Android Programozás

Resource Objects

- Adatforrás elemeket hivatkozás (referencia, mutató) segítségével használhatunk, ezek karakterláncok (stringek), képek, azonosítók vagy akár fájlok is lehetnek
- A mappastruktúra egységesen meg van adva az Android Studioban, ezekben találhatóak az adatforrásaink
- Ezekre az adatokra a @ segítségével hivatkozhatunk
- A stringeket a res/values/strings.xml fájlban definiálhatjuk name attribútumok segítségével (HTML-hez hasonló)
- Egyéni azonosítóknál (id) egyszer a megfelelő XML definíció helyén a + segítségével definiálhatunk egy új id-t, majd a @ segítségével plusz nélkül hivatkozhatunk rá a későbbiekben
- http://developer.android.com/guide/topics/resources/providing-resources.html

strings.xml

- res/values/strings.xml fájlban hozzuk létre a hiányzó string adatforrásainkat
- A jelenlegi helyett ez álljon:

```
<resources>
    <string name="app_name">My First App</string>
    <string name="edit_message">Enter a message</string>
    <string name="button_send">Send</string>
    <string name="action_settings">Settings</string>
</resources>
```

- Hivatkozott stringek helyett a megfelelő helyre akár inline (hardcoded string) is beírhattuk volna simán a stringünket (android:hint=„szoveg”), ez nem jó szokás, ugyanis a meglévő szövegünket hivatkozás segítségével tudjuk újra felhasználni és így elég csak egy helyen átírni (fordító figyelmeztet az inline string definíciókért)

Lokalizáció

- Az adatforrásokat (pl. stringek) hivatkozással használjuk mindenhol
- Ne írjuk be inline mindenhova külön a string szövegünket (ne használjunk hardcoded stringeket), hanem a values/strings.xml fájlban egyszer definiáljuk a stringet, majd mindenhol erre hivatkozzunk, így bizonyos feltételek mellett másmilyen nyelvű szövegeket is használhatunk
- Ilyen feltétel lehet pl a rendszer lokális nyelvbeállítása, így más nyelvű készülékeken más nyelvű szöveg fog megjelenni
- Ezt megtehetjük a Translations Editorral, jobb klikk a strings.xml-re és ott válasszuk a Translations Editort

Méretezési megfontolások

- Nemcsak a különféle eszközök képernyőméretei különböznek egymástól, hanem a felbontásaik is (density) így egy pixel minden telefonon más méretű, tehát a px mértékegység helyett valami állandóbb egységet kéne használni
- Erre vannak a density-independent pixelek *dp* mértékegységgel
- És a scale-independent pixelek *sp* mértékegységgel
- A kettő mértékegység számítása ugyanúgy történik, de az *sp* figyelembe veszi a szövegméreti felhasználói beállításokat is (szövegeknél mindig ilyet használjunk)
- Tehát ezt a két mértékegységet mindenhol és akkor minden képernyőfelbontáson ugyanolyanok lesznek a távolságok/méretek
- De ez még nem oldja meg a képernyőméreti problémák, ugyanis egy telefoni dobozméret tableten pazarlóan kicsi lenne

Különböző képernyőméretek

- A dp, sp értékeket tároljuk adatforrásokban, amiknek az egyedi attribútum nevei alkalmazásfüggőek (pl: <dimen name=„button_width”>10dp</dimen>)
- Ezek a méretértékek legyenek a res/values/dimens.xml fájlban és itt a dimen taget használjuk, majd az alkalmazási helyeken hivatkozunk rájuk „@dimen/button_width” segítségével
- Ha nagyobb képernyőméretekre is akarunk tervezni, akkor a dimens.xml fájlra jobb klikk new Value resource fájl segítségével különböző pl. képernyőméreti feltételeket adhatunk (qualifier) a fájlhoz, így a különböző fájlokban levő dimen egyforma nevű értékek közül az lesz használva, amihez tartozó fájl feltételei teljesülnek
- http://android4beginners.com/2013/07/appendix-c-everything-about-sizes-and-dimensions-in-android/

Különböző képfelbontások

- A különböző felbontású képernyők (density) miatt ahhoz, hogy egy kép minden kijelzőn ugyanakkora legyen, az oprendszer átméretez egy normális méretű referencia képet, így a nagyobb felbontású kijelzőkön csúnya pixeles átméretezett képet kapnánk
- Erre az a megoldás, hogy különböző felbontású képeket adunk meg és a kijelző felbontás alapján választ majd az oprendszer
- Ehhez az egy felbontáshoz tartozó képeknek a mappájának a nevében (vagy qualifierben) meg kell adni a felbontást (density), így alapvetően négyfajta felbontás közül választhatunk: xhdpi: 2.0 hdpi: 1.5 mdpi: 1.0 (baseline) ldpi: 0.75 A megadott számok méretszorzót jelentenek tehát, ha van egy képünk annak a referenciamérete lesz az mdpi és a többi felbontáshoz a szorzó segítségével kell méretezni (kétszer, másfélszer, vagy 0.75szer akkorára)

Képfelbontások

- A rendszer automatikusan kiválasztja a megfelelő felbontást a programunk futása alatt így jól fognak kinézni a képeink, ikonaink bármilyen felbontású kijelzőn
- Ezek a képek a megadott méretűek lesznek mindegyik kijelzőn
- A különböző képeket ezzel generáljuk le: https://romannurik.github.io/AndroidAssetStudio/icons-generic.html
- A létrejött mappastruktúrát másoljuk be a res mappánkba (android studioban res mappára jobb klikk -> Show in explorer)

9-patch png képek

- Ajánlott a png képek használata, mert átlátszóságot is meg lehet bennük adni
- Ha egy képnek a méretei arányosan vannak megadva (pl: match_parent), akkor nagyobb kijelzőn a képünk megnyúlik és eltorzul, ezért találták ki a 9-patch képeket (*.9.png) ezekben 1px széles fekete vonalakkal a képünk szélén megadhatjuk, hogy melyik része lesz nyújtható a képünknek (ott ahol pl csak háttérszín van és nincs alakzat) így a kényes alakzatok a képeinken megmaradnak minden kijelzőn
- Felül és bal oldalt a nyújtási részeket adhatjuk meg, alul és jobb oldalt a belső elemek helyeit adjuk meg
- http://radleymarx.com/blog/simple-guide-to-9-patch/
- Tool: http://romannurik.github.io/AndroidAssetStudio/nine-patches.html

ImageView

- Az ImageView konténerrel tudjuk a képeinket elhelyezni a kijelzőn, az src tulajdonságát kell beállítani android studioban megtalálhatjuk ezt jobb oldalt a Properties ablakon belül
- src-nél tallózás segítségével a Project fülnél ha beírjuk az előbb létrehozott képünk nevét, akkor a gyorskereső megtalálja nekünk, ezt válasszuk ki
- A layout:gravity tulajdonsággal akár középre is zárhatjuk a képünket
- Ha a szülőmegjelenítés LinearLayout, akkor láthatjuk, hogy egy oszlopba csak egy elemet rakhatunk, ha szabadabban szeretnénk módosítani a kinézetet, akkor válasszuk a RelativeLayoutot

Layout

- A vizuális (Design) szerkesztővel adjunk a képernyőnkhöz elemeket és állítsuk át a tulajdonságaikat a Properties ablakban
- A különböző Layoutok segítségével csoportosíthatjuk az elemeinket, ezért először ezeket kell elhelyezni
- Ezeket és ezeken belüli elemeinket HTML-hez hasonlóan pozícionálhatunk, behúzhatunk
- Megadhatunk az elemek között relatív elhelyezéseket, ezeket mind láthatjuk a Propertiesben és a vizuális szerkesztőben nyilak segítenek a relatív elhelyezkedések értelmezésében
- Ismerjük meg a működését, szerkesszük a kinézetet kedvünk szerint

Saját menü létrehozása

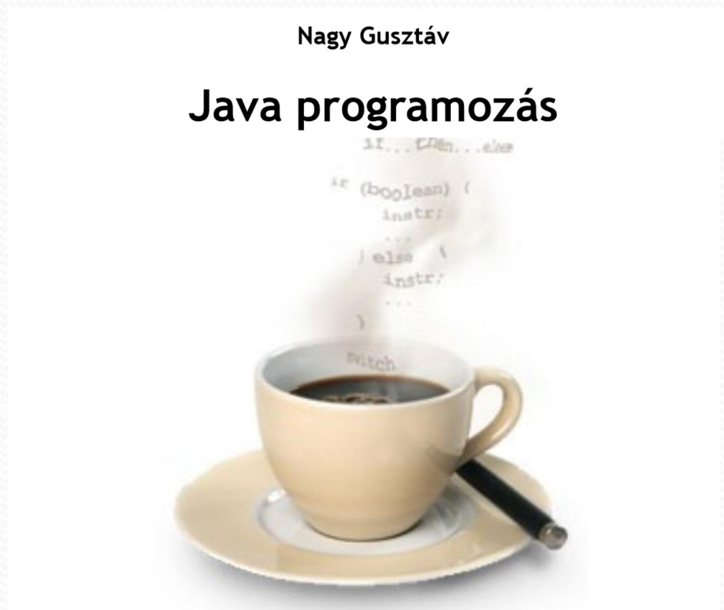
- Ahhoz, hogy könnyen jól kinéző menünk legyen, ahhoz használhatjuk az új projekt készítése során a Navigation Drawer Activity sablont
- A menu mappán belül a activity_main_drawer.xml fájlban láthatjuk az alapértelmezett menü elemeket
- Az egészet a <menu> tag veszi körbe, ezen belül csoportosításokat létrehozhatunk a <group> tag segítségével, majd a menü elemeket az <item> taggel adhatjuk meg
- http://developer.android.com/guide/topics/ui/menus.html

Android Studio API

- Ahhoz, hogy pl. egyik (akár fő) Activityből áttérjünk egy másik (mellék) Activityre, meg kellene ismernünk az Android java API függvénykönyvtárait (megoldásait), amikkel a java programozási nyelv segítségével leprogramozhatunk bármit ami csak eszünkbe jut
- Ehhez azonban elég alaposan ismerni kell a Java programozási nyelvet, ezért egy darabig a Java nyelvvel fogunk viszonylag mélyen foglalkozni
- http://developer.android.com/training/basics/firstapp/starting-activity.html

Java programozási könyv

- http://nagygusztav.hu/sites/default/files/csatol/java_programozas_1.3.pdf
- Ez alapján fogjuk megismerni a Java nyelvet



1.3. verzió

2007. február

Java programozási nyelv

- A Java programunk a JDK (Java Development Kit) segítségével fordul bájtkóddá
- A Java egy magas szintű nyelv a következő főbb jellemzőkkel
  - Egyszerű
  - Objektumorientált
  - Semleges architektúrájú
  - Hordozható
- A legtöbb Java platformra készült program asztali alkalmazás vagy Android app (applikáció)
- Kis-nagybetű érzékeny!

Megjegyzések a Javaban

- A Java nyelv a megjegyzések három típusát támogatja. Hagyományos (C stílusú) megjegyzés:
  - /* szöveg */
- Dokumentációs megjegyzés, a fordító figyelmen kívül hagyja, mint az előző típust is, de a javadoc eszköz (bin\javadoc.exe) segítségével automatikusan lehet generálni HTML dokumentációt, ami felhasználja a dokumentációs megjegyzéseket is:
  - /** dokumentáció */
- És végül a fordító figyelmen kívül hagyja a sort a // -től a sor végéig:
  - // szöveg

Osztálydefiníció, Objektumorientáltság (OO)

- A következő kód mutatja az osztálydefiníciós blokkot
- public class HelloWorld {

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

- Az osztály (class) alapvető építőeleme az objektumorientált nyelveknek. Az osztály az adatok és viselkedések összességéből álló példányok sablonját adja meg. Ezekből az osztályokból hozunk létre különböző példányokat a kódunkban, amiknek a szerkezetük (típusuk) meg fog egyezni, de az általuk tárolt adatok eltérőek lesznek. Az adatok az objektumpéldányok változóiként írhatók le, a viselkedések pedig a metódusokkal (saját függvényekkel). A valós életből egy hagyományos példa a téglalap osztály. Az osztály tartalmaz változókat a pozíció, valamint a szélesség és magasság leírására, és tartalmaz metódust a terület kiszámítására.

Osztály (class) jelzők

- A class kulcsszóval és az osztály nevével kezdődik az osztálydefiníció, majd kapcsos zárójelek között változók és metódusok következnek. A korábbi példa alkalmazásunkban nincs változó, és csak egyetlen metódus van main néven.
- Jelzők (módosítók):
  - public : jelzi, hogy a metódust más osztálybeli objektumokból is meg lehet hívni
  - static : jelzi, hogy statikus az osztálymetódus, ezeket a függvényeket példányosítás nélkül is meghívhatjuk a kódunkban a classnev.metodus(); utasítás segítségével
  - void : jelzi, hogy a metódusnak nincs visszatérési értéke
- Majd az általános Java System API segítségével kiírjuk (System.out.println(...)) az alapértelmezett (konzolablak) kimenetre a „Hello World!” szöveget

Objektumorientáltság

- *„Az objektumok az objektumorientált technológia alapjai. Néhány példa a hétköznapi életből: kutya, asztal , tv, bicikli. Ezek a valódi objektumok két jellemzővel rendelkeznek: állapottal és viselkedéssel. Például a kutya állapotát a neve, színe, fajtája, éhessége stb. jellemzi, viselkedése az ugatás, evés, csaholás, farok-csóválás stb. lehet. A bicikli állapotát a sebességfokozat, a pillanatnyi sebesség, viselkedését a gyorsulás, fékezés, sebességváltás adhatja."*
- A bicikli classból készült az én biciklim objektum:
  Változók:
  sebesség= 18km/h
  sebességfokozat = 5
  Metódusok:
  sebességváltás
  fékezés

Bicikli osztály

- class Bicikli {

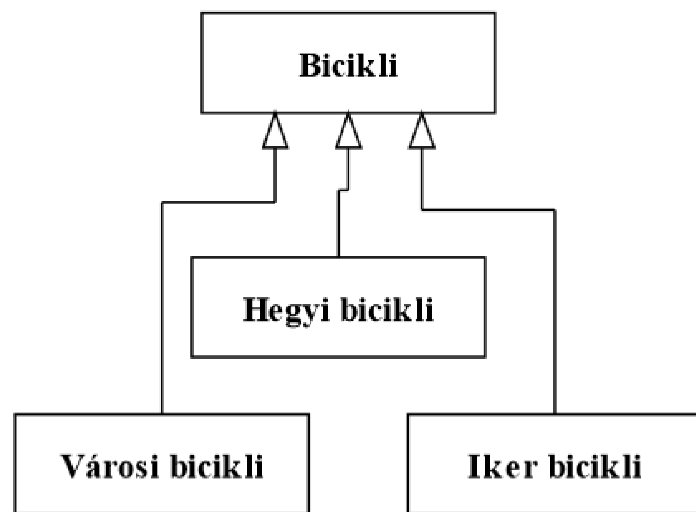
```
class Bicikli {
        // Változók
        int sebesseg;
        int sebessegfokozat;
        // Metódusok
        void sebessegvaltas() { … }
        void fekezes() { … }
}
```

- Majd ebből az osztályból létrehozunk egy új saját bicikli példányosítást:
- Bicikli enbiciklim = new Bicikli(); //Teljesen üres példány jön létre enbiciklim.sebessegvaltas(); // Függvényhívás, először állítjuk be a sebességét

Osztályleszármazás

- *„Az objektumorientált rendszerekben egyes objektumok között további összefüggéseket figyelhetünk meg. Bizonyos feltételeknek megfelelő objektumok egy másik osztályba sorolhatók. Például a hegyi vagy éppen a városi biciklik a biciklik speciális fajtái. Az objektumorientált szóhasználatban ezeket leszármazott osztálynak nevezzük. Hasonlóan, a bicikli osztály ősosztálya (szülő osztálya , bázisosztálya) a városi biciklik osztályának. Ezt az összefüggést mutatja a következő ábra:”*
- Például a Hegyi bicikli a Bicikli leszármazottja:
- **class** HegyiBicikli **extends** Bicikli {

```
	…
}
```



Osztályleszármazás

- *„Minden gyermekosztály örökli az ősosztály változó definícióit és a metódusait , de nincs ezekre korlátozva. A gyermekosztályok hozzáadhatnak változókat és metódusokat ahhoz, amit az ősosztálytól örökölt. A gyermekosztályok felül tudják írni az örökölt metódusokat , vagy speciálisabb megvalósítást tud adni azoknak."*
- *„Az öröklődés a következő előnyökkel jár:*
    - *A leszármazott osztályok tudják specializálni az ősosztálytól örökölt viselkedést. Az öröklődés segítségével az egyes osztályokat újra fel lehet használni.*
    - *A programozók meg tudnak valósítani olyan viselkedéseket , amelyek az ősosztályban még nem voltak konkrétan leírva . (Az ilyen osztályokat absztrakt, elvont osztályoknak nevezzük.) Az absztrakt ősosztályok csak részben valósítják meg a szükséges viselkedéseket, és akár más programozók fogják azt a leszármazottakban megvalósítani."*

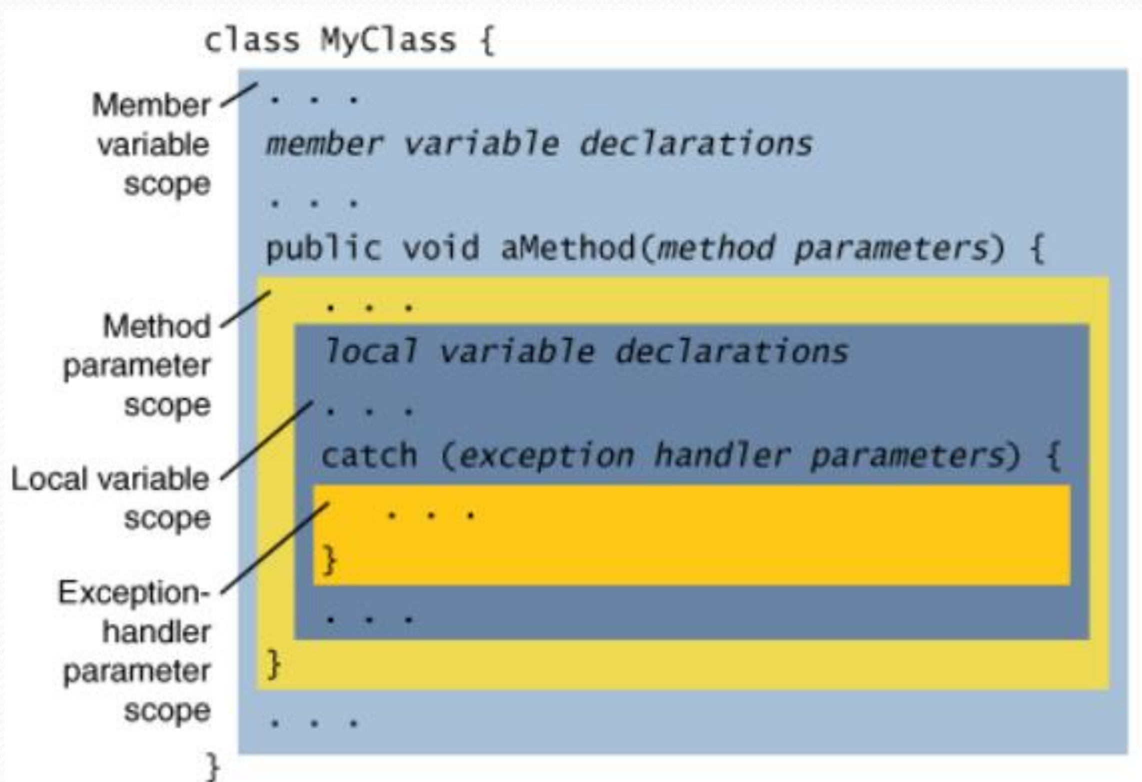
2.5. Publikus interfész

Általánosságban egy eszköznek vagy rendszernek szokás az **interfész**éről beszélni: azt írja le, hogy külső dolgok hogyan tudnak kapcsolódni hozzá. Ilyen értelemben két magyar ember között egy interfész a magyar nyelv.

Egy tetszőleges osztály esetén tehát a publikus interfész alatt az osztály kívülről is látható (publikus) felületét értjük. (Ez többnyire a publikus konstruktorokra és metódusokra korlátozódik.) Az üzenetküldés fogalmára visszautalva az osztály publikus interfésze azt határozza meg, hogy más objektumok milyen üzenetet küldhetnek az objektumnak, illetve milyen módon hozhatnak létre az osztálynak egy példányát.

Változók érvényességi határa

- A változó olyan adatelem, amely típussal és névvel van ellátva.
- A változó rendelkezik hatókörrel (érvényességi tartománnyal) is. A hatáskört a változódeklaráció helye egyértelműen meghatározza (lokális vagy globális érvényesség, lásd PHP (06. előadás))



Változók érvényességi határa

A tagváltozó (*member variable*) az osztály vagy objektum része. Az osztályon belül, de a metódusokon kívül lehet deklarálni. A tagváltozó az osztály egészében látható.

A lokális változók (*local variable*) egy kódblokkon belül vannak. A láthatóságuk a deklaráció helyétől az őket közvetlenül körülvevő blokk végéig tart.

A metódusok formális paraméterei (*method parameters*) az egész metóduson belül láthatók.

A kivételkezelő paraméterek (*exception handler parameters*) hasonlók a formális paraméterekhez.

Figyeljük meg a következő példát:

```
if (...) {
    int i = 17;
    ...
}
System.out.println("The value of i = " + i);    //error
```

Az utolsó sor kívül van az *i* lokális változó érvényességi körén, ezért a fordítás hibával leáll.

- (Egy utasításblokk {-től }-ig tart)

Adattípusok

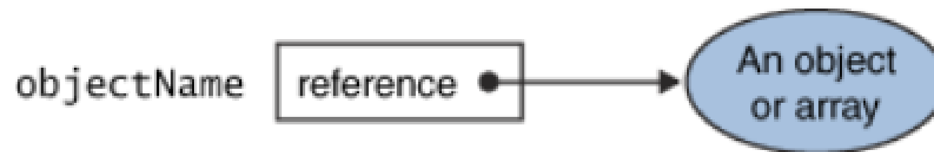
- Az objektum az állapotát változókban tárolja. Definíció : A változó olyan adatelem, amely azonosítóval van ellátva. Egy változó nevét és típusát egyértelműen meg kell adni a programunkban, ha használni akarjuk azt.
- A változó nevét és típusát a változódeklarációban adjuk meg, ami általában ehhez hasonló: *típus változónév;*
- Lehetőségünk van egyből kezdőértéket is adni a változónknak:
  - int valtozo = 4;
- Ha nem adunk kezdőértéket neki, akkor addig nem használhatjuk fel a változónk értékét
- Változót lehet véglegesen is definiálni. A végleges változó értékét nem lehet megváltoztatni az inicializálás után. Más nyelvekben ezt konstans változóknak is hívják
  - final int konstans = 10;

- Minden változó rendelkezik adattípussal. A változó adattípusa határozz a meg, hogy milyen értékeket vehet fel a változó, és milyen műveletek végezhetők vele.
- A primitív adattípusok egy egyszerű értéket képesek tárolni: számot, karaktert vagy logikai értéket

Típus	**Leírás**	**Méret/formátum**
(egészek)		
byte	bájt méretű egész	8-bit kettes komplemens
short	rövid egész	16-bit kettes komplemens
int	egész	32-bit kettes komplemens
long	hosszú egész	64-bit kettes komplemens
(valós számok)		
float	egyszeres pontosságú lebegőpontos	32-bit IEEE 754
double	dupla pontosságú lebegőpontos	64-bit IEEE 754
(egyéb típusok)		
char	karakter	16-bit Unicode karakter
boolean	logikai érték	*true* vagy *false*

Adattömb

A tömbök, az osztályok és az interfészek referencia-típusúak. A referencia-változó más nyelvek mutató vagy memóriacím fogalmára hasonlít. Az objektum neve (*objectName*) nem egy közvetlen értéket, hanem csak egy referenciát (*reference*) jelent. Az értéket közvetetten, a referencián keresztül érhetjük el:




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

- Tömbelemek elérése (*kulcs*(itt:index)->*érték* párossal)
- Index 0-tól kezdődik, tehát itt 0…9-ig tart
- Értékek elérése:
  - intArray[0] = 10; //Értékadás
    intArray[1] = 20; //Értékadás
    …
    int a = intArray[0]; //Használat

Aritmetikai operátorok

- Az operátorok egy, kettő vagy három operanduson hajtanak végre egy műveletet.

A Java programozási nyelvben sokféle aritmetikai operátor áll rendelkezésre lebegőpontos és egész számokhoz. Ezek az operátorok a + (összeadás), - (kivonás), * (szorzás), / (osztás) és % (maradékképzés). A következő táblázat összefoglalja a Java nyelv kétoperandusú aritmetikai operátorait.

operátor	használat	Leírás
+	op1 + op2	op1 és op2 összeadása, valamint sztring összefűzés
-	op1 - op2	op2 és op1 különbsége
*	op1 * op2	op1 és op2 szorzata
/	op1 / op2	op1 és op2 (egész) hányadosa
%	op1 % op2	op1 és op2 egész osztás maradéka

A művelet végrehajtása után a kifejezés értéke rendelkezésre áll. Az érték függ az operátortól és az operandusok típusától is. Aritmetikai operátorok esetén a típus alá van rendelve az operandusoknak: ha két *int* értéket adunk össze, az érték is *int* lesz.

Megjegyzendő, hogy a Javában a kifejezések kiértékelési sorrendje rögzített, vagyis egy művelet operandusai mindig balról jobbra értékelődnek ki (ha egyáltalán kiértékelődnek, lásd rövidzár kiértékelés), még a művelet elvégzése előtt.

- Zárójel segítségével aritmetikai szabályok szerint módosítható a kiértékelési sorrend
  Itt az első + jel segítségével a szöveg elejéhez hozzáfűzzük a kiértékelt számeredményt

```
int i = 37;
int j = 42;
double x = 27.475;
double y = 7.22;
System.out.println("Adding...");
System.out.println("    i + j = " + (i + j));
System.out.println("    x + y = " + (x + y));

System.out.println("Subtracting...");
System.out.println("    i - j = " + (i - j));
System.out.println("    x - y = " + (x - y));

System.out.println("Multiplying...");
System.out.println("    i * j = " + (i * j));
System.out.println("    x * y = " + (x * y));

System.out.println("Dividing...");
System.out.println("    i / j = " + (i / j));
System.out.println("    x / y = " + (x / y));
```

A program kimenete:

```
Adding...
    i + j = 79
    x + y = 34.695

Subtracting...
    i - j = -5
    x - y = 20.255

Multiplying...
    i * j = 1554
    x * y = 198.37

Dividing...
    i / j = 0
    x / y = 3.8054
```

4.1.1 Implicit konverzió

Amikor egy aritmetikai operátor egyik operandusa egész, a másik pedig lebegőpontos, akkor az eredmény is lebegőpontos lesz. Az egész érték implicit módon lebegőpontos számmá konvertálódik, mielőtt a művelet végrehajtódna. A következő táblázat összefoglalja az aritmetikai operátorok értékét az adattípusok függvényében. A szükséges konverziók még a művelet végrehajtása előtt végre fognak hajtódni.

long	az egyik operandus sem lebegőpontos, és legalább az egyik *long*
int	az egyik operandus sem lebegőpontos, és nem *long*
double	legalább az egyik operandus *double*
float	legalább az egyik operandus *float*, és a másik nem *double*

A + és - operátorok unáris (egyoperandusú) operátorokként is használhatók:

+op	*int* értékké konvertálja a *byte*, *short* és *char* értéket
-op	aritmetikai negálás

A ++ operátor növeli az operandus értékét, a -- pedig csökkenti eggyel. Mindkettőt írhatjuk az operandus elé (prefix) és után (postfix) is. A prefix forma esetén először történik az érték növelése vagy csökkentése, majd a kifejezés értéke is a megváltozott érték lesz. A postfix használat esetén fordítva történik a végrehajtás: először értékelődik ki az operandus, majd utána hajtódik végre a ++ vagy -- művelet.

4.2. Relációs operátorok

A relációs operátorok összehasonlítanak két értéket, és meghatározzák a köztük lévő kapcsolatot. Például a *!=* *true*-t ad, ha a két operandus nem egyenlő. A következő táblázatban összegyűjtöttük a relációs operátorokat:

Operátor	Alkalmazás	Leírás
>	op1 > op2	*true*-t ad vissza, ha op1 nagyobb, mint op2
>=	op1 >= op2	*true*-t ad vissza, ha op1 nagyobb vagy egyenlő, mint op2
<	op1 < op2	*true*-t ad vissza, ha op1 kisebb, mint op2
<=	op1 <= op2	*true*-t ad vissza, ha op1 kisebb vagy egyenlő, mint op2
==	op1 == op2	*true*-t ad vissza, ha op1 és op2 egyenlők
!=	op1 != op2	*true*-t ad vissza, ha op1 és op2 nem egyenlők

```java
int i = 37;
int j = 42;
int k = 42;
System.out.println("Greater than...");
System.out.println("    i > j = " + (i > j));//false
System.out.println("    j > i = " + (j > i));//true
System.out.println("    k > j = " + (k > j));//false;

System.out.println("Greater than or equal to...");
System.out.println("    i >= j = " + (i >= j));//false
System.out.println("    j >= i = " + (j >= i));//true
System.out.println("    k >= j = " + (k >= j));//true

System.out.println("Less than...");
System.out.println("    i < j = " + (i < j));//true
System.out.println("    j < i = " + (j < i));//false
System.out.println("    k < j = " + (k < j));//false

System.out.println("Less than or equal to...");
System.out.println("    i <= j = " + (i <= j));//true
System.out.println("    j <= i = " + (j <= i));//false
System.out.println("    k <= j = " + (k <= j));//true

System.out.println("Equal to...");
System.out.println("    i == j = " + (i == j));//false
System.out.println("    k == j = " + (k == j));//true

System.out.println("Not equal to...");
System.out.println("    i != j = " + (i != j));//true
System.out.println("    k != j = " + (k != j));//false
```

A fenti program kimenete:

```
Greater than...
    i > j = false
    j > i = true
    k > j = false
Greater than or equal to...
    i >= j = false
    j >= i = true
    k >= j = true
Less than...
    i < j = true
    j < i = false
    k < j = false
Less than or equal to...
    i <= j = true
    j <= i = false
    k <= j = true
Equal to...
    i == j = false
    k == j = true
Not equal to...
    i != j = true
    k != j = false
```

4.3. Logikai operátorok

A relációs operátorokat gyakran használják logikai operátorokkal együtt, így összetettebb logikai kifejezéseket hozhatunk létre. A Java programozási nyelv hatféle logikai operátort – öt bináris és egy unáris – támogat, ahogy azt a következő táblázat mutatja:

Operátor	**Alkalmazás**	**Leírás**
&&	op1 && op2	Logikai és: *true*-t ad vissza, ha op1 és op2 egyaránt *true*; op2 feltételes kiértékelésű
||	op1 || op2	Logikai vagy: *true*-t ad vissza, ha op1 vagy op2 *true*; op2 feltételes kiértékelésű
!	!op	Logikai nem: *true*-t ad vissza, ha op *false*
&	op1 & op2	Bitenkénti és: *true*-t ad vissza, ha op1 és op2 egyaránt *boolean* és *true*; op1 és op2 mindig kiértékelődik; ha mindkét operandus szám, akkor bitenkénti *és* művelet
|	op1 | op2	Bitenkénti vagy: *true*-t ad vissza, ha op1 és op2 egyaránt *boolean* vagy op1 vagy op2 *true*; op1 és op2 mindig kiértékelődik; ha mindkét operandus szám, akkor bitenkénti *vagy* művelet
^	op1 ^ op2	Bitenkénti nem: *true*-t ad vissza, ha op1 és op2 különböző — vagy egyik, vagy másik, de nem egyszerre mindkét operandus *true*

A következő példa az *&&* operátort használja a két rész-kifejezés logikai értékének összekapcsolására:

```
0 <= index && index < NUM_ENTRIES
```

Egész számok számrendszerei

- Minden decimális (10-es számrendszerű) egész szám átírható 2-es vagy 16-os (gépek által kedvelt) számrendszerekbe
- 2-es számrendszerben a számjegyeket biteknek nevezzük
- Pl: a 11-es szám 2-es számrendszerben: 1011
  Pl: a 6-os szám 2-es számrendszerben: 0110
- Így a bitenkénti műveltek érthetőek:
  pl: 11 & 6 ezt jelenti:
  1011 &
  0110
  ----- kiértékelése (1 ha mindkét szám azonos helyi értékű bitje 1, különben 0):
  0010, ez a szám a 2
- 11 & 6 == 2

4.3.1 Rövidzár kiértékelés

Bizonyos esetekben a logikai operátor második operandusa nem értékelődik ki. Például a következő esetben:

```
(numChars < LIMIT) && (...)
```

Az *&&* operátor csak akkor ad vissza *true*-t, ha mindkét operandus *true*. Tehát, ha *numChars* nagyobb vagy egyenlő, mint LIMIT, akkor *&&* bal oldala *false*, és a kifejezés visszaadott eredménye a jobb oldali operandus kiértékelése nélkül születik meg. Ilyen esetekben a fordító nem értékeli ki a jobb oldali operandust. Ilyenkor a jobb oldali kifejezés miatt közvetett mellékhatások léphetnek fel, például ha adatfolyamból olvasunk, értékeket aktualizálunk, vagy számításokat végzünk a jobb oldali operandusban. Ehhez hasonlóan, ha a || operátor bal oldali operandusa igaz, felesleges a jobboldalt kiértékelni, nem is fog megtörténni.

Ha mindkét operandus logikai, az *&* operátor hasonlóan viselkedik, mint az *&&*. Azonban *&* mindig kiértékelődik és *true*-t ad vissza, ha mindkét operandusa *true*. Ha az operandusok *boolean*-típusúak, | azonos műveletet végez, mint ||.

A fenti működés miatt nem érdemes olyan kódot készíteni, amelyik a jobboldali operandusának kiértékelése során a kiértékelésen túl mást is tesz. Például veszélyes, nehezen áttekinthető lesz a következő feltételes kifejezés:

```
if (a < b && b++ < f(c) ) {...}
```

Ha a bal oldali operandus (a<b) hamis, akkor sem a ++ operátor, sem az f függvényhívás nem fog végrehajtódni.

Értékadó operátorok

Operátor	Használat	Egyezik
+=	op1 += op2	op1 = op1 + op2
-=	op1 -= op2	op1 = op1 - op2
*=	op1 *= op2	op1 = op1 * op2
/=	op1 /= op2	op1 = op1 / op2
%=	op1 %= op2	op1 = op1 % op2
&=	op1 &= op2	op1 = op1 & op2
|=	op1 |= op2	op1 = op1 | op2
^=	op1 ^= op2	op1 = op1 ^ op2
<<=	op1 <<= op2	op1 = op1 << op2
>>=	op1 >>= op2	op1 = op1 >> op2
>>>=	op1 >>>= op2	op1 = op1 >>> op2