6.1 Adattípusok

Egyszerű adattípusok

Adattípus	tartalom
Egészek	
`byte`	8-bit integer
`short`	16-bit integer
`int`	32-bit integer
`long`	64-bit integer
Lebegőpontos	
`float`	32-bit floating point
`double`	64-bit floating point
Más típusok	
`char`	Egy karakter
`boolean`	`Igaz vagy Hamis`
`void`	semmi

6.2 Változó deklaráció

SYNTAX

adattípus változónév;

OR

adattípus változónév=érték;

OR

datatype name_1, name_2, ... ,name_n;

6.3 Stringek

Karakterláncok

A String egy Java osztály, amely állandó karaktersorozat tárolására való. Értékadáskor pl. dupla idézőjelek közé zárt konstans értékét kaphatja.

```
String greeting = "Hello World!";
```

Mint osztály, rendelkezik metódusokkal. Néhány ezek közül:

`length()`

a string hosszával tér vissza (adja eredményül).

`substring(int beginIndex,int endIndex)`

visszatérési értéke a beginIndex pozíciótól az endIndex pozícióig tartó részstring. *Remeber that position 0 is the first character in the* `String`

`valueOf(`*`anything`*`)`

szöveggé konvertálja az ***anything*** változó értékét. Például a 64 mint egész szám értékét átalakítja a "64" karaktersorozattá.

Számos egyéb metódusa van. Stringek összefűzése a + operátorral történik, ezt konkatenációnak nevezzük.

STRINGS EXAMPLES

```
1.        String greeting = "Hello World!";
2.        System.out.println(greeting);
```

3. outputs:

   Hello World!

```
4.        String greeting = "Hello World!";
5.        System.out.println(greeting.substring(0,4));
```

6. outputs:

   Hello

```
7.        String greeting = "Hello World!";
8.        System.out.println(String.valueOf(greeting.length()));
```

9. outputs:

   12

```
10.       String greeting = "Hello World!";
11.       int greeting_length = greeting.length();
12.       System.out.println("The length of "+greeting+" is
   "+greeting_length);
```

13. outputs:

    The length of Hello World! is 12

6.4 Tömbök

SYNTAX

Deklaráció

adattípus változónév[];

OR

adattípus változónév [*méret*];

OR

adattípus változónév []= {*el_1*,*el_2* , ... ,*el_n*}

Helyfoglalás a tömb elemeinek
változónév `= new adattípus`[*méret*];

Hivatkozás egy elemre
változónév[*elemsorszám*]

Megjegyzés

A tömb azonos típusú sorszámozott (indexelt) elemek sorozata. Az index a Java-ban 0-val kezdődik.

A Deklarációnál két lehetőséget láthatunk.
Az első esetben a tömbnek sem a méretét, sem az elemeit nem adjuk meg. Ekkor new konstruktorral hozunk létre tényleges tárfoglalást. Ha újra használjuk a new kostruktort ugyanezen tömbre, az nem tartalmazza majd az előzőleg abban elhelyezett értékeket.
A második eset, amikor a tömböt deklaráláskor feltöltjük az elemeinek értékével, ezzel egyúttal elvégezzük a tárfoglalást is.

6.5 Osztályok

SYNTAX

`[public] class [ extends` *SuperClass*`] {`
.
.
.
változó és metódus deklarációk
.
.
.

```
}
```

DESCRIPTION

The class is the fundamental unit of code in Java. It is a collection of variables and methods that can be thought of together as a single unit. One thinks of classes as "objects", tangible things that can be created and destroyed.

Every Java program contains at least one class. All but the most simple programs will have more than one class in existence at a time. Indeed, it is common to have many copies of the same class in existence at the same time. Of course a given thread of program flow can be only in one class's method at at time, but it can pass from class to class as a class method calls member methods from another class.

The body of the class contains the method and variable declarations that comprise the class. Method declarations are covered in a separate section. Variable declarations work like those in methods , except that you may prepend them with an access modifier like those used in methods.

The `public` declaration of a class makes the class available to other classes not in the same source code file. There can only be one `public class` per file, and it must bear the same name as the file (without the **.java** extension). You can leave out the `public` declaration if there is only one class in your program, or if your class will only be used by other classes in the same **.java** file.

If a class `extends` another class, it *inherits* the member methods and variables of this other class, called the *super class*. This means, basically, that if you have a copy of a class, you also have a copy of the variables and methods of the super class. This is an amazing invention that saves a lot of code writing.

CREATING A CLASS INSTANCE

Let us consider the following class: the `Double`. The `Double` contains methods for dealing with primitive `double` variables. For example, you might want to convert a `String` to a `double` or vice versa. The `Double` class can be thought of as a bloated `double` - it contains a `double` value as well as a bunch of methods to operate on this value.

To create a `Double`, you can use the `new` command. Consider the following:

```
     Double val = new Double(3.4);
```

This line declares the existence of the variable `val` of type `Double`. It also assigns to `val` a copy of the `Double` class initialized to the value 3.4.

Notice that the new `Double` declaration takes an argument (in this case it is the value of the `Double`). A class can contain a method with the same name as the class, called the *constructor* method. When a copy of the class is created with a `new` statement, the constructor method will be called along with the correct arguments provided. So the `Double` class has a method declared in it that looks something this:

```
public class Double {

     public Double(double value) {
```

```
        ... code here ...
    }

    ... code here ...

}
```

Accessing Class Variables and Methods

Accessing variables and methods withing a class (for example a class method calling a another method of the same class) is easy; you just use the class name. You can see this, for example, in the `main` method of the Newton class, which calls other class methods `f` and `fprime`.

Accessing variables in other classes is not much more difficult. To begin, you should know that there are two kinds of class members: instance and class members. You can use instance variables if you have an actual copy of a class on hand (for example, created using `new`). Method variables, on the other hand, can be used at any time. Here is an example with the `Double` class we saw earlier. Now the `Double` class has an instance method called `toString` that returns a `String` representation of a `Double`'s value. So you have to have an actual `Double` on hand to use this method (how else would it know what value to use?).

On the other hand, the `Double` class has a method variable called `MAX_VALUE` that contians the maximum value a `Double` (and also a `double`) can contain. You don't need to have a `Double` on hand to access this variable.

You access instance and method members differently. Method methods are prepended with the name of the class and a period, so in this case, we could use `Double.MAX_VALUE`. This would give us a `double` containing the largest possible `double` value. For instance variables, you must use the name of the instance in place of the class name, , followed by a period. So, if we created `val` as above with:

```
    Double val = new Double(3.4);
```

we could print out its value with:

```
    System.out.println( val.toString() );
```

6.6 Methods

Syntax

[*access modifier*] [*modifier*] *return type method-name*`(`*arguments*`) {`

.

.

method body

.

.

`}`

6.7 Aritmetikai, String, és más operátorok

Az operátorok

[+] [-] [*] [/] [%] [=] [++] [--] [+=] [-=] [*=] [/=]

Megjegyzés

Az operátorok ugyanolyan értéket adnak vissza, mint az operandusok típusa közül a magasabb típus. Pl.: egy float és egy double típusú szám összege double lesz.

Részletezve:

Művelet: `szintaxis`
Magyarázat

összeadás: *érték_1 + érték_2*
Számok esetén a szokásos összeadás. Stringeknél összefűzést jelent, azaz a `"One is "+1+" and all alone"` művelet eredménye az **One is 1 and all alone** szöveg

---

kivonás: *value_1 - value_2*
Számok esetén a szokásos kivonás.

---

szorzás: *value_1 * value_2*
Számok esetén a szokásos szorzás

---

osztás: *value_1 / value_2*
Az osztás eredménye lebegőpontos számokra a lebegőpontos hányados. Egész operandusok esetén az eredmény is egész, mégpedig a hányados abszolút értéke lefelé kerekítve, előjele pedig a hányados előjele. pl: egészekre: -8/3 ➔ -2 (-2,6666.. absz. ért. lefelé kerekítve)

---

modulo: *value_1 % value_2*
Csak egész számokra értelmezett, a value_1-nek value_2-vel való osztásakor keletkező maradékot adja. Pl: 13 % 4 = 1

értékadás: *változónév = value*

---

inkrementálás (növelés 1-gyel): *változónév* ++ OR ++ *változónév*
Az **increment** operátor egyet ad a változóhoz. A két formája abban különbözik, hogy a növelés előbb történik meg, mint ahogyan a változó értéke részt vesz valamilyen műveletben, vagy a behelyettesítés után. Az alábbi példában az első esetben az y változó megkapja az x változó 3 értékét, majd az x növekszik 1-gyel, így a kimenetben x 4, y 3 lesz. A második esetben az inkrementálás az értékadás előtt következik be, így mindkét változó értéke 4 lesz.. So

```
int x=3;
```

```
int y=x++;
System.out.println("x is "+x+". y is "+y+".");
```
would output **x is 4. y is 3.** whereas
```
int x=3;
int y=++x;
System.out.println("x is "+x+". y is "+y+".");
```
would output **x is 4. y is 4.**.

---

decrement: *variable* `--` OR `--` *variable*
A **decrement** (csökkentés) operátor kivon egyet a változóból Működése megegyezik az increment operátoréval.

---

plusz értékadás: *variable* `+=` *value*

A **plus assigment** operátor egyszerű formája a *variable* = *variable* + *value* kifejezésnek. Például: ossz=ossz + tag helyett írhatjuk: ossz += tag. A következő három művelet működése is hasonló.

---

minus assignment: *variable* `-=` *value*

The **minus assigment** operator is a short form for *variable* = *variable* - *value*.

---

multiplication assignment: *variable* `*=` *value*

The **multiplication assigment** operator is a short form for *variable* = *variable* * *value*.

---

division assignment: *variable* `/=` *value*

The **minus assigment** operator is a short form for *variable* = *variable* / *value*.

6.8 Logikai (Boolean) Operátorok

THE OPERATORS

[!] [&&] [||] [==] [!=] [<] [>] [<=] [>=]

Megjegyzés

A szintaxis operátoronként változik, azonban mindegyik művelet boolean típusú értékkel tér vissza. Az alábbiakban a változók boolean_1; boolean_2,.. nevet kapnak

---

not: `!` *boolean_1*

Logikai NEM művelet

---

and: *boolean_1* `&&` *boolean_2*

Logikai ÉS operátor. Értéke IGAZ, ha mindkét operandus IGAZ, különben HAMIS

---

or: *boolean_1* `||` *boolean_2*

Logikai VAGY operátor. Értéke IGAZ, ha legalább az egyik operandus IGAZ, különben HAMIS

---

egyenlő: *kifejezés_1 == kifejezés_2*
Eredménye IGAZ, ha a két kifejezés értéke pontosan egyenlő, különben HAMIS

---

nem egyenlő: *expression_1 != expression_2*
Eredménye IGAZ, ha a két kifejezés értéke eltérő, különben HAMIS.

---

kisebb mint: *value_1 < value_2*

---

nagyobb mint: *value_1 > value_2*
Numerikus értékekre értelmezett

---

kisebb, vagy egyenlő: *value_1 <= value_2*
Numerikus értékekre értelmezett

---

nagyobb, vagy egyenlő: *value_1 >= value_2*
Numerikus értékekre értelmezett

Konstansok

e: `Math.E`
double típusú, értéke a természetes alapú logaritmus alapszáma (közelítőleg) 2,718281…
Pi: `Math.PI`
double. A matematikai π értékét közelíti.
infinity: `Double.POSITIVE_INFINITY`
Returns a `double` representation of positive infinity.
-infinity: `Double.NEGATIVE_INFINITY`
Returns a `double` representation of negative infinity.
"
not a number: `Double.NaN`
Returns a `double` representation of an IEEE NaN. The NaN is often generated during floating point calculations to signify an exceptional situation. For example, `Math.sqrt(-1)` will return a NaN, since Java does not know implicitly about complex numbers.

Függvények

---

abs `Math.abs(`*value*`)`
Returns the absolute integer value of a.

---

arc cosine `Math.acos(`*value*`)`
Returns the arc cosine of a, in the range of 0.0 through Pi.

---

arc sine `Math.asin(`*value*`)`
Returns the arc sine of a, in the range of -Pi/2 through Pi/2.

---

arc tangent `Math.atan(`*value*`)`
Returns the arc tangent of a, in the range of -Pi/2 through Pi/2.

---

ceiling `Math.ceil(`*value*`)`
: Returns the smallest whole number greater than or equal to a.

---

cosine `Math.cos(`*value*`)`
: Returns the trigonometric cosine of an angle.

---

exp `Math.exp(`*value*`)`
: Returns the exponential number e(2.718...) raised to the power of a.

---

floor `Math.floor(`*value*`)`
: Returns the "floor" or largest whole number less than or equal to a.

---

logarithm `Math.log(`*value*`)`
: Returns the natural logarithm (base e) of *value*. To get base 10 logarithm, use `Math.log(`*value*`)/Math.log(10)`.

---

max `Math.max(`*value_1*`,`*value_2*`)`
: Returns the greater of *value_1* and *value_2*.

---

min `Math.min(`*value_1*`,`*value_2*`)`
: Returns the lesser of *value_1* and *value_2*.

---

power `Math.pow(`*value_1*`,`*value_2*`)`
: Returns *value_1* to the power of *value_2*.

random number `Math.random()`
: Returns a pseudo-random number between 0.0 and 1.0.

---

round `Math.round(`*value*`)`
: Rounds off a floating point value by first adding 0.5 to it and then returning the largest integer that is less than or equal to this new value.

---

sine `Math.sin(`*value*`)`
: Returns the trigonometric sine of an angle.

---

sqare root `Math.sqrt(`*value*`)`
: Returns the square root of a.

---

tangent `Math.tan(`*value*`)`
: Returns the trigonometric tangent of an angle.

6.10 do-while ciklus

SYNTAX

```
do {
    utasítások ;
} while(feltétel);
```

PROGRAM FLOW

1. The *statement(s)* are executed.
2. The boolean expression *condition* is evaluated. If *condition* evaluates as true, then flow returns to step 1.

EXAMPLE

The following example uses iteration to find an approximate solution to x=f(x). The user must specify the function f(x), a starting point x1 and a precision prec > 0. The loop computes x2 = f(x1). If |x2-x1| is smaller than prec, the loop replaces x1 by x2 and terminates. Otherwise it replaces x1 by x2 and repeats. To trap infinite loops, a count is kept of the number of iterations. The loop also terminates if the count exceeds a specified count_limit.

```
int count = 0 ;
int count_limit = 1000 ;
double error, x2 ;

do {
     x2 = f(x1) ;
     error = Math.abs(x2-x1) ;
     x1 = x2 ;
     count++ ;
} while ( (error > prec) && (count < count_limit) )  ;
```

COMMON ERRORS

Don't forget the semicolon after the condition.

6.11 while loop

SYNTAX

```
while(condition) {
    statements ;
}
```

OR

```
while (condition) statement;
```

PROGRAM FLOW

1. The boolean expression *condition* is first evaluated. If it evaluates to true, flow continues to step 2, otherwise the loop terminates.
2. The *statement(s)* are executed. Flow continues with step 1.

EXAMPLE

The following example uses the method of bisection to find an approximate solution to f(x)=0. The user must specify the function f(x), two points x1 and x2 such that f(x1) and f(x2) are of opposite sign and a precision prec > 0. When the loop terminates x1 and x2 have new values with |x1-x2| smaller than prec and f(x1) and f(x2) are of still of opposite sign. So that, assuming that f(x) is continuous, f(x)=0 has a solution between x1 and x2.

```
double x3 ;
int sign ;
if ( f(x1) >= 0 )  sign =1;
else               sign = -1 ;

while ( Math.abs(x1-x2) > prec ) {
      x3 = (x1+x2)/2 ;
      if ( sign*f(x3) >= 0 ) x1 = x3 ;
      else                   x2 = x3 ;
}
```

COMMON ERRORS

Don't forget the semicolon after the condition.

6.12 for loop

SYNTAX

```
for (initialization ; test ; increment) {
     statements ;
}
```

OR

```
for (initialization ; test ; increment)
     statement ;
```

PROGRAM FLOW

1. The *initialization* is first executed. This is typically something like `int i=0`, which creates a new variable with initial value 0, to act as a counter. Variables that you declare in this part of the for loop cease to exist after the execution of the loop is completed. Multiple, comma separated, expressions are allowed in the initialization section. But declaration expressions may not be mixed with other expressions.
2. The boolean expression *test* is then evaluated. This is typically something like `i<10`. Multiple, comma separated, expressions are not allowed. If *test* evaluates to true, flow contiues to step 3. Otherwise the loop exits.
3. The *statement(s)* are executed.

4. Then the statement *increment* is executed. It is typically something like `i++` , which increments `i` by one or `i+=2` , which increments `i` by two. Multiple, comma separated, expressions are allowed in the increment section.
5. Flow returns to step 2.

EXAMPLE

The following example computes an approximate value for the integral of f(x) from x1 to x2 using nsteps steps of the trapezoidal rule.

```
double dx = (x2-x1)/nsteps ;
double x = x1+dx ;
double integral = 0 ;

for (int i=1 ; i< nsteps ; i++, x+= dx) {
     integral += f(x) ;
}

integral = dx*(integral +( f(x1)+f(x2) )/2) ;
```

COMMON ERRORS

Putting a semicolon after the closing) as in

```
for (initialization ; test ; increment) ;
     statement ;
```

terminates the for loop immediatley. *statement* is NOT PART OF THE LOOP. If you find you are plagued by this problem, try always using the multiple statement format even if you have just one statement in the loop body.

6.13 if-else

SYNTAX

```
if (condition) {
     true_statements ;
}
```

OR

```
if (condition) true_statement ;
```

OR

```
if (condition) {
     true_statements ;
} else {
     false_statements ;
}
```

PROGRAM FLOW

1. First, the boolean expression *condition* is evaluated.
2. If *condition* evaluates to `true`, the *true_statement(s)* are executed.
3. If *condition* evaluates to `false` and an `else` clause exists, the *false_statement(s)* are executed.
4. Flow exits the if-else structure.

EXAMPLE

The following example tests to see if a user supplied function `f` comes within a `tolerance` of zero and prints an appropriate message.

```
if( Math.abs(f(x)) <= tolerance) {
 System.out.println("Zero found at x="+x);
}
else {
 System.out.println("Failed to find a zero");
}
```

COMMON ERRORS

Make sure that if you want to test equality in the *condition* to use the boolean operator `==` and not the assignment operator `=`. Consider the following bad piece of code.

```
double x=0;
if(x=1) {
  System.out.println("x is 0");
} else {
  System.out.println("x is not 0. x is "+x);
}
```

The output will be
x is not 0. x is 1