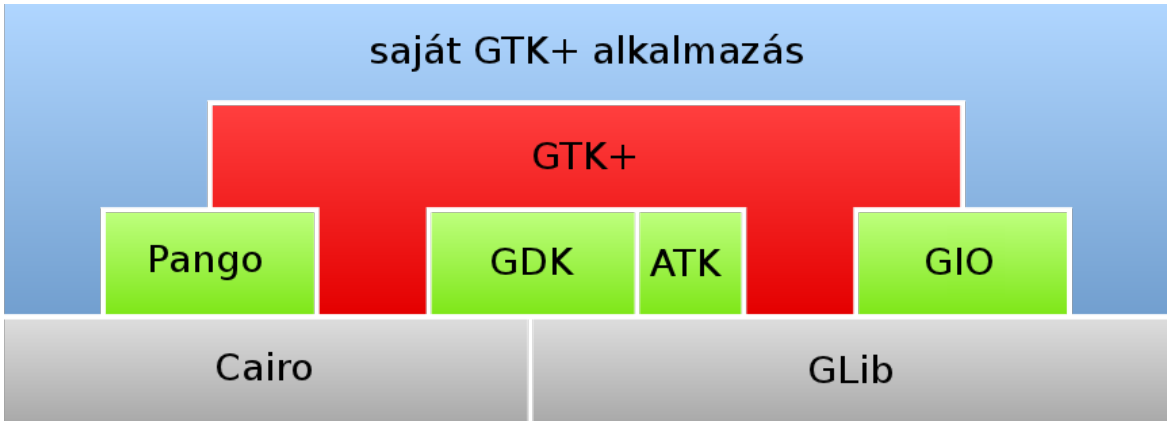
A példa lefordításához szükséges a `--pkg gio-2.0` parancs. A fordítóprogram automatikusan átírja a Vala nyelv `lower_case_with_underscores` elnevezési konvencióját a D-Bus rendszer `camelCase` stílusára.

6. A GNOME platform bemutatása

A GNOME (eredetileg a *GNU Network Object Model Environment* rövidítése) egy grafikus felhasználói felület és asztali környezet Linux disztribúciókhoz és egyéb Unix-szerű operációs rendszerekhez. A GNOME platform legújabb mérföldköve, a 3-as változat nemrégiben (2011. április 6.-án) debütált, izgalmas változásokat hozva a több mint 9 évig fejlesztett 2-es verzió után. A jegyzetben már ezt a változatot fogjuk használni.

A programozó számára a platform egy függvénykönytár gyűjtemény, amely megkönnyíti az alkalmazás-fejlesztést. A platform legfontosabb részei:

- **GTK+**: grafikus interfész eszközkészlet



- a függvénykönyvtárt támogató más könyvtárak:
  - **Pango**: betűtípus és szöveg renderelő könyvtár, mely jól támogatja a lokalizációt
  - **GDK**: ablak- és eseménykezelő könyvtár
  - **ATK**: akadálymentesítési függvénykönyvtár
  - **GIO**: fájlrendszer és hálózati támogatást biztosító könyvtár
  - **Cairo**: 2D grafikus könyvtár, hardveres támogatással több operációs rendszeren
  - **GLib**: a legalacsonyabb szintű könyvár, mely a GNOME alapját képezi

Emellett fontos megjegyezni még pár magasabb szintű könyvtárat, melyeket ugyan nem fogunk használni, de jó tudni, hogy micsodák. Ezek:

- **GStreamer**: a GNOME audio- és videókomponense, mely szinte minden igényt kielégít; a rendszer architektúrájának köszönhetően beépülő modulokkal bővíthető
- **WebKitGTK+**: a WebKit HTML motor portja GTK+ alkalmazásokhoz
- **Clutter**: OpenGL alapú grafikai megvalósítások grafikus felületekhez – egyébiránt ez a *MeeGo* mobil platform egyik fő komponense is.

A platform nagyvonalú bemutatása után bele is vágunk az első GTK+ programunk elkészítéséhez, majd a Glade felülettervező program ismertetéséhez.

7. GTK+ Helló világ!

A következő program egy rendkívül egyszerű GTK+ alkalmazás, mely létrehoz egy ablakot, az ismerős „Helló világ!” üzenettel.

Main.vala:

```
using Gtk;

int main(string [] args) {

    Gtk.init(ref args);

    var window = new Window(WindowType.TOPLEVEL);

    var label = new Label("Helló világ!");
    window.add(label);

    window.title = "GTK+ Helló világ!";
    window.set_has_resize_grip(false);
    window.set_default_size(300,50);
    window.set_position(WindowPosition.CENTER);
    window.destroy.connect (Gtk.main_quit);

    window.show_all();

    Gtk.main();
    return 0;

}
```

Tekintsük meg bővebben a programot! A programban importáljuk a Gtk a névteret a típusok direkt eléréséhez. A `main` függvényben inicializáljuk a GTK+ függvénykönyvtárat, és átadjuk az argumentumokat – referencia szerint. Ezután létrehozunk egy új ablakot, mint felsőszintű ablak. A következő sorban létrehozunk egy címkét, a „Helló világ!” felirattal, majd ezt a címkét hozzáadjuk az ablakhoz.

A következő sorokban beállítjuk az ablak címét, kikapcsoljuk az átméretezési pöcköt az ablak jobb alsó részén, beállítjuk a kezdőméreteit és a pozícióját, valamint egy visszahívási eseményt adunk meg az ablak bezárásához – ez az alap `main_quit` metódus lesz.

Ezután megjelenítjük az ablakot, és elindítjuk a GTK+ főciklusát. A metódus végi visszatérési utasítást sosem fogjuk elérni, csak szintaktikai okokból van rá szükség.

A program lefordításához szükséges a *valac* fordítóprogramnak megadni a GTK+ csomagot, ehhez szükséges a `--pgk gtk+-3.0` kapcsoló megadása. Így a 2. fejezetben található Makefile-t módosítjuk eszerint.

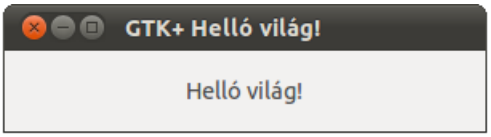
Makefile:

```
PROGRAM = build/Main

build:
    mkdir build;
    valac --pkg gtk+-3.0 Main.vala;
    mv Main build/Main

clean:
    rm -rf build
```

Ezután már nincs más dolgunk, mint kipróbálni a programot! A projekt egyébként megtekinthető a **GtkHello.zip** fájlban. A fordítás és futtatás után a következő képernyő fogad minket:



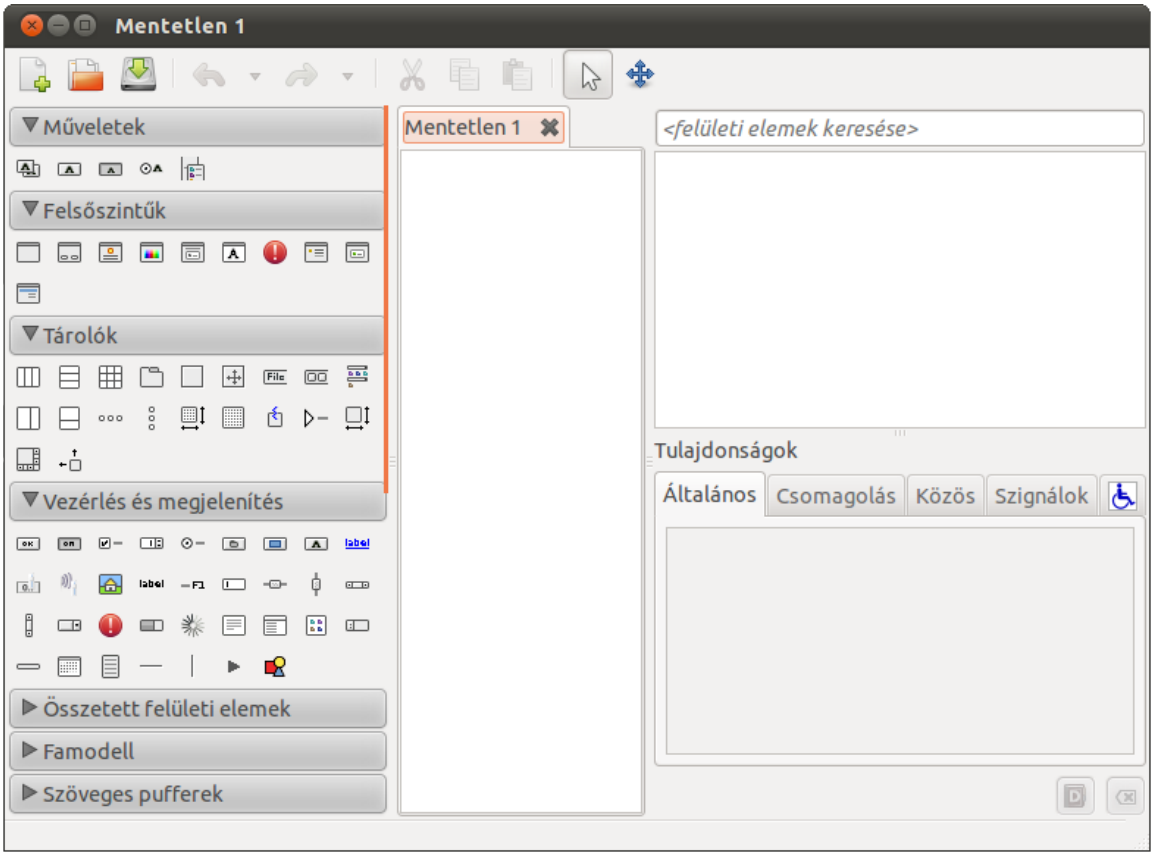
Ezek után már el is kezdhetnénk komolyabb programokat írni, azonban igen célszerű megtanulni a grafikus felület elkészítésének egy egyszerűbb módját. Ez a GTK+ környezetben a Glade grafikus felület tervező eszköz, és az ehhez kapcsolódó `Builder` osztály. Egyébként ilyen úgynevezett gyors alkalmazás-fejlesztő eszköz (RAD - *Rapid Application Development*) szinte minden grafikus eszközkészlethez létezik.

8. A Glade felülettervező alkalmazás

Ahogy azt már említettük, a Glade a GNOME projekt grafikus felület tervező alkalmazása. Ezen segédprogram segítségével könnyen, és gyorsan tervezhetünk meg teljes grafikus felületeket anélkül, hogy egy sor kódot is írnunk kellene.

Ez úgy lehetséges, hogy a program egy XML fájlt hoz létre, amely tárolja a grafikus felület leírását. A programban aztán futásidőben a `Builder` osztály egy példánya a leírás alapján elkészíti a tényleges grafikus felületet. Ez természetesen plusz erőforrást igényel, de a fejlesztési idő csökkentése, és a kód átláthatóságának növekedése miatt megéri.

A Glade program használata rendkívül egyszerű, köszönhetően a minimalista felületnek.



A bal oldali panelen láthatóak a GTK+ felületelemek, melyekből felépíthetjük a grafikus felületünket. Középen lesz látható a felület látványterve, a jobb oldalon pedig a felületi elemek listáját láthatjuk, valamint az aktuális elem paramétereit.

Most egy példa keretében elkészítjük a 7. fejezetben lévő „Helló világ!” program Glade változatát. Először is hozzuk létre a Glade fájlt, amely leírja a felületet:

- Adjunk hozzá egy új felsőszintű ablakot, és nevezzük el `window`-nak.
- Állítsuk be a tulajdonságait:
  - Ablakpozíció: középen
  - Alapértelmezett szélesség és magasság: 300×50
  - Átméretezési fogantyú: Nem
- A *Szignálok* fülön a GtkObject destroy eseményére állítsuk be: `gtk_main_quit`
- Adjunk hozzá egy címkét, és nevezzük el `label`-nek.
- Az *Általános* fülön a Címke mező értéke legyen: Helló világ!

Mentsük le Glade fájlt a projektmappába. Most módosítsuk az eredeti „Helló világ!” programot! Az új forráskód a következő lesz:

```
using Gtk;

int main(string [] args) {
    Gtk.init(ref args);
    try {
        var builder = new Builder();
        builder.add_from_file("main.glade");
        builder.connect_signals(null);
        var window = builder.get_object("window") as Window;
        window.show_all();
        Gtk.main();
    } catch (Error e) {
        stderr.printf ("Betöltési hiba: %s\n", e.message);
        return 1;
    }
    return 0;
}
```

Az új programban példányosítjuk a `Builder` osztályt, majd betöltjük a Glade fájlt. A következő sor annyit tesz, hogy a szignálkezelést a rendszerre bízzuk. Ha a `null` helyett a `this` kulcsszót használnánk egy osztályban, az azt jelentené, hogy mi magunk akarjuk kezelni őket.

Ezután kinyerjük a főablak objektumot a `builder`-ből. Ez azért kellett, mert a következő sorban megjelenítjük az ablakot. Végül a GTK+ főciklust inicializáljuk. Ha esetleg hiba történt a betöltés során, akkor a dobott kivételt elkapjuk a `catch` blokkban, és kiírjuk a hibaüzeneteket. Ez a program egyébként ekvivalens az eredeti „Helló világ!” programmal.

A program lefordításához megint szükség van a Makefile módosítására, a Glade használata miatt. Az új Makefile a következőképpen néz ki:

```
PROGRAM = build/Main

build:
    mkdir build;
    valac --pkg gtk+-3.0 --pkg gmodule-2.0 Main.vala;
    mv Main build/Main

clean:
    rm -rf build
```

Ezzel már a program lefordítható, és ki is próbálható. A kész projekt megtekinthető a **GtkHelloGlade.zip** fájlban.

Innentől már tényleg elkezdhetünk dolgozni, és hasznos alkalmazásokat készíteni. A következő fejezetben bonyolultabb, komplexebb példaprogramokat fogunk készíteni.