A runtime alrendszer

Az *Objective-C* nyelv rengeteg dolgot nem fordítási időben, hanem futtatáskor dönt el és amikor csak lehet, dinamikusan teszi azt. Ez azt jelenti, hogy nem csak fordítóra, hanem runtime-ra is szükség van *Objective-C* kód futtatásához. A runtime egyfajta „operációs rendszerként" működik a nyelv számára.

Objective-C program három különféle szinten kommunikál a runtime-mal.

1. A nyelv forráskódján keresztül. A runtime a legnagyobb részben automatikusan és a háttérben működik.
2. A *Foundation framework*-ben található NSObject metódushívásain keresztül. A legtöbb *Cocoa* objektum NSObject származék, melyek így öröklik annak metódusait is. Néhány ilyen segítségével információkat kérhetünk a runtime-tól:
    a. `[anObj class]` - az anObj objektum osztályát adja vissza.
    b. `[anObj isKindOfClass: aClass]` – aClass ősosztálya-e anObj objektumnak?
    c. `[anObj isMemberOfClass: aClass]` – anObj példánya-e aClass osztálynak?
    d. `[anObj respondsToSelector: aSel]` – létezik-e anObj objektumnak aSel metódusa?
    e. `[anObj conformsToProtocol: aProtocol]` – megvalósítja-e anObj az aProtocol interfészt?
3. Direkt hívásokkal. A runtime egy dinamikus könyvtár publikus interfésszel, amely rendelkezik jópár függvénnyel és adatstruktúrával (leírásuk a `/usr/include/objc` mappában található meg)..

Az NSObject a *Java* nyelv `getString()` metódusához hasonlóan rendelkezik egy, a példány szöveges leírását visszaadó description metódussal. Ennek segítségével például program hibakövetésekor a `gdb`-ben a `print-object` paranccsal szöveges leírást kapunk az adott példányról egy esetleges pointer skalár helyett.

Az NSObject alábbi fontos szolgáltatásokat nyújta még:

- Új példányok allokálása és inicializálása, ill. példány deallokálása, ha már nincs többet rá szükség
- Üzenettovábbítás más példányok felé
- Új modulok dinamikus betöltése futás közben

Memóriakezelés

A *Cocoa* – a Javához hasonlóan – **referencia számlálásos** memóriakezelési technikát használ. Egy létrehozott objektum addig él, amíg referenciaszámlálója nagyobb nullánál, azaz más példányok birtokolják. Ha ez lenullázódik, a szemétgyűjtő eltakarítja a feleslegessé vált objektumot.

Új példány allokálása és inicializálása

Ez *Objective-C*-ben két lépésben történik:

- Dinamikus memóriafoglalás az új példány számára (`alloc`).
- A példány inicializálása (`init`).

Megjegyzés: az *Objective-C* nem ismeri a *C++* és *Java* nyelvekben megszokott konstruktorokat, helyette van az `alloc` osztálymetódus, ill. az `init` példánymetódus.

Egy tipikus példány létrehozás:

```
id anObject = [[Rectangle alloc] init];
```

A konstrukció `alloc` és `init`-re való szétválasztása lehetőséget nyújt példány létrehozásának finomítására, mint például új NSString példány létrehozását C stringből:

```
id aString = [[NSString alloc] initWithCString: str length: len];
```

Mind az `alloc`-nak, mint az `init`-nek lehetősége van sikertelen allokálás (ez nem tipikus), ill. inicializálás esetén `nil`-t („üres” példányt) visszaadnia. Ez fontos különbség a *C++*-szal és a *Javával* szemben, ahol konstrukciót csak egy kivétel kiváltásával tudjuk megszakítani. Egy példa erre:

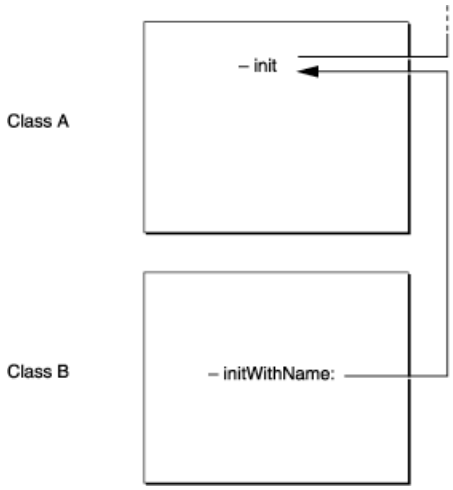
```
id anObject = [[SomeClass alloc] init];
if ( anObject )
    [anObject someOtherMessage];
else
    ...
```

Alosztály példányának inicializálása

Az `init` végrehajtása során tekintettel kell lennünk az ősosztályok által deklarált példányváltozók inicializálására is. Mivel *Objective-C*-ben ez nem automatikus (mivel nincs konstruktor), ezt nekünk kell a `[super init…]` hívással megtennünk.

```
- initWithName:(char *)string
{
    if ( self = [super init] ) {
        name = (char *)NSZoneMalloc([self zone],
            strlen(string) + 1);
        strcpy(name, string);
        return self;
    }
    return nil;
}
```

A `[super init …]` segítségével láncoljuk össze a példányinicializációt az gyökérosztálytól egészen a kiinduló alosztályig.



Példány tulajdonlása

Egy *Objective-C* programban objektumok keletkeznek és dobódnak el. Az esetek legtöbb részében egy objektum saját célra hoz létre más példányokat és miután már nincs rájuk szükség, megszabadul tőlük. Azonban ha objektum átad objektumot egy másiknak, akkor annak birtoklása – ezáltal a felelősség – kérdése elmosódik. Ki birtokolja a példányt? Kinek a felelőssége felszabadítani azt, ha már nincs rá többet szükség?

Vegyük példaként az alábbit. Legyen egy osztályunk és annak egy példánymetódusa, ami `Sprocket` osztály példányainak tömbjét adja vissza.

```
- (NSArray *) sprockets
```

Ez a deklaráció nem állít semmit, kinek kell majd felszabadítania a visszaadott tömböt. Ha ez a tömb osztályunk példányváltozója, mi felelünk érte, de ha a metódus hozta létre és adja át másnak a tömböt, akkor az felel érte, aki kapja. Ideális esetben ezzel a kérdéssel nem kellene találkoznunk. Ezért a *Cocoa* az alábbit követeli meg:

Ha

1. Létrehozunk `alloc`-kal egy példányt
2. Klónozunk `copy`-val egy példányt
3. Vagy `retain`-nel fogunk példányt, azaz explicite birtokoljuk

Akkor mi felelünk érte és nekünk kell azt felszabadítanunk `release`-zel.
Ha azonban nem mi kreáltuk vagy klónoztuk a példány a fent megadott módokon, akkor ne birtokoljuk azt és főleg ne szabadítsuk fel!

Példány megtartása

Időnként szeretnénk, ha egy megkapott objektum nem dobódna el használata után azonnal – például eltároljuk azt egy példányváltozóban – azaz továbbra is igényt tartunk rá. Ekkor csak mi tudjuk, meddig van rá pontosan szükségünk. Hogy ezt kifejezhessük, használjuk az objektum `retain` metódusát (megjegyzés: ez felel meg pontosan a nyelvben a referencia számláló növelésének).

Ha objektumunk lehetőséget nyújt arra, hogy a fő `Sprocket` objektumot kicserélhessünk egy másikra (vagy üresre állítjuk), az a következőképpen nézhet ki:

```
- (void)setMainSprocket:(Sprocket *)newSprocket
{
    [mainSprocket autorelease];
    mainSprocket = [newSprocket retain]; /* Claim the new Sprocket. */
    return;
}
```

Egy apró hiba azonban van ezzel. Mi van ha a `newSprocket` megegyezik a `mainSprocket`-tel? Első lépésben közöljük az objektummal, hogy rá már nincs szükség, majd utána mondanánk rá egy *megtartást*. Ha az első lépést követően a szemétgyűjtő eltakarítja az objektumot, a második lépés minden bizonnyal kivételt vált ki. Apró módosítással így néz ki helyesen ez a metódus:

```
- (void)setMainSprocket:(Sprocket *)newSprocket

{
    Sprocket *oldSprocket = mainSprocket;
    mainSprocket = [newSprocket retain];
    [oldSprocket release];
    return;
}
```

Deallokálás (példány felszabadítása)

Az NSObject osztály `dealloc` metódusa végzi el a példány dekonstrukcióját. Ennek meghívásával felszabadul az objektum memóriarekesze, a példány érvénytelenné válik. A `dealloc` metódust sohasem hívjuk közvetlenül, helyette a `release` vagy `autorelease` hívásokkal értesítjük a szemétgyűjtőt, hogy viheti a példányt (a `dealloc`-ot legfeljebb felüldefiniáljuk egy sajáttal, ahol az épp felszabadítandó objektum példányváltozóival hivatkozott objektumoknak üzenhetünk, végük). Ezek hívása csökkenti a példány referencia számlálóját, amely ha eléri a zérust, a példányt a szemétgyűjtő eltávolítja. A deallokálás szabályai:

- Ha egy példányt mi hoztunk létre `alloc`-kal vagy `copy`-val másoltuk, illetve `retain`-nel megtartottuk, `release`-zel kell felszabadítanunk, ha nincs többet már rá szükség.
- Ha valamely más módon jutottunk a példányhoz, tilos `release`-elnünk (hisz ki tudja, épp hányan hivatkoznak még erre a példányra?), hacsak a programblokkunkon belül nem hívtunk meg rá `retain`-t.
- Az `autorelease` ugyanaz mint a `release`, csak késleltetett és nem azonnali eltávolítást jelent.

Példányváltozók felszabadítása

A `dealloc` metódus célja, hogy felszabadítson minden, a példány birtokolta memóriarekeszt. Alapesetben az NSObject csak magát az objektumot szabadítja fel, de az általunk ebből származtatott alosztály példányváltozóiban megtartott példányok felszabadításáért mi felelünk. Egy példa saját `dealloc` metódusra:

```
- dealloc {
    [companion release];
    free(privateMemory);
    vm_deallocate(task_self(), sharedMemory, memorySize);
    [super dealloc];
}
```

Az első sor egy példányváltozó felszabadítása, a második saját memóriaterület felszabadítása (megjegyzés: ez a *Cocoa* környezetben nem szokványos. Általános memóriaterület foglalására az `NSData` osztályt használjuk), a harmadik megosztott memória felszabadítása. A negyedik sor végzi el az alapesetet, magát az objektumot szabadítja fel.

Objektumok megjelölése késleltetett deallokálással

Előfordul olyan eset, hogy átadunk egy általunk létrehozott példányt, de mi az átadás előtt le kívánunk mondani róla, nekünk többé az már nem kell. Ha az objektum átadása előtt `release`-et mondanánk rá, könnyen előfordulhat, hogy aki megkapta a példány referenciáját, az már érvénytelen lesz addigra. Erre az esetre jó az `autorelease`, amikor ugyan mi jelezzük vele, hogy nekünk már nem kell, de a szemétgyűjtő nem azonnal takarítja el a felszabadítandó objektumot, hanem egy távolabbi időpontban. Ez biztosít lehetőséget számunkra, hogy a fogadó megkapva a referenciát azonnal kimondhasson egy `retain`-t, ezzel átvéve a példány feletti birtoklás felelősségét. Egy példa erre:

```
- (NSArray *)sprockets
{
    NSArray *array;
    array = [[NSArray alloc] initWithObjects:mainSprocket
                                auxiliarySprocket, nil];
    return [array autorelease];
}
```

Üzenettovábbítás

Olyan üzenetet küldeni egy objektumnak, amely azt nem ismeri – hiba. Azonban a hiba megállapítása előtt a runtime a fogadó objektumnak ad egy második lehetőséget az üzenet kezelésére: üzen az objektumnak a `forwardInvocation:` üzenettel, ahol az egyetlen paraméter az eredeti üzenet `NSInvocation` példányba csomagolva (megj.: az `NSInvocation` szolgál a statikus metódus szignatúra dinamikus helyettesítésére).

Ha implementáljuk a `forwardInvocation:` metódust, ez lesz az alapértelmezett kezelője az ismeretlen üzeneteknek és nem lesz automatikus hiba kiváltás. A metódus nevében jelzi, ezt üzenettovábbításra használatos.

Hogy közelebbről szemügyre vehessük, mire is jó ez, tekintsük a következő példát. Tegyük fel, hogy van egy objektumunk, amely a `negotiate:` metódusra válaszolva beleveszi egy másik hasonló objektum válaszát. Ezt egyszerűen meg lehet tenni, ha először annak a `negotiate:` metódusát hívjuk meg sajátunkból valahol.

Továbbá azt is szeretnénk, hogy objektumunk válasza erre az üzenetre pontosan a másik objektum válasza legyen. Ennek egyik módja a metódus öröklése a másik objektum osztályából. Erre azonban nem mindig van lehetőség. De miért is ne lehetne a két objektum osztálya más öröklődési hierarchiában?

Annak ellenére, hogy esetleg objektumunk nem rendelkezik örökölt `negotiate:` metódussal, még „kölcsönözhet” egyet magának a másik példánytól, egyszerűen továbbítva neki a kapott üzenetet:

```
- negotiate
{
    if ( [someOtherObject respondsTo:@selector(negotiate)] )
        return [someOtherObject negotiate];
    return self;
}
```

Azonban ez a megoldás kissé nehézkes, különösen akkor, ha számos üzenetetet kellene továbbítani más objektumoknak. Minden metódust külön implementálni kellene, amit továbbítani szeretnénk. Ezen felül nem biztos, hogy minden esetről előre tudunk a kód megírása idején.

Erre ajánl kevésbé ötletszerű dinamikus alternatívát a forwardInvocation: üzenet: ha a másik objektum nem válaszol a kérdéses üzenetre (azaz nem rendelkezik a megfelelő metódussal), a runtime automatikusan üzen a `forwardInvocation:` metódusnak. Minden objektum örökli ezt az `NSObject` ősosztálytól, ami egyszerűen meghívja a `doesNotRecognizeSelector:` metódust. A `forwardInvocation:` metódust felülírva megírhatjuk saját üzenettovábbítónkat.

Üzenettovábbításhoz két dolgot kell tenni:

- Meghatározni, mely üzeneteket kívánunk továbbítani
- Majd elküldeni azt az eredeti paraméterekkel

Üzenetet az `invokeWithTarget:` metódus segítségével küldhetünk:

```
- (void)forwardInvocation:(NSInvocation *)anInvocation
{
    if ([someOtherObject respondsToSelector:
            [anInvocation selector]])
        [anInvocation invokeWithTarget:someOtherObject];
    else
        [super forwardInvocation:anInvocation];
}
```

A továbbított üzenet visszatérő értéke az `anInvocation` üzenet objektumban tárolva kerül vissza az eredeti hívóhoz.

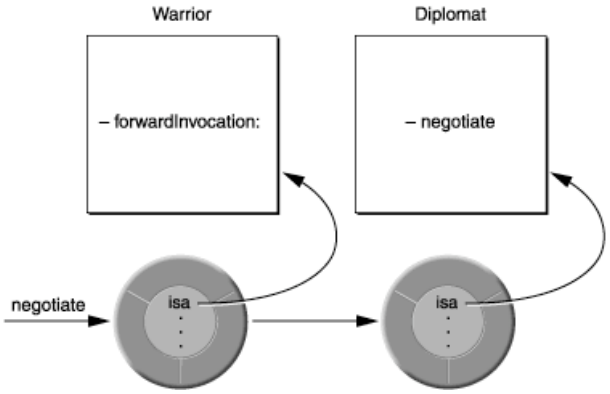
A `forwardInvocation:` metódus gyakorlatilag egy elosztó központhoz hasonlítható, ahol az ismeretlen üzenetek szétválogatásával és a megfelelő címzetthez továbbításával foglalkoznak. De ez lehet egy szimpla üzenetátadó egy másik objektum (vagy objektumok) részére vagy egyszerűen egy „elnyelő”, mely minden hibajelzés nélkül elnyeli a kapott üzeneteket.

Megjegyzés: a `forwardInvocation:` metódus csak akkor kapja meg az üzenetet, ha a névleges fogadó nem kezeli azt le. Azaz, ha a `negotiate:` metódust továbbítani szeretnénk, akkor ennek az objektumnak nem szabad rendelkeznie ilyennel deklaráltan.

Üzenettovábbítással kapcsolatos további információkat az `NSInvocation` osztály leírásában talál a *Foundation framework* referenciájában.

Üzenettovábbítás és az öröklődési lánc

Üzenetek továbbítása egy kicsit az öröklődést utánozza, alkalmas lehet a többszörös öröklődés „szimulálására” is. Az alábbi ábrán látható módon az üzenetet fogadó objektum megvizsgálja az üzenet típusát és a megválaszolásra hivatott objektumnak küldi tovább.



Az illusztációban a `Warrior` (harcos) osztály egy példánya továbbítja a `negotiate` (tárgyal) üzenetet egy `Diplomat` (diplomata) példánynak. Úgy tűnik, a `Warrior` úgy tárgyal, mint egy `Diplomat`, láthatóan válaszol a `negotiate` üzenetre (habár valójában a `Diplomat` teszi ezt meg).

Az objektum, amely továbbítja az üzenetet tehát „örököl” metódusokat két öröklési ágból, a sajátjából és a másik ágból, ahol válaszolnak a fenti üzenetre. Úgy tűnik tehát, hogy a `Warrior` osztály örököl a `Diplomat`-ból is, nem csak a sajátjából.

Objektumok helyettesítése

Az üzenettovábbítás technikája alkalmas olyan „könnyűsúlyú” (lightweight) fedőobjektumok fejlesztésére is, amelyek eltakarnak más, kritikusabb vagy távoli objektumokat.

Stróman (proxy) objektum használata távoli üzenetküldés esetén is egy ilyen helyettesítés. A stróman felel az adminisztratív tevékenységért, az átvett paraméterek átmásolását és a visszatérő érték fogadását a kapcsolaton keresztül, stb. Ez azonban nem haladhatja meg az eredeti funkcionalitást, pusztán csak egy távoli objektumnak biztosít helyben teret.

Másfajta helyettesítések is lehetségesek. Tegyük fel, hogy van egy olyan objektumunk, amely sok adattal dolgozik, pl. egy összetett képpel, egy nagyobb fájl vagy diszk tartalmával. Egy ilyen objektum használata bizony idő és erőforrásigényes lehet, egyszerűbb ugyanezt lazábban tenni, csak akkor, amikor ténylegesen szükségünk van rá, vagy van a programnak „üres” ideje az időrabló tevékenységre. Szükség lesz egy „helykitöltő” (placeholder) objektumra, ill. a valódi tevékenységet végző egy vagy több továbbira.

Ilyen esetben először csak a fedőobjektumot hozzuk létre, melynek lehetnek saját feladatai is, pl. eláruk a beolvasandó adatról információkat (méret, előrelátható olvasási idő, stb.), de többnyire csak a helyet foglalja a *tényleges* objektumnak és amikor annak az ideje eljő, továbbítja az érkező üzeneteket afele. Ha a fedőobjektum kap egy üzenetet a `forwardInvocation:` metóduson keresztül, ellenőrzi a fedettet, hogy az létezik-e már, beolvasta-e az adatokat, stb. Ebben az esetben minden üzenetet továbbít úgy viselkedve, mintha tényleg ő lenne az elfedett objektum.

Üzenettovábbítás és öröklődés

Habár az üzenettovábbítás az öröklődés utánzása, az `NSObject` osztály nem keveri össze a kettőt. A `respondsToSelector:`, `isKindOfClass:` és hasonló metódusok csak az öröklődési láncot nézik végig, sose a továbbításit. Ha a fent említett példában a `Warrior` osztályt megkérdezzük, válaszol-e a `negotiate`-re,

```
if ( [aWarrior respondsToSelector:@selector(negotiate)] )
    ...
```

akkor a válasz `NO` lenne, annak ellenére, hogy ténylegesen válaszol rá.

Sok esetben a helyes válasz ilyenkor a `NO`, de néha nem az. Ha az üzenettovábbítás technikáját objektum helyettesítésre vagy funkcióbővítésre használjuk, ennek ugyanolyan transzparensnek kellene lennie, mint az öröklődésnek. Ha ez az eset áll fent és szeretnénk, ha az üzenettovábbítási technikánk úgy viselkedjen, mintha valóban öröklődést alkalmaznánk, akkor a `respondToSelector:` metódust definiáljuk felül és adjuk hozzá a továbbítással kapcsolatos részt:

```
- (BOOL)respondsToSelector:(SEL)aSelector
{
    if ( [super respondsToSelector:aSelector] )
        return YES;
    else {
        /* Here, test whether the aSelector message can     *
         * be forwarded to another object and whether that  *
         * object can respond to it. Return YES if it can.  */
    }
    return NO;
}
```

Itt az else ágba írjuk meg a kiegészítést.

A `respondsToSelector:` és az `isKindOfClass:` metódusokon túl az `instancesRespondToSelector:`-t is szükség lehet kibővíteni. Amennyiben protokollokat is használunk, úgy vegyük ide még a `conformsToProtocol:` metódust is. Hasonló módon, ha továbbítunk üzeneteket távoli objektumoknak, akkor a `methodSignatureForSelector:` átdefiniálását is el kell végeznünk.

Dinamikus betöltés

Objective-C program – hasonlóan a *Java*hoz - képes futásidőben betölteni és becsatolni osztályokat és kategóriákat. Az így betöltött kód beépül a futó kódba és ugyanúgy viselkedik, mintha a program indítása óta ott lett volna.

Dinamikus betöltés sokmindenre használható. Például a *Mac OS X* operációs rendszer vezérlőpultjának (System Preferences) elemei is dinamikusan töltödnek be – akkor, amikor valóban szükség van rájuk.

A *Cocoa* környezetben a dinamikus betöltés gyakran használják alkalmazások specializálására. Mások modulokat írnak ilyen módon, melyeket program tölt be futásidőben.

Habár dinamikus betöltésre az *Objective-C* runtime-nak megvan a saját eszközkészlete (lásd az `objc/objc-load.h` állományban), erre van egy sokkal kényelmesebb, objektum-orientált környezetbe integrált lehetőség, az `NSBundle` osztály és szolgáltatásai.

Távoli üzenetváltás

Mint sok más programozási nyev, az *Objective-C* nyelvet is úgy tervezték eredetileg, hogy egy processzben fusson.

Mindemellett az objektum-orientált modell, ahol az üzenetváltás két többnyire öntartalmazó egység között folyik futásidőben üzenetek által, alkalmas processzek közti kommunikációra is. Nem nehéz elképzelni az *Objective-C* üzeneteket olyan objektumok között, melyek más-más processzhez vagy ugyanabban a processzben, de más szálhoz tartoznak.

Például egy tipikus szerver-kliens üzenetváltás során a kliens küldhet egy kérést a megfelelő szerver objektumnak és a szerver esetleg egy adott üzenetet csak egyfajta kliens objektumnak küld el.

Vagy képzeljünk el egy interaktív alkalmazást, melynek néha intenzíven kell dolgoznia, hogy teljesítsen egy felhasználói parancsot. Ilyenkor megjeleníthet egy egyszerű párbeszédpanelt, amely jelzi a felhasználó fele, hogy várjon, mert épp a program elfoglalt. Más esetben szét kell választania a számításigényes műveletet a felhasználói felülettől, hogy az tudjon parancsokat fogadni folyamatosan. Objektumok a két taszkban tudnának kommunikálni *Objective-C* üzeneteken keresztül.

Hasonló módon, számos processz részt tudna venni egy közös dokumentum szerkesztésében. Ezek lehetnek különféle adatspecifikus szerkesztők (szöveg, kép, stb.). Egy taszk felelhetne az egységes felhasználói felület megjelenítéséért, azon parancsok megjelenítéséért, melyek az aktuális dokumentum elemmel kapcsolatosak. Minden résztvevő taszkok meg lehet írni *Objective-C*-ben, így a nyelv üzenetei játszhatják a főszerepet a felület és az eszközök, ill. eszközök egymás közti kommunikációjában.

Elosztott objektumok (Distributed Objects)

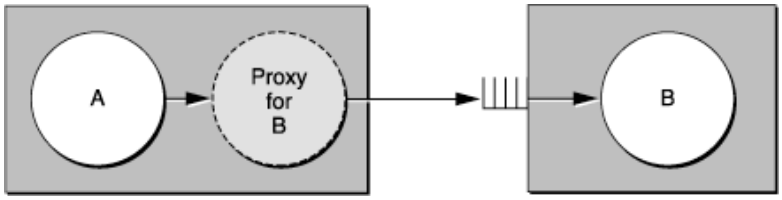
Távoli üzenetváltás *Objective-C*-ben runtime szolgáltatást igényel, ezen keresztül teremt kapcsolatot két processzbeli objektum, ill. azonosítja azokat az üzeneteket, melyek távoli objektumoknak szánták és továbbítja az adatokat az egyik objektum tárterületéről a másikába. Továbbá közvetíteni kell a két taszk ütemezője között: tárolni kell az üzeneteket addig, amíg a fogadó szabaddá nem válik az átvételre.

A *Cocoa* tartalmaz egy **elosztott objektum** (**distributed objects)** architektúrát, amely egy lényeges kiegészítése a runtime rendszernek. Használatával küldhetünk *Objective-C* üzeneteket taszkok vagy szálak közt. Megjegyzés: az elosztott objektum kezelő az *Objective-C* runtime-jára épül és nem befolyásolja a *Cocoa* objektumok alapvető működését.

```
- (BOOL)respondsToSelector:(SEL)aSelector
{
    if ( [super respondsToSelector:aSelector] )
        return YES;
    else {
        /* Here, test whether the aSelector message can     *
         * be forwarded to another object and whether that  *
         * object can respond to it. Return YES if it can.  */
    }
    return NO;
}
```

Távoli üzenetküldéshez először a programnak kapcsolatot kell teremtenie a távoli fogadóval. A létrejött kapcsolat során lesz egy *stróman* (proxy) objektumunk, ami képviseli a távolit, így programunk ezen keresztül kommunikál a távolival és ez a legtöbb esetben meg is fog felelni programunknak.

Távoli üzenetváltást az alábbi ábra szemlélteti. **A** objektum kommunikál **B**-vel strómanon keresztül, majd a **B**-nek szánt üzenetek addig várnak egy sorban tárolódva, amíg **B** kész nem lesz fogadni őket.



A küldő és a fogadó más taszkokban vannak és függetlenül ütemezettek egymáshoz képest. Azonban így nincs garancia arra, hogy a fogadó készen áll a fogadásra akkor, amikor a küldő a küldésre igen. Ennélfogva az érkező üzenetek egy sorba helyeztetnek, melyből a fogadó program kiveszi őket, amikor szükséges.

A *stróman* objektum nem a távoli objektum nevében cselekszik és nincs is szüksége annak az osztályát elérnie. Ez nem egy objektum másolat, inkább egy könnyűsúlyú helyettesítése annak. Bizonyos értelemben transzparens, egyszerűen továbbítja a kapott üzeneteket a fogadó fele és lebonyolítja a processzközti üzenetváltást. A fő feladata, hogy helyben tárat biztosítson egy olyan objektumnak, melynek erre nincs lehetősége. Azonban a stróman nem teljesen transzparens. Például nem engedi a távoli objektum példányváltozóinak az értékét lekérdezni, ill. átállítani.

A távoli fogadó tipikusan névtelen objektum, annak osztályát eltakarja a távoli program. A küldő programnak nincs is szüksége arra, hogy tudja, a másik programot mire tervezték és milyen osztályokat használ. Nem is kell, hogy ugyanazokat az osztályokat használja. Csak annyit kell tudni, milyen üzenetekre reagál a fogadó.

Ebből a célból az az objektum, melyet arra terveztek, hogy fogadjon távoli üzeneteket, formális protokoll típusú interfészét meghirdeti. Mind a küldő, mind a fogadó alkalmazás deklarálja a protokollt - ugyanazt importálja be mindkettő. A fogadó azért deklarálja azt, mert távoli objektumnak komformnak kell lennie ahhoz. A küldő program azért deklarálja a protokolt, hogy informálhassa a fordító programot a küldendő üzenetekről és mert esetleg használja a `conformsToProtocol:` metódust, ill. a `@protocol()` direktívát a távol fogadó azonosítására. A küldönek nem szükséges a protokoll minden metódusát implementálnia, egyszerűen csak azért deklarálja azt, mert az üzenetküldést ő kezdeményezi a távoli fogadó fele.

Az elosztott objektumok architektúráról - benne az `NSProxy` és az `NSConnection` osztályokkal - a Foundation framework referenciájában olvashat bővebben.

Nyelvi támogatás

A távoli üzenetváltás nem csak számos ravasz lehetőséget ad a program tervezéséhez, de néhány érdekes problémát is az *Objective-C* nyelv számára. A kérdések nagy része a távoli üzenetváltás hatékonyságával kapcsolatos és kétoldali bonyolító taszk szétválasztásának fokából is eredhetnek.

Az *Objective-C* hatféle típusminősítőt különböztet meg formális protokoll metódusdeklarációiban:

```
oneway
in
out
inout
bycopy
byref
```

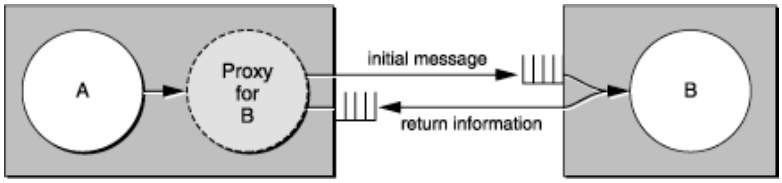
A kulcsszavak csak formális protokollokban használhatók, osztályokban és kategóriákban nem. Azonban, ha egy osztály vagy kategória adoptálja ezt a protokollt, az implementációban használhatjuk ezeket a metódus deklarációkban.

Szinron és aszinkron üzenetek

Nézzük a következő metódust egy egyszerű visszatérési értékkel:

```
- (BOOL)canDance;
```

Amikor a `canDance` üzenetet elküldjük ugyanabban a programban található fogadónak, egyszerűen annak metódusa lesz meghívva annak visszatérési értékét kapjuk meg. De amikor a fogadó egy másik programban található, két további üzenetre lesz szükségünk. Egy, amely eljut a távoli objektumhoz meghívva annak metódusát és egy másik, amely visszaérkezik az eredménnyel. Ezt mutatja az alábbi illuszráció:



A legtöbb üzenet - legalul - ehhez hasonló kétirányú (vagy “round trip”) távoli procedúrahívás (RPC). A küldő program megvárja a fogadót, amíg az meghívja a megfelelő metódust, elvégzi a műveletet és visszaküldi annak eredményét jelezve ezzel, hogy befejezte a feldolgozást. Az feldolgozásra várakozás - mégha eredmény nélküli is - a két program közti kommunikáció koordinálásának előnye, mivel szinkronban tartja őket. Ezért hívják gyakran a round-trip üzeneteket **szinkron** üzeneteknek.

Azonban vannak esetek, amikor nem szükségszerűen okos dolog válaszra várakozni. Néha hatékonyabb átadni az üzenetet és visszatérni, ezáltal ráhagyva a fogadóra, hogy azt és akkor kezdjen az üzenettel, amit és amikor akarja. Eközben a küldő folytatja tovább tevékenységét. *Objective-C* erre az esetre a `oneway` kulcsszót biztosítja, amely **aszinkron** üzenetekre használt metódusokat jelöl.

```
- (oneway void)waltzAtWill;
```

Habár a `oneway` (a `const`-hoz hasonlóan) típusminősítő és kombinálható más típusnevekkel, mint például `oneway float` vagy `oneway id`, azonban csak a `oneway void`-nak van értelme ebben az esetben, mivel aszinkron üzeneteknek nincs és nem is lehet érvényes visszatérési értéke.

Pointer paraméterek

Most tekintsük az alábbi metódust, amely pointer paraméterrel rendelkezik. Pointert cím szerinti paraméterátadásoknál használunk.

```
- setTune:(struct tune *)aSong
{
    tune = *aSong;
    ...
}
```

Ugyanez a paraméter sorrend használható információ cím szerinti visszaadására is.

```
- getTune:(struct tune *)theSong
{
    *theSong = tune;
    ...
}
```

Távoli üzenetváltásnál pointerek használata kicsit módosít(?) a küldés végrehajtásában. Egyik esetben sem lehet a pointer címét egyszerűen átadni távoli objektumnak, mivel az a lokális memóriaterületre mutat és ennek a címnek sok értelme nincs a fogadó memóriaterületén. A runtime-nak ilyenkor némileg tennie kell ezt-azt a színfalak mögött.

Ha a paraméter cím szerinti, a runtime rendszernek ezt értékké kell alakítania és így átküldeni a fogadóhoz, ott egy lokális címre elhelyezni és annak mutatóját átadni a fogadónak.

A másik esetben ha a pointer cím szerinti eredményhez kell, akkor a mutatott címről nem kell semmit sem elküldeni, csak a fogadó által küldött adattal feltölteni.

Az első esetben az információtovábbítás a round trip egyik ágán folyik, a másik esetben a visszairány a másik ágon. Mivel ezek az esetek a runtime távoli üzenetváltó alrendszerének teljesen különbözö részein történhetnek, ezért az *Objective-C* az alábbi módosító kulcsszavakat biztosítja:

Az `in` kulcsszó cím szerinti paraméter *küldését* jelzi:

```
- setTune:(in struct tune *)aSong;
```

Ehhez hasonlóan az `out` kulcsszó cím szerinti eredmény *fogadását* jelzi:

```
- getTune:(out struct tune *)theSong;
```

A harmadik, `inout` kulcsszó pedig cím szerinti paraméter *mindkét irányú küldését* jelzi:

```
- adjustTune:(inout struct tune *)aSong;
```

A *Cocoa* elosztott objektum kezelője az `inout` kulcsszót tekinti alapértelmezettnek a pointer paraméterek esetében. Ettől csak a `const`-tal megjelöltek képeznek kivételt, itt az `in` kulcsszó az alapértelmezett. Az `inout` a legbiztonságosabb, egyben a legköltségesebb megközelítés, hiszen adatot kell küldeni és fogadni. Az `in` az egyetlen kulcsszó, amely érték szerinti paraméterek esetén értelmes. Amíg az `in` bármilyen típusú paraméterre alkalmazható, addig az `out` és az `inout` csak pointerek esetén van értelme.

C-ben a pointerek néha összetett típust jelölnek, ilyen például a string is, amely karakterek sorozatára mutató pointer (`char *`). Habár jelölésben és megvalósításban itt lehet különbség az elvonatkoztatás mértékében, a koncepcióban azonban nem. Egy string önmagában entitás és nem egy pointer vagy valami más.

Ilyen esetekben az elosztott objektumok alrendszer automatikusan értékké alakítja a pointer tartalmát, majd továbbítja azt. Tehát az `out` és az `inout` kulcsszavaknak nincs értelmük egyszerű karakterre

mutató pointerek esetén. Ez csak plusz munkát jelent egy string cím szerinti továbbítása vagy fogadása során.

```
- getTuneTitle:(out char **)theTitle;
```

Hasonlóan igaz ez objektumokra is:

```
- adjustRectangle:(inout Rectangle **)theRect;
```

Az átalakításokat dinamikusan a runtime végzi el és nem a fordító.

Strómanok és klónok

Végezetül tekintsünk egy olyan metódust, melynek egy objektum típusú formális paramétere van:

```
- danceWith:(id)aPartner;
```

A `danceWith:` üzenet egy objektumot fog küldeni a fogadónak. Ha a küldő és a fogadó ugyanahhoz a programhoz tartozik, akkor ők ugyanarra az `aPartner` objektumra fognak hivatkozni.

Ez igaz akkor is, ha a fogadó egy távoli programban van, kivéve, ha annak hivatkoznia kell az objektumra strómanon keresztül - lévén az objektum nem az ő terében található. A `danceWith:` üzettel szállított pointer szimplán egy *strómanra* mutat, a neki küldött üzenetek továbbítódnak a valódi távoli objektumnak, a visszatérő értékek pedig ide.

Vannak esetek, amikor a *stróman* nem eléggé hatékony, amikor egyszerűbb egy objektum klónját átküldeni a távoli processznek, így az működni tud ott, a saját címterületén. Az *Objective-C* ezt az esetet a `bycopy` kulcsszóval teszi lehetővé a programozók számára.

```
- danceWith:(bycopy id)aClone;
```

A `bycopy` visszatérő értékekhez is használható:

```
- (bycopy)dancer;
```

Hasonló módon kombinálható az `out` kulcsszóval. Ekkor a cím szerint átveendő érték lemásolódik, mintsem *strómannal* helyettesítődne:

```
- getDancer:(bycopy out id *)theDancer;
```

> **Megjegyzés:** Amikor egy objektum klónja átadódik egy másik alkalmazásnak, az nem lehet ismeretlen. Annak, aki kapja, rendelkeznie kell a kapott objektum osztályának fizikai megvalósításával!

A `bycopy` bizonyos osztályok esetén több értelmet nyer - például olyan osztályoknál, melyek más objektumok gyűjteményét tartalmazzák -, melyeket gyakran úgy írunk meg, hogy klónt küldünk róluk a túloldalra a szokásos referencia helyett. Ezt azonban felülbírálhatjuk a `byref` kulcsszóval. Ebből kifolyólag az ezen a metóduson átküldött vagy ettől fogadott objektumok referencia szerint lesznek továbbítva. A legtöbb *Objective-C* objektum esetében ez az alapértelmezett mód, ritkán lehet szükségünk a `byref` kulcsszóra.

Az egyetlen típus, melynél fontos lehet a `bycopy` vagy `byref` kulcsszó, az egy objektum, amely akár dinamikus `id` típusú vagy statikus, osztálynévvel megadott típus.

Habár a `bycopy` vagy `byref` kulcsszó nem használható osztály vagy kategória deklarációkban, használható viszont formális protokollokban. Például írhatunk egy `foo` formális protokollt az alábbi módon:

```
@Protocol foo
- (bycopy)array;
@end
```

Majd ezt osztály vagy kategória adoptálhatja. Ennek segítségével olyan protokollokat konstruálhatunk, melyet "súgnak", hogy egyes objektumokat milyen módon kellene küldeni vagy fogadni.

Típusok kódolás

A fordító oly módon támogatja a runtime alrendszert, hogy kódolja a formális paraméterek és visszatérő értékek típusait minden egyes metódusnál. A kódolás `char *` (C string) típusú és a metódus szignatúrájához rendelt. Ez a kódolási séma hasznos lehet más esetekben is, ilyenkor az `@encode()` fordítói direktívával kaphatjuk meg a kódolást: a megadott típushoz az `@encode()` visszadja annak kódolását string formában. A kódolandó típus lehet egy alaptípus - például `int` -, pointer, rekord vagy únió, de akár egy osztálynév is. Bármi, ami megadható a *C* `sizeof()` operátorának argumentumként.

```
char *buf1 = @encode(int **);
char *buf2 = @encode(struct key);
char *buf3 = @encode(Rectangle);
```

Az alábbi táblázat tartalmazza a típus kódok listáját. Jegyezzük meg, hogy számos kód fedi azokat a kódokat, melyeket archiválás vagy elküldés céljából használunk. Akárhogy is, vannak olyan kódok a listában, melyek nem használhatóak bekódoló írásakor, illetve olyanok, melyeket használhatunk ilyen írásakor, azonban az `@encode()` nem generálja őket.

Kód	Jelentés
`c`	char
`i`	int
`s`	short
`l`	long
`q`	long long
`C`	unsigned char
`I`	unsigned int
`S`	unsigned short
`L`	unsigned long
`Q`	unsigned long long
`f`	float
`d`	double
`B`	C++ vagy C99 bool

v	void
*	char *
@	objektum
#	Class
:	SEL (szelektor)
[tömb]	tömb
{név=típus,...}	struktúra
(típus,...)	únió
bnum	bitmező *num* bitekkel
^type	type típusú pointer
?	ismeretlen típus (általában függvény pointerekre alkalmazzák)

A tömb típus kódja szögletes zárójelek közé van zárva, az elemek száma pedig mindjárt a nyitó zárójel után következik. Például egy 12 float típusú pointert tartalmazó tömb kódolása:

```
[12^f]
```

A struktúrák kapcsos, az únió pedig sima zárójelek között helyezkednek el. Először egy rekord neve, majd az egyenlőség jel után a típusa következik, és így tovább. Például legyen egy struktúra a következő:

```
typedef struct example {
    id   anObject;
    char *aString;
    int  anInt;
} Example;
```

A kódolása így néz ki:

```
{example=@*i}
```

Hasonló módon néz ki, ha a definiált típus neve (`Example`) vagy struktúra címke (`example`) kerül paraméterként az `@encode()` direktívához. Az erre a struktúrára mutató pointer kódolása a következő:

```
^{example=@*i}
```

De egy újabb szintű indirekció már eltünteti a belső típusspecifikációt:

```
^^{example}
```

Az objektumok ugyanúgy kódolódnak, mint a struktúrák. Az NSObject osztályneve átadva az `@encode()`-nak ezt eredményezi:

```
{NSObject=#}
```

Az NSObject osztálynak csak egy példányváltozója van, az `isa`, ami Class típusú.

Habár az `@encode()` nem adja őket vissza, a runtime rendszer használja a következő, táblázatba foglalt kódolásokat a metódus deklarációban levő típusminősítő kulcsszavakra.

Kód	Jelentés
`r`	`const`
`n`	`in`
`N`	`inout`
`o`	`out`
`O`	`bycopy`
`R`	`byref`
`V`	`oneway`

Ez a dokumentáció az *Apple Computers, Inc.: The Objective-C Programming Language* elektronikus dokumentum *The Runtime System* fejezetének fordítása. Fordította Sebestyén Gábor (segabor@elte.hu) 2005-ben az **ELTE IK Programozási nyelvek 3** tárgy keretében. A fordítás nem terjed ki a teljes eredeti szövegre!

Az eredeti angol nyelvű dokumentáció elérhető az alábbi linken:

```
http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/
RuntimeOverview/chapter_4_section_1.html#
```