Programozási nyelvek Java

Kitlei Róbert

Programozási nyelvek és Fordítóprogramok tanszék
ELTE Informatikai Kar

A program dinamikus szerkezete

- ***vezérlés*** vagy ***végrehajtás*** (control flow, execution): a program utasításainak végrehajtási sorrendje
  - ◇ ***belépési pont*** (entry point): a programkód olyan pontja, ahová a vezérlés kívülről odakerülhet
  - ◇ ***főprogram***: a teljes program belépési pontjához tartozó alprogram
    - ▸ Javában ez a ***main***
  - ◇ procedurális paradigma: a főprogram alprogramokat hív meg, az alprogramok további alprogramokat és így tovább
  - ◇ futási idejű fogalom
  - ◇ statikusan is lehet következtetéseket levonni róla
    - ▸ optimalizálás: különböző változókat közös helyen lehet tárolni, ha nem "zavarják egymást"
    - ▸ kód vizsgálata: a kód melyik részeire terjedhet el egy érték?
    - ▸ unreachable code: vannak-e elérhetetlen kódrészletek?
    - ▸ dead code: vannak-e olyan számítások, amelyeknek az eredményét nem használjuk?

Hívási lánc

- ***hívási lánc*** (call chain): egy adott időpillanatban a meghívott alprogramok sorozata a belépési ponttól kiindulva
  - ◇ amikor egy meghívott alprogram befejezi a futását, visszatér, és az eggyel kijjebb levő alprogram fut tovább
  - ◇ a teljes program akkor ér véget, amikor a legkülső alprogram (a főprogram) is véget ér
- ***aktív*** alprogram: a hívási lánc egy eleme (azaz még nem tért vissza)
  - ◇ a hívó ("külső") paramétereket ad át a hívott ("belső") alprogramnak
  - ◇ mindegyik alprogram lokális változókat tarthat nyilván a futása során
  - ◇ az aktív alprogramok közül a legbelső ***fut*** (utasításokat hajt végre)

Hívási lánc

- ***aktivációs rekord*** (activation record): az aktív alprogram paramétereit és lokális változóit tartalmazza
  - ◇ ***(futási idejű) verem*** (stack, call stack): a hívási lánc aktivációs rekordjainak leggyakrabban használt tárolási módja
    - ▸ új alprogram hívásakor bővül egy szinttel
    - ▸ alprogram befejeztével visszabomlik az utolsó szint
    - ▸ ... azaz tényleg veremként működik (LIFO - Last In, First Out)
    - ▸ a verem további szerepe: a kifejezések kiértékelése során átmenetileg itt tárolódnak a részkifejezések értékei
  - ◇ ***veremkeret*** (stack frame): a veremben tárolt aktivációs rekord
- ***rekurzív*** (recursive) alprogram: a hívási láncban többször szerepelhet
  - ◇ ***reentráns***: ugyanaz a kódrészlet több példányban fut
  - ◇ vagy önmagát hívja meg közvetlenül, vagy több alprogramon keresztül érhet vissza a hívás
  - ◇ ekkor az alprogramnak több ***futó példány***a van a hívási láncon
  - ◇ a program “elszáll” (crash) vagy “lefagy” (freeze), ha végtelen rekurzióba kerül

Memória

- ***memória*** vagy ***(operatív) tár***: bájtok sorozata
  - ◇ ***memóriacím*** (memory address): egy bájt sorszáma a memóriában
  - ◇ a verem a memóriában kap helyet
- ***allokáció***: példányosításkor az objektumhoz hozzárendelődik egy terület a memóriában (diszjunkt más objektumok területétől)
  - ◇ a terület kezdőcímét az objektum ***cím***ének nevezzük
  - ◇ a területen belül minden egyes példányváltozónak van helye
  - ◇ ***mutató*** (pointer): olyan változó, ami egy memóriacímet tartalmaz
    - ▸ ***nullmutató***: egy kitüntetett számérték (nem mindig 0)
    - ▸ mutató ***feloldása*** (***meghivatkozása***): a memóriacímének elérése
      - `null` feloldása mindig sikertelen (`NullPointerException`)
    - ▸ ***mutatóaritmetika*** (pointer arithmetic): a mutató memóriacímének egész számként való kezelése, vele műveletek végzése
      - veszélyes: könnyű érvénytelen helyre állítani a mutatót
  - ◇ ***hivatkozás*** vagy ***referencia*** (reference): olyan mutató, amely csak érvényes objektumra mutathat (vagy `null`)
    - ▸ Javában csak hivatkozások használhatóak, mutatóaritmetika nem

Heap

- ***heap***: a dinamikusan lefoglalt adatok tárterülete a memóriában
  - ◇ Javában az összes objektum ide kerül

```
void f() {
    Point p = new Point(); // kiértékelésének menete:
    ...
}
```

1. először a `new Point()` kifejezés értékelődik ki
   i. a futtató rendszer az objektumnak megfelelő méretű tárterületet allokál a heap-en
   ii. az objektum inicializálódik (erről később)
   iii. a kifejezés értéke az objektumra mutató hivatkozás
2. a deklarált `p` változónak helyet készít a rendszer a legbelső, `f` metódushoz tartozó aktivációs rekordon
3. erre a helyre bemásolódik a hivatkozás értéke (az objektum memóriacíme), innentől `p` az új objektumra hivatkozik

Élettartam

- ***élettartam*** (lifetime vagy extent): a futási időnek az a része, amíg egy érték (objektum) a memóriában megtalálható, elérhető, használható
  - ◇ a verembe kerülő értékek (lokális változók, formális paraméterek) csak addig élnek, amíg az aktivációs rekord létezik
    - ▸ tehát amint az őket tartalmazó alprogram kilép, elpusztulnak
    - ▸ technikailag lehet, hogy egy darabig megmaradnak a veremkeret lebontása után is, de akkor már társzemétként tekintünk rájuk
  - ◇ a heap-re kerülő objektumok addig élnek, amíg van rájuk hivatkozás máshonnan
    - ▸ pusztán attól, hogy véget ér az alprogram, amelynek futása során készültek, még megmaradhatnak
      - az alprogram visszatérhetett az objektumra való hivatkozással
      - a objektumra való hivatkozással feltölthettük egy másik objektum mezőjét

Szemétgyűjtés

- más nyelveken van lehetőség explicit ***felszabadítani*** (deallocate) a tárterületeket, Javában nincsen
- ***szemétgyűjtés*** (garbage collection, GC): a futtató rendszer időnként felderíti és felszabadítja az elérhetetlenné vált objektumokat
  - ◇ automatikus memóriakezelést biztosít: csak lefoglalni tudjuk az objektumokat
  - ◇ megjelenhet a nyelvbe beépítve (pl. Java)
    - ▸ ilyenkor általában minden objektumra hat
    - ▸ lehet (részlegesen) kikapcsolható, kikerülhető
  - ◇ egyes nyelvekhez kiegészítőként elérhető
- használatának oka: a kézi vezérlésű memóriakezelés bonyolult lehet
  - ◇ gyakran nehéz megszabni, mikor kell egy objektumot felszabadítani
    - ▸ nagy baj, ha felszabadítunk egy még használatban levő objektumot
  - ◇ könnyű elfeledkezni a deallokációról → memóriaszivárgás

Szemétgyűjtés: hivatkozásszámlálás

- ***(egyszerű) hivatkozásszámlálás*** (reference counting): mindegyik objektumról nyilvántartjuk, hány hivatkozás van rá
  - ◇ gyors módszer: csak az objektum számlálóját kell növelni/csökkenteni
    - ▸ $+1$: `new`, hivatkozás objektumra állítása
    - ▸ -1: hivatkozás átállítása az objektumról
  - ◇ nem elég jó: hivatkozási kör nem szabadul fel
- léteznek különféle javítások hozzá

Szemétgyűjtés: mark and sweep

- ***bejáró szemétgyűjtő*** (tracing GC): felkutatja az elérhető objektumokat, és a többit felszabadítja
  - ◇ ***gyökérkészlet*** (root set): olyan hivatkozások, amelyek még biztosan használatban vannak
    - ▸ ezek elsősorban a veremben tárolt hivatkozások
    - ▸ cél: csak az innen elérhető objektumok maradjanak meg
  - ◇ ***mark and sweep*** algoritmus működése:
    1. ***mark*** fázis: a gyökérkészlet elemeiből kiindulva bejárjuk a hivatkozások gráfját
       - ▸ megjelöljük az érintett objektumokat
    2. ***sweep*** fázis: azok az objektumok, amelyek a végére érintetlenek maradnak, elérhetetlenek, ezeket felszabadítjuk
- naiv implementáció: a program nem futhat közben (“stop the world”)
  - ◇ ha futhatna, átrajzolhatná a hivatkozási gráfot, és ezzel érvényteleníthetné az eddigi jelöléseket
  - ◇ továbbfejlesztések: inkrementális, párhuzamos, generációs, realtime

Hivatkozásfajták

- ***strong reference***: az eddig tárgyalt Java referenciák
  - ◇ ha egy objektum elérhető a gyökérkészletből, nem szabadítható fel
- ***soft reference*** és ***weak reference***: ha csak ilyen hivatkozások vannak egy objektumra, nincsen garancia arra, hogy a memóriában marad
  - ◇ a szemétgyűjtő eltávolíthatja, pl. ha fogyóban van a memória
  - ◇ a "weak" hamarabb szabadul fel, mint a "soft"

```
import java.lang.ref.WeakReference;

WeakReference<Data> cacheData = new WeakReference<Data>(data);
Data data = cacheData.get();
          // --------------- strong reference-t kapunk
          //                 vagy null-t, ha már nem érhető el
if (data != null) { /* még él, lehet használni */ }
```

Élettartam vége

- objektum élettartamának akkor van vége, amikor a szemétgyűjtő eltünteti a memóriából
  - ◇ arra nincsen befolyásunk, hogy ez mikor történik meg
- legtöbbször nem is kell tudomást vennünk róla és veszélyes, de ha mégis...
  - ◇ értesítést kaphatunk róla: meghívódik a `finalize` metódusa
    - ▸ ennek a metódusnak adható saját törzs
      - "újjá tud éleszteni" objektumokat: új hivatkozást állíthat már begyűjtésre ítélt objektumokra
      - "halott" objektumok metódusait hívhatja meg
  - ◇ másfajta értesülés: ***phantom reference***
    - ▸ nem használható a hivatkozott objektum elérésére
    - ▸ ha a szemétgyűjtő megjelölte begyűjtésre, bekerül egy `ReferenceQueue`-ba, ez mutatja, hogy felszabadult az objektum
    - ▸ rugalmasabb megközelítés a `finalize` használatánál
- mindkét megközelítés igen-igen ritkán használatos, kerülendő!

Alapfogalmak

- elnevezések
  1. ***egyed*** vagy ***entitás*** (entity): koncepcionálisan minden mástól elkülöníthető, az ábrázolás alapegysége
  2. ***típus*** (type): valamilyen szempontból hasonló egyedek összessége
     - ◇ a típus elképzelhető egyedek halmazaként
  3. ***altípus*** (subtype): egyedei beletartoznak egy másik típusba is
     - ◇ ***bázistípus*** (supertype): a "másik típus"
       - ▸ ***szülőtípus***: közvetlen bázistípus; másik irány: gyermek
     - ◇ halmazokkal: altípus egyedei $\subseteq$ őstípus egyedei
- ezeknek a programozási nyelvekbeli megfelelői
  1. ***objektum*** $\leftarrow$ egyed
  2. ***osztály*** $\leftarrow$ típus
  3. ***öröklődés*** vagy ***leszármazás*** (inheritance) $\leftarrow$ altípusosság
- ***objektum-orientált***nak (object oriented, OO) nevezhetünk egy nyelvet, ha megvan benne (1.), (1. és 2.) vagy mindhárom jellemző
  - ◇ a Java rendelkezik mindhárom jellemzővel

Egyedek/objektumok/értékek

- ***jellemző***k (property)
  - ◇ pl. egy adott ember neve Nagy Béla, magassága 180cm, kora 45 év
  - ◇ az, hogy mit tekintünk jellemzőnek, attól függ, hogy mennyire megyünk a részletekbe (milyen absztrakciós szinten dolgozunk)
    - ▸ pl. Kovács Béla minden egyes sejtjét ábrázolhatnánk
- ***művelet***ek (operation)
  - ◇ az egyedek jellemzőit a valóságban nem tudjuk közvetlenül megváltoztatni
  - ◇ az egyedeken különböző műveleteket lehet végrehajtani
    - ▸ ezek megváltoztathatják az egyedek jellemzőit
- ***invariáns*** (invariant): olyan feltétel, amely mindig teljesül egy egyedre
  - ◇ pl. se egy ember kora, se a magassága nem lehet negatív
  - ◇ több jellemzőből együttesen is adódhat
    - ▸ pl. egy négyzet mindegyik oldala egyforma hosszú

Egyedek/objektumok/értékek

- ábrázolás OO programozási nyelvekben
  - ◇ objektum ← egyed
  - ◇ adattag/mező ← jellemző
  - ◇ metódus ← művelet
  - ◇ invariáns: a legtöbb programozási nyelv jelenleg nem ad hozzá széleskörű támogatást
    - ▸ kutatási célú nyelvekben elérhetőek
      - pl. a fordítóprogram ellenőrizni/garantálni tudja, hogy egy lista mindig rendezve tartalmazza-e az elemeket
    - ▸ ***assert***: adott feltétel ellenőrzése futási időben
      - gyengébb eszköz, de sokkal könnyebben használható
- az objektumok a programban használható értékek
  - ◇ ... a primitív típusok mellett
  - ◇ kényelmesebb lenne, ha minden érték objektum lenne, de a primitív típusok sokkal hatékonyabbak

Típusok/osztályok

- ***osztálydefiníció*** (class definition)
  - ◇ bevezeti az osztály nevét
  - ◇ a `.java` forrásfájlok főként osztálydefiníciókat tartalmaznak

```
class Point { int x; int y; }
```

- ***példányosítás*** (instantiation): objektum létrehozása
  - ◇ a létrejövő objektum az osztály ***példány***a (instance)
  - ◇ ***példányváltozó*** (instance variable): az osztály összes adattagjához létrejön egy-egy változó, ami az objektumhoz tartozik
    - ▸ az adattag nevével lehet elérni
  - ◇ a példányosítás kifejezés mellékhatásos (létrejön egy objektum), visszatérési értéke az objektum hivatkozása

```
new Point()  // Point típusú objektumot létrehozó kifejezés
             // a zárójeleket kötelező kitenni
(new Point()).x // új pont létrehozása, az x adattag elérése
new Point().x   // zárójelezés nélkül is érvényes
```

Referenciák egyenlősége

- minden egyes példányosítás új, a korábbiaktól különböző objektumot készít el
  - ◇ referenciák között az == operátor azt vizsgálja, hogy a két hivatkozás ugyanoda mutat-e (azaz ugyanarról az objektumról van-e szó)

```
Point p  = new Point();   // (1)
Point p2 = new Point();   // (2)
System.out.println(p == p2); // false, (1) és (2) különbözik
System.out.println(new Point() == new Point());
        // false, változók nélkül, de ugyanaz, mint előbb

String s  = beolvas();    // tfh az "abc" szöveget      (3)
String s2 = beolvas();    // kapjuk kétszer             (4)
System.out.println(s == s2);  // false a fentiek miatt
System.out.println("abc" == "abc");
   // true! a String speciális, a fordító optimalizálja,
   //        és csak egy szöveg objektum keletkezik
System.out.println("abc" == new String("abc")); // false
```

Final minősítő

- primitív típusokra: ***immutable***: a változó értéke nem változtatható meg
  - ◇ konvenció: csupa nagybetűs változónév

```
final int MAX_SOROK_SZÁMA = 200;
MAX_SOROK_SZÁMA = 300;                    // fordítási hiba
```

- referenciatípusokra: az objektumhivatkozás nem állítható át
  - ◇ azaz a mutató immutable
  - ◇ .. maga az objektum viszont megváltozhat!

```
class C { int x; void setX(int x) { this.x = x; } }
final C c = new C();
c.x = 62;          // lefordul
c.setX(51);        // lefordul
c = new C();       // fordítási hiba
```

Példányváltozó inicializálása

- a példányváltozók nem maradhatnak kitöltetlenül; akkor is kapnak implicit értéket, ha nem írjuk le

```
class Data { boolean b; int x; double d; String s; }
```

- ugyanez kiírva az implicit kapott értékeket:

```
class Data {
    boolean b = false; int x = 0; double d = 0.0;
    String  s = null;   // nem "" a kezdőértéke,
                        // mert a String referenciatípus!
}
```

- nem célszerű az implicit értékekre hagyatkozni
  - ◇ ha kiírjuk az inicializációt, azzal jelezzük, hogy a szándékaink szerint kapta az implicittel megegyező értéket a példányváltozó, nem csupán elfelejtettük odaírni a megfelelő értéket

Osztályszintű változók

- ***osztályszintű*** (statikus) adattag: mindig pontosan egy példány van belőle, és ez magához az osztályhoz tartozik
  - ◇ nem objektumonként keletkezik belőle egy példány
  - ◇ létezik, mielőtt az osztálynak létezne akár egyetlen példánya is
  - ◇ a heap-en tárolódik

```
class C {
    static int si;
           int d;
}

C cobj = new C();   // létrejön (1) objektum
C.si;          // OK: statikus mező hivatkozható osztálynévvel
C.d;           // hibás: d példányszintű
cobj.si;       // OK: objektum segítségével is hivatkozható
cobj.d;        // OK: a cobj (vagyis (1)) d-jét érjük el
```

Osztályszintű változó kezdőértéke

- akkor kap értéket, amikor az osztály betöltődik
  - ◇ mindenféleképpen az első használat előtt
  - ◇ implicit értéket kap a példányszintű mezőkhöz hasonlóan, ha nincsen kitöltve
- a kódban feljebb szereplő `static` változó értékét felhasználhatja
  - ◇ a példányszintű inicializáció felhasználhat static változót

```
class C {
    // maxIdx később lesz deklarálva, de használható itt
    final int IDX = ++maxIdx;              // példányszintű

    static final int START  = 12;          // osztályszintű
    static       int maxIdx = START;       // osztályszintű
}
```

Inicializáló blokk

- ***inicializáló blokk***: osztályba írt blokk, példány inicializálásakor fut le
  - ◇ példányváltozó(k) kezdőértékét határozza meg hosszabb számítással
  - ◇ a példányváltozók és a blokkok sorrendje számít
  - ◇ ***statikus inicializáló blokk***: hasonló eszköz osztályszintű változókhoz

```
class C {
    static final int    VSN = .....;
    static final String INIT_NAME;

    static { if (VSN >= 3)    INIT_NAME = "VSN3";
             else             INIT_NAME = "VSN_OLD";  }

    final List<String> names = ......;
    final String name;

    {   if (names.contains("X"))    name = INIT_NAME + "X";
        else                        name = INIT_NAME;       }
}
```

Konstruktor

- ***konstruktor***: az objektum létrehozásakor lefutó kódrészlet
  - ◇ inicializálja az objektumot
    - ▸ az inicializáció célja: az invariánsok beállítása
      - más szóval: érvényes objektum készítése
    - ▸ ennek módja: a konstruktor beállítja az objektum példányváltozóit
  - ◇ neve megegyezik az osztály nevével

```
class Point {
    int x;
    int y;

    Point(int x, int y) { this.x = x; this.y = y; }
}
```

Konstruktor

- ha nem írunk mi magunk, ***alapértelmezett*** (default) konstruktor készül
  - ◇ nincsenek paraméterei
  - ◇ a törzse (majdnem) üres
  - ◇ láthatósága megegyezik az osztályáéval

```
class Point {
    // ilyen konstruktor készül
    // ha nem írunk kézzel
    Point() {
        /* "majdnem" üres */
    }
}
```

```
public class Point {

    // itt pedig ilyen
    public Point() {
        /* "majdnem" üres */
    }
}
```

- ha legalább egy konstruktort megírtunk, nem készül alapértelmezett
  - ◇ a kézzel írott konstruktorok között lehet paraméter nélküli is

```
public class Point { public Point(int x) { ... } }

new Point();  // fordítási hiba, most nem készült ilyen!
```

Konstruktor

```
class Point {
    // ha ez az egyetlen konstruktor,
    // kívülről az osztály nem példányosítható közvetlenül
    private Point() { }

    static Point createPoint() { // csak egy
        return new Point();      // metódus segítségével,
    }                            // aminek van joga elérni
}
```

- az osztály neve metódusnak is adható, bár ez nem jó ötlet

```
class Point {
    // figyelem: ez itt nem konstruktor!
    public void Point() { }
        // ---- emiatt
}
```

Konstruktor

- konstruktor meghívhat másik konstruktort
  - ez a `this` kulcsszó egy speciális felhasználása
  - csak a konstruktor törzsének elejére kerülhet, csak egyszer

```
public class Point {
    int x;
    int y;

    public Point(int x, int y) { this.x = x; this.y = y; }
    public Point()             { this(0, 0); }
}
```

- ***túlterhelés*** (overloading): egy névhez több különböző jelentés is tartozik
  - metódushíváskor megkülönböztetés: a paraméterek száma és típusai alapján

```
new Point();        // az alsót hívja
new Point(3, 5);    // a felsőt hívja
```

Paraméterátadás

- paraméterátadási (kiértékelési) stratégiák metódus/konstruktorhíváskor
  - ◇ ***érték szerinti*** paraméterátadás (call-by-value): híváskor a formális paraméter felveszi az aktuális paraméter értékét
  - ◇ ***megosztás szerinti*** paraméterátadás (call-by-sharing): az aktuális paraméter egy mutató/hivatkozás, ezt veszi fel a formális paraméter
    - ▸ ez alias-t létesít a hivatkozott objektumra
- Java: primitív típusokat érték, referenciákat megosztás szerint ad át

```
class Point {
    int x;
    int y;

    Point(int x, int y) { this.x = x; this.y = y; }
       // -----  -----  érték szerint kapja meg
    Point(Point p)      { this(p.x, p.y); }
                                  // ---  --- érték szerint adja át
       // ------- megosztás szerint kapja meg
}
```

Enkapszuláció

- ***egységbe zárás***, ***enkapszuláció***: az objektum felelős a belső állapotáért
  - ◇ cél: ne lehessen elrontani az invariánsokat
    - ▸ az állapotot kívülről csak műveletek hívásával lehessen megváltoztatni
  - ◇ az élettartam során csak saját kód kezeli a saját adatokat
    1. létrehozáskor konstruktorral gondoskodunk az invariánsok beállításáról
    2. mivel a belső állapot csak műveletek meghívásával változhat meg, a műveleteknek van egy "szerződése"
       a. a művelet feltételezheti, hogy az invariánsok fennállnak, amikor meghívják
       b. kilépésre olyan állapotba kell hoznia az objektumot, ahol fennállnak az invariánsok
- mi törheti meg az enkapszulációt?
  - ◇ ha az adattagok kívülről hozzáférhetők (publikusak)
  - ◇ ha az objektum "kiadja" a reprezentációjának egy részét

Enkapszuláció megsértése

```
class Point {  private int[] coords = { 0, 0 };
               int  getX()      { return coords[0]; }
               void setX(int x) { coords[0] = x;    } }
```

- az interfészt jól kell kitalálni
  - ◇ mi a helyzet, ha a másfajta gettert/settert használunk?

```
class PointN {
    private int[] coords;
    PointN(int[] coords)    { this.coords = coords; } // !!!
    int[] get()             { return coords;        } // !!!
    void  set(int[] coords) { this.coords = coords; } // !!!
}

int[] coords = { 1, 2, 3, 4, 5 };
PointN p = new PointN(coords);
coords[2] = -7;                    // rossz: p "private"
System.out.println(p.get()[2]); // mezője is ide hivatkozik!
```

Enkapszuláció megtartása

- az objektum ne adja ki a belült kezelt referenciákat
  - ne is vegyen át olyan adatot a reprezentációjába, ami kívülről elérhető

```
class PointN {
    private int[] coords;
    PointN(int[] coords)    { set(coords);                 }
    int[] get()             { return coords.clone();       }
    void  set(int[] coords) { this.coords = coords.clone(); }
}
```

- ***shallow copy***: csak a legkülső hivatkozások válnak külön
  - a clone művelet ilyen másolatot készít a tömbökről
  - ez néha nem elég, a belső hivatkozásokat is másolni kell: ***deep copy***

```
Point[][] ptss  = { { new Point(1, 2) } };
Point[][] ptss2 = ptss.clone();   // ptss[0]   != ptss2[0]
ptss[0][0].x = 100;               // ptss[0][0]== ptss2[0][0]
System.out.println(ptss2[0][0].x);// 100, a shallow cp miatt
```

Metódus lokális változói

- ha létrehozunk egy objektumot, azt a későbbiekben általában használni is szeretnénk, ehhez egy ***lokális változó***t (local variable) használhatunk
  - ◇ természetesen egy másik objektum példányváltozóját is ráállíthatjuk
  - ◇ ***inicializáció***: ***kezdeti értékadás***
    - itt metódus lokális változóival kapcsolatban vizsgáljuk
    - objektumok példányváltozóinak inicializációját lásd később

```
Point p;     // deklarációs utasítás inicializáció nélkül
Point p2 = new Point();   // deklaráció inicializációval (1)
int    x  = p.x;      // fordítási hiba, p-nek nincsen értéke
p2    = new Point(); // értékadás, (1) pont elérhetetlen  (2)
p     = new Point(); // értékadó utasítás                 (3)
p2    = p;           // (2) is elérhetetlen,
                     // p és p2 is a (3) pontra hivatkozik
p2.x = 5;           // (3) x adattagja az 5 értéket veszi fel
p.y  = p2.x;        // (3) x és y adattagja egyaránt 5 lesz
```

Változó inicializálása

- metódus lokális változóit kötelező inicializálni használat előtt

```
Point p;          // ez egy lokális változó egy metódusban
int   v = p.x;    // fordítási hiba, p nem kapott kezdőértéket
```

- az inicializáció történhet a deklaráció után
  - ◇ ha ennek nincsen különös oka, jobb, ha a deklarációnál történik

```
Point p; // p értéke feltételtől függ, itt még nem ismert
if (feltétel) { p = new Point(1,2); }
else          { p = new Point(3,4); }
int v = p.x;  // garantált, hogy p-nek itt már van értéke
```

- a fordítóprogram ellenőrzi, hogy minden ágon kapott-e értéket a változó
  - ◇ bonyolult feltételekről nem tudja belátni, hogy mindig teljesülnek-e
  - ◇ elméletileg sem megoldható (eldöntési probléma)

```
Point p;
if (mindigTrue()) { p = new Point(1,2); }
int v = p.x;  // a p változó lehet, hogy nem inicializált
```

Változó inicializálása

- szóba jöhet még: változó null-ra inicializálása

```
Point p = null; // legtöbbször rossz ötlet így inicializálni
int   v = p.x;  // NullPointerException jön futási időben
```

- `null`: feltalálója (Tony Hoare) szerint: "billion-dollar mistake"
  - ◇ minden változó felveheti a `null`-t
  - ◇ azaz bármilyen olyan helyen megjelenhet, ahol érvényes objektumra számítanánk
    - ▸ igen messzire el tud terjedni
    - ▸ amikor előjön egy ponton, nehéz lehet megtalálni, honnan származik
  - ◇ nem kötelező ellenőrizni feloldás előtt, hogy `null`-t használunk-e
  - ◇ nagyon veszélyes, kerülendő!
- egyre több nyelvben ki van kényszerítve a használat előtti ellenőrzés
  - ◇ Java 8: `T` referenciatípushoz `java.util.Optional<T>`

Deklaráció

- ***deklaráció***: nevet adunk valaminek (pl. változó, metódus, osztály)
- ***hatókör*/*láthatóság*** (scope): azon kódrész, ahol a deklaráció érvényes
  - ◇ szinte mindig fordítási idejű (statikus) fogalom (lexical scoping)
    - ▸ … néhol nem az (pl. Lisp, Perl), ami sokkal nehezebben követhető
  - ◇ ***elfedés*** (hiding, masking): hatókörön belül újradeklarálunk egy nevet
    - ▸ a név a belső jelentésében hat, a külsőhöz minősített név szükséges
    - ▸ ***közvetlen láthatóság*** (visibility): a név nincsen elfedve

```
class C {
    int x;                  // (1)
    void f(int x) {  // (2)
        System.out.println(x);       // (2)-re hivatkozik
        System.out.println(this.x); // (1)-re hivatkozik
    }
    boolean g() { return  x == this.x; }  // mindig true
                        //  ---   ------ nincsen elfedés,
                        //                ugyanarra hivatkoznak
}
```

Hatókör: blokk

- több utasítás egybefoglalása
- a benne deklarált változók használatának korlátozása
- legjellemzőbb használata: elágazások ágaként és ciklusok magjaként
  - ◇ a metódusok törzse technikailag nem blokk utasítás (bár azt is kapcsos zárójel veszi körbe)
- változók láthatóságát is lehet vele korlátozni

```
{
    int tízSzámÖsszege = 0;
    for (int i = 0; i < 10; ++i) {
        tízSzámÖsszege += számotBeolvas();
    }
    // tízSzámÖsszege itt használható
}

// tízSzámÖsszege itt már nem használható
```

Hatókör: elfedés

- Java: lokális változó (formális paraméter is) elfedheti adattag nevét
  - ◇ ... de nem fedhet el másik lokális változót

```
class C {
    int x;              // (1)
    void f(int x) {  // (2)
        int x; // érvénytelen: (2) ezzel azonos nevű
               //                lokális változó

        int y; // (3)
        {
            int y; // érvénytelen (3) miatt
        }
    }
}
```

Hatókör: elfedés: getter/setter

- a `setter` metódusokban sokszor használunk elfedést

```
class Point {
    private int x;
    private int y;

    public int getX() { return /*this.*/x; }
    public int getY() { return /*this.*/y; }

    public void setX(int x) {
                  // ----- a formális paraméter neve is
                  //       megegyezhet adattagéval
               this.x = x;
             // ------      a példányváltozóra hivatkozik
             //         --- a formális paraméterre hivatkozik
    }
}
```

Láthatósági minősítő

- ***láthatósági minősítő*** vagy ***hozzáférési módosító*** (access modifier): (osztály, adattag, metódus, konstruktor) nevének hatókörét szűkíti
  - ◇ `public`: ezek összessége alkotja az objektum ***interfész***ét: ezek a szolgáltatások elérhetőek az objektumon a külvilág számára

```
package p1;
public class A {
    private   int priv;
              int pkg;  // félnyilvános (package private)
    protected int prot; // igen ritkán használjuk
    public    int publ;
}
class B { A a;    int f(){return a.prv+a.pkg+a.prt+a.pub; }}
                                  // XXXXX
==========================================================
package p2;                       // XXXXX XXXXX
class C extends A{int f(){return    prv+  pkg+  prt+  pub; }}
class D { A a;    int f(){return a.prv+a.pkg+a.prt+a.pub; }}
                                  // XXXXX XXXXX XXXXX
```

Láthatósági minősítő: osztályok

```
package p1;
public class A { }   // publikus osztály
class B {}   // félnyilvános osztály
class Elérem {
    void f() {
        new A(); // OK
        new B(); // OK
    }
}
```

```
==========================================================
```

```
package p2;
class CsakAPublicotÉremEl {
    int f() {
        new A(); // OK
        new B(); // fordítási hiba:
                 // error: B is not public in p1;
                 // cannot be accessed from outside package
    }
}
```

Metódus részei

- metódusdeklaráció

```
class Point { int x; int y;
              // törzs (body) ---------------------
    void move(int dx, int dy) { x += dx; y += dy; }
 //       ---- ---      ---     szignatúra (signature)
 //            --------------    formális paraméterlista
 // ----                         visszatérési érték típusa
 //      ----                    művelet neve (azonosítója)
 // ------------------------- fejléc (header)
}
```

- ***példánymetódus***: a példányon hat
  - ◇ a példányváltozókkal ellentétben nem a példányokban tároljuk őket

```
Point p = new Point(); // (1)
p.move(3, 5*4+8); // "move" példánymetódus meghívása (1)-re
   //  -------- a konkrét hívás aktuális paraméterlistája
      // a paraméter kifejezések kiértékelődnek,
      // ezek az értékek adódnak át a formális paraméterekbe
```

Metódus: this

```
class Point {
    void move(int dx, int dy) {
        x += dx;
        y += dy;
    }

    void move(/* Point this, */ int dx, int dy) {
            // ---------------- implicit formális paraméter
        this.x += dx;
        this.y += dy;
     // ------          az adattagokra hivatkozik
     //            ---- a formális paraméterre hivatkozik
    }
}
```

- a `this` minden ***példányszintű*** metódus implicit paramétere

```
Point p = new Point();  // létrejön (1), p rá hivatkozik
p.move(3, 5);           // a hívásban this (1)-re hivatkozik
```

Metódus: nevek használata, láncolás

```
class Point {
    int getX() { return x; }
    int getY() { return y; }

    void setX(int x) {
            // ----- a formális paraméter neve
            //       megegyezhet adattagéval
        this.x = x;
     // ------       a példányváltozóra hivatkozik
     //         --- a formális paraméterre hivatkozik
    }

    Point setY(int y) { this.y = y; return this; }
 // -----                           ----------
 // ha ilyen stílusban írjuk a kódot,
 // a hívások láncolhatóak (method chaining):
 //     p.setX(3).setY(-2).move(11, 5);
}
```

Metódus: visszatérési érték

- minden metódushoz fel kell tüntetni a visszatérési értékének a típusát
  - ◇ ha függvény (tiszta alprogram), akkor ez valódi típus
  - ◇ ha eljárás, akkor nem célja értéket kiszámítani, ezért nincsen visszatérési értéke, amit a ***void*** kulcsszó jelöl
- a formális paraméterek a metódus futása során lokális változók benne

```
class C {
    void f(int x) { if (x>5) return; System.out.println(x); }
                            // ------ void metódus return-je
                            //        csak önmagában állhat

    int g(int a, int b) {
        return a*a+b/2;
            // ------- nem-void visszatérési érték esetén
            //         ide kötelező (megfelelő típusú)
            //         kifejezést írni
    }
}
```

Metódus: osztályszintű metódus

- az osztályszintű (`static`) metódusnak nincsen implicit `this` paramétere

```
class C {
    static void f(int n) { ... }
}
```

- meghívható objektumra (pl. változón keresztül) is és az osztály nevén is

```
new C().f(1);
C.f(-9);
```

- statikus metódusból csak másik statikus metódus hívható
  - ◇ ... hacsak nincsen kéznél egy objektumhivatkozás
  - ◇ legismertebb `static` metódus: `main`

Öröklődés

- ***öröklődés***, ***származtatás*** (inheritance)

```
class A {
    int  a;
    void f(char c) { ... }
}

class B extends A {
    int b;
    int g(String s) { ... }
}

class C extends B { ... }
```

- A a ***szülőosztály***a B-nek (B-nek nem lehet más szülője)
- B egy ***gyermekosztály***a A-nak (A-nak lehetnek más gyermekei)
  - ◇ A és B ***bázisosztály***ai (***ősosztály***ai) C-nek
  - ◇ B és C ***alosztály***ai (***leszármazott***jai) A-nak

Öröklődési reláció

- öröklődés célja
  - ◇ a metódusok kódjának újrafelhasználása, és ezzel
  - ◇ a kód redundanciájának csökkentése
- öröklődési reláció: parciális rendezés az osztályok között
  - ◇ irányított gráfot képez
    - ▸ egy osztály nem lehet önmaga őse, ezért a gráf irányított kört nem tartalmaz
    - ▸ Javában az osztályok között nincsen többszörös öröklődés, ezért a gráf erdő
    - ▸ minden osztálynak őse a `java.lang.Object` osztály, ezért a gráf fa
    - ▸ összefoglalva: ***irányított fa***

Öröklődési reláció: miért egyszeres (osztályok között)?

- ha lehetne többszörös öröklődés osztályok között, előfordulhatna az alábbi helyzet

```
class A                  { void f() { /*  A-beli törzs */ } }
class B1 extends A       { void f() { /* B1-beli törzs */ } }
class B2 extends A       { void f() { /* B2-beli törzs */ } }
class C  extends B1, B2  { void f() { /* ??????????????? */ } }
```

- `B1` és `B2` is új törzset ad `f`-nek
- ***diamond inheritance problem*** (rombusz/káró öröklés): milyen törzzsel rendelkezzen `C`?
  - ◇ meg kell feleljen mind `B1`, mind `B2`-nek (lásd: helyettesítési elv)
- a Java tervezői úgy döntöttek, hogy metódustörzseket és változókat ne lehessen több forrásból örökölni

Öröklődés: mezők, metódusok

- mi öröklődik?
  - ◇ a mezők
  - ◇ a metódusok (a fejléceik és a törzsük egyaránt)

```
class A /* extends Object */ {      // ha nem írjuk ki,
    public  int    val;              // az Object a szülő
    private String txt;

    String f() { return txt + val; }
}

class B extends A { }
```

- az öröklött mezők/metódusok nem feltétlenül érhetőek el

```
B b = new B();
System.out.println(b.val);  // működik
System.out.println(b.txt);  // txt nem hozzáférhető
System.out.println(b.f());  // működik
```

Öröklődés: konstruktorok

- a gyermek konstruktorának kötelező áthívnia a szülőére

```
class A {
    public A(int val, String txt) { ... }
}

class B extends A {
    public B(String txt) {
        super(8, txt); // csak konstruktor törzsének
        ...            // legelején állhat
    }

    public B() {
        this("abc");   // áthív a másik konstruktorra,
    }                  // azon keresztül hívja a szülőét
}
```

Öröklődés: konstruktorok

- a konstruktorok nem öröklődnek
  - ◇ ha a szülőéhez hasonló konstruktorokra van szükségünk, azokat újra meg kell írni

```
class A {
    public A(char c) { ... }
}

class B1 extends A { }
class B2 extends A {
    public B2(char c) { super(c); ... }
}


new B1('x');  // B1-nek nincsen ilyen konstruktora
new B2('o');  // működik
```

Öröklődés: konstruktor: `super`

- ha egy konstruktor elején nem szerepel sem `this`, sem `super`, akkor automatikusan `super()` generálódik
  - ◇ fordítási hiba, ha a szülőnek nincsen paraméter nélküli konstruktora

```
class A {
    // tfh nem írtunk paraméter nélküli konstruktort
    // alapértelmezett sem generálódik(ami paraméter nélküli)
    public A(int val, String txt) { ... }
}

class B extends A {
    public B(String txt) {
     // super();     // implicit generálódik
        ...          // par. nélkülit hívna, de nincsen A-ban
    }                // fordítási hiba
}

class B2 extends A { /* B2() { super(); } */ }
                     // ---------------- ugyanúgy nem jó
```

Öröklődés: helytelen használat

- az öröklődés technikailag megengedi, hogy pusztán a bázisosztály mezőkkel és metódusokkal való kiterjesztésére használjuk

```
class XNégyzet {
    int aOldalHossza;
}
```

```
class XTéglalap extends XNégyzet {
    int bOldalHossza;
}
```

- az öröklődés ***altípusosság***ot (subtyping) indukál: a leszármazott használható minden olyan helyen, ahol a bázistípus
  - ◇ ha `class B extends A`, akkor $B \subseteq A$-ként tekintünk rájuk

```
// a fenti osztályok ezért rosszak: négyzetként kezelhetünk
XNégyzet n = new XTéglalap(); // egy (általános) téglalapot
```

```
class JóTéglalap {
    int aOldalHossza;
    int bOldalHossza;
}
```

```
class JóNégyzet extends JóTéglalap {
    // ezzel viszont felesleges
    // tárhelyet kell foglalnunk
}
```

Helyettesítési elv

- ***Liskov-féle helyettesítési elv*** (Liskov substitution principle, 1987): egy altípusra mindig álljon fenn az őstípus minden tulajdonsága
  - ◇ az altípus minden ponton helyettesíthesse az őstípust
  - ◇ az öröklődés (ha van a nyelvben) egy természetesen adódó eszköz erre
- egy másik lehetséges megközelítés: ***aggregáció*** (tagként felvétel)
  - ◇ ***kompozíció***: olyan aggregáció, ahol a tartalmazó élettartamának végén a tartalmazott élettartama is befejeződik
    - ▸ néha ez lehet valóban a megoldás (Javában is)
  - ◇ az elv betartásához duplikálni kell az őstípus interfészét az altípusban

```
class Ember    {                        public void f() {......} }
class Hallgato { private Ember e; public void f() {e.f();} }
```

- ***duck typing***: "if it walks like a duck and quacks like a duck then it is a duck"
  - ◇ futási időben vizsgálja a szükséges mezők/metódusok meglétét
  - ◇ Javában megoldható, de sokkal célszerűbb az öröklődést használni

Polimorfizmus

- ***polimorfizmus*** (többalakúság): különböző adattípusok közös kezelése
  - ◇ az öröklődés ennek egy speciális esete: ***altípusos polimorfizmus***

```
class A {}
class B extends A {}

class C {
    public static void f(A a){ ... }
    public static A    g()   { return new B(); }
                // ---                    ------- érvényes
                // de kifelé már A statikus típusúnak látszik
}

C.f(new B());  // meghívható, pedig az "a" formális paraméter
               // statikus típusa A
A a = C.g();   // működik
B b = C.g();   // nem jó, a kif. statikus típusa A; megoldás:
B b = (B)C.g(); // downcast: konverzió a hierarchiában lefele
```

Statikus és dinamikus típus

- ***statikus típus***: a változó fordítási időben ismert típusa
- ***dinamikus típus***: a változóhoz kötődő érték típusa
  - ◇ csak futási időben ismert biztosan
  - ◇ időben változhat, mert különböző értékekre hivatkozhat a változó
  - ◇ mindig a statikus típus altípusa

```
List<A> as = new ArrayList<>();
Random rnd = new java.util.Random();
for (int i = 0; i < 100; ++i) {
    A a;

    if (rnd.nextBoolean()) a = new B1(); // tfh B1 extends A
    else                   a = new B2(); // tfh B2 extends A

    // "a" dinamikus típusa itt vagy B1, vagy B2
    as.add(a);
}
```

Upcast, downcast

- ***upcast***: konverzió az öröklődési hierarchiában felfele (alosztályról bázisosztályra) futási időben
  - ◇ mindig megengedett
  - ◇ csak egymásból származó típusok között megengedett
  - ◇ nem alakítja át az objektum reprezentációját
- ***downcast***: futási idejű konverzió bázisosztályról alosztályra
  - ◇ futási időben megvizsgálja, a dinamikus típus megfelelő-e

```
class A           { int a; }
class B extends A { int b; }

new B().a       = 1;   // valójában a következő történik:
((A)new B()).a  = 1;   // upcast, OK

((B)new A()).a  = 1;   // downcast, runtime ClassCastException
((Object)new B()).a = 1; // fordítási hiba
                       // az Object-nek nincsen "a" mezője
```

Az `instanceof` operátor

- ***instanceof***: típusvizsgáló operátor
  - ◇ csupa kisbetűvel írandó!
  - ◇ jobb op: egy típus neve
  - ◇ bal op: kifejezés
  - ◇ értéke igaz, ha a kifejezés dinamikus típusa altípusa a jobb oldalinak

```
class A             { int a; }
class B extends A { int b; }

null instanceof T       // mindig hamis, bármi is T
new A() instanceof A    // true
new B() instanceof A    // true
new A() instanceof B    // false
"abc"   instanceof A    // fordítási hiba:  incompatible types:
                          // String cannot be converted to A
A a = new B();
a instanceof B          // true (nem "a" statikus típusát nézi)
```

Az `instanceof` operátor

- az `instanceof`-ot általában ritkán használjuk kézzel

```
class A { void doSomethingA() { ... } }
class B { void doSomethingB() { ... } }

Object o = ...;
if      (o instanceof A)   ((A)o).doSomethingA();
else if (o instanceof B)   ((B)o).doSomethingB();
```

- ehelyett jobban illeszkedik az objektum-orientáltsághoz, ha közös ősosztályt veszünk fel (ha megtehető)

```
class Base            { void do() { ... } }
class A extends Base { void do() { ... } } // felüldefiniálás
class B extends Base { void do() { ... } } // (override)

Object o = ...;
((Base)o).do();
```

Felüldefiniálás

```
// segédtípusok a felüldefiniálás példáihoz
class T1 {}
class T2 extends T1 {}

// ennek az osztálynak az f metódusát fogjuk felüldefiniálni
class Base { T1 f(T1 t) { return t; }
```

- ***felüldefiniálás*** (overriding): a metódus a leszármazottban új törzset kap
- a láthatóság félnyilvánosról teljesen nyilvánosra változhat

```
class Ext extends Base { public T1 f(T1 t) { return t; } }
                        // ------ félnyilvános -> nyilvános
```

Felüldefiniálás

- felüldefiniálás során a paraméterek típusa nem változhat (altípusra sem)
  - ◇ ha fel van tüntetve az `@Override` annotáció, a fordító hibajelzést ad

```
class Ext extends Base {
    @Override
    public Type1 f(Type2 t) { return t; }}
                // ----- fordítási hiba
```

- annotáció nélkül sem helytelen a kód, de mást jelent
  - ◇ így túlterheljük a metódust; általában nem ez a célunk

```
class Ext extends Base {
    public T1 f(T2 t) { return t; }}
              // -- @Override nélkül érvényes, viszont...

new Ext().f(new T1()) instanceof T2)  // == false
       // ------------ Base.f -et hívja
new Ext().f(new T2()) instanceof T2)  // == true
       // ------------ Ext.f -et hívja
```

Felüldefiniálás

- a visszatérés típusa szűkülhet altípusra
  - ez egy **kovariáns** változás: ahogy a típusunk szűkül (`Base`→`Ext`, azaz ős→leszármazott), ugyanolyan módon változik a visszatérési érték típusa is

```
class Ext extends Base {
    @Override
    public T2 f(T1 t) { return new T2(); }
        // -- megengedett ez   -------- és ez is
}
```

- a paraméterek típusáról megállapítottuk, hogy **invariáns**
  - más nyelvekben előfordul **kontravariáns** változás is

Felüldefiniálás

- a `super` kulcsszó segítségével az ős műveletét/változóját/konstruktorát érjük el
  - ◇ `((Base)this)` nem helyettesítheti

```
class Base { T1 f(T1 t) { return t; }
class Ext extends Base {
    T1 f(T1 t) { return t == null ? new T2() : super.f(t); }}

Base b  = new Base();
Ext  e  = new Ext();
Base b2 = new Ext();
T1   t1 = new T1();
b.f(t1);     // == t1
b.f(null);   // == null
e.f(t1);     // == t1
e.f(null);   // egy új T2 példány
b2.f(t1);    // == t1
b2.f(null); // egy új T2 példány; lásd következő oldal
```

Dinamikus kötés

- ***dinamikus kötés*** (dynamic binding, late binding): példánymetódus hívásakor a dinamikus típus (a példány típusa) határozza meg, melyik osztályban megadott törzs fut le
  - ◇ a futtató rendszer felfele keres a bázistípusokon
    - ▸ a legközelebbi definíciót választja
  - ◇ némi extra költséggel jár futási időben
  - ◇ a művelet meghívhatóságát a fordítóprogram ellenőrzi még fordítási időben a statikus típusok alapján ⇒ garantáltan lesz meghívható törzs

```
class A           { void f() { ... } } // (1)
class B extends A {                    }
class C extends B { void f() { ... } } // (2)
class D extends C {                    }

new A().f();    // keresési sorrend: A    -> (1)-et hívja
new B().f();    // keresési sorrend: B, A -> (1)-et hívja
new C().f();    // keresési sorrend: C    -> (2)-t hívja
new D().f();    // keresési sorrend: D, C -> (2)-t hívja
```

Új törzs: statikus metódusok

- az osztályszintű metódusokat nem lehet felüldefiniálni
  - ◇ leszármazott statikus metódusa ***elfedi*** a hasonló bázisosztálybelit
  - ◇ nincsen dinamikus kötés közöttük, fordítási időben dől el, mit hívunk
  - ◇ általában sokkal jobb ötlet új nevet adni az új metódusnak

```
class A           {static void f(){System.out.println("A");}}
class B extends A {}
class C extends B {static void f(){System.out.println("C");}}

A.f(); // -> A
B.f(); // -> A, mert a (statikus) hierarchiában legközelebb
C.f(); // -> C                                ez található felfelé

A a = new C();
a.f();      // -> A, mert "a" statikus típusa alapján dől el
((C)a).f(); // -> C, mert a kif. statikus típusa most C
```

Új törzs: adattagok

- a változókat (akár példányszintűek, akár osztályszintűek) is csak ***elfedni*** lehet
  - ◇ a régi és az új változó egyszerre van jelen a példányban/osztályban
  - ◇ statikus kötéssel dől el, melyiket érjük el hozzáféréskor
  - ◇ általában sokkal jobb ötlet új nevet adni az új változónak

```
class A           { char x='A'; char a()  {return x;} }
class B extends A { char x='B'; char b()  {return x;}
                                char bS() {return super.x;}}

A a  = new A();
A ab = new B();
B b  = new B();
System.out.printf("%c %c %c %c %c %c %c %c %c %c%n",
   a.x,ab.x,b.x,((A)b).x,a.a(),ab.a(),b.a(),b.b(),b.bS());
//   A    A   B        A   A      A     A     B     A
```

A `final` módosító osztályra és metódusra

- `final` osztályból nem származtathatunk le

```
final class FinalClass { ... }

class C2 extends FinalClass { ... } // helytelen
```

- `final` metódust nem definiálhatunk felül

```
class BaseClass {
    final void f() { ... }
}

class C2 extends BaseClass {
    final void f() { /* új törzs */ } // helytelen
}
```

Interfészek

- ***interfész*** (interface): legalább kétfajta jelentésben használjuk

1. osztály interfésze: a rajta kívülről elérhető szolgáltatások összessége
   - leggyakrabban: a publikus metódusai
2. nyelvi elemként: az osztályhoz hasonló referenciatípus
   - csak publikus hozzáférésű metódusfejléceket tartalmazhat
     - ezek csak "ígéretek", törzs nem tartozik hozzájuk
   - ... és `public static final` minősítővel ellátott adattagokat
   - ... és néhány további lehetőséget (statikus metódusok, default metódusok)

```
interface IFace {
    public int f(int n);
           int f2(int n);// explicit jelölés nélkül is public

    static final int DATE = 19700101;
}
```

Interfészek: megvalósítás

- az osztályok ***megvalósít***hatnak (***implementál***hatnak) egy vagy több interfészt
  - ◇ minden megígért művelethez törzset kell adniuk
  - ◇ az interfészek többszörös öröklődése nem probléma: nincsenek törzsek, amelyek ütközhetnének

```
interface F1 { public double f(int n); }
interface F2 { public double f(int k); }
                                // ↓↓↓

class Impl implements F1, F2 {
    public double f(int z) { return 2 * z + 1; }
                    // --- a paraméterek nevei eltérhetnek
}

F1 f1  = new F1();   // HIBA: interfész nem példányosítható
F1 f1b = new Impl(); // csak az implementáló osztály
System.out.println(f1b.f(9)); // F1 metódusai meghívhatóak
```

Interfészek: használat

- az alábbiakban `Impl2` ***megvalósítja*** `F`-et

```
interface F           { public double f(int n); }
interface G extends F { public void   g(F f);   }

class Impl implements G {
    public double f(int z) { ... }
    public void   g(F f)   { System.out.println(f); }
}

class Impl2 extends Impl { }


// ---------- Impl -> G, tehát elérhető rá a g() metódus
   new Impl().g(new Impl2());
              // ----------- Impl2 -> Impl -> G -> F
              //             így használható g() paraméterekén
```

Interfészek: öröklődés

- az interfészek között van többszörös öröklődés
  - ◇ ütközés általában nincsen (a törzsnélküliség miatt)

```
interface F1 { double f(int n); void g(); }
interface F2 { int    f();      void g(); }

interface F3 extends F1, F2 { public String f(String s); }

class Impl implements F3 {
    public double f(int n)    { ... }
    public int    f()         { ... }
    public String f(String s) { ... }
                // ---------- az "f" név túl van terhelve,
                //            mindegyiket meg kell valósítani

    public void g() { ... } // egy közös fv. valósítja meg
 // ------ az implementáló osztályban muszáj kiírni
}
```

Interfészek: alapértelmezett törzs

- az interfészek, ha jól vannak megtervezve, nem változnak
  - ◇ ... de ha mégis, az elrontaná a programot
  - ◇ ... még abban az egyszerű esetben is, ha "csak" bővülne az interfész
    - ▸ az összes implementáló osztályban törzset kellene adni az új metódusnak
- Java 8 megoldás: az interfészben adható alapértelmezett törzs

```
interface F {
    default double f(int n) { return 2 * n + 1; }
    void g();
}

class Impl implements F {
    // f-nek itt nem kötelező törzset adni

    public void g() { ... }   // ... g-nek viszont kötelező
}
```

Interfészek: függvénnyel paraméterezés

- szerepelt említés szintjén: lehet "függvényeket" paraméterként átadni

```
IntUnaryOperator f = x -> c*x;
f.applyAsInt(123);
```

- az `IntUnaryOperator` interfész a `java.util.function` csomagban
- Java 8: ***funkcionális interfész*** (functional interface): olyan interfész, amelynek pontosan egy metódusa van
  - ◇ a metódusnak akárhány paramétere lehet (nulla is)
  - ◇ az ilyen interfészekkel használható a fenti szintaktikus cukor
- valójában a rövidítés az `IntUnaryOperator` egy (névtelen) leszármazottjának egy példányát takarja

```
class MyFun implements IntUnaryOperator {
    public int applyAsInt(int x) { return c*x; }
}

IntUnaryOperator f = new MyFun();
```

Interfészek: függvénnyel paraméterezés

- ***névtelen osztály*** (anonymous class): új osztály és annak példányának létrehozása egyszerre
  - ◇ az új osztálynak az őse lehet osztály vagy interfész
  - ◇ a "névtelen" osztályoknak (`MyClass$1`) és függvényeknek (`lambda$metodusNeve$0`) a fordító generál nevet

```
class MyClass {
    int f(int par1, IntUnaryOperator parFun) {
        int c = 36;
        IntUnaryOperator op = new IntUnaryOperator() {
  // a példányosítást jelzik  ---                    --
  // az ősosztály/interfész neve  ----------------
            public int applyAsInt(int x) {
                return c*x;
            }
        };
        return op.applyAsInt(parFun.applyAsInt(par1));
    }
}
```

Interfészek: függvénnyel paraméterezés

- probléma: a külső metódus már lefutott, amikor a névtelen függvény futna; hogyan érhető el a metódus lokális változója, ami már megszűnt?
  - ◇ Java 8: ***effectively final*** változó: olyan lokális változó, amelynek értéke inicializálás után nem változik meg
    - tehát megkaphatná a `final` módosítót is
    - névtelen függvényben csak a függvény környezetének (effectively) final változóit szabad használni
    - mivel `final` a változó, a futtató rendszer tudja, melyik értéket kell a függvénynek használnia

```
int c = 36;
IntUnaryOperator f = x -> c*x;
// ++c;  // ha itt lenne, "c" nem lenne effectively final
```

- trükk: ha mégis változtatható változót szeretnénk, betehetjük egy (egyelemű) tömbbe

```
int[] c = { 36 };
IntUnaryOperator f = x -> { ++c[0]; return c[0]*x; };
```

Absztrakt osztályok

- olyan osztály, amelynek bizonyos műveletei nincsenek kifejtve
  - ◇ (a Java 8 új eszközeivel) legtöbbször érdemesebb interfészt használni
  - ◇ mivel osztály, egyszeresen örökölhető
  - ◇ mivel osztály, lehetnek (nem-static) adattagjai és műveletei
  - ◇ nem-absztrakt osztálynak lehet absztrakt gyermeke

```
abstract class AbsCl {
    int        adat;
    boolean[] adat2;

    abstract String f();  // törzs megadása fordítási hiba

    void g(int n) { /* g törzse */ }
}

class MyCl extends AbsCl { ... }

AbsCl ac  = new AbsCl(); // példányosítása fordítási hiba
AbsCl ac2 = new MyCl();  // így helyes
```

Tömbök

- a tömbök objektumok Javában, referencián keresztül érhetőek el
- a primitív elemekből álló tömbök közvetlenül tartalmazzák az elemeket
- a tömb objektumok hossza nem változhatható meg
  - ◇ ha arra van szükség, pl. `ArrayList` használható

```
int[] ints1 = new int[]{ 1, 2, 3 };   // ez a teljes alak
int[] ints2 = { 1, 2, 3 };            // néha rövidíthető

int[1] rossz1 = {1};     // a sor elején a [] típust jelöl
int[]  rossz2 = new int[3]{ 1, 2, 3 };     // itt szintén
int[]  rossz3 = new int[]; // példányosítás: kell a méret

int t1[], t2[]; // több tömb is deklarálható egyszerre
                // nehezebben olvasható, mint 2 deklarációban
```

- ha nem adott kezdeti kitöltés, az értékek `0`, `0.0` vagy `false` lesznek

```
int[] ints1 = new int[5];
int[] ints1 = { 0, 0, 0, 0, 0 };  // kifejtve
```

Tömbök

- az objektumtömbök referenciákat tartalmaznak

```
String[] strs1 = new String[]{ "A", "B" };
String[] strs2 = { "A", "B" };
```

- ***aliasing*** vagy ***sharing***: egy egyed több különböző néven is elérhető

```
class X { public int x = 1; }

X[] xs = new X[2];
xs[1] = xs[0] = new X();
System.out.println(xs[0] == xs[1]);  // true

xs[0].x = 2;
System.out.println(xs[1].x);         // 2
```

- a kitöltetlen helyekre `null` kerül

```
String[] strs1 = new String[3];
String[] strs2 = { null, null, null }; // kifejtve
```

Tömbök tömbjei

```
String[][] strs1  = new String[][]{ {"a"}, {"b"} };
String[][] strs1b = { {"a"}, {"b"} };
String[][] strs2  = new String[2][1]; // { {null}, {null} }
String[][] strs3  = new String[1][2]; // { {null, null} }
String[][] strs4  = new String[1][];  // { null }

String[][] rossz = new String[][]; // min. 1 dim. mérete kell
```

- az `int[][]` külső tömbje és belső tömbjei különböző jellegűek

```
int[][] ints = { {1,2,3}, {4,5} };
               //  - - -    - -  az értékeik vannak a tömbben
               // -------  -----  hivatkozások a belső tömbökre
```

- tömbök tömbje potenciálisan "jagged array": a sorok hosszai eltérhetnek

```
String[][] strs5 = new String[2][];
strs5[0] = new String[]{};
strs5[1] = new String[]{"A", "B"};
         // ------------ ide ezt kötelező kiírni
```

Csomagoló osztályok

- a primitív típusok hatékonyak, de néha nem használhatóak
  - ◇ a legtöbb adatszerkezetbe csak objektumokat lehet tenni
    - ▸ a tömbök ebből a szempontból kivételek
- ***csomagoló osztály***ok (wrapper class): a primitív típusoknak megfelelő osztályok

```
byte short int     long char      boolean float double
Byte Short Integer Long Character Boolean Float Double
```

Csomagoló osztályok

- ***becsomagolt*** (boxed) érték: a szokásos alakjához képest egy extra mutatón/hivatkozáson keresztül érjük el
- ***autoboxing*** és ***unboxing***: a primitív típus és a csomagolója közötti konverziót nem kötelező jelölni
  - ◇ a csomagolásnak és a referencia használatának költsége van ⇒ csak akkor használjuk, ha szükséges

```
Integer i1 = 3;
Integer i2 = Integer.valueOf(3);    // valójában ezt jelenti

int     i3 = i1;
int     i4 = i1.intValue();         // valójában ezt jelenti
```

- a referenciák `null` értéket is felvehetnek

```
Integer i1 = null;
int     i2 = i1.intValue();         // NullPointerException
```

Csomagoló osztályok

- az operátoroknak is többletköltsége van
  - ◇ valójában mindig a primitív típussal végezzük a számítást
  - ◇ a fordítóprogram generálja a be- és kicsomagolás kódját

```
Integer a = new Integer(3);
Integer b = new Integer(4);


Integer c = a + b;

// teljesen kiírva:
Integer c = Integer.valueOf(a.intValue() + b.intValue());
```