Programozási nyelvek – Java
Objektumelvű programozás

Kozsik Tamás

Absztrakció - típusmegvalósítás

- Egységbe zárás (encapsulation)
- Információelrejtés

Az OOP paradigmában a kód fő rendező elve a számításokhoz használt adatok és a rajtuk végezhető alapvető műveletek összefogása objektumokká. A program nem más, mint az objektumok összessége és a közöttük megvalósuló interakciók. Az objektum-orientált tervezésben a feladatokat nem egy számítógépközeli fogalmi (absztrakciós) szinten próbáljuk megoldani. Ehelyett felépítjük az adott alkalmazási terület modelljét: objektumok segítségével modellezzük az adott alkalmazási terület fogalmait, elemi eseményeit. Ezután az így felépített modell segítségével készítjük el a probléma megoldását. Tehát a megoldás a problématér absztrakciós szintjén kerül megfogalmazásra. Ezért van az, hogy az OO szoftverfejlesztésben a fogalomalkotás (absztrakció) egy igen fontos tevékenység.

Az osztály alapú OO nyelvek az objektum típusát az osztálydefiníciók segítségével fogalmazzák meg. Így az osztály a típusmegvalósítás eszközévé válik. A modellépítés során az egyes fogalmakat osztályok segítségével írjuk le. Egy osztály tartalmazza az általa megvalósított fogalom számítógépes implementációját. A program futása során az objektumok interakcióba kerülnek egymással, de ezek az interakciók az alkalmazási terület logikai szintjén vannak definiálva, a fogalmak számítógépes implementációjának (az objektum belső működésének) módjától függetlenül.

Az absztrakció azt jelenti, hogy a fogalomalkotás során a szétválasztjuk a modell szempontjából releváns dolgokat az implementációs részletektől: az objektumok más objektumokból történő használata során elvonatkoztathatunk az implementációs részletektől. Az osztálydefinícióban ez úgy jelenik meg, hogy az implementációs részleteket elrejtjük a külvilág elől. Az osztály definiálja az objektumainak a nyilvános interfészét, azaz azon tulajdonságait és műveleteit, melyek az alkalmazási terület modelljének megfelelően a többi objektum számára relevánsak. Minden más információt elrejt az osztálydefiníció a külvilág elől.

1 Egységbe zárás

Osztály, objektum, példányosítás

Point.java

```
class Point {       // osztálydefiníció
    int x, y;       // mezők
}
```

. . .

Main.java

```
class Main {
    public static void main( String[] args ){   // főprogram
        Point p = new Point();     // példányosítás (heap)
        p.x = 3;                   // objektum állapotának
        p.y = 3;                      // módosítása
```

```
    }
}
```

Két külön fájlba elkészíthetjük a fenti két osztálydefiníciót. A főprogramban létrehozunk egy *példányt* (objektumot) a `Point` osztályból. A példányosítás a `new` operátor segítségével történik. Ez a C-beli `malloc` megfelelője: a dinamikus tárhelyen allokál memóriát az objektum számára, és visszaad egy (típushelyes) hivatkozást az objektumra. Ezt a hivatkozást *referenciának* szokás nevezni, és ez olyasvalami, mint a pointer a C-ben.

A Java nyelvben mindig minden objektum a heapen jön létre, és ezeket az objektumokat referenciákon keresztül, azaz indirekt módon használjuk. A `p.x` kifejezés a C-beli `p->x` megfelelője: feloldja a `p` hivatkozást, és az elért objektumból előkeresi az `x` mezőt (dereferálás és szelekció).

Fordítás, futtatás

```
$ ls
Main.java  Point.java
$ javac *.java
$ ls
Main.class  Main.java  Point.class  Point.java
$ java Point
Error: Main method not found in class Point, please define
 the main method as:
   public static void main(String[] args)
$ java Main
$
```

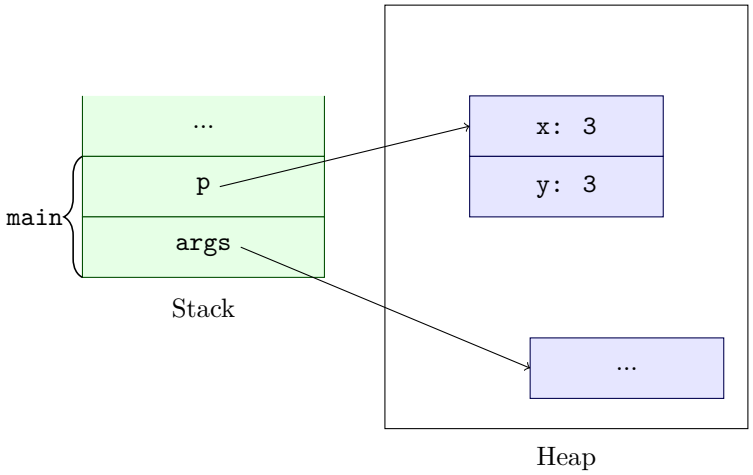
Ha elkészült a két forrásfájl, egy paranccsal mindkettőt lefordíthatjuk: a fordításhoz mindig a lefordítandó *forrásfájlok nevét* kell megadni! A fordítóprogram minden osztálydefinícióból generál egy bájtkódot tartalmazó, `.class` kiterjesztésű fájlt (tárgykód).

A `Point` osztály nem egy futtatható osztály, mert nem szerepel benne a főprogramként elvárt módon definiált `main` metódus. Ha megpróbáljuk mégis lefuttatni, a JVM hibát jelez. A `Main` osztály viszont futtatható, és hiba nélkül le is fut. Figyeljük meg, hogy a JVM indításánál nem fájlnevet, hanem *osztálynevet* kellett megadni! Ha megpróbálnánk ezt a parancsot végrehajtani: `java Main.class`, hibát kapnánk, mert a JVM nem találná a `Main.clas.class` fájlt, aminek tartalmaznia kellene a `Main.class` nevű osztály kódját.

```
Error: Could not find or load main class Main.class
Caused by: java.lang.ClassNotFoundException: Main.class
```

Elvileg az fentiek alapján az is hibás, ha a programot a `java Main.java` paranccsal próbáljuk futtatni, hiszen ez azt jelentené, hogy a `Main.java` nevű osztályt kívánjuk végrehajtani, amelynek a kódja a `Main.java.class` fájlban kell, hogy legyen – és persze ez a fájl sem létezik a mi esetünkben. Azonban azt is el kell itt mondanunk, hogy a 11-es JDK-t kezdve az egyetlen forrásfájlból álló programjainkat úgy is lefuttathatjuk, hogy nem fordítjuk le előzetesen a `javac` paranccsal. A fenti példában ez az előfeltétel nem teljesül, de ha egy forrásfájlba másolnánk mindkét osztálydefiníciót, pl. a `Main.java` fájlba, akkor a `java Main.java` parancs végrehajtaná a programunkat (a háttérben titokban meghívva a fordítóprogramot). Egyelőre nem szeretnénk elmerülni ebben a témában: maradjunk annyiban, hogy minden osztályt külön fájlba írunk, ezeknek a fájloknak a neve megegyezik a tárolt osztály nevével (kis- és nagybetűk is számítanak!), és a fordításnál a forrásfájl nevét (kiterjesztéssel), futtatásnál pedig az osztálynevet (azaz `.class` nélkül) adjuk meg.

Stack és heap



Javában a program végrehajtása során két fontos adatterületre van szükségünk: a végrehajtási veremre, ahol a metódushívások nyomon követését szolgáló aktivációs rekordok találhatók (bennük a lokális változókkal), valamint a dinamikus tárhelyre, ahol az objektumok jönnek létre. A példánkban a főprogram aktivációs rekordja bekerül a verem aljára, benne két lokális változóval: az `args` a parancssori argumentumokat tároló tömbre hivatkozó referencia (ezt a tömb objektumot a JVM hozza létre számunkra a program indításakor), míg a `p` által hivatkozott objektumot mi magunk hozzuk létre, és töltjük fel adatokkal.

A fentiekből leszűrhetjük azt a következtetést is, hogy a Javában a tömbök valójában objektumok. Van nekik egy adattagjuk is, a korábban már látott `.length`, mely a hosszukat tárolja (csak olvasható adattag), de speciálisak abban az értelemben, hogy az indexelés műveletet is használhatjuk rájuk.

Mezők inicializációja

```
class Point {
    int x = 3, y = 3;
}

class Main {
    public static void main( String[] args ){
        Point p = new Point();
        System.out.println(p.x + " " + p.y);    // 3 3
    }
}
```

A mezőket bevezető deklarációkban elhelyezhetünk kezdőérték beállítását végző értékadásokat is. Ezek hatása az lesz, hogy amikor az objektum létrejön, az értékadás a mezőn végrehajtódik. Ez egy lehetséges módja az objektumok inicializálásának.

Mező alapértelmezett inicializációja

Automatikusan egy nulla-szerű értékre!

```
class Point {
    int x, y = 3;
}

class Main {
    public static void main( String[] args ){
        Point p = new Point();
        System.out.println(p.x + " " + p.y);    // 0 3
```

```
    }
}
```

A Java annyira igyekszik segíteni a programozókat, hogy inkább minden mezőt inicializál, nehogy memóriaszemetet találjunk egy változóban. Ha egy mezőt nem inicializál explicit módon a programozó, akkor az a mező egy nulla-szerű értékre inicializálódik majd. Például egy szám nullára, egy karakter a nulla kódú karakterre, egy logikai típusú változó pedig hamisra.

1.1 Metódusok

Metódus

```
class Point {
    int x, y;   // 0, 0
    void move( int dx, int dy ){   // implicit paraméter: this
        this.x += dx;
        this.y += dy;
    }
}

class Main {
    public static void main( String[] args ){
        Point p = new Point();
        p.move(3,3);              // p -> this, 3 -> dx, 3 -> dy
    }
}
```

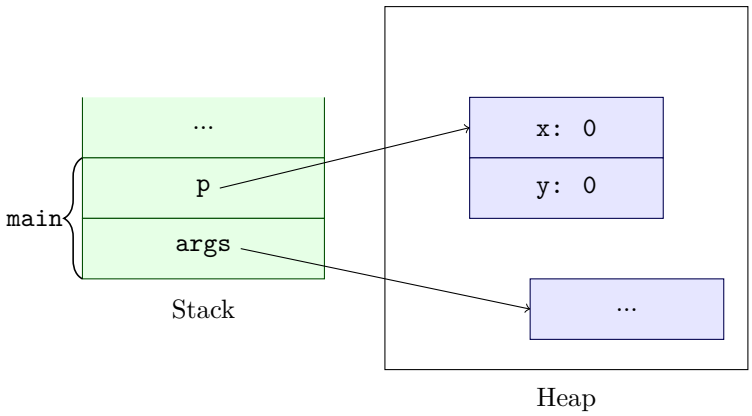
Most bővítsük ki az osztálydefiníciót egy metódussal. Mint látjuk, a `move` metódus hívásakor három paramétert adunk át: a `p` referenciát, valamint a `3` értéket kétszer is. Az OO jelölésrendszert követve a `p` paraméter kitüntetett helyre került. A `move` művelet logikailag a `Point` osztály elemi műveleteként lett meghatározva, ezért azt a `Point` objektumot, amin az eltolás műveletet végrehajtjuk, kitüntetett paraméternek tekintjük. Később megtanuljuk majd, hogy ez nem pusztán szintaktikus megkülönböztetés. Az OOP meghatározó tulajdonságainak nevezett *felüldefiniálás* és *dinamikus kötés* épp erre a kitüntetett paraméterre vonatkozik majd.

A `p` paraméter kiemelése az aktuális paraméterlistából visszaköszön a `Point` osztály definíciójából is. A `move` művelet formális paraméterlistája is csak a két `int` típusú paramétert tartalmazza. A metóduson belül a kitüntetett paraméterre a `this` kulcsszóval hivatkozhatunk, azaz a kitüntetett paraméter implicit módon jelenik meg a `move` definíciójában.

Nézzük meg, hogy a paraméterátadást hogyan követhetjük nyomon a végrehajtási verem és a dinamikus tárhely segítségével.

Metódus aktivációs rekordja – 1

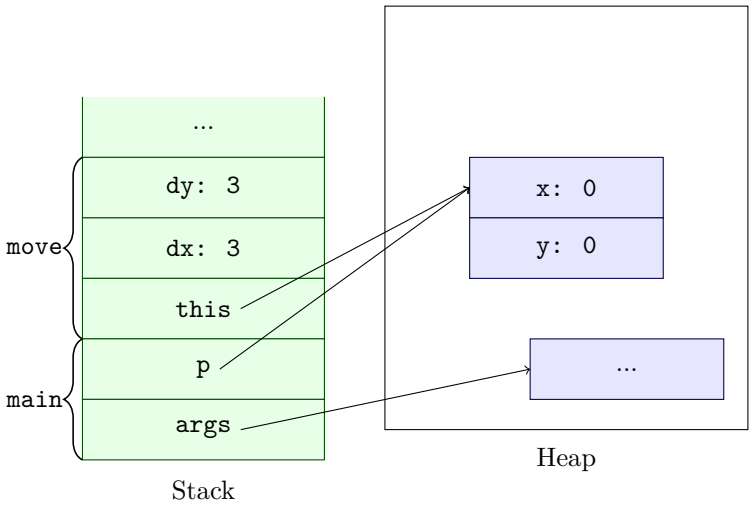
```
Point p = new Point();
```



Amikor kiértékeljük a `new` kifejezést, létrejön a heapen egy `Point` objektum, a mezői pedig inicializálódnak az alapértelmezett nulla értékre. A `p` változó inicializáló értékadása elmenti a `p`-be a létrejövő objektum hivatkozását.

Metódus aktivációs rekordja – 2

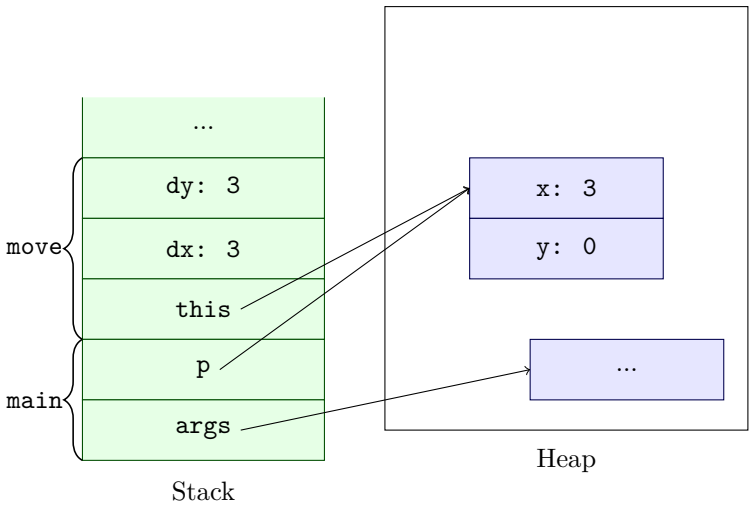
```
p.move(3,3);
```



Amikor meghívjuk a `move` metódust, az aktivációs rekordja bekerül a verembe. Ebben az aktivációs rekordban három változó kap helyet: egy a kitüntetett paraméternek (`this`), egy a `dx`, és egy a `dy` paraméternek felel meg. A `this` megkapja a `p` aktuális paraméter értékét, amely egy referencia a heapen tárolt `Point` objektumra. Így a `this` és a `p` egymás *aliasai* lesznek, ugyanarra az objektumra hivatkoznak. A `dx` és `dy` változókba bemásolódik a `3` érték.

Metódus aktivációs rekordja – 3

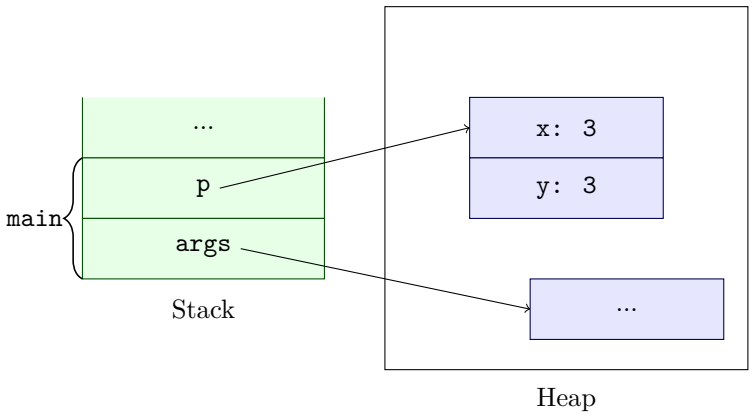
```
this.x += dx;
```



A `Point` objektum `x` mezőjének növelése a fenti utasítással történik. A `this.x` kifejezésben a pont dereferálást (a heapen tárolt objektum felkeresését) és szelekciót (egy mező kiválasztását) is jelenti. A két lépés segítségével megtaláljuk azt a tárhelyet, amelyben az ott talált értéket `dx`-szel növelni kell. A `dx` értékét az aktivációs rekordból keressük elő.

Metódus aktivációs rekordja – 4

```
System.out.println(p.x + " " + p.y);
```



Amikor a `move` hívása befejeződik, az aktivációs rekord törlődik a veremből, és visszakerül a vezérlés a `main` metódusba. A `p` referencián keresztül felkeresve a `point` objektumot, benne a `3` értékű mezőket találjuk.

A `this` implicit lehet

```
class Point {
    int x, y;   // 0, 0
    void move( int dx, int dy ){
        this.x += dx;
        y += dy;
    }
}

class Main {
    public static void main( String[] args ){
        Point p = new Point();
        p.move(3,3);
```

```
    }
}
```

Általában a `this` elhagyható egy metódus definíciójában. Amikor a fenti kódban az `y += dy` utasítást értelmezi a fordítóprogram, akkor az `y`-t automatikusan `this.y`-nak veszi – más értelme nincs itt az `y` névnek. Ez összhangban van a C nyelv kapcsán megismert, és a Java nyelvben is hasonlóan működő *hatókör* fogalmához. A `move` definíció az `y` mező deklarációjának hatókörében van, ezért hivatkozhatunk a `move`-on belül az `y` mezőre.

1.2 Konstruktorok

Inicializálás konstruktorral

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

class Main {
    public static void main( String[] args ){
        Point p = new Point(0,3);
        System.out.println(p.x + " " + p.y);   // 0 3
    }
}
```

Van, amikor a mezőkre megfogalmazott kezdő értékadásnál összetettebb módon szeretnénk inicializálni az objektumainkat, például szeretnénk az inicializálást paraméterezhetővé tenni. Ilyen esetben *konstruktorra* van szükségünk. A konstruktor egy olyan metódusszerű művelet, amely csak és kizárólag az objektum létrehozása utáni inicializáció során futhat le. Ugyanúgy kaphat paramétereket, mint egy metódus, és inicializálhatja az objektum mezőit például ezen paraméterek segítségével. A fenti példában a `main` létrehoz egy `Point` objektumot, és inicializálja a két paraméteres konstruktor segítségével. Ez a konstruktor a két aktuális paraméter alapján inicilizálja az objektum `x`, illetve `y` mezőjét. Fontos kihangsúlyozni, hogy a konstruktort csak a `new` operátor segítségével képzett kifejezéssel tudjuk végrehajtatni.

A konstruktorok neve mindig az osztálynév (ebben az esetben a `Point`). Szintaktikusan abban különböznek a metódusoktól, hogy nincs visszatérési típus megadva a név előtt. (A `move` esetén a formális paraméterlista előtt az áll, hogy `void move`, míg a konstruktor esetén csak annyi, hogy `Point`.)

Kicsit szerencsétlen az, hogy a Java megengedi, hogy egy metódust ugyanúgy hívjunk, mint a tartalmazó osztályt. (Ilyet szándékosan sosem csinálunk, már csak azért sem, mert az osztályok neve nagybetűvel, a metódusok neve kisbetűvel kezdődik az elnevezési konvenciók szerint.) Elvileg tehát véthetünk egy olyan hibát, hogy egy konstruktor elé kitesszük a `void` kulcsszót – ez azt jelenti, hogy nem egy konstruktorról, hanem egy metódusról van szó. Ha a `Point` osztály esetén így járnánk el, azt tapasztalnánk, hogy a `Point` osztály értelmes, azaz a fordítóprogram elfogadja, lefordítja. Persze az értelme nem az, mint amit mi szerettünk volna! Ez ki is derül akkor, amikor a `Main` osztályt megpróbáljuk lefordítani: a konstruktorhívás két paraméterrel hibás lesz, hiszen így a `Point` osztályban nem lesz két `int` paraméteres konstruktor.

Inicializálás konstruktorral – a `this` elhagyható

```
class Point {
    int x, y;
    Point( int initialX, int initialY ){
        x = initialX;
        y = initialY;
```

```
    }
}

class Main {
    public static void main( String[] args ){
        Point p = new Point(0,3);
        System.out.println(p.x + " " + p.y);   // 0 3
    }
}
```

Észrevehető, hogy ebben a példában a konstruktor törzsében elhagyható a `this` hivatkozás.

Nevek újrahasznosítása

```
class Point {
    int x, y;
    Point( int x, int y ){   // elfedés
        this.x = x;          // minősített (qualified) név
        this.y = y;          // konvenció
    }
}

class Main {
    public static void main( String[] args ){
        Point p = new Point(0,3);
        System.out.println(p.x + " " + p.y);   // 0 3
    }
}
```

Van egy olyan konvenció is, hogy a konstruktor paramétereit pont úgy nevezzük el, mint azt a mezőt, amit inicializálni szeretnénk vele. A példánkban az `x` mező inicializálására szolgáló paramétert `x`-nek (az `y`-ra hasonlóan). A korábbról már ismert hatóköri-elfedési szabályokat felidézve észrevehető, hogy a formális paraméter deklarációja a konstruktor törzsében elfedi a mező deklarációját, így pl. az `x` névvel a konstruktor formális paraméterére hivatkozunk, míg a mezőt csak minősített névvel, `this.x` formában érhetjük el. Ez egy olyan eset, amikor a `this` nem hagyható el, mert jelentést megkülönböztető szerepe van.

Paraméter nélküli konstruktor

```
class Point {
    int x, y;
    Point(){}
}

class Main {
    public static void main( String[] args ){
        Point p = new Point();
        System.out.println(p.x + " " + p.y);   // 0 0
    }
}
```

Egy konstruktornak nem kötelező paramétert várnia. Definiálhatunk *paraméter nélküli konstruktort* is. A fenti példában egy üres törzsű, semmitmondó konstruktort adtunk meg – nyilván azt is lehetővé teszi a nyelv, hogy bonyolult, de nem paraméterezett számítást végrehajtva inicializáljuk az objektumunkat egy ilyen konstruktorban.

Alapértelmezett (default) konstruktor

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

class Main {
    public static void main( String[] args ){
        Point p = new Point();
        System.out.println(p.x + " " + p.y);   // 0 0
    }
}
```

Generálodik egy paraméter nélküli, üres konstruktor

`Point(){}`

Amikor egy osztályba egyáltalán nem írunk konstruktort, akkor automatikusan generálódik egy paraméter nélküli, üres törzsű konstruktor. Emiatt kijelenthetjük, hogy az osztályainkban mindig lesz konstruktor: vagy az, amit mi definiálunk, vagy ez az automatikusan generálódó, úgynevezett default (alapértelmezett) konstruktor. Fontos, hogy ez utóbbi valóban csak akkor jön létre, ha nem írtunk explicit módon konstruktort az osztályhoz.

Ne tévesszük össze a paraméter nélküli és a default konstruktorok fogalmát. A default konstruktor paraméter nélküli, de a paraméter nélküli konstruktor nem feltétlenül a default konstruktor (mint az előző dia mutatta, explicit módon is definiálhatunk paraméter nélküli konstruktort).

Ezen a ponton be kell vallanunk, hogy a default konstruktor nem teljesen üres törzsű: az öröklődés fogalmának tárgyalásakor erre a kérdésre még visszatérünk, de addig is tekintsünk úgy ezekre a default konstruktorokra, mintha üres törzsűek lennének.

2 Információelrejtés

Egységbe zárás

```
class Time {
    int hour;
    int minute;
    Time( int hour, int minute ){
        this.hour = hour;
        this.minute = minute;
    }
    void aMinutePassed(){
        if( minute < 59 ){
            ++minute;
        } else { ... }
    } // (C) Monty Python
}
```

```
Time morning = new Time(6,10);
morning.aMinutePassed();
int hour = morning.hour;
```

Ebben a példában egységbe zárjuk az időt reprezentáló óra és perc adattagokat, egy konstruktort és az aMinutePassed műveletet. Ez utóbbinak a kódja alább olvasható. A használatot szemléltető kódban 6:10-kor felébredünk, még szunyókálunk egy percet, majd megnézzük, hogy még nem múlt el 7 óra. A `Time` osztály ilyen használatta tökéletesen rendeltetésszerű.

```
    void aMinutePassed(){
        if( minute < 59 ){
            ++minute;
        } else {
            minute = 0;
            if( hour < 23 ){
                ++hour;
            } else {
                hour = 0;
            }
        }
    }
```

Típusinvariáns

```
class Time {
    int hour;                          // 0 <= hour < 24
    int minute;                        // 0 <= minute < 60
    Time( int hour, int minute ){
        this.hour = hour;
        this.minute = minute;
    }
    void aMinutePassed(){
        if( minute < 59 ){
            ++minute;
        } else { ... }
    }
}
```

Van itt azonban egy fontos dolog, ami az alkalmazási terület logikájából következik: az `hour` mező értéke 0 és 23, a `minute` mező értéke pedig 0 és 59 közé kell essen: csak azokat a `Time` objektumokat tartjuk értelmes objektumoknak, amelyekre ez a tulajdonság fennáll. Programozáselméletben az ilyen tulajdonságokat típusinvariánsnak (az OOP-ben speciálisan gyakran osztályinvariánsnak) nevezzük. Jól működő programot úgy készíthetünk, ha odafigyelünk arra, hogy a programban használt `Time` objektumokra mindig teljesüljön ez a tulajdonság.

Értelmetlen érték létrehozása

```
class Time {
    int hour;
    int minute;
    Time( int hour, int minute ){
        this.hour = hour;
        this.minute = minute;
    }
    void aMinutePassed(){
        if( minute < 59 ){
            ++minute;
        } else { ... }
```

```
    }
}
```

```
Time morning = new Time(6,10);
morning.aMinutePassed();
int hour = morning.hour;

morning.hour = -1;
morning = new Time(24,-1);
```

A típusinvariáns biztosítása érdekében meg szeretnénk akadályozni, hogy a programba olyan sorok kerüljenek, mint például az, amely az `hour` mezőt `-1`-re állítja, vagy amely egy `Time` objektumot a konstruktor segítségével `-1` perc értékre inicializál.

Létrehozásnál típusinvariáns biztosítása

```
class Time {
    int hour;                          // 0 <= hour < 24
    int minute;                        // 0 <= minute < 60
    Time( int hour, int minute ){
        if (0 <= hour && hour < 24 && 0 <= minute && minute < 60){
            this.hour = hour;
            this.minute = minute;
        }
    }
    void aMinutePassed(){
        if( minute < 59 ){
            ++minute;
        } else { ... }
    }
}
```

Kezdjük először a konstruktorral. Alakítsuk át olyanra, hogy csak akkor végezze el az újonnan létrejött objektum inicializációját, ha a paraméterek értéke megfelelő. És mit tegyünk akkor, ha a paraméterek nem megfelelőek? A fenti megoldás nem lesz jó: ha pusztán csak eltekintünk a paraméterek értékével történő inicializációtól, ez ebben az esetben helyes kezdőértéket eredményez: az óra és a perc is nullára inicializálódik. Mégsem tartjuk ezt jó megoldásnak, ugyanis ebben az esetben egyszerűen ignoráltuk a hibás konstruktorhívást, így a hiba észrevétlen marad.

Azt, amikor elsikkad a hiba, mert ahelyett, hogy a hibát detektáló programrész jelezte volna, inkább ignorálta, és egy alapértelmezett egyéb tevékenységgel helyettesítette, szokás *silent faiulure*-nek is nevezni. A név magáért beszél…

Kerüljük el a „silent failure'' jelenséget

```
class Time {
    int hour;                          // 0 <= hour < 24
    int minute;                        // 0 <= minute < 60
    Time( int hour, int minute ){
        if (0 <= hour && hour < 24 && 0 <= minute && minute < 60){
            this.hour = hour;
            this.minute = minute;
        } else {
```

```
            throw new IllegalArgumentException("Invalid time!");
        }
    }
    void aMinutePassed(){
        ...
    }
}
```

A megoldás tehát a hiba jelzése. Ez történhetne úgy, hogy kiírunk egy hibaüzenetet a képernyőre – de mi van akkor, ha a kódunk nem olyan környezetben fut majd, hogy a felhasználó látja a képernyőt, vagy ha lát is egy felhasználói felületet, az a felület nem jelzi ki a szabványos kimenetre (Javában `System.out`) kiírt üzeneteket? És egyáltalán, milyen alapon írna ki bármit is a szabványos kimenetre a szóban forgó kódrészlet? Nem az ő dolga, és esetleg ezzel a kiírással össze is zavarhatja a teljes program kimenetét. Egy programban a kimenetre történő kiírás, illetve a felhasználói felület kezelése egy jól körülhatárolható programkomponens feladata kell legyen. Ez biztosítja az átláthatóságot, és persze azt is, hogy a programkomponens cseréjével könnyen lecserélhessük a szoftver felhasználói felületét (például egy szöveges felületről egy webes felületre).

Azt mondhatjuk hát, hogy egy kiírás a szabványos kimenetre nem egy igazán jó módja a hibák jelzésének. Egy fokkal jobb megoldás az, ha a szabványos hibakimenetre írunk (Javában `System.err`). Professzionális alkalmazásokat úgy szoktak elkészíteni, hogy a hibákat, figyelmeztetéseket és nyomkövetési célú rendszerüzeneteket egy úgynevezett *logger* programkomponensen keresztül írják ki, ez a megoldás sokkal rugalmasabb, konfigurálhatóbb, testre szabhatóbb a szabványos hibakimenetre történő kiírásnál. De még ebben az esetben is igaz lehet az, hogy nem kívánunk minden programkomponensbe kiíró utasításokat tenni. És különben is, miután kiírtuk, hogy hiba van, hogyan tovább? Két lehetőség is lenne: az egyik, hogy elfogadunk egy helyettesítő megoldást, és megpróbáljuk ezzel folytatni a működést (mint a példánkban a `0`, `0` értékekkel történő `Time` inicializálás), vagy megszakítjuk a program további futását.

A programozási nyelvekben meghonosodott egy olyan technika, illetve nyelvi elem, amellyel ez a kérdéskör egész jól kezelhetővé válik. Ahogy a fenti példában látjuk, abban az esetben, amikor a paraméterek nem megfelelőek, és ezért nem lehet elvégezni velük az objektumunk inicializálását, *kiváltunk egy kivételt* (mégpedig Javában ilyen esetben jellemzően a szabványos programkönyvtárban definiált `IllegalArgumentException` kivételt), amely a konstruktort hívő programrésznek jelzi, hogy hiba történt.

Kivétel

- Futás közben lép fel
- Problémát jelezhetünk vele
  - `throw` utasítás
- Jelezhet „dinamikus szemantikai hibát''
- Program leállását eredményezheti
- Lekezelhető a programban
  - `try`–`catch` utasítás

A kivételek azt jelzik, hogy a program végrehajtása során kivételes esemény lépett fel, valami elromlott, valami hiba történt, valami olyan dolog következett be, amely eltéríti a programot a normális végrehajtási menettől. A programozó maga is jelezhet problémát kivételekkel, mint ahogy a példánkban láttuk. A szabványos programkönyvtárak is számos esetben kivétellel jelzik azt, ha valami nem az elvártaknak megfelelően működik. Sőt, maga a nyelv is úgy van kitalálva, hogy bizonyos szemantikai hibákat kivételek formájában jelez.

Egy programozási nyelv definícióját a nyelv szabályai adják meg. A szabályok vonatkozhatnak a programban használható elemi szimbólumok formájára (lexikális szabályok, a *tokeneket* írják le), a szimbólumokból való építkezés módjára (szintaktikus szabályok, az összetett struktúrák szerkezetét írják le), valamint arra, hogy milyen jól felépített struktúrákat tartunk értelmesnek (szemantikus szabályok). Ezen utóbbi szabályok szoktak a legbonyolultabbak lenni. Ebbe a kategóriába tartoznak a hatóköri szabályok és a jóltípusozottság is. A Java nyelv fordítóprogramja a lexikális és szintaktikus szabályok mellett a szemantikus szabályok nagy részének betartását is ellenőrzi, mielőtt lefordítaná progra-

munkat. A fordítási időben ellenőrzött szemantikai szabályokat szokás a nyelv *statikus szemantikai szabályainak* is nevezni. Vannak olyan szemantikai szabályok, melyeknek való megfelelést azonban nem lehet hatékonyan ellenőrizni fordítási időben. A Java nyelvben ilyen például az a szabály, hogy egy tömböt csak olyan számmal indexelhetünk, amely nullánál nem kisebb, de a tömb méreténél kisebb. Ez egy olyan szabály, amelyet nehéz lenne teljes általánosságában már fordítási időben ellenőrizni, hiszen egy tömb indexeléséhez használt kifejezés értékének meghatározása általában nem végezhető el fordítási időben. Futás közben persze kiderül, mert a program kiszámolja, az indexeléshez használt kifejezés értékét, és ekkor elvégezhető a fenti szabály ellenőrzése. Azokat a nyelvi szabályokat, amelyeket fordítási időben nem lehet (vagy nem érdemes) ellenőrizni, csak futás közben, szokás a nyelv *dinamikus szemantikai szabályainak* nevezni. A futás közben kiderített szemantikai hibákat („értelmetlenség'' a programban) a nyelv szintén kivétel kiváltásával jelezheti. Például a Java nyelvben egy tömb hibás indexelése `ArrayIndexOutOfBoundsException` kivételt eredményez. Ez a programozó szemszögéből nézve sokkal kényelmesebb, mint a C nyelv megközelítése, melyben a tömbök hibás indexelése egyszerűen „undefined behaviour''-t eredményez, azaz nem feltétlenül derül ki, hogy értelmetlen a program, illetve misztikusnak tűnő további hibákat okozhat az észre nem vett hiba.

A kivételeket – a fenti példa alapján – a `throw`-utasítással lehet kiváltani a Java nyelvben. Angolul a *raise an exception*, valamint Java esetén gyakran a *throw an exception* kifejezéseket használjuk.

Amikor egy meghívott metódusban fellép egy kivétel, a metódus végrehajtása megszakad, és a vezérlés visszakerül a hívóba. A hívó dönthet úgy, hogy *lekezeli a kivételt* (angolul *handle the exception* vagy *catch the exception*), ez a Javában a try-catch utasítással történhet. Erről az utasításról később beszélünk majd. Ha a kivételt nem kezeljük le, az tovább terjed a hívási lánc mentén, azaz a hívó hívójában, majd az azt hívóba, és így tovább, amíg el nem érjük a főprogramot (a legelsőként meghívott, `main` metódust). Ha ez sem kezeli le a kivételt, a program végrehajtása megszakad, és egy hibaüzenet íródik ki a szabványos hibakimenetre.

Futási hiba

```
class Main {
    public static void main( String[] args ){
        Time morning = new Time(24,-1);
    }
}
```

```
$ javac Time.java
$ javac Main.java
$ java Main
Exception in thread "main" java.lang.IllegalArgumentException:
Invalid time!
    at Time.<init>(Time.java:9)
    at Main.main(Main.java:3)
$
```

A programunk ebben az esetben nem kezeli le a konstruktorhívás során fellépő `IllegalArgumentException` kivételt, így a program futási hibával leáll. A kiírt futásihiba-jelzésben hasznos információkat találunk a hiba fajtájáról (`IllegalArgumentException`), esetleges hibaüzenetről (az általunk megadott „Invalid time!'' üzenet), a kivétel fellépésének pontjáról (a `Time.java` forrásfájl 9. sorában, a `Time` osztály konstruktorában), valamint a kivétel terjedéséről (a kivételt kiváltó konstruktort a `Main.java` forrásfájl 3. sorában hívta meg a `Main` osztály `main` nevű metódusa).

2.1 private

A mezők közvetlenül manipulálhatók

```
class Time {
    int hour;                          // 0 <= hour < 24
    int minute;                        // 0 <= minute < 60
    ...
}
```

```
class Main {
    public static void main( String[] args ){
        Time morning = new Time(6,10);
        morning.aMinutePassed();

        morning.hour = -1;             // ajjaj!
    }
}
```

A konstruktor rendbe szedése után térjünk rá a másik problémánkra. A `Time` objektumok inicializációja garantáltan értelmes objektumokat eredményez már, de ezután egy sima értékadással elromolhat a típusinvariáns.

A megoldás erre a problémára az, hogy az objektumon kívülről nem engedjük meg a mezőkre vonatkozó értékadást, csak az osztálydefiníción belülről. Ez azért jó megoldás, mert így csak az osztálydefiníció elkészítése során kell különös gondossággal eljárni, hogy ne kerüljön bele hibás értékadás – az összes többi osztály készítése során biztosak lehetünk abban, hogy nem fogjuk a `Time` típusinvariánsát megsérteni.

Mező elrejtése: `private`

```
class Time {
    private int hour;                  // 0 <= hour < 24
    private int minute;                // 0 <= minute < 60
    ...
}
```

```
class Main {
    public static void main( String[] args ){
        Time morning = new Time(6,10);
        morning.aMinutePassed();

        morning.hour = -1;             // fordítási hiba
    }
}
```

A `private` kulcsszóval ellátott deklarációk csak a tartalmazó osztálydefiníció számára láthatók. Azzal, hogy az `hour` és `minute` mezőket priváttá tettük, elértük azt, hogy a többi osztályban a két mezőre vonatkozó hivatkozás fordítási hibát okozzon. A két mező kikerült a `Time` objektumok interfészéből, és a belső működés részévé, a külvilág elől elrejtett információvá vált.

Igen ám, de akkor hogyan lehet megtudni, vagy akár megváltoztatni a `Time` objektumokban az óra és perc értékeket? Természetesen biztonságosra megírt metódusok segítségével!

Idióma: privát állapot csak műveleteken keresztül

```
class Time {
    private int hour;                   // 0 <= hour < 24
    private int minute;                 // 0 <= minute < 60
    Time( int hour, int minute ){ ... }
    int getHour(){ return hour; }
    int getMinute(){ return minute; }
    void setHour( int hour ){
        if( 0 <= hour && hour <= 23 ){
            this.hour = hour;
        } else {
            throw new IllegalArgumentException("Invalid hour!");
        }
    }
    void setMinute( int minute ){ ... }
    void aMinutePassed(){ ... }
}
```

Az OOP-ban egy közismert idióma, hogy az objektum állapotát leíró mezőket elrejtjük a külvilág elől, és a hozzáférést metódusokba csomagoljuk. Készíthetünk a két adattaghoz lekérdező és beállító műveleteket, és a beállító műveletekben ugyanúgy gondoskodhatunk a típusinvariáns megtartásáról, mint a konstruktorokban. A `setHour` és a `setMinute` metódusok ellenőrzik a kapott paramétereket, és értelmetlen értékek esetén kivételt váltanak ki.

Getter-setter konvenció

Lekérdező és beállító művelet neve

```
class Time {
    private int hour;                   // 0 <= hour < 24
    int getHour(){ return hour; }
    void setHour( int hour ){
        if( 0 <= hour && hour <= 23 ){
            this.hour = hour;
        } else {
            throw new IllegalArgumentException("Invalid hour!");
        }
    }
    ...
}
```

A Java nyelv (és több más nyelv) kapcsán kialakult konvenció az, hogy a fenti idióma szerint elkészített lekérdező és beállító műveletek nevét a szóban forgó attribútum nevéből képezzük a `get-`, illetve `set-` előtag hozzáadásával, természetesen alkalmazva a *camel-case* szabályait (azaz az attribútum nevének első betűjét nagybetűre alakítjuk). Így nyeri el a metódusnév a Java elnevezési konvenciónak megfelelő alakot (minden, az azonosítóban szereplő szót nagybetűvel kezdünk és kisbetűkkel folytatunk – kivéve a legelső szót, mert az kisbetűvel is kezdjük).

A konvenció kiterjed a *setter*-metódus paraméterének elnevezésére is: a paraméter neve megegyezik a beállítandó attribútum nevével. Ebből az is következik, hogy az elfedés miatt az attribútumra minősített névvel kell hivatkoznunk a setter-metóduson belül.

Ez a getter-setter konvencióval elkészített, a privát mezőket lekérdező és beállító metódusokon keresztül elérhetővé tévő megoldás nagyon jellemző a Java programokra. A nyelv egyszerűsége itt hátrányként tűnik fel, mert egy osztály elkészítésénél ezen struktúra kialakítása egy szükséges, rutinszerű tevékenységgé válik. Annyira közhelyes és sablonos szerkezetről van szó, hogy elegánsabb lenne csak annyit mondani, hogy „ezek és ezek a mezők vannak, természetesen a szokásos getterekkel és setterekkel ellátva’’. Angolul ezt úgy mondjuk, hogy a Java osztálydefiníciók sok *boiler-plate* kódot tartalmaznak.

Reprezentáció változtatása

```
class Time {
    private short minutes;
    Time( int hour, int minute ){
        if (0 <= hour && hour < 24 && 0 <= minute && minute < 60){
            minutes = 60*hour + minute;
        } else {
            throw new IllegalArgumentException("Invalid time!");
        }
    }
    int getHour(){ return minutes / 60; }
    int getMinute(){ return minutes % 60; }
    void setHour( int hour ){
        if( 0 <= hour && hour <= 23 ){
            minutes = 60 * hour + getMinute();
        } else {
            throw new IllegalArgumentException("Invalid hour!");
        }
    }
    void setMinute( int minute ){ ... }
    void aMinutePassed(){ ... }
}
```

A külvilág elől elrejtett mezők egy pozitív hatása az is, hogy az osztály megvalósítását (például a belső állapot leírására szolgáló adatstruktúra szerkezetét) könnyebb megváltoztatni így, hogy az osztálydefiníción kívül ezekre az adatokra nem lehet hivatkozni. Egy szoftver életciklusa során egy-egy osztály definíciója keresztül szokott menni egy-egy változáson. Akár hatékonysági okokból, akár azért, mert egy új funkciót építünk be a kódba, egy osztály belső felépítését meg szeretnénk változtatni. Persze ezt akkor könnyű megtenni, ha a változás hatása nem terjed túlságosan messzire – ideális esetben csak magára a megváltoztatott osztályra. Ha az osztály interfészében történik változás, az nyilván minden olyan osztályra kihatással van, amely a megváltozott osztállyal kapcsolatban áll, azt használja. Ha a változás a privát dolgokra, illetve a metódusok belsejére szorítkozik, a nyelv szabályai értelmében biztosak lehetünk abban, hogy az összes többi osztály ebből a változásból „semmit nem vesz észre''. Sőt, még csak újra sem kell fordítani a többi osztályt: elegendő azt, amelyiket megváltoztattuk.

A `Time` osztály kapcsán dönthetünk például úgy, hogy a korábbi, két `int` típusú mezőt egy darab, `short` típusúval helyettesítjük oly módon, hogy az aktuális napból eddig eltelt percek számát tároljuk. (Ez az érték kényelmesen elfér egy `short`-ban, hiszen ez a típus a Javában egy 16 biten tárolt előjeles egész szám. Vegyük észre, hogy az adott napban eltelt másodpercek száma is biztonsággal ábrázolható ezzel a típussal.)

A két privát mezőt lecseréljük egy másikra, és természetesen módosítjuk a konstruktor, valamint a getter-setter és egyéb metódusok törzsét, de kintről nézve mindezek a módosítások észrevétlenek maradnak.

Információ elrejtése

- Osztályhoz szűk interfész
  - Ez „látszik'' más osztályokból
  - A lehető legkevesebb kapcsolat
- Priváttá tett implementációs részletek
  - Segédműveletek
  - Mezők

Előnyök

- Típusinvariáns megőrzése könnyebb
- Kód könnyebb evolúciója (reprezentációváltás)
- Kevesebb kapcsolat, kisebb komplexitás

Összefoglalva az információ elrejtésének elvét: egy osztály definiálásakor meghatározzuk, hogy milyen információkra van szüksége a többi osztálynak, ezek alkotják az új osztály interfészét. Minél szűkebb ez az interfész, annál kevesebbféle kapcsolat alakulhat ki más osztályokkal. A sokféle kapcsolat a kód komplexitását (bonyolultságát) növeli, így a kód érthetőségét (és ebből kifolyólag karbantarthatóságát is) csökkenti, azaz negatív hatású. Az osztályok interfészének kialakításakor minimalistának kell lennünk: a lehető legszűkebb interfészt érdemes kialakítani, amivel a megoldandó feladatokat még kezelni tudjuk. Legyen benn minden, ami kell, de semmi felesleges. Amit lehet, rejtsünk el a külvilág elől. Az osztály mezőit, a segédműveleteket érdemes lehet priváttá tenni.