# Bevezető

## A Java nyelv rétegei

A Java nyelv a megszületése óta eltelt harminc év során professzionális, robosztus nyelvvé vált. Logikailag jól elkülöníthető rétegekből áll, amelyek hagymahéjszerűen épülnek egymásra:

- szintaktikai szabályosság a nyelvben,
- algoritmus alapú strukturált építkezés,
- erős típusosság,
- objektumorientáltság,
- projekt alapú közösségi fejlesztéstámogatás.

Az érettségi feladatok megoldásához az első két héj biztos ismerete elegendő, azonban a Java nyelv egymásra rétegződő héjai egymásra „héjképző” módon hatnak. Ennek következményeként már a legegyszerűbb algoritmusok írása közben is szembesülni fogunk az erős típusosságot megvalósító környezettel, illetve azzal, hogy a számunkra szükséges metódusokat a fejlesztők a nyelvben érvényesülő objektumorientált szemlélet miatt segédobjektumokba építették. A nyelvvel történő ismerkedés korai szakaszában tisztáznunk kell tehát a programnyelvi típus fogalmát illetve a metódusok meghívásának mikéntjét is.

## Programtervezés természetes nyelven

Egy programozási feladat, bármely bonyolult is, amennyiben megoldható, megoldásához minden esetben elegendő egy papír és egy ceruza, továbbá a strukturált programozási paradigma három elemének - *elemi utasítás*, *elágazás*, *ismétlés* – beszélt nyelvi szintű felhasználása.

Persze hiába írunk papír és ceruza segítségével logikailag helyes algoritmusokat magyar nyelven, és hiába építjük össze ezeket logikailag szintén helyesen, mindez önmagában még nem eredményez lefordítható és ezáltal futtatható kódot gépi fejlesztőkörnyezet nélkül. Algoritmusaink mégiscsak gondolatainkból állnak, és gondolatainkat rajtunk kívül vajon ki és hogyan „futtathatná”? És ki biztosíthatna ellenőrzés nélkül afelől, hogy gondolataink mindenképpen helyesek?

Egy egyszerű és rövid algoritmust viszonylag könnyű átlátni és ezáltal helyességét elvi, logikai szinten megállapítani. Azonban minél összetettebb és bonyolultabb az általunk írt algoritmus, annál szerteágazóbb az a logikai gráf, amelynek megfeleltethető, emiatt átláthatósága jelentősen csökken. Mivel a mai napig nem áll rendelkezésünkre emberi nyelven megfogalmazott összefüggő állításcsoportok igazságának olyan általánosan használható korrekt nyelvi elemző módszere – melynek megalkotásáról már Leibniz is álmodott *calculus ratiocinator* fantázianévvel illetve azt – amelynek segítségével tetszőlegesen bonyolult probléma papírral és ceruzával, természetes nyelvi alapon megvalósított megoldásának helyességét megfelelően rövid idő alatt, megbízhatóan ellenőrizhetnénk, ezért számítógépet építünk, és gondolatainkat a gép számára lefordítható *programozási nyelvbe* ágyazzuk. Algoritmusaink, végső soron tehát gondolataink helyességének teszteléséhez egy működő (megvalósított) programozási nyelvet használunk fel!

## Algoritmusok mondatszerű leírása

Egy *algoritmus* egymást követő elemi utasítások véges sorozatából áll, és elvárjuk, hogy az utasítások mindegyike végrehajtható legyen.

Az ógörög szóösszetétel első tagja , második tagja . Az *algoritmus* szóösszetételt ezért mi fordítjuk. És valóban: ,ha azt közölni szeretnénk valakivel! Különösen így van ez, ha mindezt egy géppel szeretnénk megtenni. A gép ugyanis semmilyen metakommunikatív jelzésünket nem tudja értelmezni, sőt utasításaink való világra vonatkozó képzeti tartalmát sem képes elemezni!

Az algoritmus definíciójában használt *elemi utasítás* kifejezés tartalmát nehéz precízen leírni. Ha azt mondjuk egy gépnek: *„Oldd meg a feladatot!”*, az *„Oldd meg”* kifejezés semmiképpen sem minősül elemi utasításnak. Egy feladat megoldása többnyire részlépésekből áll, és előfordulhat, hogy egy-egy részlépés további részlépésekre bontható. Az egyes utasítások egyre elemibb összetevőkre bontásának nyilván van egy ésszerű határa.

Az elemi utasítások számának véges volta illetve az egyes elemi utasítások végrehajthatóságának kívánalma azért szükséges, mert azt szeretnénk, hogy algoritmusunk véges idő alatt lefuthasson, és ne akadjon el egy végrehajthatatlan részlépés közbeiktatása miatt. Szeretnénk látni az eredményt!

Az elemi utasításokkal kapcsolatban megfogalmazott *„egymást követő”* kitétel gondolkodásunk egyfajta linearitására tesz utalást. Amikor problémát oldunk meg és eljutunk egy közbenső lépcsőfokra, jólesően megpihenhetünk, megállapíthatjuk addig elért eredményeinket, majd a részeredményekre alapozva tovább léphetünk, folytathatjuk a megoldást. Neumann János, az elektronikus számítógép elvi alapjainak kidolgozásakor megjegyezte, hogy véleménye szerint az emberi agy nem-lineáris megoldási stratégiákat is felhasznál a gyors és eredményes problémamegoldás érdekében.

## A strukturált programozás elemei

Az elmúlt évszázadok tudományos eredményei arra tanítanak bennünket, hogy a minket körülvevő természetben tapasztalható sokféleség minden esetben visszavezethető valamilyen egyszerű, de legalábbis egyszerűbb, kiindulásul választható alapra. A XVIII. - XIX. század folyamán például az elektromágnesességgel kapcsolatban összegyűlt, egyébként több ezer oldalon leírt kísérleti tényeket sikerült négy alapegyenletre visszavezetni, de gondolhatunk a periódusos rendszerre is. Néhány alkotóelemből szerves vegyületek milliói képesek létrejönni és stabil állapotban megmaradni, az élet anyagi alapjait alkotva. A strukturált programozási paradigma szerint, akkor írunk jó programot, ha a következő három építőelemre hagyatkozunk: *elemi utasítás, elágazás, ismétlés*. Bizonyítható, hogy ezen elemek segítségével minden, egyébként megoldható feladat megoldása leírható.

## Elemi utasítás

*„Rakj rendet a szobában, utána porszívózz légy szíves!”* Ebben a mondatban két utasítás szerepel: *rendrakás*, *porszívózás*. A két utasítás nem tekinthető elemi utasításnak, hiszen a rendrakás valószínűleg több résztevékenységből áll, illetve a porszívózáshoz elő kell keresni a porszívót, csatlakoztatni kell a hálózathoz, be kell kapcsolni és így tovább. Ennek ellenére a *„Készülődés születésnapi bulira”* nevű programban nem kell lemondanunk ezen fogalmak használatáról, pusztán azért mert nem elemi utasítások. A szereplő fogalmakat használhatjuk az őket megvalósító elemi utasítások csoportjának megnevezéseként. Az elemi utasítások csoportját ebben az esetben metódusnak (*method – eljárás*) fogjuk nevezni. Magukat a metódusokat is egységbe foghatjuk *KellTakarítás* néven, ezáltal egy újabb metódushoz jutunk, amely éppen a két szükséges metódust tartalmazza.

**Eljárás** KellTakaritás
  Rendrakás
  Porszívózás
**Eljárás vége**

Amikor hasonló absztrakciókkal élve röviden leírjuk elvégzendő feladatainkat, azaz nyelvi formába öntjük azokat, valójában már algoritmizálunk! Az egyes algoritmusok fentiekhez hasonló jellegű leírását *mondatszerű leírás*nak nevezzük. Ha minden esetben ragaszkodunk ehhez a szabványos formátumhoz, algoritmusaink a magunk és mások számára is könnyen áttekinthetővé válnak. Az áttekinthetőséget tovább növelhetjük a mondatszerű leírásokban meghonosodott néhány további szokás betartásával. Például a logikailag összetartozó részeknél a margótól számítva azonos mértékű baloldali behúzást alkalmazunk. Ezáltal jól látható lesz, hogy hol van a *KellTakarítás* eljárás eleje illetve vége, és az is, hogy az eljárás pontosan két utasítást tartalmaz.

Ha tovább gondoljuk a fenti, egyébként életszerű példát, felmerülhet bennünk: takarítás és takarítás között sok különbség lehet. Hogy csak a legegyszerűbbet említsük, másképpen kell takarítani egy kertet, másképpen kell takarítani egy udvart, és másképpen kell takarítani egy szobát. Talán jelezni kellene mindezt. Tovább menve, ha végeztünk a takarítással, takarításunk minősége értékelhetővé válik. Jó lenne, ha ez is megjelenhetne mondatszerű leírásunkban! És végül, talán jelezni kellene azt is, hogy a *Rendrakás* illetve *Porszívózás* nevek nem elemi utasításokat takarnak, hanem eljárásokat. (A *KellTakarítás* név használatakor ezt explicit módon jeleztük az *Eljárás KellTakarítás* kifejezésforma használatával).

**Eljárás** KellTakaritás (**helyiség** *szoba*): **logikai**
  Rendrakás()
  Porszívózás()
**Eljárás vége**

Ha egy név után zárójelpárt írunk, az megállapodás szerint azt jelenti, nem elemi utasítást takar a név, hanem eljárást. A természetes nyelvek is használnak hasonló megfontolások alapján úgynevezett minősített neveket, ilyenek például a tulajdonnevek, és ezt jelzik is, például azzal, hogy nagybetűvel kezdik a nevet.

A *KellTakarítás* nevű eljárásunk után írt zárójelbe most már beírhatjuk a takarítandó terület jellegét. Ezt az eljárás paraméterének nevezzük. Paraméterünk most éppen a felkérésben szereplő, meghatározott szoba a lakásunkban. A szoba *helyiség* típusú. A típus segítségével a takarítandó terület jellegét is meg tudtuk adni. Ez olyan esetekben lehet fontos, ha a *szoba* szabványos név helyett valaki nem a természetes nyelv szabályozott névteréből kíván azonosítót választani, például „*bungi”*-nak szeretné nevezni szobáját, vagy éppen „x”-nek. Ezek a nevek nem feltétlenül vagy egyáltalán nem hordozzák magukban azt az információt, hogy szobaszerű helyiséget kell takarítanunk. A típus szerepeltetése szabadabbá teheti a névválasztást, de ami még fontosabb, a típusok előzetes megállapodás alapján történő megválasztása és pontos leírásuk megkönnyítheti szabványos eljárások írását. Egy korrekt fejlesztőkörnyezet például hibát fog jelezni, ha *fürdőszoba* típusú helyiséggel paraméterezzük *Fűnyírás* nevű eljárásunkat.

Eljárásunknak van visszaadott értéke is, amit szokás *visszatérési értéknek* nevezni. Ennek az értéknek a kettőspont után szerepel a típusa. A *KellTakarítás* eljárás visszatérési értéke *logikai - boolean* típusú. Az eljárás feladatának elvégzése után visszajelzést ad arról, megfelelő-e a takarítás minősége: *igen – true*, vagy *nem – false*. *KellTakarítás* nevű eljárásunk ezáltal egy elágazás vezérlésére is alkalmassá válik.

Végül megadjuk algoritmusunk Java nyelvben használható mondatszerű leírását is. Alább látható, hogy az *Eljárás* és *Eljárás vége* formális jelzéseket elhagytuk, és egy kapcsos zárójelpárral helyettesítettük azokat, továbbá az eljárás fejlécét tartalmazó sor elején tüntettük fel a visszatérési érték típusát, elhagyva a kettőspontot. A *KellTakarítás* eljárás egyes sorait pontosvesszővel zártuk az egyértelműség kedvéért. A meglepő az, hogy ezek a minimális változtatások lehetővé teszik, hogy ezt a kódot a Java fordító már képes legyen lefordítani a processzor számára érthető és ezáltal végrehajtható parancsokká.

Új kódunk talán legérdekesebb, eddig nem említett részlete a korábbi mondatszerű leírásban nem szereplő *SzülőiIntelmek.* kifejezés. Igen … , kikerülhetetlenül megjelent a Java nyelv objektumorientált szemléletéről hírt adó első „fecske”: mi az, hogy SzülőiIntelmek.*Rendrakás()*; ??

```
logikai KellTakaritás (helyiség szoba) {
   SzülőiIntelmek.Rendrakás();
   SzülőiIntelmek.Porszívózás();
}
```

A kódban szereplő *KellTakarítás* eljárást kimerítően és precízen definiáltuk, pontosan látunk minden szükséges információt. De hol van definiálva a két belső eljárás. Beszéltünk már róluk, azonban precízen sehol sem definiáltuk őket. Ha tehát *Rendrakás();* és *Porszívózás();* formában adnánk oda a Java fordítónak az eljárásokat, nem tudna velük mit kezdeni: mi az, hogy *Rendrakás()* és mi az, hogy *Porszívózás()?* Nagyon fontos „előtag" tehát a *SzülőiIntelmek.* karaktersorozat, ugyanis arra utal, létezik egy *SzülőiIntelmek* objektum „valahol" – persze a *KellTakarítás* eljárás számára elérhető módon – és ebben az objektumban definiálva van sok hasznos dolog mellet az is, hogyan kell „Rendetrakni" és hogyan kell „Porszívózni". A kérdéses két sorral tehát ennek a „távoli", de legalábbis független és önálló objektumnak egy-egy már definiált metódusát hívjuk meg a *KellTakarítás* eljárás belsejében.

Kódunk imént vázolt kialakítása első látásra némileg furcsának tűnhet, és akár azt is gondolhatjuk, mindez túl mesterkélt, a valóságban ilyen nincs is! Pedig van! A SzülőiIntelmek „objektum" ugyanis létezik, mégpedig a fejünkben van elraktározva emlékek formájában. Pontosan emlékszünk, amikor szüleink megmutatták gyerekkorunkban egy végigjátszott nap után, hogyan kell rendet rakni a szobánkban. Mire kell ügyelnünk, hogy a végén szüleinknek is tetsző eredményre juthassunk. Ezek az imprinting-ek ott vannak emlékeinkben és vezérlik mindenkori rendrakásainkat. Amikor tehát meghalljuk a *„Rakj rendet a szobában, utána porszívózz légy szíves!"* kérést, meg kell hívnunk a jelenben egy önálló „távoli" emlékobjektumot.

### Elágazás

```
üres KészülődésSzületésnapiBulira() {
   helyiség szoba;
   szoba = példányosít helyiség();
   logikai takarítvaSzoba;
   takarítvaSzoba = KellTakarítás (szoba);
   Ha (takarítvaSzoba) {
         Üzenőfal(„Köszi!");
   } különben {
            Üzenőfal(„Ma még ki lesz takarítva?");
      }
}
```

Folytatva az előző fejezetben tárgyalt példánkat kiegészítettük a kódot. A *KészülődésSzületésnapiBulira* nevű eljárás egy új eljárás. Eljárás voltát jelzi a neve után feltüntetett zárójelpár, ami most egyébként üres, azaz az eljárásnak nincs paramétere. Az új eljárásunk neve előtti *üres - void* kifejezés arra utal, a *KészülődésSzületésnapiBulira()* eljárásnak nincs visszaadott értéke.

Bevezettünk egy *takarítvaSzoba* nevű logikai típusú változót, amely tárolja a *KellTakarítás (szoba)* eljárás lefutása után az eljárás által

visszaadott értéket. Ezzel a logikai értékkel vezéreljük az elágazást. Ha a takarítás megfelelő, köszönetet kapunk, különben elmarasztalást. Az alábbiakban megadjuk az elágazás programozási környezettől független mondatszerű leírását is:

**Ha** *feltétel igaz*
**akkor**
utasítás $_1$
…
utasítás $_n$
**akkor ág vége**
**különben**
utasítás $_{n+1}$
…
utasítás $_{n+m}$
**különben ág vége**
**elágazás vége**

Java nyelvi környezetben az elágazást vezérlő feltételt zárójelbe tettük, az *akkor* és *akkor ág vége*, illetve a *különben* és *különben ág vége* kifejezéseket a megfelelő kapcsos zárójelekkel helyettesítettük, és az egyértelműség miatt elhagytuk az *elágazás vége* kifejezésnek megfeleltethető ötödik zárójelet.

Amennyiben nem kívánunk műveletet végrehajtani a vezérlő feltétel nem teljesülése esetén, akkor a *különben ág* elhagyható.

```
Ha feltétel igaz
  akkor
    utasítás 1
    …
    utasítás n
  akkor ág vége
```

vagy

```
Ha feltétel igaz
  utasítás 1
  …
  utasítás n
elágazás vége
```

```
Ha (takarítvaSzoba) {
  Üzenőfal(„Köszi!”);
}
```

Végül röviden szót ejtünk a Java nyelvű környezetben világosszürke színnel jelzett kódsorról is. Ez a kódsor már tipikusan kötődik a Java nyelven megvalósított fejlesztőkörnyezet követelményeihez. A *helyiség szoba;* kódsorral azt jeleztük a fordító számára, hogy szeretnénk a későbbiekben használni egy *szoba* nevű és *helyiség* típusú változót. Ezt követően a szoba nevű objektum már a program „látókörében van”, „tudja”, hogy előbb-utóbb egy valódi szobával kell majd valamit csinálnia, de a valódi szoba még nem létezik. (Például egy takarítással foglalkozó céggel tárgyalunk, közöljük, hogy ki kellene takarítani egy ilyen és ilyen szobát, de a címünket még nem adtuk meg.) A takarítás elvégzéséhez ténylegesen létre kell hoznunk egy valódi szobát, vagy rá kell mutatnunk a házban a mi szobánk helyére.

A későbbiekben még sok szó fog esni a *példányosítás – new* parancsról.

## Ismétlés

## Számlálós ciklus

Tételezzük fel, hogy kerítésünk egy szakaszát öt vasoszlop tartja. Mindegyik oszlopot le szeretnénk festeni időjárásálló festékkel. A cselekvéssort a következő algoritmussal adhatjuk meg:

**Ciklus i = 1 – 5**
  OszlopFestés
**ciklus vége**

Mivel pontosan ismerjük a cselekvéssor szükséges ismétléseinek számát, ezért ezt a ciklus fajtát *számlálós ciklusnak* nevezzük. Az *i* változó a ciklus számlálója, a ciklus belsejében szereplő ismétlendő utasítások pedig a ciklus magját képezik. A ciklusmag a mi esetünkben most egyetlen ismétlendő eljárásból áll: *OszlopFestés*. Az *i* ciklusszámláló értéke a ciklusmag minden végrehajtása után automatikusan eggyel nő, a megadott alsó határtól indulva egészen a megadott felső határig. A ciklusszámláló nevének megválasztását az indokolja, hogy *i* egy *jelzőszám,* azaz *index*. Leggyakrabban az i, j, k változóneveket használjuk erre a célra.

Sok esetben kihasználhatjuk a ciklusszámláló automatikus és szabályos növekedését a ciklusmag belsejében. Sorszámozzuk meg az oszlopokat. Ezután némileg „beszédesebbé” tehetjük algoritmusunkat, ugyanis az *Oszlopfestés* nevű eljárás nem jelzi számunkra explicit módon, hogy öt különböző oszlopot akarunk-e lefesteni, vagy egyetlen oszlopot ötször. Megváltoztatjuk ezért az eljárás nevét: *Festés ( oszlop(i) )*. Ily módon jelezni tudjuk, hogy a ciklus első végrehajtásakor az első oszlopot, második végrehajtásakor egy másik oszlopot, a másodikat, festjük le és így tovább. Módosított ciklusunk Java nyelvű megvalósításának mondatszerű leírása:

**Ciklus** (i=1; i<6; differencia(i)=+1) **{**
  Festés( oszlop(i) );
**}**

A legtöbb programozási nyelv lehetővé teszi, hogy a differencia ($\Delta$) ne csak egész értéket vehessen fel, sőt akár negatív érték is lehessen. Például a fordított sorrendű festést a következőképpen jelölhetjük *i = 5 - 1*, vagy *(i=5; i>0, $\Delta$i=-1)*

## Elöl tesztelős ciklus

**Ciklus amíg** van barack a fán
  Szedd a barackot
**Ciklus vége**

Az őszibarackszedés algoritmusa elöl tesztelős ciklus segítségével valósítható meg. Sajnos ugyanis elképzelhető, hogy elfagyott a termés, azaz egyetlen barack sincs a fán, ezért kell *előzetesen* ellenőriznünk a fa állapotát.

A *ciklus amíg* kifejezést követő feltételt *belépési feltétel*nek nevezzük. Ha a belépési feltétel hamis, a ciklus egyszer sem hajtódik végre.

## Hátul tesztelős ciklus

*- Hú, de meleg van! Kérhetek egy kis hideg szörpöt!*
*- Persze, de csak kispoharam van, nem baj?*
*- Nem, dehogy!*
*Készül a szörp …, megiszom.*
*- Ez nagyon jól esett, kaphatok még?*
*- Persze!*
*Készül az újabb szörp …, megiszom.*
*- Hmm!*
*- Kérsz még?*
*- Igen!*
*Készül az újabb szörp …, megiszom.*
*- Köszönöm, elég, ez „életmentő” volt!*

Algoritmizáljuk a meleg nyári napokon gyakran előforduló előbbi párbeszéd programozási szempontból lényeges elemeit!

```
Ciklus
   SzörpKészítés
   SzörpFogyasztás
amíg szomjas vagyok
```

*vagy:*

```
Ciklus
   SzörpKészítés
   SzörpFogyasztás
amíg nem nem vagyok szomjas
```

A hátul tesztelős ciklus mindenképpen lefut legalább egyszer és egészen addig ismétlődik, amíg a bent maradási feltétel igaz marad (bal oldalon szereplő algoritmus) vagy a kilépési feltétel igazzá nem válik (jobb oldalon szereplő algoritmus). Egyes nyelvek a bent maradási, egyes nyelvek a kilépési feltétel megfogalmazását várják el. A két különböző megközelítésben az egyes feltételek nyilván egymás tagadásai. A Java nyelvben a ciklusban való *bent maradási feltételt* kell megfogalmazni!

Megjegyzendő, hogy a kilépési feltétel megfogalmazását váró nyelvekben a kettős tagadás miatt a vezérlés számára mégiscsak a bent maradási feltétel kerül megadásra:

(amíg nem)((nem)(vagyok szomjas))=(amíg)(nem)((nem)(vagyok szomjas)) = (amíg) ((nem) (nem)) (vagyok szomjas) = (amíg) (vagyok szomjas).

Az előbbi két példa Java nyelvű megvalósításának mondatszerű leírása:

```
Ciklus amíg (van barack a fán) {
   Szedd a barackot();
}
```

```
Ciklus {
   SzörpKészítés();
   SzörpFogyasztás();
} amíg (szomjas vagyok)
```

## Egy algoritmus programnyelvi állapotai

Algoritmusaink programnyelvi megvalósításuk során meghatározott „fejlődési” utat járnak be. A fejlődési út végén az algoritmus programnyelvi megvalósítása akkor tekinthető ideálisnak, ha egy szintaktikailag, szemantikailag és logikailag hibátlan, továbbá hatékony, hibatűrő, és elegáns programot sikerült létrehoznunk. Vizsgáljuk meg röviden a fejlődési út egyes állomásait!

*Szintaktikai állapot*

A kód kezdetben tartalmazhat formális nyelvi hibákat, például hiányzik a sor végén szükséges pontosvessző vagy egy zárójelpár záró tagja. Ezeket a hibákat a fordító fordítási időben a kód futtatása előtt észreveszi, ezért ezeket a hibákat *fordítási idejű hibáknak* nevezzük. A szintaktikai hibafigyelés nyelvi megvalósítástól függően kiegészülhet egy minimális szemantikai ellenőrzéssel is. Például deklarálunk egy tízelemű tömböt, miközben explicit módon hivatkozunk a kódban egy tizenegyedik elemre. Ezt észre lehet venni a tartományok terjedelmének figyelésével. Egy másik gyakori hibalehetőség például, hogy egy egész típusú változónak valós típusú lebegőpontos értéket próbálunk átadni. Ez szintén észrevehető a beépített típusleírások alapján.

*Szemantikai állapot*

A kód már szintaktikailag hibátlan, de futási időben mégis „lefagy”, kivételt dob. Például az előbb említett tízelemű tömb tizenegyedik elemére ugyan nem hivatkozunk explicit módon, azonban egy rosszul beállított ciklusban egyesével növelve a ciklusváltozó értékét mégiscsak eljutunk a nem létező tizenegyedik elemhez. A fordító ezt a hibát és az ehhez hasonló hibákat csak akkor lenne képes előre észrevenni, ha képes lenne elméletben is futtatni, elemezni a lefordított algoritmusunkat, ezáltal végigpásztázva annak minden lehetséges állapotát. Mivel erre a fordító nem képes, ténylegesen végig kell futtatnia a kódot, ekkor szembesülve az esetleges szemantikai hibákkal. Ezek a hibák tehát csak a kód futtatása során derülhetnek ki, ezért az ilyen jellegű hibákat *futás idejű hibáknak* nevezzük.

*Logikai állapot*

Elképzelhető, hogy egy program ugyan hibamentesen lefut saját programkörnyezetében, de mégsem a kitűzött feladatot oldja meg, azaz *logikailag hibás*. A logikai hibákat csak a programozó tudja javítani programjának alapos tesztelését követően. Az érettségi követelményrendszere az eddig említett három feltétel megvalósítását várja el a vizsgázótól, azaz programja legyen szintaktikailag, szemantikailag és logikailag hibátlan.

*Hatékonyság*

Egy program hatékonysága több tényezőn is múlhat. Elsősorban a végrehajtás sebessége illetve az erőforrások mértékletes használata befolyásolja hatékonyságát, de fontos szempont lehet a kód átláthatósága, könnyű fejleszthetősége is.

*Hibatűrés*

Egy program hibatűrővé tételével kapcsolatos munkálatok sok esetben összemérhetőek magának a programnak a létrehozásával. Érzékeny programok esetében az is előfordulhat, hogy a programot ténylegesen megvalósító kódmagot a kódmagnál jóval nagyobb terjedelmű „program védművel” kell körülvenni. Egy hétköznapi példa: pontosan három karakter hosszúságú választ várunk a felhasználótól, azonban a felhasználó véletlenül vagy készakarva folyamatosan nyomva tart egy billentyűt. Programunknak ezt az egyszerű, gyakran előforduló hibafaktort is tudni kell kezelnie.

*Elegancia*

Egy program eleganciája nehezen meghatározható fogalom, a programozó illetve programozó társai inkább csak érzik, semmint szavakba tudnák mindezt önteni. Mindenesetre ne lepődjünk meg akkor, amikor egy ismert programozó, Donald Knuth *„A programozás művészete*” címet adja kétkötetes könyvének.