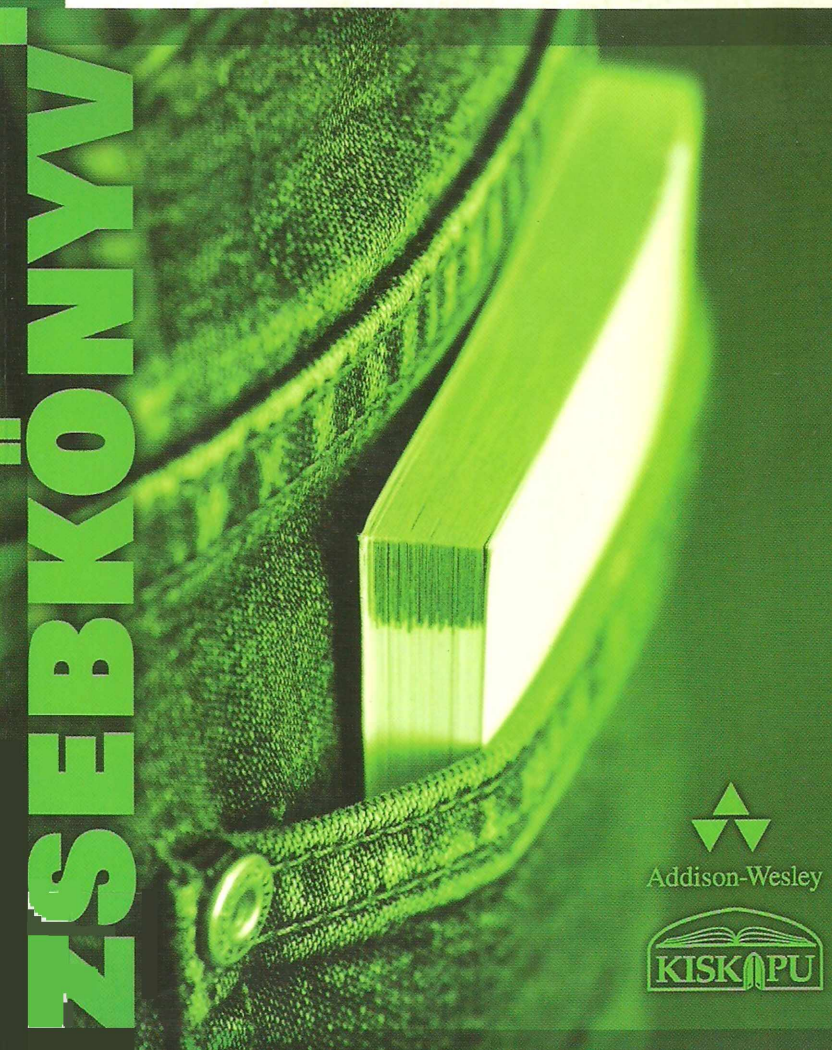
Java

Peter J DePasquale

ZSEBKÖNYV



Addison-Wesley

KISKAPU

Java zsebkönyv

Peter J. DePasquale

Ha gyorsan szeretnénk információkhoz jutni, miközben Java programokat írunk, lapozzuk fel a *Java zsebkönyvet*, amely tömören felsorolja a Java leggyakrabban használt kulcsszavait és programozási felületeit. A kötet világos szerkezetű, könnyen áttekinthető; megtalálható benne a Java-utasítások formája (példákkal illusztrálva) és a kulcsszavak leírása, valamint tippeket is ad a programozáshoz. Mindezek nélkülözhetetlen kézikönyvvé teszik, amely nem hiányozhat a hátizsákunkból.

A zsebkönyv számos hasznos információt tartalmaz a kezdő és középhaladó programozó hallgatók, illetve azok számára, akik egyetlen könnyen forgatható kötetben szeretnék látni a Java nyelv jellemzőit.

Addison-Wesley

http://www.kiskapukiado.hu/121

Témakör: Programozás, Java
Felhasználói szint: Kezdő–Haladó
ISBN szám: 963 9637 10 6
Ára: 1300 Ft

Peter J. DePasquale

Java zsebkönyv

Budapest, 2006

A fordítás a következő angol eredeti alapján készült:

Peter J. DePasquale: Java Backpack Reference Guide

Authorized translation from the English language edition, entitled ADDISON-WESLEY'S JAVA BACKPACK REFERENCE GUIDE, 1st Edition, ISBN 0321304276, by DEPASQUALE, PETER, published by Pearson Education, Inc, publishing as Addison-Wesley.

Lektor: *Rézműves László*
Fordítás: *Vlaskovits Dóra*
Műszaki szerkesztés, tördelés: *Csutak Hoffmann Levente*
Borító: *Bognár Tamás*

Felelős kiadó a Kiskapu Kft. ügyvezető igazgatója

© 2006 Kiskapu Kft.
1081 Budapest, Népszínház u. 31. I. 7.
Telefon: (+36-1) 477-0443 Fax: (+36-1) 303-1619
www.kiskapukiado.hu
e-mail: kiado@kiskapu.hu

ISBN: 963 9637 10 6

Készült a debreceni Kinizsi Nyomdában
Felelős vezető: *Bördős János*

Köszönetnyilvánítás

Többen is segítették javaslataikkal ezen útmutató megírását. Köszönettel tartozom Philip Isenhournak, Deborah Knoxnak, Cara Cockingnak, J.A.N. Lee-nek és Joseph Chase-nek, akik mind a tartalmat, mind formát illetően hasznos ötleteket adtak.

Külön köszönet jár John Lewisnak, egyik legközelebbi barátomnak, amiért megvilágította számomra a tanári életstílus lényegét és támogatta ötleteimet, valamint Michael Hirsch szerkesztőnek (annak ellenére, hogy valójában nem szerkeszt semmit), akinek előrelátása, hozzájárulása és útmutatása nélkül ez a könyv nem jelenhetett volna meg.

Köszönöm mindenkinek a segítséget és támogatást.

Ajánlás

Ezt a könyvet szüleimnek, Peternek és Penny-nek ajánlom. Mindig hittek benne, hogy bármi lehetek, ami csak akarok, de arra buzdítottak, hogy valami mást csináljak a tanítás helyett. Ironikus módon éppen a tanítás az, amihez visszatértem, és amit a legjobban élvezek.

Megjegyzés

Az egyes fenntartott kifejezések utasításformájának összefoglalásakor megpróbáltam tömören fogalmazni és a témánál maradni. Fontosnak tartom megjegyezni, hogy ez az útmutató elsőéves programozó hallgatók számára készült.

Ha egy utasítás egy része elhagyható, akkor az szögletes zárójelben szerepel. Azokra a helyekre, ahová további utasítások kerülhetnek – ilyen például a tagfüggvények vagy osztályok törzse –, három pontot tettem. Például:

```
[módosító] [abstract] visszatérési-típus
    ➥ függvénynév ([paraméterdeklarációk]) {...}
```

Ebben a példában az `abstract` kulcsszó kiemelt, mivel ennek nyelvtanáról és működéséről van szó. Az **`abstract`** ugyan módosító, de ebben az utasítástípusban más módosítók (például *public, private*) is szerepelhetnek, ezért a `[módosító]` jelölés is szerepel.

Tartalomjegyzék

A Java alapjai

Vezérlő-jelsorozatok a Java nyelvben

Vezérlősorozat	Jelentés
\'	egyszeres idézőjel
\"	kettős idézőjel
\\	fordított perjel
\b	visszatörlés
\n	új sor
\r	kocsivissza
\t	tabulátor
\f	lapváltás
\000 és \377 között	nyolcas számrendszerben megadott vezérlőkódok; az \u0000 és \u00FF közötti Unicode karaktereknek felelnek meg
\uXXXX és \uXXXX között	Unicode karakterek (ahol minden „X" tizenhatos számrendszerbeli számjegy [0-9, a-f vagy A-F]

Szűkítő átalakítások a Java nyelvben

Miből	Mivé
byte	char
short	byte vagy char
char	byte vagy short
int	byte, short vagy char
long	byte, short, char vagy int
float	byte, short, char, int vagy long
double	byte, short, char, int, long vagy float

Láthatósági módosítók a Java nyelvben

Módosító	Osztályok és felületek	Függvények és változók
alapértelmezett (nincs módosító)	Saját csomagjában látható.	A saját osztályával megegyező csomagban lévő bármely osztály számára látható.
public	Bárhol látható.	Bárhol látható.
protected	Nem alkalmazható.	A saját osztályával megegyező csomagban lévő bármely osztály számára látható.
private	Csak abban az osztályban látható, amelyben található.	Semelyik másik osztály által nem látható.

Műveletek elsőbbsége a Java nyelvben

Az alacsonyabb elsőbbségi szintű műveletek kiértékelése előbb történik meg egy kifejezésben, mint a magasabb szintűeké (első oszlop). Az azonos szintű műveletek kiértékelése a megadott társítás szerint történik (negyedik oszlop)

Elsőbbségi szint	Műveleti jel	Művelet	Társítás
1	[]	tömbindexelés	balról jobbra
	.	objektumtag-hivatkozás	
	(paraméterek)	paraméterkiértékelés és függvényhívás	
	++	utótagos növelés	
	--	utótagos csökkentés	
2	++	előtagos növelés	jobbról balra
	--	előtagos csökkentés	
	+	egytényezős plusz	
	-	egytényezős mínusz	
	~	bitenkénti NEM	
	!	logikai NEM	
3	`new`	objektumpéldányosítás	jobbról balra
	(*típus*)	típusátalakítás	
4	*	szorzás	balról jobbra
	/	osztás	
	%	maradékképzés	
5	+	összeadás	balról jobbra
	+	karakterlánc-összefűzés	
	-	kivonás	
6	<<	balra léptetés	balról jobbra
	>>	jobbra léptetés	
	>>>	jobbra léptetés és feltöltés nullával	

Elsőbbségi szint	Műveleti jel	Művelet	Társítás
7	<	kisebb	balról jobbra
	<=	kisebb vagy egyenlő	
	>	nagyobb	
	>=	nagyobb vagy egyenlő	
	`instanceof`	típus-összehasonlítás	
8	==	egyenlő	balról jobbra
	!=	nem egyenlő	
9	&	bitenkénti ÉS	balról jobbra
	&	logikai ÉS	
10	^	bitenkénti KIZÁRÓ VAGY	balról jobbra
	^	logikai KIZÁRÓ VAGY	
11	|	bitenkénti VAGY	balról jobbra
	|	logikai VAGY	
12	&&	logikai ÉS	balról jobbra
13	||	logikai VAGY	balról jobbra
14	?:	feltételes művelet	jobbról balra
15	=	értékadás	jobbról balra
	+=	összeadás, majd értékadás	
	+=	karakterlánc-összefűzés, majd értékadás	
	-=	kivonás, majd értékadás	

Elsőbbségi szint	Műveleti jel	Művelet	Társítás
15	*=	szorzás, majd értékadás	jobbról balra
	/=	osztás, majd értékadás	
	%=	maradékképzés, majd értékadás	
	<<=	balra léptetés, majd értékadás	
	>>=	jobbra léptetés (előjel), majd értékadás	
	>>>=	jobbra léptetés (nulla), majd értékadás	
	&=	bitenkénti ÉS, majd értékadás	
	&=	logikai ÉS, majd értékadás	
	^=	bitenkénti KIZÁRÓ VAGY, majd értékadás	
	^=	logikai KIZÁRÓ VAGY, majd értékadás	
	|=	bitenkénti VAGY, majd értékadás	
	|=	logikai VAGY, majd értékadás	

Néhány Java-csomag áttekintése

Csomag	Feladat
java.applet	Kisalkalmazások létrehozása
java.awt	Grafika és grafikus felhasználó felületek (GUI) létrehozása
java.awt.Color	Színeket jelképező és kezelő objektumok létrehozása
java.beans	Java babszemek (JavaBeans) fejlesztése
java.io	Bemeneti és kimeneti tevékenységek elvégzése átmeneti tárak, adatfolyamok stb. használatával
java.lang	Általános nyelvi támogatás
java.math	Nagy pontosságú matematikai számítások és műveletek elvégzése
java.net	Hálózati kommunikáció bonyolítása
java.rmi	Távoli függvényhívás (RMI) végrehajtása
java.sql	Adatbázisokkal való együttműködés
java.text	Szöveg formázása kimenetre
java.util	Általános segédprogramok (Random, Scanner, System osztályok stb.)
java.util.jar	JAR fájlok olvasása és írása
java.util.regex	Karaktersorozatok összehasonlítása szabályos kifejezésekkel
java.util.zip	ZIP és GZIP fájlok olvasása és írása
java.crypto	Titkosító műveletek elvégzése
java.swing	Grafikus felhasználói felületek létrehozása a java.awt csomag alapján
java.xml.parsers	XML dokumentumok feldolgozása

A Java gyakran használt végrehajtható alkalmazásai

appletviewer – A Java **appletviewer** (kisalkalmazás-megjelenítő) programjának segítségével a programozónak lehetősége van Java kisalkalmazások megtekintésére és végrehajtására külön böngésző használata nélkül. Az **appletviewer** egy egyszerű felülettel rendelkezik, amely megjeleníti a kisalkalmazást, és alkalmas hibakeresésre is. A kisalkalmazást tartalmazó HTML oldal URL-jét vagy fájlnevét paraméterként kapja meg a kisalkalmazás-megjelenítő program.

Használat:

```
appletviewer <kapcsolók> url(ek)
```

A `<kapcsolók>` a következők lehetnek:

`-debug` A kisalkalmazás-megjelenítő elindítása a Java hibakeresőben

`-encoding <kódolás>` A HTML fájlok által használt karakterkódolás megadása

`-J<futásidejű jelző>` Paraméter átadása a `java` értelmezőnek

> A `-J` kapcsoló nem szabványos, és előzetes figyelmeztetés nélkül megváltozhat.

jar – A **jar** futtatható program Java archív fájlok létrehozására, olvasására és bővítésére alkalmas. A `jar` fájlok általában bájtkódfájlokat, valamint más olyan fájlokat tartalmaznak, amelyekre az alkalmazásnak futásidőben szüksége van (például GUI gombok képei, ikonok stb.). A **jar**

fájl „manifest" fájljának (META-INF/MANIFEST.MF) módosításával a `jar` fájlok beállíthatók úgy, hogy a `jar` fájl kibontása nélkül végrehajthatóak legyenek.

Használat:

```
jar {ctxu}[vfm0Mi][jar-fájl]
➥ [manifest-fájl][-C könyvtár] fájlok...
```

Kapcsolók:

`-c`	új archívum létrehozása
`-t`	az archívum tartalomjegyzékének megjelenítése
`-x`	a megnevezett fájlok (vagy az összes fájl) kicsomagolása az archívumból
`-u`	meglévő archívum frissítése
`-v`	bőbeszédű kimenet létrehozása a szabványos kimeneten
`-f`	archív fájl nevének megadása
`-m`	manifest információk beágyazása megadott manifest fájlból
`-0`	csak tárolás; ZIP tömörítés használatának mellőzése
`-M`	ne jöjjön létre manifest fájl a bejegyzésekhez
`-i`	indexinformációk létrehozása a megadott jar fájlokhoz
`-C`	váltás a megadott alkönyvtárra, és a következő fájl beágyazása

Ha bármelyik fájl könyvtár, akkor annak feldolgozása rekurzív módon történik.
A manifest fájl és az archív fájl nevét az `m` és `f` kapcsolóknak megfelelő sorrendben kell megadni.

1. példa: két osztályfájl archiválása egy classes.jar nevű archívumban:

```
jar cvf classes.jar Foo.class Bar.class
```

2. példa: egy meglévő, mymanifest nevű manifest fájl használata és a foo/ könyvtárban lévő összes fájl archiválása a classes.jar fájlban:

```
jar cvfm classes.jar mymanifest -C foo/ .
```

java – A Java értelmezője a **java** futtatható program. Ennek a programnak a használatával futtathatók a Java programok a Java forráskódból létrehozott bájtkódfájlok értelmezése által. A bájtkódfájlok a Java fordítóprogram használatával történő fordítás eredményeképp jönnek létre.

Használat:

```
java [-kapcsolók] osztály [paraméterek...]
        (osztály futtatásához)
```

vagy

```
java [-kapcsolók] -jar jarfájl
   [paraméterek...](jar fájl futtatásához)
```

A <kapcsolók> a következők lehetnek:

-client	az ügyfél VM kiválasztásához
-server	a kiszolgáló VM kiválasztásához
-hotspot	az ügyfél VM szinonimája [elavult]. Az alapértelmezett VM az ügyfél.

-cp <a könyvtárak és zip/jar fájlok osztálykeresési útvonala>

`-classpath < a könyvtárak és zip/jar fájlok osztálykeresési útvonala>`
Pontosvesszőkkel tagolt lista azokról a könyvtárakról, JAR archívumokról és ZIP archívumokról, amelyekben az osztályfájlok keresendők.

`-D<név>=<érték>`
rendszertulajdonság beállítása

`-verbose[:class|gc|ini]`
bőbeszédű kimenet bekapcsolása

`-version` a termékváltozat kiírása és kilépés

`-version:<érték>`
a futtatáshoz a megadott változat szükséges

`-showversion` a termékváltozat kiírása és folytatás

`-jre-restrict-search | -jre-no-restrict-search`
privát felhasználói JRE-k befoglalása/kizárása a változatkeresésbe

`-? -help` súgóüzenet kiíratása

`-X` súgó kiíratása a nem szabványos kapcsolókról

`-ea[:<csomagnév>...|:<osztálynév>]`
`-enableassertions[:<csomagnév>...|:<osztálynév>]`
előfeltételezések bekapcsolása

`-da[:<csomagnév>...|:<osztálynév>]`
`-disableassertions[:<csomagnév>...|:<osztálynév>]`
előfeltételezések kikapcsolása

`-esa | -enablesystemassertions`
rendszer-előfeltételezések bekapcsolása

`-dsa | -disablesystemassertions`
rendszer-előfeltételezések kikapcsolása

`-agentlib:<programkönyvtár_neve>[=<kapcsolók>]`
a `<programkönyvtár_neve>` natív ügynök-programkönyvtár betöltése; például: `-agentlib:hprof`; lásd még: `agentlib:jdwp=help` és `-agentlib:hpof=help`

`-agentpath:<elérési_út>[=<kapcsolók>]`
natív ügynök-programkönyvtár betöltése teljes elérési úttal

`-javaagent:<osztálynév>[=<kapcsolók>]`
Java programozási nyelvi ügynök betöltése; lásd `java.lang.instrument`

`javac` – A Java fordítója a **`javac`** futtatható program, amely Java forráskódfájlokat fordít futtatható bájtkódfájlokra. A Java bájtkód-fájlok (Java alkalmazások) a **`java`** programmal futtathatók.

Használat:

```
javac <kapcsolók> <forrásfájlok>
```

A lehetséges kapcsolók:

`-g` Minden hibakeresési információ előállítása

`-g:none` Nincsenek hibakeresési információk

`-g:{lines,vars,source}`
Részleges hibakeresési információk előállítása

`-nowarn` Nincsenek figyelmeztetések

`-verbose` Kimeneti üzenetek a fordító tevékenységéről

`-deprecation` Elavult API-kat használó forráshelyek kiíratása

`-classpath <útvonal>`
A felhasználói osztályfájlok helyének megadása

`-cp <útvonal>` A felhasználói osztályfájlok helyének megadása

`-sourcepath <útvonal>`
A bemeneti forrásfájlok helyének megadása

`-bootclasspath <útvonal>`
A bootstrap osztályfájlok helyének felülírása

`-extdirs <könyvtárak>`
A telepített bővítmények helyének felülírása

`-endorseddirs <könyvtárak>`
A jóváhagyott szabványok elérési útjának felülírása

`-d <könyvtár>` A létrehozott osztályfájlok helyének meghatározása

`-encoding <kódolás>`
A forrásfájlok által használt karakterkódolás megadása

`-source <kiadás>`
Forrásmegfelelőség biztosítása a megadott kiadáshoz

`-target <kiadás>`
Osztályfájlok létrehozása a megadott VM-változathoz

`-version` Változatinformációk

`-help` A szabványos kapcsolók kivonatának megjelenítése

`-X`	A nem szabványos kapcsolók kivonatának megjelenítése
`-J<jelző>`	A <jelző> átadása közvetlenül a futásidejű rendszernek

`javadoc` – A **`javadoc`** futtatható program HTML oldalak sorozatát hozza létre Java forráskódfájlok egy megadott készletéből. A javadoc programozóbarát dokumentációt készít a Java API dokumentáció stílusában.

Használat:

```
javadoc [kapcsolók] [csomagnevek]
                [forrásfájlok] [@fájlok]
```

`-overview <fájl>`	Összefoglaló dokumentáció beolvasása HTML fájlból
`-public`	Csak a nyilvános osztályok és tagok megjelenítése
`-protected`	A védett és nyilvános osztályok és tagok megjelenítése (alapértelmezett beállítás)
`-package`	Csomag-, védett és nyilvános osztályok és tagok megjelenítése
`-private`	Minden osztály és tag megjelenítése
`-help`	Parancssori kapcsolók megjelenítése és kilépés
`-doclet <osztály>`	Kimenet létrehozása másik docleten keresztül
`-docletpath <útvonal>`	A doclet osztályfájlok helyének meghatározása

`-sourcepath <útvonallista>`
: A forrásfájlok helyének meghatározása

`-classpath <útvonallista>`
: A felhasználói osztályfájlok helyének meghatározása

`-exclude <csomaglista>`
: Kizárandó csomagok listájának megadása

`-subpackages <alcsomaglista>`
: Rekurzív módon betöltendő alcsomagok megadása

`-breakiterator`
: Az első mondat kiszámítása a `BreakIterator` használatával

`-bootclasspath <útvonallista>`
: A bootstrap osztálybetöltő által betöltött osztályfájlok helyének felülírása

`-source <kiadás>`
: Forrásmegfelelőség biztosítása a megadott kiadáshoz

`-extdirs <könyvtárlista>`
: A telepített bővítmények helyének felülbírálása

`-verbose`
: Kimeneti üzenetek a javadoc tevékenységéről

`-locale <név>`
: Használandó nyelvi beállítás megadása; például `en_US` vagy `en_US_WIN`

`-encoding <név>`
: Forrásfájl kódolási név

`-quiet`
: Állapotüzenetek megjelenítésének mellőzése

`-J<jelző>`
: A `<jelző>` átadása közvetlenül a futásidejű rendszernek

Fenntartott szavak a Javában

abstract (módosító)

Utasításforma

```
[ módosító ][ abstract ] class osztálynév
    [ extends osztálynév ]
    [ implements osztálynév [, osztálynév…] ] {…}
```

vagy

```
[ módosító ] [ abstract ] adattípus függvénynév
    ( [ paraméterdeklarációk ] )
    [ throws kivételtípus [, kivételtípus…] ] {…}
```

Leírás

Az **abstract** (elvont) módosító osztályokra és függvényekre is alkalmazható. Osztályokra alkalmazva megakadályozza a kérdéses osztály példányosítását. Az **abstract** függvényeket egy alosztályban kell megvalósítani. Mindkét esetben arról van szó, hogy az osztály vagy az **abstract** függvények teljes meghatározását egy alosztály adja meg.

Példa

```
abstract class Motion {
   private int xDirection = 0, yDirection =0;

   // Az abstract move függvényt egy megfelelő
   // alosztályban kell megvalósítani.
   abstract void move (int x, int y);

   // Az objektum mozgásának két irányát leíró
   // karakterláncokat adja vissza.
   public String toString(){
      return xDirection + "," + yDirection;
   }
}
```

Tippek

- Az **abstract** függvényeket általában annak jelzésére használják, hogy a függvényt az alosztályokban kell megvalósítani. Minden többgyermekes alosztálynak egyedi megvalósítása lehet.
- Ha egy osztály egy vagy több elvont függvényt tartalmaz, akkor magának az osztálynak is elvontnak kell lennie.
- Konstruktorok nem módosíthatók az **abstract** kulcsszóval.
- Az **abstract** függvények nem lehetnek egyben privát függvények is.
- A felületosztályok alapértelmezés szerint elvontak.

Lásd még

- class
- interface

assert (vezérlőutasítás)

Utasításforma

assert *logikai kifejezés*;

vagy

assert *logikai-kifejezés* : *kifejezés*;

Leírás

Az **assert** (előfeltételezés) fenntartott szó egy megadott logikai kifejezéssel kapcsolatos előfeltételezés ellenőrzésére szolgál egy programban. Az **assert** utasítást egy logikai kifejezés követi, amelynek kiértékelése megelőzi az azt követő kód végrehajtását. Amennyiben az előfeltételezés „false" (hamis) eredményt ad, a rendszer egy **AssertionException** kivételt vált ki. Ha az előfeltételezés „true" (igaz) eredményt ad, a rendszer nem vált ki kivételt, és a feldolgozás úgy folytatódik, mintha az előfeltételezés nem lenne jelen. Ily módon használva az előfeltételezés igen hatékony programozási szerkezet lehet.

Példa

```
public class AssertTest {
 // Egyszerű előfeltételezés-ellenőrzés.
 // Beállítunk egy változót, majd előfeltételezünk
 // egy ismerten hamis eredményt.
 // Ennek a kódnak egy AssertionException kivételt
 // kell kiváltania. A kódot mindenképpen
 // az -ea kapcsolóval futtassuk a parancssorból.
 public static void main (String[] args) {
   int alpha = 5;
   assert (alpha < 5);
   alpha ++
 }
}
```

Tippek

- Tehetünk előfeltételezéseket a programba, és azt lefordíthatjuk, de alapértelmezés szerint végrehajtáskor az előfeltételezések kikapcsolt állapotúak. Az előfeltételezések bekapcsolásához és ellenőrzéséhez a kód végrehajtásakor használjuk az `-enableassertions` vagy az `-ea` parancssori kapcsolót. Példa: `java -ea AssertTest`
- Az `assert` utasítás másik formájának (amelyben két kifejezés szerepel) használatával egy értéket adhatunk át az AssertionException konstruktorának, ezáltal további információkat szolgáltathatunk a kivétel kiváltásakor (és a kimenet létrehozásakor). Az AssertionException konstruktora egyetlen boolean, char, double, float, int, long vagy Object típusú értéket fogad. Ez utóbbi esetben az Object típus karakterlánccá alakítva adódik át. Ezáltal a fenti előfeltételezés így írható újra: `assert (alpha < 5): "alpha < 5";`

Lásd még

- boolean
- false
- true

boolean (adattípus)

Utasításforma

```
[ módosító ] boolean változónév
                    ➥ [ = kezdőérték-kifejezés ];
```

Leírás

A `boolean` (Boole-féle, logikai) fenntartott szó használatával egy vagy több `boolean` adattípusú, „true" (igaz) vagy „false" (hamis) értéket tartalmazó változót vezethetünk be.

Példa

```
// Egy found nevű logikai változót vezet be,
// és kezdőértékét "igaz"-ra állítja:
boolean found = true;

// Bevezeti a minFound és maxFound nevű logikai
// változókat, és a maxFound kezdőértékét hamisra
// állítja.
boolean minFound, maxFound = false;
```

Tippek

- Egy **boolean** változó bevezetésekor a változó kezdőértéke is beállítható, ha a változó neve után egy egyenlőségjelet teszünk, és beírjuk a kezdőérték-kifejezést (az utasításformában ezt a szögletes zárójellel jeleztük).
- Több **boolean** változó is bevezethető egyazon utasításban, ha a változók neveit vesszővel választjuk el egymástól (lásd a második példát).
- Az osztályváltozóként megadott, de kezdőértékkel el nem látott **boolean** változók az alapértelmezett „false" értéket kapják.
- Függvényváltozóként a **boolean** változókat használat előtt mindenképpen el kell látnunk kezdőértékkel.
- A java.lang.Boolean osztály a **boolean** alap adattípus burkolóosztálya.
- A **boolean** változók nem alakíthatók át másik adattípusra, és más adattípusok sem alakíthatók át **boolean** értékekre.

Lásd még

- false
- true
- java.lang.Boolean

break (vezérlőutasítás)

Utasításforma

```
break [ címke ];
```

Leírás

A **break** (kitörés) utasítás (címke nélkül használva) leállítja annak a switch, while, do vagy for (a **break** céljaként ismert) ciklusnak a végrehajtását, amelyikben található, és a kód végrehajtását a ciklus után folytatja.

A **break** utasítás tartalmazhat egy nem kötelező címkét is. Ha van ilyen, akkor a kód végrehajtása a címkét viselő utasításnál folytatódik.

Példák

❶

```
// Egész típusú, age (életkor) nevű változót
// használó switch utasítás. A break
// utasítás megakadályozza, hogy a végrehajtás
// a következő case ággal folytatódjon.
switch (age) {
  case 16:
    System.out.println ("Of legal driving age.");
    break;

  case 18:
    System.out.println ("Of legal voting age.");
    break;
}
```

❷

```
// Végignézzük az osztályfelsorolást (diákok
// neveit tartalmazó String tömb), hogy az
// tartalmaz-e egy megadott célt (String).
// Ha a cél nem található, a keresés véget ér.
```

```
// A keresés eredményének jelölésére a keresés
// eredményétől függetlenül beállítunk
// egy logikai jelzőt.
for (int index = 0; index < classSize; index ++) {
  if (classList[index].equals (target)) {
    found = true;
    break;
  } else
    found = false;
}
```

Tippek

- Amennyiben a **break** utasítás nem switch, while, do vagy for ciklusban található, fordítási hiba lép fel.
- Ha egy **break** utasítás egy try blokkon belül van, a vezérlés a hozzá tartozó finally záradék végrehajtása után tér vissza a **break** céljához.
- A többszörösen beágyazott ciklusokból néha egy címkével rendelkező **break** utasítással szoktak kilépni.
- A **break** utasítás használata (kivéve, ha a switch utasítással együtt használjuk) rossz programozási gyakorlatnak számít.
- Amennyiben hiányzik a **break** utasítás egy case záradék végéről, a végrehajtás a következő **case** címkével és annak utasításaival (amennyiben vannak ilyenek) folytatódik.

Lásd még

- case
- do
- finally
- for
- switch
- try
- while

byte (adattípus)

Utasításforma

```
[ módosító ] byte változónév [
                          ➥ = kezdőérték-kifejezés ];
```

Leírás

A **byte** (bájt) fenntartott szó egy vagy több **byte** adattípusú változó bevezetésére szolgál. A Java nyelvben a **byte** 8 bites előjeles érték; legkisebb értéke –128, legnagyobb értéke pedig 127.

Példa

```
// Egy alpha nevű, byte típusú változót vezet
// be, és kezdőértékét 19-re állítja.
byte alpha = 19;

// Bevezeti a beta és a gamma nevű, byte
// típusú változót, és a gamma kezdőértékét
// -5-re állítja.
byte beta, gamma = -5;
```

Tippek

- A **byte** típusú változó bevezetésekor beállítható a változó kezdőértéke is, ha a változó neve után egy egyenlőségjelet teszünk, és beírjuk a kezdőérték-kifejezést (az utasításformában ezt a szögletes zárójellel jeleztük).
- Több **byte** típusú változó is bevezethető egyazon utasításban, ha a változók neveit vesszővel választjuk el egymástól (lásd a második példát).
- Az osztályváltozóként megadott, de kezdőértékkel el nem látott **byte** változók az alapértelmezett nulla értéket kapják.

- Függvényváltozóként a `byte` változót használat előtt mindenképpen el kell látnunk kezdőértékkel.
- A java.lang.Byte osztály a `byte` alap adattípus burkolóosztálya.

Lásd még

- java.lang.Byte

case (vezérlőutasítás)

Utasításforma

```
switch ( kifejezés ) {
  // Egy vagy több ilyen formátumú case záradék:
  [case állandókifejezés:
       utasítás(ok);
       [ break; ]
  ]

  [ default:
       utasítás(ok);
       [ break; ]
  ]
}
```

Leírás

A `case` (eset) címkét switch utasításokon belül használjuk kódblokkok meghatározására, amelyek akkor hajtódnak végre, amikor a switch kifejezés a `case` címke állandójának értékét eredményezi.

Példák

❶

```
char letter = 'i';
String characterType = "unknown";
```

```
// switch utasítás a "letter" változó értéke
// alapján
switch (letter) {
   // Ha a "letter" 'a', 'e', 'i', 'o' vagy 'u',
   // akkor a characterType legyen "vowel",
   case 'a':
   case 'e':
   case 'i':
   case 'o':
   case 'a':
     characterType = "vowel";
     break;

   // egyébként a típust jelölő characterType
   // változó értéke legyen "consonant".
   default:
     characterType = "consonant";
     break;
}
```

❷

```
int value = 3;

// switch utasítás a "value" változó értéke
// alapján
switch (value) {

   // Ha az érték 1, akkor eggyel növeljük.
   case 1:
     value ++;
     break;

   // Ha az érték 3, akkor kettővel növeljük.
   case 3:
     value += 2;
     break;
}
```

Tippek

- Az állandókifejezésnek char, byte, short vagy integer típusúnak kell lennie.
- Az ugyanazon switch utasításon belül található állandókifejezéseknek egyedinek kell lenniük. A **case** címkéket általában egy vagy több break utasítással lezárt utasítás követi. Amennyiben hiányzik a break utasítás, a végrehajtás a következő **case** címkével és annak utasításaival (ha vannak ilyenek) folytatódik. A **case** címkék követhetik is egymást ezzel a módszerrel, ahogy az az első példában látható.
- A default címke (a **case** címkéhez hasonlóan) egy utasításblokk meghatározására használható, amelynek végrehajtása akkor történik meg, ha egyik **case** címke sem egyezik meg az állandókifejezéssel.

Lásd még

- break
- default
- switch

catch (vezérlőutasítás)

Utasításforma

```
try {
        utasítás(ok);
}
 [ catch (kivételtípus kivételazonosító-név) {
        utasítás(ok);
} ]
[ finally {
        utasítás(ok);
} ]
```

Leírás

A `catch` (elfogás) fenntartott szó a nagyobb try/`catch`/finally utasítás része. Ezen fenntartott szavak mindegyike egy egy vagy több utasításból álló blokkot foglal magában, amelynek végrehajtása bizonyos körülményektől függ. Először a try blokk végrehajtása történik meg, utasításai végrehajtásának befejezéséig vagy egy kivétel-kiváltásáig.

Kivétel esetén sorban egymás után egyenként megtörténik a `catch` záradékok vizsgálata annak meghatározására, hogy a kiváltott kivétel megfelel-e egy megadott kivételtípusnak. Akkor van illeszkedés, ha a kiváltott kivétel típusa hozzárendelhető az elkapott típushoz. Amennyiben van illeszkedés, megtörténik a megfelelő `catch` záradékban található utasítás(ok) végrehajtása.

Ha nem lép fel kivétel egy try blokk végrehajtása során, akkor a feldolgozás az utolsó `catch` blokk után folytatódik. Amennyiben van finally záradék, akkor a rendszer azt is végrehajtja, egyébként az utolsó `catch` záradékot követő utasítás(ok) következnek.

Példa

```
// Ez a függvény egy új File objektumot próbál
// létrehozni a megadott fájlnévből.
// Az objektum létrehozó- és canWrite függvénye
// kivételek kiváltását eredményezheti,
// több okból kifolyólag. Ezért ennek a kódnak
// a lefordításához két catch utasítás szükséges.
public void checkFile (String filename) {
    try {
      File inputFile = new File (filename);
      if (inputFile.canWrite ())
        System.out.println ("The file: "
          ➥ + filename + " can be written to.");
    }
```

```
    catch (NullPointerException nullPtr) {
      System.err.println (nullPtr);
    }
    catch (SecurityException securityExpt) {
      System.err.println (securityExpt);
    }
}
```

Tippek

- Egy try blokkhoz nulla vagy több **catch** záradék tartozhat, amennyiben ezen záradékok mindegyike különböző típusú kivételeket fog el. Vegyük figyelembe, hogy a különböző **catch** záradékokhoz tartozhatnak olyan kivételtípusok, amelyek egymás alosztályai.
- A **catch** záradékot általában egy lehetséges hiba kijavítására és a feldolgozás folytatására használják a programozók, néhányan azonban figyelmeztető üzenetet íratnak ki vele, vagy arra használják, hogy nyomon kövessék a program futását. Kritikus helyzetben lehetőség van a program leállítására egy **catch** záradékon belül.

Lásd még

- finally
- try

char (adattípus)

Utasításforma

```
[ módosító ] char változónév [
                        ➥ = kezdőérték-kifejezés ];
```

Leírás

A **char** (karakter) fenntartott szó egy vagy több **char** adattípusú (karakter adatértékeket tartalmazó) változó beveze-

tésére szolgál. A Java nyelvben a karakterek 16 bites előjel nélküli értékek, amelyek a Unicode karakterkészlet karaktereit jelölik. A Unicode által támogatott több mint 65 000 karakterszimbólum közül csak 128 használatos hagyományosan programozásra és angol nyelvű információk megjelenítésére.

Példák

❶

```
// Bevezeti a firstLetter nevű char típusú
// változót, és annak kezdőértékéül a 'p'
// karaktert adja.
char firstLetter = 'p';
```

❷

```
// Bevezeti az alpha és beta nevű char
// változókat, és a beta kezdőértékéül a '*'
// karaktert állítja be.
char alpha, beta = '*';
```

Tippek

- Egy **char** típusú változó bevezetésekor beállítható a változó kezdőértéke is, ha a változó neve után egy egyenlőségjelet teszünk, és beírjuk a kezdőérték-kifejezést (az utasításformában ezt a szögletes zárójellel jeleztük).
- Több **char** típusú változó is bevezethető egyazon utasításban, ha a változók neveit vesszővel választjuk el egymástól (lásd a második példát).
- Az osztályváltozóként megadott, de kezdőértékkel el nem látott karakterváltozók az alapértelmezett '\u0000' értéket (null karakter) kapják.
- Függvényváltozóként a **char** változót használat előtt mindenképpen el kell látnunk kezdőértékkel.
- A java.lang.Character osztály a **char** alap adattípus burkolóosztálya.

- A Unicode karakterkészletről további információk a Unicode honlapján, a `http://www.unicode.org` címen találhatók.

Lásd még

- java.lang.Character

class (osztállyal kapcsolatos)

Utasításforma

```
[ módosító ] class osztálynév
   [ extends  osztálynév ]
   [ implements osztálynév [, osztálynév...] ] {
   // Itt kell meghatározni az osztályváltozókat,
   // függvényeket és belső osztályokat.
}
```

Leírás

A `class` (osztály) fenntartott szó egy osztály megvalósításának meghatározására szolgál a Java nyelvben.

Példa

```
public class Student {
    // A diák nevét és címét karakterláncként
    // tárolja.
    private String name;
    private String address;

    // Visszaadja a diák nevét.
    public String getName (){
        return name;
    }
}
```

Tippek

- Az osztályok bevezethetők egy másik osztályon belül is, így beágyazott vagy belső osztályokat hozhatunk létre.
- Egy osztály örökölhet egy másik osztálytól – ezt nevezzük öröklésnek. Az öröklés az extends fenntartott szó használatával történik. Egy osztály csak egy szülőosztályától örökölhet.
- A konkrétan nem egy másik osztályt kibővítő osztályok az Object osztályt bővítik ki.
- Osztály meghatározásából objektum a new fenntartott szó használatával hozható létre.
- Egy osztály meghatározása a következők közül eggyel vagy többel módosítható: public, protected, private, abstract, static, final, strictfp.

Lásd még

- abstract
- extends
- new
- java.lang.Object

const (nem használatos)

Utasításforma

Nincs, lásd a Leírást.

Leírás

A `const` (állandó) szó a Java programozási nyelv fenntartott szava, de jelenleg nem használatos.

Példa

Nincs.

Tippek

- Ha egy állandó értéket próbálunk meghatározni, azt megtehetjük a final fenntartott szó használatával is.

Lásd még

- final

continue (vezérlőutasítás)

Utasításforma

```
continue;
```

Leírás

A **continue** (folytatás) utasítás befejezi egy ciklus aktuális ismétlésének feldolgozását az utasítás helyén. Ez a viselkedés hasonló a break utasításéhoz, de a **continue** utasítás a ciklus teljes elhagyása helyett kiértékeli a ciklus feltételét, és újra végigmegy a cikluson, amennyiben a feltétel igaznak bizonyul.

Példa

```
while (alpha < 5) {
    // Az alpha változó növelése és a ciklus ezen
    // ismétlésének befejezése.
    alpha ++;
    if (alpha > 0)
      continue;

    // A beta változó növelésének végrehajtására
    // soha nem kerül sor.
    beta ++;
}
```

Tippek

- A példában azért szerepel az if utasítás, hogy egy lehetséges útvonalat biztosítson a beta változót növelő utasításhoz. Az if utasítás nélkül a fordító hibát jelez, mert a beta változót növelő utasítás nem érhető el (a `continue` utasítás jelenléte miatt, ami mindig hatással van a végrehajtásra). Az if utasítás felvételével a fordító nem tudja megállapítani, hogy a `continue` mindig hatással van-e a végrehajtási útvonalra (például ha alpha <= 0, akkor a `continue` kimarad).
- A `continue` utasítás do és for ciklusokban is használható.
- Amennyiben a `continue` utasítás nem do, for vagy while cikluson belül szerepel, fordítási hiba lép fel.
- A `continue` utasítás használatával vigyázni kell, mert nehezen olvashatóvá teheti a forráskódot.

Lásd még

- do
- for
- true
- while

default (vezérlőutasítás)

Utasításforma

```
default:
```

Leírás

A `default` (alapértelmezés) címke switch utasításokon belül használható olyan végrehajtható kódblokkok meghatározására, amelyek végrehajtása abban az esetben történik meg, ha a switch kifejezésének értéke nem felel meg egyik létező case címke állandókifejezésének sem.

Példa

```
int value = 3;

// switch utasítás végrehajtása az előzőleg
// meghatározott értékre.
switch (value) {
    // Ha az érték 3, kettővel növeljük.
    case 3:
        value += 2;
        break;

    // Az alapértelmezett eset: eggyel növeljük
    // az értéket.
    default:
        value ++;
        break;
}
```

Tippek

- A **default** címkét a switch utasításokban szintén szereplő case címkéktől eltérően nem követi állandókifejezés.
- A **default** címke általában a switch utasításblokk utolsó helyén szerepel, így könnyebben megkülönböztethető a case címkéktől. Mindazonáltal ha a **default** címke előbb szerepel a címkék listájában, az sem akadályozza a switch utasításblokkban később megtalált case címkék helyes illesztését.

Lásd még

- case
- switch

do (vezérlőutasítás)

Utasításforma

```
do
        utasítás;
while ( logikai-kifejezés );
```

Leírás

A `do` fenntartott szó használatával olyan ciklus hozható létre, amely egyszer vagy többször végrehajtja a megadott utasítást, amíg a while fenntartott szót követő kifejezés hamis értéket nem ad.

Példa

```
// A ciklus ismétlése és az alpha változó
// értékének növelése addig, amíg az alpha
// értéke meg nem haladja az 5-öt.
do
   alpha += beta;
while (alpha < 5);
```

Tippek

- A `do` ciklus utasításrészének végrehajtása legalább egyszer mindig megtörténik.
- Amennyiben a logikai kifejezés nem logikai eredményt ad (igaz vagy hamis), fordításkor hiba lép fel.
- Több utasítást is végrehajtathatunk a `do` cikluson belül, ha a végrehajtandó utasításokat kapcsos zárójelek közé tesszük és így utasításblokkot hozunk létre.
- Győződjünk meg róla, hogy a logikai kifejezés megváltozik a ciklus belsejében. Amennyiben a kifejezés értéke nem változik, a ciklus végrehajtása vég nélkül folytatódik. Ezt a helyzetet *végtelen ciklusnak* nevezzük.

- Ne felejtsük el, hogy egy logikai kifejezés több logikai műveletjellel összekapcsolt logikai kifejezésből is állhat.
- A `do` ciklus tartalmazhat más ciklusokat is (`do`, for, while), ezáltal beágyazott ciklusokat hozhatunk létre.
- A `do` ciklusokban használhatunk break és continue utasításokat.
- Gyakran elkövetett hiba, hogy elmarad a logikai kifejezés utáni pontosvessző (;).

Lásd még

- break
- continue
- do
- false
- for
- true
- while

double (adattípus)

Utasításforma

```
[ módosító ] double változónév [
                        ➥ = kezdőérték-kifejezés ];
```

Leírás

A `double` (kétszeres) fenntartott szó egy vagy több `double` adattípusú változó bevezetésére szolgál. A Java nyelvben a `double` típus 64 bites előjeles értékeket jelent. A `double` adattípus lehetséges legkisebb értéke körülbelül –1,7E+308, legnagyobb értéke körülbelül 1,7E+308.
A `double` adattípusú értékek legfeljebb tizenöt értékes számjegyből állhatnak.

Példák

❶

```
// Bevezeti az alpha nevű double típusú változót,
// és annak kezdőértékét 45,6964-re állítja.
double alpha = 45.6964;
```

❷

```
// Bevezeti a beta és gamma nevű double típusú
// változókat, és a gamma kezdőértékét
// -55,112D-re állítja.
double beta, gamma = -55.112D;
```

Tippek

- Egy `double` típusú változó bevezetésekor beállítható a változó kezdőértéke is, ha a változó neve után egy egyenlőségjelet teszünk, és beírjuk a kezdőérték-kifejezést (az utasításformában ezt a szögletes zárójellel jeleztük).
- Több `double` típusú változó is bevezethető egyazon utasításban, ha a változók neveit vesszővel választjuk el egymástól (lásd a második példát).
- Az osztályváltozóként megadott, de kezdőértékkel el nem látott `double` típusú változók az alapértelmezett nulla értéket kapják.
- Függvényváltozóként a `double` változót használat előtt mindenképpen el kell látnunk kezdőértékkel.
- A lebegőpontos literál értékekkel való összekeverés veszélyének csökkentése érdekében a `double` literál értékek mögé a `d` vagy a `D` karakter írható (lásd a második példát).
- A java.lang.Double osztály a `double` alap adattípus burkolóosztálya.

Lásd még

- java.lang.Double

else (vezérlőutasítás)

Utasításforma

```
if ( logikai-kifejezés )
        utasítás;
[ else
        utasítás;
]
```

Leírás

Az **else** (egyébként) fenntartott szó használatával alternatív utasítás adható meg, amelynek végrehajtására akkor kerül sor, ha az if (ha) utasítás logikai kifejezése hamis eredményt ad. Az if utasítással együtt használva a programozó egy-egy utat hozhat létre az igaz és a hamis eredménynek.

Példák

❶

```
// Ellenőrizzük, hogy a tanfolyam hallgatóinak
// száma egyenlő-e 25-tel. Ha így van, lezárjuk
// ezt a munkamenetet, egyébként nyitva hagyjuk.
if (numberOfStudents == 25)
    classFull = true;
else
    classFull = false;
```

❷

```
// Ha a felhasználó által bevitt érték a "Java"
// karakterlánc, akkor kiírjuk két másik
// objektumközpontú nyelv nevét, különben két
// eljárásközpontú nyelv nevét írjuk ki.
if (enteredValue.equals ("Java")) {
    System.out.println ("Smalltalk");
    System.out.println ("C++");
} else {
    System.out.println ("Pascal");
    System.out.println ("Fortran");
}
```

Tippek

- Az `else` fenntartott szó (és a hozzá tartozó utasítás) nem kötelező része az if feltételes utasításnak.
- Több utasítást is végrehajtathatunk az `else` záradék részeként, ha a végrehajtandó utasításokat kapcsos zárójelek közé tesszük, és így utasításblokkot hozunk létre (lásd a második példát).

Lásd még

- false
- if
- true

enum (adattípus)

Utasításforma

`enum` *adattípus* { *érték1*, *érték2*, *érték3*,...}

Leírás

Az `enum` (felsorolás) fenntartott szó egy új adattípust és a típus által tárolható megfelelő értékeket határozza meg. Az ebből az új adattípusból létrehozott azonosítók csak az adattípus meghatározásában felsorolt értékeket tárolhatják (*érték1*, *érték2*, *érték3*).

Példa

```
public class U2 {
  // A Member adattípus meghatározása és
  // kizárólag a következő értékek engedélyezése.
  enum Member {Bono, Larry, Edge, Adam};

  // Megadott azonosítók létrehozása a zenekar
  // egyes tagjaihoz, és azok megfelelő
```

```
    // értékeinek beállítása a tagok zenekarban
    // betöltött szerepe alapján.
    public static void main (String[] args) {
      Member vocals = Member.Bono;
      Member bass = Member.Adam;
      Member drums = Member.Larry;
      Member lead = Member.Edge;

      // A zenekartagokhoz tartozó értékek
      // kiíratása különböző szempontok szerint.
      System.out.println ("Vocals: " + vocals);
      System.out.println ("Drums: "
                          ➥ + drums.name());
      System.out.println ("Lead Guitar: "
                          ➥ + lead.ordinal());
      System.out.println ("Bass Guitar: " + bass);
    }
}
```

Tippek

- Az **enum** típus különleges Java osztály, amely a Comparable (összehasonlítható) és a Serializable (adatfolyamba írható) felületeket is megvalósítja.
- Az **enum** típus értékeinek tárolása egész értékekként történik. Az első azonosító nullás (0) értékként, a második egyes (1) értékként tárolódik, és így tovább.
- Az azonosítók értékének karakterlánc-ábrázolása az **enum** típus **name** (név) függvényével érhető el.
- Az azonosítók értékéhez tartozó számérték az **enum** típus **ordinal** (sorszám) függvényével érhető el.
- Ha érvénytelen értéket (olyat, ami nem szerepel a megadott lehetséges értékek között) próbálunk kiosztani egy **enum** változónak, fordítási hiba lép fel.
- Számértékek nem rendelhetők **enum** típushoz.

- Az **enum** típusok egy továbbfejlesztett változata lehetővé teszi konstruktorok, függvények és példányváltozók használatát is a felsoroláson belül.

Lásd még

- java.lang.Comparable
- java.io.Serializable

extends (osztállyal kapcsolatos)

Utasításforma

```
[ módosító ] class osztálynév
   [ extends  osztálynév ]
   [ implements osztálynév [, osztálynév...] ] {...}
```

vagy

```
[ módosító ] interface felületnév extends
                                    ➥ felületnév {...}
```

Leírás

Az **extends** (bővíti) fenntartott szó egy osztály meghatározásának módosítására szolgál, és azt jelzi, hogy a gyermekosztály a megnevezett szülőosztályt (osztálynév) bővíti.

Példa

```
public class Rectangle extends Shape {
        ...
}
```

Tippek

- Az öröklés az az elv, amely szerint egy osztályt egy másik (létező) osztályból származtatunk.

- A létező osztályt (a példában Shape) szülőosztálynak vagy szuperosztálynak nevezzük. Az új osztályt (származtatott osztály; ebben a példában a Rectangle) gyermekosztálynak vagy alosztálynak is nevezzük.
- A Java programozási nyelvben egy osztály csak egy szülőosztálytól örökölhet, ezért az **extends** fenntartott szó használatakor soha nem nevezünk meg egy kibővítendő osztálynál többet.
- Mind az elvont, mind pedig a felületosztályok kibővíthetők.
- Ha egy származtatott osztály nem szülőosztály kibővítésével jön létre, akkor a java.lang.Object osztály az alapértelmezett szülőosztály.

Lásd még

- abstract
- class
- interface

final (módosító)

Utasításforma

```
[ módosító ] [ final ] class osztálynév
  [ extends osztálynév ]
  [ implements osztálynév [, osztálynév…] ] {…}
```

vagy

```
[ módosító ] [ final ] adattípus függvénynév ( [
[ final ] paraméterdeklarációk ] )
  [ throws kivételtípus [, kivételtípus…] ] {…}
```

vagy

```
[ módosító ] [ final ] adattípus változónév [
                    ➥ = kezdőérték-kifejezés ];
```

Leírás

A `final` (végleges) módosító osztályok, függvények, mezők és függvényeknek átadott paraméterek (formális paraméterek) tulajdonságainak megváltoztatására szolgál. A `final` fenntartott szó használata általában azt jelenti, hogy az általa módosított elem nem változtatható meg.

Példa

```
public final class PI {
    // Beállítja a PI értékét, amelyet ebben az
    // osztályban használunk.
    final double VALUE = 3.1415;

    // Visszaadja az rValue sugarú kör területét.
    public final double
                    ➥ getRSquared (double rValue) {
        return VALUE * rValue * rValue;
    }

    // Visszaadja PI értékét a megadott szorzóval
    // megszorozva.
    public final double
                    ➥ getMultPI (final int mult) {
        return VALUE * mult;
    }
}
```

Tippek

- A `final` módosítóval bevezetett osztályok nem lehetnek elvontak, és nem hozhatók létre belőlük alosztályok.

- A `final` módosítóval bevezetett függvények nem írhatók felül, és nem vezethetők be abstract módosítóval.
- A `final` módosítóval bevezetett mezők (osztályok vagy felületek) nem módosíthatók, és kötelező kezdőértékkel ellátni őket a bevezetés helyén.
- A `final` fenntartott szót formális paraméterekre (egy tagfüggvény paramétereire) alkalmazva megakadályozhatjuk a paraméterértékek módosulását.
- Konstruktorok nem módosíthatók a `final` fenntartott szóval.

Lásd még

- abstract
- class

finally (vezérlőutasítás)

Utasításforma

```
try {
        utasítás(ok);
}
[ catch ( kivételtípus kivételazonosító-név ) {
        utasítás(ok);
} ]
[ finally {
        utasítás(ok);
} ]
```

Leírás

A `finally` (végül) fenntartott szó a nagyobb try/catch/`finally` utasítás része. Ezen fenntartott szavak mindegyike egy egy vagy több utasításból álló blokkot foglal magában,

amelyek végrehajtása bizonyos körülményektől függ. Először a try blokk végrehajtása történik meg, utasításai végrehajtásának befejezéséig vagy egy kivétel kiváltásáig.

Kivétel esetén sorban egymás után egyenként megtörténik a catch záradékok vizsgálata annak meghatározására, hogy a kiváltott kivétel megfelel-e egy megadott catch záradéknak. Akkor van illeszkedés, ha a kiváltott kivétel típusa hozzárendelhető az elkapott típushoz. Amennyiben van illeszkedés, megtörténik a megfelelő catch záradékban található utasítás(ok) végrehajtása.

Ha a try blokk végrehajtása során nem lép fel kivétel, a feldolgozás az utolsó catch blokk után folytatódik. Amennyiben van **finally** záradék, akkor a rendszer azt is végrehajtja, egyébként pedig az utolsó catch záradékot követő utasítás következik.

Példa

```
// Ez a függvény ellenőrzi, hogy a megadott
// fájlba lehetséges-e az írás.
// Jegyezzük meg, hogy a fájl megadása
// a függvénynek átadott String típusú
// paraméterrel történik.
public void checkFile (String filename) {
   File inputFile = null;
   try {
     inputFile = new File (filename);
     if (inputFile.canWrite ())
       System.out.println ("The file: "
         ➥ + filename + " can be written to.");
   }
   // Elkap egy lehetséges NullPointerException
   // kivételt a File konstruktorából.
   catch (NullPointerException nullPtr) {
       System.err.println (nullPtr);
   }
```

```
    // Elkap egy lehetséges SecurityException
    // kivételt a canWrite tagfüggvényből.
    catch (SecurityException securityExpt) {
        System.err.println (securityExpt);
    }
    finally {
        // Ha sikeresen létrejött egy File
        // objektum, akkor kiíratjuk
        // annak karakterlánc-ábrázolását.
        if (inputFile != null)
            System.out.println (inputFile);
    }
}
```

Tippek

- A catch záradéktól eltérően a try/catch blokkok csak egy **finally** záradékot tartalmazhatnak.

Lásd még

- catch
- try

float (adattípus)

Utasításforma

```
[ módosító ] float változónév [
                        ➥ = kezdőérték-kifejezés ];
```

Leírás

A **float** fenntartott szó egy vagy több **float** adattípusú változó bevezetésére szolgál. A Java nyelvben a **float** típus 32 bites előjeles érték. A **float** adattípus lehetséges legkisebb értéke körülbelül –3,4E+8, legnagyobb értéke körülbelül 3,4E+38. A **float** adattípusú értékek legfeljebb hét értékes számjegyből állhatnak.

Példák

❶

```
// Bevezeti az alpha nevű float típusú változót,
// és annak kezdőértékét 6,964-re állítja.
float alpha = 6.964f;
```

❷

```
// Bevezeti a beta és gamma nevű double típusú
// változókat, és a gamma kezdőértékét
// -12,345F-re állítja.
float beta, gamma = -12.345F;
```

Tippek

- Egy `float` típusú változó bevezetésekor beállítható a változó kezdőértéke is, ha a változó neve után egy egyenlőségjelet teszünk, és beírjuk a kezdőérték-kifejezést (az utasításformában ezt a szögletes zárójellel jeleztük).
- Több `float` típusú változó is bevezethető egyazon utasításban, ha a változók neveit vesszővel választjuk el egymástól (lásd a második példát).
- Az osztályváltozóként megadott, de kezdőértékkel el nem látott `float` típusú változók az alapértelmezett nulla értéket kapják.
- Függvényváltozóként a `float` változót használat előtt mindenképpen el kell látnunk kezdőértékkel.
- A lebegőpontos literál értékekkel való összekeverés veszélyének csökkentése érdekében a `float` literál értékek mögé az `f` vagy az `F` karakter írható (lásd a példákat).
- A java.lang.Byte osztály a `float` alap adattípus burkolóosztálya.

Lásd még

- java.lang.Float

for (vezérlőutasítás, léptető stílusú)

Utasításforma

```
for ( gyűjteménytípus ciklusváltozó :
                                    ➥ gyűjteménynév )
        utasítás;
```

Leírás

A `for` ciklus léptető változata új szolgáltatás a Java 5.0-ás változatában. Ez a forma végigléptet objektumok egy gyűjteményén (gyűjteménynév) a gyűjtemény minden egyes objektumát bejárva. A léptetés során az éppen meglátogatott objektumra hivatkozunk ciklusváltozóként. A gyűjtemény minden objektuma egy megadott gyűjteménytípusba tartozik. A `for` ciklus ezen formájának használatával egyszerűen bejárhatjuk egy gyűjtemény minden elemét.

Példa

```
// Kiír egy listát a hallgatókról (String típusú
// objektumok listáját)
public void printList (ArrayList<Student> list) {
 for (Student member : list)
  System.out.println ("Student name: " + member);
}
```

Tippek

- Több utasítást is végrehajtathatunk a `for` cikluson belül, ha a végrehajtandó utasításokat kapcsos zárójelek közé tesszük, és így utasításblokkot hozunk létre (lásd a második példát a hagyományos `for` ciklus összefoglalásánál).
- A `for` ciklusnak ezt az új formáját `for`-each ciklusnak is nevezik.

- A `for`-each ciklus beágyazható egy másik ciklusba, és tömbökkel is működik.

Lásd még

- for (vezérlőutasítás, hagyományos stílusú)

for (vezérlőutasítás, hagyományos stílusú)

Utasításforma

```
for ( értékadó-kifejezés; tesztelő-kifejezés;
                          ➥ frissítő-kifejezés )
        utasítás;
```

Leírás

A `for` fenntartott szó használatával olyan ciklus hozható létre, amely addig hajtja végre a megadott utasítást, amíg a tesztelő kifejezés hamis eredményt nem ad. A ciklus első végrehajtási kísérlete előtt megtörténik az értékadó kifejezés végrehajtása (ami általában a ciklus vezérlőváltozóinak beállítására szolgál).

Ez után a tesztelő kifejezés végrehajtása következik. Amennyiben a tesztelő kifejezés igaz eredményt ad, megtörténik a ciklus belsejében található utasítás végrehajtása. A ciklus minden ismétlése után végrehajtódik a frissítő kifejezés, majd újra kiértékelődik a tesztelő kifejezés. Amennyiben a tesztelő kifejezés hamis eredményt ad, a ciklus végrehajtása véget ér, és a vezérlés a ciklust követő utasításhoz kerül.

Az értékadó kifejezés bevezethet egy változót is, és beállíthatja annak kezdőértékét (lásd a példákban). Amennyiben

a változó bevezetése az értékadó kifejezésben történt, a változó csak a cikluson belül érhető el, a cikluson kívül nem.

Az értékadó kifejezés és a frissítő kifejezés több, egymástól vesszővel elválasztott kifejezést is tartalmazhat (lásd a második példát), a tesztelő kifejezés pedig nulla vagy több logikai műveletjellel összekapcsolt több kifejezésből is állhat.

Példák

❶

```
// Ezt a ciklust az alpha változó irányítja,
// amely 5-ről 1-re csökken. A ciklus
// minden ismétlésénél megtörténik az alpha
// változó csökkentése és kiíratása.
for (int alpha = 5; alpha > 0; alpha--)
        System.out.println ("Az alpha értéke: "
                                        ➥ + alpha);
```

❷

```
// Ezt a ciklust a beta változó irányítja, amely
// 5-ről 1-re csökken. A beta a ciklus
// minden ismétlésénél eggyel csökken, és
// megtörténik
// a beta és gamma változók kiíratása.
for (int beta = 5, gamma = 0; beta > 0;
                              ➥ beta--, gamma++) {
        System.out.println ("A beta értéke: "
                                        ➥ + beta);
        System.out.println ("A gamma értéke: "
                                        ➥ + gamma);
}
```

Tippek

- Több utasítást is végrehajtathatunk a `for` cikluson belül, ha a végrehajtandó utasításokat kapcsos zárójelek

közé tesszük, és így utasításblokkot hozunk létre (lásd a második példát).

- Győződjünk meg róla, hogy a logikai kifejezés valóban megváltozik a ciklus belsejében. Amennyiben a kifejezés értéke soha nem változik, a ciklus végrehajtása vég nélkül folytatódik. Ezt a helyzetet *végtelen ciklusnak* nevezzük.
- A `for` ciklus tartalmazhat más ciklusokat (do, `for`, while) is, ezáltal beágyazott ciklusokat hozhatunk létre.
- A `for` ciklusokban használhatunk break és continue utasításokat.

Lásd még

- break
- continue
- do
- for (vezérlőutasítás, léptető stílusú)
- while

goto (nem használatos)

Utasításforma

Nincs, lásd a Leírást.

Leírás

A `goto` szó a Java programozási nyelv fenntartott szava, de jelenleg nem használatos.

Példa

Nincs.

Tippek

- A `goto` utasítást más programozási nyelvek (például a C és a C++) arra használják, hogy a végrehajtást a program egy másik részére irányítsák át.

Lásd még

- Nincs.

if (vezérlőutasítás)

Utasításforma

```
if ( logikai-kifejezés )
        utasítás;
[ else
        utasítás;
]
```

Leírás

Az `if` (ha) fenntartott szó használatával feltételes utasítás építhető fel, amelynek végrehajtására akkor kerül sor, ha a logikai kifejezés igaz eredményt ad. Az utasítás végrehajtása csak abban az esetben történik meg, ha a logikai kifejezés igaz eredményt ad. Amennyiben a logikai kifejezés hamis, és van else záradék (az utasítás után), akkor az else utasítás végrehajtása történik meg.

Példák

❶

```
// Ellenőrizzük, hogy a tanfolyam hallgatóinak
// száma egyenlő-e 25-tel.
// Ha így van, zárjuk le ezt a munkamenetet
// a classFull (osztályTele) jelzőnek a "true"
// (igaz) értéket adva.
```

```
if (numberOfStudents == 25)
        classFull = true;
```

❷

```
// Ha a felhasználó által bevitt érték a "Java"
// karakterlánc, kiírjuk két másik
// objektumközpontú nyelv nevét.
if (enteredValue.equals ("Java")) {
        System.out.println ("Smalltalk");
        System.out.println ("C++");
}
```

Tippek

- Amennyiben a logikai kifejezés nem logikai eredményt („true" vagy „false") ad, fordítási hiba lép fel.
- Amennyiben a logikai kifejezés értéke „true", több utasítást is végrehajtathatunk, ha a végrehajtandó utasításokat kapcsos zárójelek közé tesszük, és így utasításblokkot hozunk létre (lásd a második példát).
- Az `if` utasítások (az `if` fenntartott szó, a logikai kifejezés, és az utasítás) egymásba ágyazhatók, tehát egy `if` utasítás végrehajtható egy másik `if` utasítás eredményeképp.

Lásd még

- boolean
- false
- true

implements (osztállyal kapcsolatos)

Utasításforma

```
[ módosító ] class osztálynév
  [ extends  osztálynév ]
  [ implements osztálynév [, osztálynév…] ] {…}
```

Leírás

Az `implements` (megvalósítja) fenntartott szó azt jelzi, hogy az adott osztály a meghatározott felületosztály függvényeinek megvalósításait tartalmazza.

Példa

```
public class Student implements Undergraduate {…}
```

Tippek

- Az `implements` és az extends fenntartott szó együtt arra használható, hogy egy osztály meghatározását módosítsuk.
- Egy osztály megvalósíthat több felületet is, ha az `implements` fenntartott szó után vesszővel elválasztva felsoroljuk a felületosztályok neveit.

Lásd még

- class
- extends

import (osztállyal kapcsolatos)

Utasításforma

`import` *típusnév*;

vagy

`import static` *típusnév.azonosítónév*;

Leírás

Az **import** (betöltés, beágyazás) hozzáférést biztosít a megadott osztályhoz. Az **import** deklarációt tartalmazó fájlból a nyilvános statikus függvényeket és adatokat érhetjük el, ezen kívül (amennyiben ez engedélyezett) az **import** deklarációt tartalmazó fájlban lévő osztályok példányosíthatják a megadott osztályt.

Példa

```
import java.util.Scanner;
import java.swing.*;
```

Tippek

- Alapértelmezés szerint minden osztály betölti a java.lang csomagot, ezért soha nem szükséges konkrétan beágyazni azt, ellentétben a többi csomaggal.
- Az **import** deklaráció valójában nem ágyazza be a megadott típusnév forráskódját az **import** utasítást tartalmazó fájlba, csak a fordítónak nyújt információkat.
- Ha a betöltendő osztály megnevezésekor az osztály nevét csillaggal (*) helyettesítjük, azzal azt mondjuk, hogy a csomagban lévő összes osztályt be szeretnénk tölteni (lásd a második példát).
- Azáltal, hogy a static fenntartott szót írjuk az **import** fenntartott szó és a betöltendő osztály neve közé, és hozzáírunk egy pontot és egy azonosítónevet, a programozók közvetlenül hivatkozhatnak a betöltött osztályban lévő statikus adatmezőkre. Példa:

```
// Betölti a PI statikus azonosítót
// a Math osztályból.
import static java.lang.Math.PI;

public class import {
 public static void main (String[] args) {
```

```
    // Megszerzi a PI értékét a java.lang.Math
    // osztályból
    System.out.println ("Two * PI = " + PI * 2.0);
  }
}
```

Lásd még

- static

instanceof (osztállyal kapcsolatos)

Utasításforma

hivatkozásnév **instanceof** *osztálynév*

Leírás

Az **instanceof** (példánya) fenntartott szó egy műveletet jelöl, ami annak érvényesítésére szolgál, hogy egy objektumra mutató adott hivatkozás (hivatkozásnév) egy megadott felület vagy osztály (osztálynév) példánya. Az **instanceof** művelet logikai eredményt („true" vagy „false") ad.

Példa

```
// Ellenőrizzük, hogy a miscObject egy String
// objektum. Amennyiben az, a típusát
// változtassuk vissza String-re, és írassuk ki
// a karakterlánc-ábrázolását.
if (miscObject instanceof String)
      System.out.println ("Student name: "
                  ➥ + ((String) miscObject));
```

Tippek

- Az **instanceof** műveletet gyakran használjuk annak biztosítására, hogy egy objektum egy bizonyos típusba tartozzon, mielőtt a típusát arra a típusra változtatnánk.
- A Java 5.0-ás kiadása előtt ez a helyzet leggyakrabban akkor fordult elő, amikor objektumokat helyeztünk egy

gyűjteménybe (listába, verembe, sorba stb.), amely a tartalmát Object típusú elemekként tartotta számon. Ha eltávolítottuk az objektumok egyikét a gyűjteményből, akkor használat előtt annak típusát valós osztályára kellett visszaváltoztatni. Ez az érvényesség-ellenőrzés (az **instanceof** művelet használatával) általában szerepelt a típusátalakítás előtt, annak érdekében, hogy biztosítsa az átalakítás sikerességét. A Java 5.0 gyűjteményeket támogató új általános típusával ez a fajta típusátalakítás elavulttá vált.

Lásd még

- class
- false
- interface
- true

int (adattípus)

Utasításforma

```
[ módosító ] int változónév [
                          ➥ = kezdőérték-kifejezés ];
```

Leírás

Az **int** (egész) fenntartott szó egy vagy több **int** adattípusú változó bevezetésére szolgál. A Java nyelvben a karakterek 32 bites előjeles értékek. Az **int** legkisebb értéke –2 147 483 648, legnagyobb értéke pedig 2 147 483 647.

Példák

❶

```
// Bevezeti az alpha nevű int típusú változót,
// és annak kezdőértékét 19-re állítja.
int alpha = 19;
```

❷

```
// Bevezeti a beta és gamma nevű int változókat,
// és a gamma kezdőértékét -5-re állítja.
int beta, gamma = -5;
```

Tippek

- Egy **int** típusú változó bevezetésekor beállítható a változó kezdőértéke is, ha a változó neve után egy egyenlőségjelet teszünk, és beírjuk a kezdőérték-kifejezést (az utasításformában ezt a szögletes zárójellel jeleztük).
- Több **int** típusú változó is bevezethető egyazon utasításban, ha a változók neveit vesszővel választjuk el egymástól (lásd a második példát).
- Az osztályváltozóként megadott, de kezdőértékkel el nem látott **int** változók az alapértelmezett nulla értéket kapják.
- Függvényváltozóként az **int** változót használat előtt mindenképpen el kell látnunk kezdőértékkel.
- A java.lang.Integer osztály az **int** alap adattípus burkolóosztálya.

Lásd még

- java.lang.Integer

interface (osztállyal kapcsolatos)

Utasításforma

```
[ módosító ] interface felületnév extends
                                    ➥ felületnév {
        állandó-meghatározások;
        elvont-függvények;
```

Leírás

Az **interface** (felület) egy vagy több állandó azonosítót vagy elvont függvényt határoz meg. Külön osztály valósítja meg a felületosztályt, és szolgáltatja az elvont függvények meghatározását. A felületeket a megvalósító osztályok által feltételezett tulajdonságok (azonosítók) és viselkedések (függvények) szervezésének elősegítésére használjuk.

Példa

```
// A ShapeInterface (AlakFelület) két, bármely
// alakzat által megvalósítható elvont függvényt
// határoz meg, illetve egy különféle alakzatfüg-
// gő számításokban használható, weightMultiplier
// (súlySzorzó) nevű változót is tartalmaz.
public interface ShapeInterface {
    public final double weightMultiplier = 69.64;

    public int getArea ();
    public int getCircumference ();
}
```

Tippek

- Az osztályok úgy valósítanak meg egy felületet, hogy az implements fenntartott szót használják a meghatározásukban, illetve megvalósítják a felület elvont függvényeit.
- Az **interface** osztály nem példányosítható.
- Az **interface** osztályokban lévő függvények nem tartalmazhatnak utasításokat. Ez azt jelenti, hogy megvalósításuknak abban az osztályban kell szerepelnie, amelyik megvalósítja a felületet.
- Több osztály is megvalósíthatja ugyanazt a felületet, és különböző meghatározásokat adhatnak a felület elvont függvényeire.
- Egy osztály egyszerre egynél több felületet is megvalósíthat.

Lásd még

- abstract
- class
- implements

long (adattípus)

Utasításforma

```
[ módosító ] long változónév [
                        ➥ = kezdőérték-kifejezés ];
```

Leírás

A `long` fenntartott szó egy vagy több `long` adattípusú változó bevezetésére szolgál. A Java nyelvben a `long` típus 64 bites előjeles értékeket jelent. A `long` adattípus legkisebb értéke –9 223 372 036 854 775 808, legnagyobb értéke pedig 9 223 372 036 854 775 807.

Példák

❶

```
// Bevezeti az alpha nevű long típusú változót,
// és annak kezdőértékét 19-re állítja.
long alpha = 19;
```

❷

```
// Bevezeti a beta és gamma nevű long típusú
// változókat, és a gamma kezdőértékét -55063L-re
// állítja.
long beta, gamma = -55063L;
```

Tippek

- Egy `long` típusú változó bevezetésekor beállítható a változó kezdőértéke is, ha a változó neve után egy egyen-

lőségjelet teszünk, és beírjuk a kezdőérték-kifejezést (az utasításformában ezt a szögletes zárójellel jeleztük).
- Több `long` típusú változó is bevezethető egyazon utasításban, ha a változók neveit vesszővel választjuk el egymástól (lásd a második példát).
- Az osztályváltozóként megadott, de kezdőértékkel el nem látott `long` típusú változók az alapértelmezett nulla értéket kapják.
- Függvényváltozóként a `long` típusú változót használat előtt mindenképpen el kell látnunk kezdőértékkel.
- A lebegőpontos literál értékekkel való összekeverés veszélyének csökkentése érdekében a `long` literál értékek mögé az `l` vagy az `L` karakter írható (lásd a második példát).
- A java.lang.Long osztály a `long` alap adattípus burkolóosztálya.

Lásd még

- java.lang.Long

native (módosító)

Utasításforma

```
[ módosító ] [ native ] adattípus
      ➥ függvénynév ( [ paraméterdeklarációk ] )
  [ throws kivételtípus [, kivételtípus...] ];
```

Leírás

A `native` (natív) fenntartott szó annak jelzésére szolgál, hogy egy függvényt a Java nyelvből hívunk meg, de annak megvalósítása egy másik programozási nyelven (például C++ vagy C) történt.

Példa

```
// Két natív, másik nyelven megvalósított
// függvényt vezet be.
public native int getAge();
public native void getAge (int newAge);
```

Tippek

- Figyeljük meg, hogy a **native** függvény tartalmi része hiányzik, és a formális paraméterlistát csak egy pontosvessző követi.
- Konstruktor nem módosítható a **native** fenntartott szóval.
- A **native** függvények nem vezethetők be az abstract vagy a strictfp kulcsszóval.

Lásd még

- abstract
- strictfp

new (osztállyal kapcsolatos)

Utasításforma

változónév = **new** *osztálynév* ([*paraméterlista*]);

Leírás

A **new** (új) fenntartott szóval egy objektum (példány) hozható létre a megadott osztályból. Az objektum példányosítása általában a hozzárendelés jobb oldalán történik.
A példányosítás végrehajtja az osztály konstruktorát, és nulla vagy több paramétert tartalmazhat.

Példák

❶

```
// Létrehoz egy új Integer objektumot.
Integer result = new Integer ("45");
```

❷

```
//Létrehoz egy új Student (hallgató) objektumot
// az alapértelmezett konstruktor meghívásával.
Student freshman = null;
freshman = new Student ();
```

Tippek

- Objektumok nem példányosíthatók anélkül, hogy változókhoz rendelnénk azokat. Ez leggyakrabban olyankor történik, amikor létrehozunk egy objektumot, és paraméterként átadjuk azt egy függvénynek.
- Az objektum példányosításának nem kell feltétlenül a változó bevezetésével egyidejűleg megtörténnie. Elvégezhető később is, és a változóhoz hozzárendelhető egy hivatkozás, ami az újonnan létrehozott objektumra mutat (lásd a második példát).
- Objektum adott osztályból (osztálynév) történő példányosításakor a programozó meghívja az osztály egyik konstruktorát (az osztály nevével megegyező nevű, és void visszatérési típusú függvények). A konstruktorhívásnak meg kell felelnie a konstruktor paraméterlistájának (sorrendben, típusban, és a paraméterek számában).

Lásd még

- void

package (osztállyal kapcsolatos)

Utasításforma

package *csomagnév*;

Leírás

A **package** (csomag) fenntartott szó használatával egy osztály vagy felület egy az osztálynévvel megadott egységbe (csomagba) helyezhető.

Példa

```
package com.peterdepasquale;

// A Faculty osztály a com.peterdepasquale
// csomagban található.
public class Faculty {
    String lastName;
    String firstName;
}
```

Tippek

- Ha használjuk, akkor a megnevezett csomagnak a forráskód első elemeként kell szerepelnie, az import utasítások és bármiféle osztály vagy felület meghatározása előtt.
- Ha egy osztályt nem helyezünk egy megadott csomagba a **package** fenntartott szó használatával, akkor a csomag az „unnamed" (név nélküli) vagy „default" (alapértelmezett) csomagba kerül.
- A csomagok elősegítik a forráskód kapcsolódó szolgáltatásainak modulokba szervezését.
- A csomagok elnevezésének általános szabványa a tartománynév összetevőinek fordított sorrendben történő használata (például edu.tcnj.depasquale).

- Alcsomagok is létrehozhatók, ha további összetevőket fűzünk a csomagnévhez (például edu.tcnj.depasquale.webSpider).
- Érdemes a csomagokat beszédes nevekkel ellátni.

Lásd még

- class
- interface

private (módosító)

Utasításforma

```
[ private ] [ módosító ] class osztálynév
  [ extends osztálynév ]
  [ implements osztálynév [, osztálynév…] ] {…}
```

vagy

```
[ private ] [ módosító ] adattípus
     ➥ függvénynév ( [ paraméterdeklarációk ] )
  [throws kivételtípus [, kivételtípus…]] {…}
```

vagy

```
[ private ] [ módosító ] adattípus változónév [
                                ➥ = kezdőérték ];
```

Leírás

A **private** (saját) fenntartott szó osztályok, függvények vagy adatmezők láthatóságának módosítására szolgál. Osztály esetében a **private** láthatóság csak belső osztályokra alkalmazható. Ha függvényt vagy adatmező-meghatározást

módosítunk a **private** fenntartott szóval, megakadályozhatjuk, hogy azok elérhetők legyenek az alosztályokból. A **private** függvények és adatmezők nem érhetők el más osztályokból.

Példa

```
public class PrivateExample {
   // A privateField (sajátMező) adatmezőt nem
   // öröklik alosztályok, és csak ezen osztály
   // példányai érhetik el.
   private double privateField = 0.0;

   // Az incrementField (mezőNövelése) függvényt
   // nem öröklik alosztályok, és csak ezen
   // osztály példányai érhetik el.
   private void incrementField () {
      privateField += 3.1415;
   }
}
```

Tipp

- A **private** módosító elősegíti egy osztály függvényeinek és adatmezőinek betokozását. Egy osztály, függvény vagy mező láthatósági módosítójának kiválasztása fontos szoftverfejlesztési döntés.

Lásd még

Nincs.

protected (módosító)

Utasításforma

```
[ protected ] [ módosító ] adattípus függvénynév
                    ➥ ( [ paraméterdeklarációk ] )
   [ throws kivételtípus [, kivételtípus…] ] {…}
```

vagy

```
[ protected ] [ módosító ] adattípus változónév
                                    ➥ [ = kezdőérték ];
```

Leírás

A **protected** (védett) fenntartott szó függvények és adatmezők láthatóságának módosítására szolgál. Több szempontból is módosítja azok elérhetőségét: először is, a **protected** függvények és adatmezők öröklődnek, és elérhetők az alosztályok által; másodszor, az osztályukkal megegyező csomagban lévő összes többi osztály is elérheti őket. A függvény vagy adatmező meghatározását tartalmazó csomagon kívüli osztályok nem férhetnek hozzá a védett elem(ek)hez.

Példa

```
public class ProtectedExample {
  // Ezt az adatmezőt öröklik az alosztályok,
  // és elérheti ez az osztály és annak
  // alosztályai is.
  protected double protectedField = 0.0;

  // Az incrementField (mezőNövelése) függvényt
  // öröklik az alosztályok, és elérheti ez az
  // osztály és annak alosztályai is.
  protected void incrementField () {
     protectedField += 3.1415;
  }
}
```

Tipp

- A **protected** láthatóság elősegíti egy osztály függvényeinek és adatmezőinek betokozását. Egy osztály,

függvény vagy mező láthatósági módosítójának kiválasztása fontos szoftverfejlesztési döntés.
- Osztályok meghatározása nem módosítható a `protected` fenntartott szóval.

Lásd még

Nincs.

public (módosító)

Utasításforma

```
[ public ] [ módosító ] class osztálynév
  [ extends osztálynév ]
  [ implements osztálynév [, osztálynév…] ] {…}
```

vagy

```
[ módosító ] [ public ] interface felületnév {…}
```

vagy

```
[ public ] [ módosító ] adattípus
     ➥ függvénynév ( [ paraméterdeklarációk ] )
  [ throws kivételtípus [, kivételtípus…]] {…}
```

vagy

```
[ public ] [ módosító ] adattípus változónév [
                              ➥ = kezdőérték ];
```

Leírás

A `public` (nyilvános) fenntartott szó osztályok, függvények vagy adatmezők láthatóságának módosítására szolgál a Java nyelvben. Ha osztály meghatározásánál használjuk, a `public` módosító elérhetővé teszi az osztályt minden más kód számára (bár a belső adatmezők és függvények más láthatósággal is rendelkezhetnek). A `public` fenntartott szóval módosított felületeket bármelyik osztály elérheti.

Példa

```
public class Circle {
   // A PI értéke elérhető ebben és minden más
   // osztályban, de nem változtatható meg.
   public static final double PI = 3.1415927d;

   // A statikus area (terület) függvény egy kör
   // területét adja vissza a sugár alapján.
   // A függvény a meghatározott PI értéket
   // használja.
   public static double area (double radius) {
      return PI * radius * radius;
   }
}
```

Tippek

- Amennyiben egy adott osztály konstruktora nem `public` (private, protected, vagy nem tartozik hozzá láthatósági módosító), akkor csak maga az osztály hozhat létre példányokat az osztályból.
- A nem nyilvános felületeket csak a felülettel megegyező csomagban lévő osztályok érhetik el.

Lásd még

Nincs.

return (vezérlőutasítás)

Utasításforma

`return` [*kifejezés*];

Leírás

A `return` (visszaad) fenntartott szó önmagában vagy a nem kötelező kifejezéssel használva visszaadja a vezérlést a hívó függvénynek. Ekkor az aktuális függvény végrehajtása megszűnik, és a végrehajtás a hívó függvényben vagy konstruktorban folytatódik.

Példák

❶

```
// A getPostalCode függvény egy irányítószámot
// (egy egész számot) ad vissza.
public int getPostalCode () {
       return postalCode;
}
```

❷

```
// A setPostalCode függvény beállítja az
// irányítószámot a megadott értékre,
// és nem ad vissza értéket.
public void setPostalCode (int value) {
       postalCode = value;
       return;
}
```

Tippek

- Amennyiben szerepel kifejezés a `return` utasításban, a kifejezés eredményezte adattípusnak meg kell egyeznie a függvény aláírásában visszaadásra megadott adattípussal, tehát a kifejezésnek meg kell egyeznie a függvény visszatérési típusával.

- Lehetőség van több **return** utasítás használatára is, de jó programozási gyakorlatnak számít, ha egy függvényben csak egy **return** utasítást alkalmazunk.
- Ha egy függvény egyáltalán nem ad vissza adatot (a visszatérési típus void), akkor nem kell szerepelnie **return** utasításnak. A második példában jelen van ugyan, de nem ad vissza semmit, tehát jelenléte nem okoz figyelmeztetést vagy hibát.
- A konstruktorok tartalmazhatnak **return** utasítást, de nem szerepelhet bennük kifejezés.

Lásd még

- void

short (adattípus)

Utasításforma

```
[ módosító ] short változónév [
                              ➥ = kezdőérték-kifejezés ];
```

Leírás

A **short** (rövid) fenntartott szó egy vagy több **short** adattípusú változó bevezetésére szolgál. A Java nyelvben a **short** típus 12 bites előjeles értékeket jelent. A **short** adattípus legkisebb értéke –32 768, legnagyobb értéke pedig 32 767.

Példák

❶

```
// Bevezeti az alpha nevű long típusú változót,
// és annak kezdőértékét 19-re állítja.
short alpha = 19;
```

❷

```
// Bevezeti a beta és gamma nevű long típusú
// változókat, és a gamma kezdőértékét
// -5-re állítja.
short beta, gamma = -5;
```

Tippek

- Egy **short** típusú változó bevezetésekor beállítható a változó kezdőértéke is, ha a változó neve után egy egyenlőségjelet teszünk, és beírjuk a kezdőérték-kifejezést (az utasításformában ezt a szögletes zárójellel jeleztük).
- Több **short** típusú változó is bevezethető egyazon utasításban, ha a változók neveit vesszővel választjuk el egymástól (lásd a második példát).
- Az osztályváltozóként megadott, de kezdőértékkel el nem látott **short** típusú változók az alapértelmezett nulla értéket kapják.
- Függvényváltozóként a **short** típusú változót használat előtt mindenképpen el kell látnunk kezdőértékkel.
- A java.lang.Short osztály a **short** alap adattípus burkolóosztálya.

Lásd még

- java.lang.Short

static (módosító)

Utasításforma

```
[ módosító ] [ static ] adattípus
      ➥ függvénynév ( [ paraméterdeklarációk ] )
  [ throws kivételtípus [, kivételtípus...] ] {...}
```

vagy

```
[ módosító ] [ static ] adattípus változónév [
                                    ➥ = kezdőérték ];
```

vagy

```
import static típusnév.azonosítónév;
```

Leírás

A **static** (statikus) módosító használatával úgy módosítható egy függvény vagy adatmező, hogy az egy osztályfüggvény (vagy osztályadatmező) legyen példányfüggvény (vagy példányadatmező) helyett. Tehát a függvénynek vagy adatmezőnek csak egy másolata lesz az összes objektum számára abban az osztályban, ahol szerepel.

Példa

```
public class Course {
   // A static crsPrefix azonosítót használjuk
   // a TCNJ előtag kiíratására.
   private static String crsPrefix = "TCNJ";

   // A getPrefix függvény a crsPrefix statikus
   // adatot adja vissza
   public static String getPrefix () {
      return crsPrefix;
   }
}
```

Tippek

- Egy **static** függvény csak **static** adatokra és más **static** függvényekre hivatkozhat, nincs tudomása az őt tartalmazó osztály példányairól. Egy osztály minden példánya hozzáférhet a példányadatokhoz és példány-

függvényekhez, valamint az osztály `static` adataihoz és `static` függvényeihez.

- A `static` változók értékének megváltozásakor az osztály minden példánya látja a változást.
- Konstruktorok nem módosíthatók a `static` fenntartott szóval.
- Az állandó (final fenntartott szóval bevezetett) adatmezőket gyakran módosítják a `static` fenntartott szóval, hogy memóriát takarítsanak meg. (Pazarlás több másolatot fenntartani egy értékről, ami nem változtatható meg.)
- A `static` függvények általában egy szolgáltatást nyújtanak más osztályoknak. Így van ez például a java.lang.Math osztályban lévő nyilvános statikus függvények esetében is, mivel ez az osztály különféle matematikai függvényeket nyújt.
- A `static` módosító használható az import fenntartott szóval együtt is, megadott `static` azonosítók betöltésére egy megadott csomagból.

Lásd még

- import

strictfp (módosító)

Utasításforma

```
[ módosító ] strictfp class osztálynév
  [ extends osztálynév ]
  [ implements osztálynév [, osztálynév…] ] {…}
```

vagy

```
[ módosító ] strictfp interface felületnév {…}
```

vagy

```
[ módosító ] [ strictfp ] adattípus
     ➥ függvénynév ( [ paraméterdeklarációk ] )
  [ throws kivételtípus [, kivételtípus…] ] {…}
```

Leírás

A **strictfp** fenntartott szó egy osztály, felület vagy függvény módosítására szolgál, és arra kényszeríti azt, hogy szigorú számítási módot használjon minden lebegőpontos (float és double típusú változókat érintő) számítás során abban a blokkban, amelyben a módosító szerepel.
A **strictfp** módosító használata nélkül a lebegőpontos kifejezések felvehetnek egy köztes formát, amely nagyobb az eredményezett adattípus megengedhető legnagyobb (vagy legkisebb) értékénél.

Ha a szigorú lebegőpontos számítási mód van érvényben, akkor a köztes számítások nem sérthetik meg az adattípus legnagyobb (vagy legkisebb) értékét. Ha megsérül a legnagyobb (vagy legkisebb) megengedett érték, akkor az eredményezett kifejezés „Infinity" (végtelen), vagy a legkisebb érték esetében „–Infinity" (mínusz végtelen) lesz.

Példa

```
public strictfp class Government {
  // A calcTaxes függvény a megadott éves
  // fizetés alapján kiszámítja a fizetendő
  // összeget. A függvény szigorú lebegőpontos
  // számítást alkalmaz.
  private strictfp float
                  ➥ calcTaxes (float salary) {
      return salary * 0.338f;
  }
}
```

Tippek

- Ha a `strictfp` módosítót alkalmazzuk egy függvényre, akkor a függvényben lévő minden kód végrehajtása a szigorú lebegőpontos számítási módnak megfelelően történik.
- Ha a `strictfp` módosítót alkalmazzuk egy osztályra vagy felületre, akkor az azokban szereplő minden kód szigorú módon értékelődik ki.
- Konstruktorok nem módosíthatók a `strictfp` fenntartott szóval.
- Egy felületen belül lévő függvények nem módosíthatók a `strictfp` fenntartott szóval.

Lásd még

Nincs.

super (osztállyal kapcsolatos)

Utasításforma

```
super.függvénynév ( );
```

vagy

```
super.változónév
```

Leírás

A `super` fenntartott szó egy osztály nem statikus függvényeiben használható a szülőosztályra vagy annak adatmezőire való hivatkozáshoz.

Példa

```
public class Beta extends Alpha {
  // A Beta konstruktor meghívja az Alpha
  // konstruktorát, amely egy egész értéket
  // fogad. A super (függvénynév nélkül)
  // függvényhívás egy az Alpha osztályban
  // található konstruktorra hivatkozik.
  // Az Alpha osztályban lévő count változót
  // is megnöveljük.
  public Beta (int value) {
    super (value);
    super.count ++;
  }

  // Ez a Beta konstruktor az Alpha osztály
  // getRandomValue függvényét hívja meg,
  // és kiírja a szülőosztály count adatmezőjét.
  public Beta () {
    System.out.println (super.getRandomValue ());
  }
}
```

Tippek

- A **super** fenntartott szó használatával a szülőosztály egy megadott konstruktorát hívhatjuk meg. Ha a szülőosztály konstruktorának hívása szerepel az alosztály konstruktorában, akkor annak az első helyen kell lennie (ahogy a példában látható).
- A szülőosztályban lévő függvényeket és adatmezőket (amelyeket felülbírál az alosztály) a **super** fenntartott szó használatával érheti el az alosztály. Ezen fenntartott szó használatával könnyebben megadható, hogy egy függvényhívás vagy mezőhivatkozás melyik osztályra hivatkozik.

Lásd még

Nincs.

switch (vezérlőutasítás)

Utasításforma

```
switch ( kifejezés ) {
  // Egy vagy több ilyen formátumú case záradék:
  [ case állandókifejezés:
      utasítás(ok);
      [ break; ]
  ]

  [ default:
      utasítás(ok);
      [ break; ]
  ]
}
```

Leírás

A **switch** (váltás) utasítással választást tehetünk lehetővé több útvonal között. A választás alapja a kifejezés eredménye és a **switch** utasítás, amely egy case címkét tartalmaz a kifejezés eredményének megfelelő útvonalhoz.

Példák

❶

```
char letter = 'i';
String characterType = "unknown";

// switch utasítás a letter változó értéke
// alapján
switch (letter) {
  // Ha a 'letter' 'a', 'e', 'i', 'o' vagy 'u',
  // akkor a characterType legyen "vowel"
  case 'a':
  case 'e':
  case 'i':
  case 'o':
  case 'u':
```

```
        characterType = "vowel";
        break;

    // egyébként a típust jelölő characterType
    // változó értéke legyen "consonant"
    default:
        characterType = "consonant";
        break;
}
```

❷

```
int value = 3;

// switch utasítás a value változó értéke
// alapján
switch (value) {
    // Ha az érték 1, akkor eggyel növeljük.
    case 1:
        value ++;
        break;

    // Ha az érték 3, akkor kettővel növeljük.
    case 3:
        value += 2;
        break;
}
```

Tippek

- A **switch** kifejezésnek char, byte, short vagy integer típusúnak kell lennie.
- A **switch** utasításblokkon belül minden utasításnak egy case vagy egy default címkéhez kell tartoznia.

Lásd még

- break
- case
- default

synchronized (módosító)

Utasításforma

```
synchronized ( objektumhivatkozás ) {
[ módosító ] synchronized adattípus
        ➥ függvénynév ( [ paraméterlista ] ) {…}
```

Leírás

A **synchronized** (összehangolt) fenntartott szó egy függvény módosítására vagy egy **synchronized** utasítás létrehozására szolgál. Az eredmény mindkét esetben egy olyan függvény vagy kódblokk, amelynek végrehajtása egyszerre csak egy szálon történik. Az összehangolás megakadályozza egy adott utasítás vagy utasításblokk egyidejű elérését.

Bizonyos fejlesztési helyzetekben (szálakkal való munka során) a programozó biztosíthatja, hogy egy adott objektumot egyszerre csak egy szál módosítson vagy hivatkozzon rá. A példában arról gondoskodunk, hogy egy adott időpontban csak egy szál hívhassa meg az addStudent (felveszHallgató) függvényt.

Az összehangolás úgy valósul meg, hogy egy szál zárol egy adott objektumhivatkozást (összehangolt utasítások esetében), vagy beágyaz egy osztályt (összehangolt függvények esetében). Amennyiben a zárolás nem hajtható végre, a szál addig vár, amíg a zárolás megszerezhetővé válik. Az összehangolt függvény vagy utasítás befejezése után a fenntartott zárolás megszűnik. Előfordulhat, hogy egy függvény vagy utasítás megakadályozza a zárolás feloldását (például egy végtelen ciklusban), ami a *holtpont*nak nevezett helyzethez vezet.

Példa

```
import java.util.Vector;

public class CourseSection {
   // Az enrollment (beiratkozás) vektor egy
   // listát tárol a tanfolyami csoport
   // hallgatóiról.
   private Vector<Student> enrollment
                        ➥ = new Vector<Student> ();

   // Felvesz egy hallgatót ebbe a tanfolyami
   // csoportba. A függvény összehangolt,
   // így elkerüljük, hogy egyszerre több szál is
   // végrehajthassa a függvényt.
   public synchronized void
               ➥ addStudent (Student enrollee) {
      if (enrollee != null)
            enrollment.add (enrollee);
   }

   // Eltávolítja az első hallgatót az enrollment
   // vektorból. A több egyidejű elérés megaka-
   // dályozása érdekében a vektorszerkesztést
   // magába foglaló kód összehangolt.
   public void removeFirst () {
      synchronized (enrollment) {
            enrollment.removeElementAt (0);
      }
   }
}
```

Tippek

- Az összehangolt utasítások a megnevezett objektumhivatkozáshoz rendelt zárat próbálják zárolni.
- Az összehangolt példányfüggvények a függvényt tartalmazó osztálypéldányhoz rendelt zárat próbálják zárolni.
- Az összehangolt statikus függvények a függvényt tartalmazó Class objektumhoz (java.lang.Class) rendelt zárat próbálják zárolni.

- Konstruktor nem módosítható a **synchronized** fenntartott szóval.

Lásd még

Nincs.

this (osztállyal kapcsolatos)

Utasításforma

this.*adatmező*

vagy

this.*függvénynév* ([*paraméterlista*]);

Leírás

A Java nyelv **this** (ez) fenntartott szava egy objektumon belül magára az objektumra való hivatkozásra használatos. Általában konstruktorokban és példányfüggvényekben használjuk a konstruktor vagy függvény nevével megegyező nevű példányadatmezőkre való hivatkozáshoz. A **this** fenntartott szó használatával tehát tisztázható, hogy melyik változóra (a formális paraméterre vagy a példányadatmezőre) kívánunk hivatkozni.

Példa

```
public class Section {
  // A sectionNumber (csoportSzám) és
  // sectionEnrollment (csoportBeiratkozás) nevű
  // példányváltozók a tanfolyami csoport egyes
  // példányainak kulcsfontosságú információit
  // követik nyomon.
  private int sectionNumber;
```

```
    private int sectionEnrollment;

    // A Section (Csoport) konstruktor két értéket
    // fogad: a csoport számát és a csoport
    // méretét.
    public Section (int sectionNumber,
                    ➥ int sectionEnrollment) {
        this.sectionNumber = sectionNumber;
        this.sectionEnrollment = sectionEnrollment;
    }
}
```

Tipp

- A **this** fenntartott szó technikai szempontból hivatkozás az aktuális objektumra. Függvény paramétereként is használható; például: enrollment.add (**this**);

Lásd még

Nincs.

throw (vezérlőutasítás)

Utasításforma

throw *kifejezés*;

Leírás

A **throw** (dobás) fenntartott szó egy kivételpéldány kiváltására („dobására") szolgál. A kiváltott, de egy catch blokk által el nem kapott kivételek a hívó függvény(ek)hez kerülnek, amíg elkapásukra sor nem kerül. Amennyiben egy kivétel elkapása nem történik meg, a kivétel végrehajtási szála megszűnik, amikor eléri a hívási láncnak (a függvényhívások sorozatának) a tetejét.

Példa

```
public class ThrowingTester {
   // A main függvény egy új RuntimeException
   // kivételobjektumot hoz létre és ad át,
   // amennyiben nincsenek parancssori paraméterek.
   public static void main (String[] args)
                        ➥ throws RuntimeException {
      if (args.length < 1)
          throw new RuntimeException
          ➥ ("command line parameters missing!");
      else
          System.out.println ("Hello " + args[0]);
   }
}
```

Tippek

- A **throw** utasításban szereplő kivételtípusnak meg kell egyeznie a **throw** utasítást tartalmazó függvény throws záradékában lévő kivételtípusok egyikével (vagy a java.lang.Throwable osztály vagy annak egy alosztályának egy példányának kell lennie).
- Nem minden kivételt vált ki a programozó forráskódja: néhány kivétel futásidőben felmerülő helyzetek eredménye. A kivételek lehetnek file-not-found (a fájl nem található), divide-by-zero (nullával való osztás) kivételek, out-of-memory (nincs elég memória) és más hasonló, nem tervezett hibák.

Lásd még

- catch

throws (osztállyal kapcsolatos)

Utasításforma

```
[ módosító ] adattípus
      ➥ függvénynév ( [ paraméterdeklarációk ] )
   [ throws kivételtípus [, kivételtípus…] ] {…}
```

Leírás

A **throws** (dob) fenntartott szóval egy függvény vagy konstruktor módosítható, annak jelzésére, hogy az abban lévő kód egy példányt dobhat a megadott kivételből (kivételtípus).

Példa

```
import java.io.*;
import java.util.*;

public class ThrowingTester {
   // A main függvény egy FileNotFoundException
   // kivételt (ellenőrzött kivétel) vált ki,
   // amennyiben nem található a Scanner objektum
   // létrehozásakor használt bemeneti fájl.
   // Amikor a program megpróbálja beolvasni
   // a fájlból az első karakterláncot,
   // NoSuchElementException kivétel jöhet létre,
   // ha a bemeneti fájl jelen van ugyan,
   // de üres. A NoSuchElementException nem
   // ellenőrzött kivétel, és kiválthatja
   // a következő függvény hívása, de nem kell
   // felsorolni a throws záradékban.
   public static void main (String[] args)
               ➥ throws FileNotFoundException {
     Scanner scan = new Scanner
               ➥ (new File ("input.txt"));
     System.out.println ("Hello, " + scan.next ());
   }
}
```

Tippek

- A Java nyelvben kétféle kivétel létezik: ellenőrzött és nem ellenőrzött. Az ellenőrzött kivételeket vagy el kell kapni (a try/catch blokkok használatával), vagy fel kell sorolni a függvény meghatározásának **throws** záradékában (és ezáltal jelezni, hogy a kivétel továbbadódik). A nem ellenőrzött kivételek RuntimeException típusú (vagy annak egy alosztályából létrehozott) objektumok, és nem igényelnek **throws** záradékot.
- Elvont függvény felülírásakor vagy megvalósításakor nem vehetünk fel kivételobjektumokat az eredeti függvény **throws** záradékába.
- Egy függvény vagy konstruktor több kivétel kiváltását is támogathatja. Ehhez egymás után vesszővel elválasztva kell megadni a kivételtípusokat a **throws** záradékban.
- Fordítási hibát okoz, ha a kivételtípus nem a java.lang.Thowable osztályból vagy annak egy alosztályából származik.

Lásd még

- catch
- throw
- try

transient (módosító)

Utasításforma

```
[ módosító ] transient adattípus változónév;
```

Leírás

A **transient** (átmeneti) fenntartott szó annak jelölésére használható, hogy a fenntartott szó által módosított példánymező nem állandó az azt befoglaló objektum adatfo-

lyamba való írásakor. Tehát a **transient** fenntartott szó megakadályozza a megadott példányadatok adatfolyamba írását.

Példa

```
import java.io.Serializable

// A SerializeStudent osztály alapinformációkat
// tartalmaz az egyes hallgatókról.
// A SerializeStudent objektumok adatfolyamba
// írásakor azonban a folyamba csak a név és
// a cím kerül.
public class SerializedStudent
                    ➥ implements Serializable {
      String name;
      String address;
      transient double gpa;

      // Itt szerepelnének az osztály-
      // és példányfüggvények.
}
```

Tippek

- Fontoljuk meg, hogy mikor használjuk a **transient** fenntartott szót. Használata csökkentheti a lemezre kiírt adatmennyiséget.
- A programozók biztonsági óvintézkedésként is használhatják a **transient** fenntartott szót, hogy megelőzzék a kényes adatok kiadását.

Lásd még

- java.io.Serializable

try (vezérlőutasítás)

Utasításforma

```
try {
        utasítás(ok);
}
  [ catch (kivételtípus kivételazonosító-név) {
        utasítás(ok);
} ]
[ finally {
        utasítás(ok);
} ]
```

Leírás

A `try` (próbálkozás) fenntartott szó a nagyobb `try`/ catch/ finally utasítás része. Ezen fenntartott szavak mindegyike egy egy vagy több utasításból álló blokkot foglal magában, melyek végrehajtása bizonyos körülményektől függ. Először a `try` blokk végrehajtása történik meg, utasításai végrehajtásának befejezéséig, vagy egy kivétel kiváltásáig.

Kivétel esetén sorban egymás után egyenként megtörténik a catch záradékok vizsgálata annak meghatározására, hogy a kiváltott kivétel megfelel-e egy megadott kivételtípusnak. Akkor van illeszkedés, ha a kiváltott kivétel típusa hozzárendelhető az elkapott típushoz. Amennyiben van illeszkedés, megtörténik a megfelelő catch záradékban található utasítás(ok) végrehajtása.

Ha nem lép fel kivétel egy `try` blokk végrehajtása során, akkor a feldolgozás az utolsó catch blokk után folytatódik. Amennyiben van finally záradék, akkor a rendszer azt is végrehajtja, egyébként pedig az utolsó catch záradékot követő utasítás következik.

Példa

```
// Ez a függvény egy új File objektumot próbál
// létrehozni a megadott fájlnévből.
// Az objektum létrehozó- és canWrite függvénye
// kivételek kiváltását eredményezheti, több
// okból kifolyólag. Ezért ennek a kódnak
// a lefordításához két catch utasítás szükséges.
public void checkFile (String filename) {
    try {
      File inputFile = new File (filename);
      if (inputFile.canWrite ())
        System.out.println ("The file: "
          ➥ + filename + " can be written to.");
    }
    catch (NullPointerException nullPtr) {
        System.err.println (nullPtr);
    }
    catch (SecurityException securityExpt) {
        System.err.println (securityExpt);
    }
}
```

Tippek

- A finally záradék nem kötelező a `try`/ catch utasításban.
- Egy `try` blokkhoz nulla vagy több catch záradék tartozhat, amennyiben ezeknek a záradékoknak mindegyike különböző típusú kivételeket kap el. Vegyük figyelembe, hogy a különböző catch záradékokhoz tartozhatnak olyan kivételtípusok is, amelyek egymás alosztályai.

Lásd még

- catch
- finally

void (adattípus)

Utasításforma

```
[ módosító ] void függvénynév ( [
                ➥ paraméterdeklarációk ] )
   [ throws kivételtípus [, kivételtípus...] ]  {...}
```

Leírás

A **void** fenntartott szó annak a jelölésére szolgál, hogy egy függvény végrehatásának befejezése után nem adunk vissza semmilyen adatértéket.

Példa

```
// A setName függvény a karakter nevét a cName
// paraméter értékére állítja, de nem ad
// vissza semmit a meghívó függvénynek.
public void setName (String cName) {
        characterName = cName;
}
```

Tippek

- Ha egy függvény bevezetésekor a **void** fenntartott szót használjuk, győződjünk meg arról, hogy a függvény nem tartalmaz return utasítást, vagy a return utasítás (ha használjuk) nem ad vissza értéket. A **void** fenntartott szó megakadályozza, hogy egy függvény értéket adjon vissza a meghívó függvénynek.
- Ha értéket szeretnénk visszaadni egy függvény végrehajtásának befejeződésekor, használjuk a return fenntartott szót.

Lásd még

- return

volatile (módosító)

Utasításforma

```
[módosító] volatile adattípus változónév [
                    ➥ = kezdőérték-kifejezés ];
```

Leírás

A `volatile` (változékony) fenntartott szó egy példányváltozó bevezetésének módosítására szolgál, és azt jelzi, hogy a változó tartalmát nagy mértékben módosíthatják más szálak. Egy `volatile` módosítóval ellátott változó (a szálban tárolt) helyi értékének kiolvasási kísérletekor az olvasó szál azáltal biztosítja az érték helyességét, hogy a változó mestermásolatára hivatkozik.

Példa

```
// A WorkshopSection (MűhelyCsoport) osztály
// beiratkozási és elnevezési információkat
// tartalmaz az egyes kínált műhelyekről.
// Figyeljük meg, hogy a currentEnrollment
// (aktuálisBeiratkozás) példánymező nagy
// mértékben ki van téve a műhelyek beiratkozási
// adatait feldolgozó szálak változtatásainak,
// tehát változékony példánymezőről van szó.
public class WorkshopSection {
   private volatile int currentEnrollment;
   private String courseName;
   private final int maxSize = 24;
}
```

Tipp

- Final változók nem vezethetők be a `volatile` módosítóval.

Lásd még

- final

while (vezérlőutasítás)

Utasításforma

```
while ( logikai-kifejezés )
        utasítás;
```

Leírás

A **while** (amíg) ciklus addig ismétli az utasítást (vagy kapcsos zárójelben található utasításokat), amíg a logikai kifejezés hamis értéket nem ad. Szerkezetéből adódóan lehetséges, hogy az utasítás végrehajtása egyáltalán nem történik meg. Amennyiben a kifejezés első kiértékelése hamis értéket ad, az utasítás végrehajtására soha nem kerül sor. A ciklust vezérlő logikai kifejezés kiértékelése megelőzi a ciklus utasításrészének (törzsének) végrehajtását.

Példák

❶

```
// A ciklus ismétlése addig, amíg az alpha
// változó értéke már nem kisebb a target
// változóénál, és az alpha növelése kettővel
// a ciklus minden ismétlésekor.
while (alpha < target)
        alpha = alpha + 2;
```

❷

```
// A ciklus ismétlése addig, amíg az alpha
// értéke kisebb a target változó értékénél, és
// a beta nagyobb, mint 5.
// A ciklus törzse megváltoztatja az alpha és
// beta változók értékét.
while (alpha < target && beta > 5) {
        alpha++;
        beta = beta -1;
}
```

Tippek

- A `while` ciklus utasításrésze lehet, hogy egyszer sem hajtódik végre. A végrehajtás a logikai kifejezés értékétől függ.
- Amennyiben a logikai kifejezés nem logikai (igaz vagy hamis) eredményt ad, fordításkor hiba lép fel.
- Több utasítást is végrehajtathatunk a `while` cikluson belül, ha a végrehajtandó utasításokat kapcsos zárójelek közé tesszük, és így utasításblokkot hozunk létre (lásd a második példát).
- Győződjünk meg róla, hogy a logikai kifejezés megváltozik a ciklus belsejében. Amennyiben a kifejezés értéke nem változik, a ciklus végrehajtása vég nélkül folytatódik. Ezt a helyzetet *végtelen ciklusnak* nevezzük.
- Ne felejtsük el, hogy egy logikai kifejezés több logikai műveletjellel összekapcsolt logikai kifejezésből is állhat.
- A `while` ciklus tartalmazhat más ciklusokat (do, for, `while`) is, ezáltal *beágyazott ciklusokat* hozhatunk létre.
- A `while` ciklusokban használhatunk break és continue utasításokat.

Lásd még

- break
- continue
- do
- false
- for
- true

Gyakran használt Java API osztályok és felületek

Boolean (osztály)

Leírás

A boolean alap adattípus burkolóosztályaként a `java.lang.Boolean` a boolean adattípussal való különféle átalakítások elvégzésére szolgál, valamint objektumburkolót biztosít a boolean típusú adatok számára.

Osztálymódosítók

```
public final class Boolean
extends Object
implements Serializable, Comparable<Boolean>
```

Konstruktorok

```
Boolean (boolean)
Boolean (String)
```

Nyilvános statikus mezők

```
Boolean FALSE
Boolean TRUE
Class <Boolean> TYPE
```

Nyilvános statikus függvények

```
boolean getBoolean (String)
boolean parseBoolean (String)
String toString (boolean)
Boolean valueOf (boolean)
Boolean valueOf (String)
```

Nyilvános példányfüggvények

```
boolean booleanValue ()
int compareTo (Boolean)
boolean equals (Object)
int hashCode ()
String toString ()
```

Byte (osztály)

Leírás

A byte alap adattípus burkolóosztályaként a `java.lang.Byte` a byte adattípussal való különféle átalakítások elvégzésére szolgál, valamint objektumburkolót biztosít a byte típusú adatok számára.

Osztálymódosítók

```
public final class Byte
extends Number
implements Comparable<Byte>
```

Konstruktorok

```
Byte (byte)
Byte (String)
```

Nyilvános statikus mezők

```
byte MAX_VALUE
byte MIN_VALUE
int SIZE
Class<Byte> TYPE
```

Nyilvános statikus függvények

```
Byte decode (String)
byte parseByte (String)
byte parseByte (String, int)
String toString (byte)
Byte valueOf (byte)
Byte valueOf (String)
Byte valueOf (String, int)
```

Nyilvános példányfüggvények

```
byte byteValue ()
int compareTo (Byte)
double doubleValue ()
boolean equals (Object)
float floatValue ()
int hashCode ()
int intValue ()
long longValue ()
short shortValue ()
String toString ()
```

Character (osztály)

Leírás

A char alap adattípus burkolóosztályaként a `java.lang.Character` a char adattípussal való különféle átalakítások elvégzésére szolgál, valamint objektumburkolót biztosít a char típusú adatok számára.

Osztálymódosítók

```
public final class Character
extends Object
implements Serializable, Comparable<Character>
```

Konstruktor

```
Character (char)
```

Nyilvános statikus mezők

```
byte COMBINING_SPACING_MARK
byte CONNECTOR_PUNCTATION
byte CONTROL
byte CURRENCY_SYMBOL
byte DASH_PUNCTATION
byte DECIMAL_DIGIT_NUMBER
byte DIRECTIONALITY_ARABIC_NUMBER
byte DIRECTIONALITY_BOUNDARY_NEUTRAL
byte DIRECTIONALITY_COMMON_NUMBER_SEPARATOR
byte DIRECTIONALITY_EUROPEAN_NUMBER
byte DIRECTIONALITY_EUROPEAN_NUMBER_SEPARATOR
byte DIRECTIONALITY_EUROPEAN_NUMBER_TERMINATOR
byte DIRECTIONALITY_LEFT_TO_RIGHT
byte DIRECTIONALITY_LEFT_TO_RIGHT_EMBEDDING
byte DIRECTIONALITY_LEFT_TO_RIGHT_OVERRIDE
byte DIRECTIONALITY_NONSPACING_MARK
byte DIRECTIONALITY_OTHER_NEUTRALS
byte DIRECTIONALITY_PARAGRAPH_SEPARATOR
byte DIRECTIONALITY_POP_DIRECTIONAL_FORMAT
byte DIRECTIONALITY_RIGHT_TO_LEFT
byte DIRECTIONALITY_RIGHT_TO_LEFT_ARABIC
byte DIRECTIONALITY_RIGHT_TO_LEFT_EMBEDDING
byte DIRECTIONALITY_RIGHT_TO_LEFT_OVERRIDE
byte DIRECTIONALITY_SEGMENT_SEPARATOR
byte DIRECTIONALITY_UNDEFINED
byte DIRECTIONALITY_WHITESPACE
byte ENCLOSING_MARK
byte END_PUNCTATION
byte FINAL_QUOTE_PUNCTATION
byte FORMAT
byte INITIAL_QUOTE_PUNCTATION
byte LETTER_NUMBER
byte LINE_SEPARATOR
byte LOWERCASE_LETTER
byte MATH_SYMBOL
int MAX_CODE_POINT
char MAX_HIGH_SURROGATE
char MAX_LOW_SURROGATE
int MAX_RADIX
char MAX_SURROGATE
```

```
char MAX_VALUE
int MIN_CODE_POINT
char MIN_HIGH_SURROGATE
char MIN_LOW_SURROGATE
int MIN_RADIX
int MIN_SUPPLEMENTARY_CODE_POINT
char MIN_SURROGATE
char MIN_VALUE
byte MODIFIER_LETTER
byte MODIFIER_SYMBOL
byte NON_SPACING_MARK
byte OTHER_LETTER
byte OTHER_NUMBER
byte OTHER_PUNCTATION
byte OTHER_SYMBOL
byte PARAGRAPH_SEPARATOR
byte PRIVATE_USE
int SIZE
byte SPACE_SEPARATOR
byte START_PUNCTATION
byte SURROGATE
byte TITLECASE_LETTER
Class <Character> TYPE
byte UNASSIGNED
byte UPPERCASE_LETTER
```

Nyilvános statikus függvények

```
int charCount (int)
int codePointAt (char [], int)
int codePointAt (char [], int, int)
int codePointAt (CharSequence, int)
int codePointBefore (char [], int)
int codePointBefore (char [], int, int)
int codePointBefore (CharSequence, int)
int codePointCount (char [], int, int)
int codePointCount (CharSequence, int, int)
int digit (char, int)
int digit (int, int)
char forDigit (int, int)
byte getDirectionality (char)
byte getDirectionality (int)
```

```
int getNumericValue (char)
int getNumericValue (int)
int getType (char)
int getType (int)
boolean isDefined (char)
boolean isDefined (int)
boolean isDigit (char)
boolean isDigit (int)
boolean isHighSurrogate (char)
boolean isIdentifierIgnorable (char)
boolean isIdentifierIgnorable (int)
boolean isISOControl (char)
boolean isISOControl (int)
boolean isJavaIdentifierPart (char)
boolean isJavaIdentifierPart (int)
boolean isJavaIdentifierStart (char)
boolean isJavaIdentifierStart (int)
boolean isJavaLetter (char)
boolean isJavaLetterOrDigit (char)
boolean isLetter (char)
boolean isLetter (int)
boolean isLetterOrDigit (char)
boolean isLetterOrDigit (int)
boolean isLowerCase (char)
boolean isLowerCase (int)
boolean isLowSurrogate (char)
boolean isMirrored (char)
boolean isMirrored (int)
boolean isSpace (char)
boolean isSpaceChar (char)
boolean isSpaceChar (int)
boolean isSupplementaryCodePoint (int)
boolean isSurrogatePair (char, char)
boolean isTitleCase (char)
boolean isTitleCase (int)
boolean isUnicodeIdentifierPart (char)
boolean isUnicodeIdentifierPart (int)
boolean isUnicodeIdentifierStart (char)
boolean isUnicodeIdentifierStart (int)
boolean isUpperCase (char)
boolean isUpperCase (int)
```

```
boolean isValidCodePoint (int)
boolean isWhitespace (char)
boolean isWhitespace (int)
int offsetByCodePoints (char [], int, int, int, int)
int offsetByCodePoints (CharSequence, int, int)
char reverseBytes (char)
char [] toChars (int)
int toChars (int, char [], int)
int toCodePoint (char, char)
char toLowerCase (char)
int toLowerCase (int)
String toString (char)
char toTitleCase (char)
int toTitleCase (int)
char toUpperCase (char)
int toUpperCase (int)
Character valueOf (char)
```

Nyilvános példányfüggvények

```
char charValue ()
int compareTo (Character)
boolean equals (Object)
int hashCode ()
String toString ()
```

Cloneable (felület)

Leírás

Egy osztály annak jelzésére valósítja meg a `java.lang.Cloneable` felületet, hogy a klónozási függvény használatát megengedi egy a megvalósító osztályból példányosított objektum,vagyis a `Cloneable` felületet megvalósító osztály klónozható az Object.clone függvénnyel. A megvalósító osztály eldöntheti, hogy bizonyos szolgáltatások biztosításának érdekében felülírja-e a clone függvényt.

Osztálymódosító

```
public interface Cloneable
```

Függvények

```
Nincsenek.
```

Comparable (felület)

Leírás

Egy osztály meghatározására alkalmazva a `java.lang.Comparable` felület azt jelzi, hogy az osztályban létezik a példányok természetes sorrendje. A programozó megvalósítja a compareTo függvényt, amely két objektumnak (a megadott osztály példányainak) az összehasonlítására szolgál. A compareTo függvény mindig negatív egész számot, nullát vagy pozitív egész számot ad vissza, amennyiben az objektum kisebb vagy nagyobb, mint a paraméterobjektum (T), vagy egyenlő azzal. Ez a lehetőség két objektum összehasonlítására (és ezáltal sorrend létrehozására) képezi az objektumok rendezhetőségének alapját.

Osztálymódosító

```
public interface Comparable<T>
```

Nyilvános statikus mezők

```
Nincsenek.
```

Nyilvános statikus függvények

```
Nincsenek.
```

Nyilvános példányfüggvény

```
int compareTo (T)
```

DecimalFormat (osztály)

Leírás

A `java.text.DecimalFormat` a java.text.NumberFormat alosztálya, és decimális számok feldolgozására és formázására használható, különféle formátumokban és különféle nyelvi környezetekhez. Ez az osztály általában lebegőpontos számok, egész számok, valamint tudományos jelölésű, pénz- és százalékértékek formázására szolgál.

Osztálymódosítók

```
public class DecimalFormat
extends NumberFormat
```

Konstruktorok

```
DecimalFormat ()
DecimalFormat (String)
DecimalFormat (String, DecimalFormatSymbols)
```

Nyilvános statikus mezők

Nincsenek.

Nyilvános statikus függvények

Nincsenek.

Nyilvános példányfüggvények

```
void applyLocalizedPattern (String)
void applyPattern (String)
Object clone ()
boolean equals (Object)
StringBuffer format (double, StringBuffer,
                     ➥ FieldPosition)
StringBuffer format (loong, StringBuffer,
                     ➥ FieldPosition)
StringBuffer format (Object, StringBuffer,
                     ➥ FieldPosition)
AttributedCharacterIterator
          ➥ formatToCharacterIterator (Object)
```

```
Currency getCurrency ()
DecimalFormatSybols getDecimalFormatSymbols ()
int getGroupingSize ()
int getMaximumFractionDigits ()
int getMaximumIntegerDigits ()
int getMinimumFractionDigits ()
int getMinimumIntegerDigits ()
int getMultiplier ()
String getNegativePrefix ()
String getNegativeSuffix ()
String getPositivePrefix ()
String getPositiveSuffix ()
int hashCode ()
boolean isDecimalSeparatorAlwaysShown ()
boolean isParseBigDecimal ()
Number parse (String, ParsePosition)
void setCurrency (Currency)
void setDecimalFormatSymbols
          ➥ (DecimalFormatSymbols)
void setDecimalSeparatorAlwaysShown (boolean)
void setGroupingSize (int)
void setMaximumFractionDigits (int)
void setMaximumIntegerDigits (int)
void setMinimumFractionDigits (int)
void setMinimumIntegerDigits (int)
void setMultiplier (int)
void setNegativePrefix (String)
void setNegativeSuffix (String)
void setParseBigDecimal (boolean)
void setPositivePrefix (String)
void setPositiveSuffix (String)
String toLocalizedPattern ()
String toPattern ()
```

Double (osztály)

Leírás

A double alap adattípus burkolóosztályaként a **`java.lang.Double`** a double adattípussal

való különféle átalakítások elvégzésére szolgál, valamint objektumburkolót biztosít a double típusú adatok számára.

Osztálymódosítók

```
public final class Double
extends Number
implements Comparable<Double>
```

Konstruktorok

```
Double (double)
Double (String)
```

Nyilvános statikus mezők

```
double MAX_VALUE
double MIN_VALUE
double NaN
double NEGATIVE_INFINITY
double POSITIVE_INFINITY
int SIZE
Class<Double> TYPE
```

Nyilvános statikus függvények

```
int compare (double, double)
long doubleToLongBits (double)
long doubleToRawLongBits (double)
boolean isInfinite (double)
boolean isNaN (double)
double longBitsToDouble (long)
double parseDouble (String)
String toHexString (double)
String toString (double)
Double valueOf (double)
Double valueOf (String)
```

Nyilvános példányfüggvények

```
byte byteValue ()
int compareTo (Double)
double doubleValue ()
boolean equals (Object)
```

```
float floatValue ()
int hashCode ()
int intValue ()
boolean isInfinite ()
boolean isNaN ()
long longValue ()
short shortValue ()
String toString ()
```

Float (osztály)

Leírás

A float alap adattípus burkolóosztályaként a **`java.lang.Float`** a float adattípussal való különféle átalakítások elvégzésére szolgál, valamint objektumburkolót biztosít a float típusú adatok számára.

Osztálymódosítók

```
public final class Float
extends Number
implements Comparable<Float>
```

Konstruktorok

```
Float (double)
Float (float)
Float (String)
```

Nyilvános statikus mezők

```
float MAX_VALUE
float MIN_VALUE
float NaN
float NEGATIVE_INFINITY
float POSITIVE_INFINITY
int SIZE
Class<Float> TYPE
```

Nyilvános statikus függvények

```
int compare (float, float)
int float ToIntBits (float)
int float ToRawIntBits (float)
float intBitsToFloat (int)
boolean isInfinite (float)
boolean isNaN (float)
float parseFloat (String)
String toHexString (float)
String toString (float)
Float valueOf (float)
Float valueOf (String)
```

Nyilvános példányfüggvények

```
byte byteValue ()
int compareTo (Float)
double doubleValue ()
boolean equals (Object)
float floatValue ()
int hashCode ()
int intValue ()
boolean isInfinite ()
boolean isNaN ()
long longValue ()
short shortValue ()
String toString ()
```

Integer (osztály)

Leírás

Az int alap adattípus burkolóosztályaként a **`java.lang.Integer`** az int adattípussal való különféle átalakítások elvégzésére szolgál, valamint objektumburkolót biztosít az int típusú adatok számára.

Osztálymódosítók

```
public final class Integer
extends Number
implements Comparable<Integer>
```

Konstruktorok

```
Integer (int)
Integer (String)
```

Nyilvános statikus mezők

```
int MAX_VALUE
int MIN_VALUE
int SIZE
Class<Integer> TYPE
```

Nyilvános statikus függvények

```
int bitCount (int)
Integer decode (String)
Integer getInteger (String)
Integer getInteger (String, int)
Integer getInteger (String, Integer)
int highestOneBit (int)
int lowestOneBit (int)
int numberOfLeadingZeros (int)
int numberOfTrailingZeros (int)
int parseInt (String)
int parseInt (String, int)
int reverse (int)
int reverseBytes (int)
int rotateLeft (int, int)
int rotateRight (int, int)
int signum (int)
String toBinaryString (int)
String toHexString (int)
String toOctalString (int)
String toString (int)
String toString (int, int)
Integer valueOf (int)
Integer valueOf (String)
Integer valueOf (String, int)
```

Nyilvános példányfüggvények

```
byte byteValue ()
int compareTo (Integer)
double doubleValue ()
boolean equals (Object)
float floatValue ()
int hashCode ()
int intValue ()
long longValue ()
short shortValue ()
String toString ()
```

Iterator (felület)

Leírás

A `java.util.Iterator` osztály egy felület, amelyet egy objektumgyűjteményt meghatározó osztály valósít meg. Az `Iterator` a gyűjtemény bejárására ad módot. A bejárók (léptetők) nem távolítják el az objektumot a gyűjteményből.

Osztálymódosító

```
public interface Iterator<T>
```

Nyilvános statikus mezők

```
Nincsenek.
```

Nyilvános statikus függvények

```
Nincsenek.
```

Nyilvános példányfüggvények

```
boolean hasNext ()
E next ()
void remove ()
```

Long (osztály)

Leírás

A long alap adattípus burkolóosztályaként a `java.lang.Long` a long adattípussal való különféle átalakítások elvégzésére szolgál, valamint objektumburkolót biztosít a long típusú adatok számára.

Osztálymódosítók

```
public final class Long
extends Number
implements Comparable<Long>
```

Konstruktorok

```
Long (long)
Long (String)
```

Nyilvános statikus mezők

```
long MAX_VALUE
long MIN_VALUE
long SIZE
Class<Long> TYPE
```

Nyilvános statikus függvények

```
int bitCount (long)
Long decode (String)
Long getLong (String)
Long getLong (String, long)
Long getLong (String, Long)
long highestOneBit (long)
long lowestOneBit (long)
int numberOfLeadingZeros (long)
int numberOfTrailingZeros (long)
long parseLong (String)
long parseLong (String, int)
long reverse (long)
long reverseBytes (long)
long rotateLeft (long, int)
```

```
long rotateRight (long, int)
int signum (long)
String toBinaryString (long)
String toHexString (long)
String toOctalString (long)
String toString (long)
String toString (long, int)
Long valueOf (long)
Long valueOf (String)
Long valueOf (String, int)
```

Nyilvános példányfüggvények

```
byte byteValue ()
int compareTo (Long)
double doubleValue ()
boolean equals (Object)
float floatValue ()
int hashCode ()
int intValue ()
long longValue ()
short shortValue ()
String toString ()
```

Math (osztály)

Leírás

A `java.lang.Math` nagy számú matematikai műveletet és függvényt tartalmaz. A `Math` osztály kizárólag azért létezik, hogy statikus mezőket és függvényeket biztosítson más (mind API, mind felhasználói) osztályoknak, általános célú felhasználásra.

Osztálymódosítók

```
public final class Math
extends Object
```

Konstruktorok

Nincsenek.

Nyilvános statikus mezők

```
double E
double PI
```

Nyilvános statikus függvények

```
double abs (double)
float abs (float)
int abs (int)
long abs (long)
double acos (double)
double asin (double)
double atan (double)
double atan2 (double, double)
double cbrt (double)
double ceil (double)
double cos (double)
double cosh (double)
double exp (double)
double expm1 (double)
double floor (double)
double hypot (double, double)
double IEEEremainder (double, double)
double log (double)
double log10 (double)
double log1p (double)
double max (double, double)
float max (float, float)
int max (int, int)
long max (long, long)
double min (double, double)
float min (float, float)
int min (int, int)
long min (long, long)
double pow (double, double)
double random ()
double rint (double)
long round (double)
int round (float)
double signum (double)
float signum (float)
double sin (double)
```

```
double sinh (double)
double sqrt (double)
double tan (double)
double tanh (double)
double toDegrees (double)
double toRadians (double)
double ulp (double)
float ulp (float)
```

Nyilvános példányfüggvények

```
Nincsenek.
```

NumberFormat (osztály)

Leírás

A `java.text.NumberFormat` osztály a rendszer egy adott nyelvi beállítása alapján végzi különféle számok feldolgozását és formázását.

Osztálymódosítók

```
public abstract class NumberFormat
extends Format
```

Konstruktor

```
NumberFormat ()
```

Nyilvános statikus mezők

```
int FRACTION_FIELD
int INTEGER_FIELD
```

Nyilvános statikus függvények

```
Locale [] getAvailableLocales ()
NumberFomat getCurrencyInstance ()
NumberFomat getCurrencyInstance (Locale)
NumberFomat getInstance ()
NumberFomat getInstance (Local)
NumberFomat getIntegerInstance ()
NumberFomat getIntegerInstance (Locale)
```

```
NumberFomat getNumberInstance ()
NumberFomat getNumberInstance (Locale)
NumberFomat getPercentInstance ()
NumberFomat getPercentInstance (Locale)
```

Nyilvános példányfüggvények

```
Object clone ()
boolean equals (Object)
String format (double)
String format (long)
StringBuffer format (Object, StringBuffer,
                                ➥ FieldPosition)
Currency getCurrency ()
int getMaximumFractionDigits ()
int getMaximumIntegerDigits ()
int getMinimumFractionDigits ()
int getMinimumIntegerDigits ()
int hashCode ()
boolean isGroupingUsed ()
boolean isParseIntegerOnly ()
Number parse (String)
Object parseObject (String, ParsePosition)
void setCurrency (Currency)
void setGroupingUsed (boolean)
void setMaximumFractionDigits (int)
void setMaximumIntegerDigits (int)
void setMinimumFractionDigits (int)
void setMinimumIntegerDigits (int)
void setParseIntegerOnly (boolean)
```

Elvont függvények

```
StringBuffer format (double, StringBuffer,
                                ➥ FieldPosition)
StringBuffer format (long, StringBuffer,
                                ➥ FieldPosition)
Number parse (String, ParsePosition)
```

Object (osztály)

Leírás

A `java.lang.Object` az összes osztály gyökérszülőosztálya a Java nyelvben. Olyan függvényeket tartalmaz, amelyek minden objektumra alkalmazhatók a Java nyelvben.

Osztálymódosító

```
public class Object
```

Konstruktor

```
Object ()
```

Nyilvános statikus mezők

```
Nincsenek.
```

Nyilvános statikus függvények

```
Nincsenek.
```

Nyilvános példányfüggvények

```
boolean equals(Object)
Class<? extends Object> getClass()
int hashCode()
void notify()
void notifyAll()
String toString()
void wait()
void wait (long)
void wait (long, int)
```

Védett példányfüggvények

```
Object clone ()
void finalize ()
```

Random (osztály)

Leírás

A `java.util.Random` osztály különféle formátumú ál-véletlenszerű számok létrehozására használható. A létrehozott értékek sorrendjét egy a konstruktor (vagy a setSeed függvény) által beállított magérték határozza meg. Az azonos magértéken alapuló `Random` objektumok azonos értéksorozatokat hoznak létre (ebből adódik az ál-véletlenszerűség).

Osztálymódosítók

```
public class Random
extends Object
implements Serializable
```

Konstruktorok

```
Random ()
Random (long)
```

Nyilvános statikus mezők

```
Nincsenek.
```

Nyilvános statikus függvények

```
Nincsenek.
```

Nyilvános példányfüggvények

```
boolean nextBoolean ()
void nextBytes (byte[])
double nextDouble ()
float nextFloat ()
double nextGaussian ()
int nextInt ()
int nextInt (int)
long nextLong ()
void setSeed (long)
```

Védett példányfüggvény

```
int next (int)
```

Scanner (osztály)

Leírás

A `java.util.Scanner` osztály karakterláncok elemekre bontását teszi lehetővé különféle bemeneti adatfolyamokhoz (fájlok, System.in stb.)

Osztálymódosítók

```
public final class Scanner
extends Object
implements Iterator<String>
```

Konstruktorok

```
Scanner (File)
Scanner (File, String)
Scanner (InputStream)
Scanner (InputStream, String)
Scanner (Readable)
Scanner (ReadableByteChannel)
Scanner (ReadableByteChannel, String)
Scanner (String)
```

Nyilvános statikus mezők

Nincsenek.

Nyilvános statikus függvények

Nincsenek.

Nyilvános példányfüggvények

```
void close ()
Pattern delimiter ()
String findInLine (Pattern)
String findInLine (String)
String findWithinHorizon (Pattern, int)
String findWithinHorizon (String, int)
boolean hasNext ()
boolean hasNext (Pattern)
boolean hasNext (String)
boolean hasNextBigDecimal ()
boolean hasNextBigInteger ()
boolean hasNextBigInteger (int)
```

```
boolean hasNextBoolean ()
boolean hasNextByte ()
boolean hasNextByte (int)
boolean hasNextDouble ()
boolean hasNextFloat ()
boolean hasNextInt ()
boolean hasNextInt (int)
boolean hasNextLine ()
boolean hasNextLong ()
boolean hasNextLong (int)
boolean hasNextShort ()
boolean hasNextShort (int)
IOException ioException ()
Locale locale ()
MatchResult match ()
String next ()
String next (Pattern)
String next (String)
BigDecimal nextBigDecimal ()
BigInteger nextBigInteger ()
BigInteger nextBigInteger (int)
boolean nextBoolean ()
byte nextByte ()
byte nextByte (int)
double nextDouble ()
float nextFloat ()
int nextInt ()
int nextInt (int)
String nextLine ()
long nextLong ()
long nextLong (int)
short nextShort ()
short nextShort (int)
int radix ()
void remove ()
Scanner skip (Pattern)
Scanner skip (String)
String toString ()
Scanner useDelimiter (Pattern)
Scanner useDelimiter (String)
Scanner useLocale (Locale)
Scanner useRadix (int)
```

Serializable (felület)

Leírás

Egy osztály annak jelzésére valósíthatja meg a `java.io.Serializable` felületet, hogy az osztály tartalma maradandóvá tehető a Java Serialization használatával. Amennyiben egy osztály megvalósítja a Serializable felületet, az osztály egy példánya bájtsorozatként kimeneti adatfolyamba írható (majd onnan visszaolvasható, és így visszaállítható az eredeti objektum).

Osztálymódosító

```
public interface Serializable
```

Függvények

Nincsenek.

Short (osztály)

Leírás

A short alap adattípus burkolóosztályaként a `java.lang.Short` a short adattípussal való különféle átalakítások elvégzésére szolgál, valamint objektumburkolót biztosít a short típusú adatok számára.

Osztálymódosítók

```
public final class Short
extends Number
implements Comparable<Short>
```

Konstruktorok

```
Short (short)
Short (String)
```

Nyilvános statikus mezők

```
long MAX_VALUE
long MIN_VALUE
long SIZE
Class<Short> TYPE
```

Nyilvános statikus függvények

```
Short decode (String)
short parseShort (String)
short parseShort (String, int)
short reverseBytes (short)
String toString (short)
Short valueOf (short)
Short valueOf (String)
Short valueOf (String, int)
```

Nyilvános példányfüggvények

```
byte byteValue ()
int compareTo (Short)
double doubleValue ()
boolean equals (Object)
float floatValue ()
int hashCode ()
int intValue ()
long longValue ()
short shortValue ()
String toString ()
```

String (osztály)

Leírás

A `java.lang.String` osztály karakterláncok (több karakterből álló sorozatok) ábrázolására és módosítására szolgál.

Osztálymódosítók

```
public final class String
extends Object
implements Serializable, Comparable<String>,
                                    ➥ CharSequence
```

Konstruktorok

```
String ()
String (byte[])
String (byte[], int)
String (byte[], int, int)
String (byte[], int, int, int)
String (byte[], int, int, String)
String (byte[], String)
String (char[])
String (char[], int, int)
String (int[], int, int)
String (String)
String (StringBuffer)
String (StringBuilder)
```

Nyilvános statikus mező

```
Comparator<String> CASE_INSENSITIVE_ORDER
```

Nyilvános statikus függvények

```
String copyValueOf (char[])
String copyValueOf (char[], int, int)
String format (Locale, String, Object…)
String format (String, Object…)
String valueOf (boolean)
String valueOf (char)
String valueOf (char[])
String valueOf (char[], int, int)
String valueOf (double)
String valueOf (float)
String valueOf (int)
String valueOf (long)
String valueOf (Object)
```

Nyilvános példányfüggvények

```
char charAt (int)
int codePointAt (int)
int codePointBefore (int)
int codePointCount (int, int)
int compareTo (String)
int compareToIgnoreCase (String)
String concat (String)
boolean contains (CharSequence)
```

```
boolean contentEquals (StringBuffer)
boolean endsWith (String)
boolean equals (Object)
boolean equalsIgnoreCase (String)
byte [] getBytes ()
void getBytes (int, int, byte [], int)
byte [] getBytes (String)
void getChars (int, int, char [], int)
int hashCode ()
int indexOf (int)
int indexOf (int, int)
int IndexOf (String)
int IndexOf (String, int)
String intern ()
int lastIndexOf (int)
int lastIndexOf (int, int)
int lastIndexOf (String)
int lastIndexOf (String, int)
int length ()
boolean matches (String)
boolean regionMatches (boolean, int, String,
                                        ➥ int, int)
booean regionMatches (int, String, int, int)
String replace (char, char)
String replace (String, String)
String replace (CharSequence, CharSequence)
String replaceAll (String, String)
String replaceFirst (String, String)
String [] split (String)
String [] split (String, int)
boolean startsWith (String)
boolean startsWith (String, int)
CharSequence subSequence (int, int)
String substring (int)
String substring (int, int)
char[] to CharArray ()
String toLowerCase ()
String toLowerCase (Locale)
String toString ()
String toUpperCase ()
String toUpperCase (Locale)
String trim ()
```

System (osztály)

Leírás

A `java.lang.System` osztály többféle hasznos függvénnyel segíti információk beszerzését a rendszerről. Ez az osztály nem példányosítható.

Osztálymódosítók

```
public final class System
extends Object
```

Konstruktorok

Nincsenek.

Nyilvános statikus mezők

```
PrintStream err
InputStream in
PrintStream out
```

Nyilvános statikus függvények

```
void arraycopy (Object, int, Object, int, int)
String clearProperty (String)
long currentTimeMillis ()
void exit (int)
void gc ()
Map<String, String> getenv ()
String getenv (String)
Properties getProperties ()
String getProperty (String)
String getProperty (String, String)
SecurityManager getSecurityManager ()
int identityHashCode (Object)
Channel inheritedChannel ()
void load (String)
void loadLibrary (String)
void mapLibraryName (String)
long nanoTime ()
void runFinalization ()
void runFinalizersOnExit (boolean)
```

```
void setErr (PrintStream)
void setIn (InputStream)
void SetOut (PrintStream)
void setProperties (Properties)
String setProperty (String, String)
void setSecurityManager (SecurityManger)
```

Nyilvános példányfüggvények

Nincsenek.

Tárgymutató

Látogasson el hozzánk!

Virtuális könyvesboltunk egyedülálló választékot kínál magyar és angol nyelvű számítástechnikai könyvekből

könyvespolc: ha megtetszett egy könyv, de csak később szeretné megrendelni, felteheti virtuális könyvespolcára, ahol mi megőrizzük.

rendelési napló: nyomon követheti rendeléseit és azok aktuális állapotát (pl. folyamatban, utánrendelés alatt)

egyéni beállítások: saját címjegyzéket tárolhat, segítségével vásárláskor egy gombnyomással kitöltheti a kívánt postázási adatokat.

akciók: leértékelt könyvek 10–90% kedvezménnyel!

témakörök: számtalan kategóriában böngészhet, és egy adott témakörre szűkítve is használhatja a keresőt.

www.kiskapu.hu