Java Basics

Java – Overview

Java programming language was originally developed by Sun Microsystems which was initiated by James Gosling and released in 1995 as core component of Sun Microsystems' Java platform (Java 1.0 [J2SE]).

With the advancement of Java and its widespread popularity, multiple configurations were built to suite various types of platforms. Ex: J2EE for Enterprise Applications, J2ME for Mobile Applications.

Sun Microsystems has renamed the new J2 versions as Java SE, Java EE and Java ME respectively. Java is guaranteed to be **Write Once, Run Anywhere.**

Java is:

- **Object Oriented:** In Java, everything is an Object. Java can be easily extended since it is based on the Object model.
- **Platform independent:** Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent **byte code**. This byte code is distributed over the web and interpreted by virtual Machine (JVM) on whichever platform it is being run.

- **Simple:**Java is designed to be easy to learn. If you understand the basic concept of OOP Java would be easy to master.
- **Secure:** With Java's secure feature it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.
- **Architectural-neutral :**Java compiler generates an architecture-neutral object file format which makes the compiled code to be executable on many processors, with the presence of Java runtime system.

- **Portable:** Being architectural-neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler in Java is written in ANSI C.
- **Robust:** Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.
- **Multithreaded:** With Java's multithreaded feature it is possible to write programs that can do many tasks simultaneously. This design feature allows developers to construct smoothly running interactive applications.
- **Interpreted:** Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The

development process is more rapid and analytical since the linking is an incremental and light weight process.

- **High Performance:** With the use of Just-In-Time compilers, Java enables high performance.
- **Distributed:** Java is designed for the distributed environment of the internet.
- **Dynamic:** Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

Note on JAVA language

Java as C++ descendant

In second half on nineties a simple support of object paradigm proved to be insufficient, new language Java appears. Unlike C++ it doesn’t take over all the elements without thinking about them and some of them are left out on purpose, such as pointers, but anyway it builds upon C syntax.

Generally

Very wide spread, interesting and quite high quailty environment is NetBeans.

Its main three advantages are:

- It is highly supported by Oracle
- It has high quailty *free* version
- It is applicable for many languages. Besides C++ these are JAVA, PHP, Python and many others

Java Basic Elements

When we consider a Java program it can be defined as a collection of objects that communicate via invoking each other's methods. Let us now briefly look into what do class, object, methods and instance variables mean.

Object – In real world, objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors -wagging, barking, eating. An object is an instance of a class.

Class - A class can be defined as a template/ blue print that describes the behaviors/states that object of its type support.

Methods - A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed. It is equivalent functions from classic programming languages

Instance Variables - Each object has its unique set of instance variables. An object's state is created by the values assigned to these instance variables.

Basic Syntax:

We can see Java basic syntax on a simple example:

```
public class MyFirstJavaProgram {

   /* This is my first java program.
    * This will print 'Hello World'
      as the output */

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

Remember

About Java programs, it is very important to keep in mind the following points:

Case Sensitivity - Java is case sensitive, which means identifier *Hello* and *hello* would have different meaning in Java.

Class Names - For all class names the first letter should be in **Upper** Case.
If several words are used to form a name of the class, each inner word's first letter should be in Upper Case.
Example class: MyFirstJavaClass

Method Names - All method names should start with a **Lower** Case letter.
If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case.
Example: public void myMethodName()

Program File Name - Name of the program file should exactly match the class name.
When saving the file, you should save it using the class name (Remember, Java is case sensitive!) and append '.java' to the end of the name. **If the file name and the class name do not match your program will not compile**.

Example: Assume 'MyFirstJavaProgram' is the class name. Then the file should be saved as 'MyFirstJavaProgram.java'

public static void main(String args[]) - Java program processing starts from the *main()* method which is a mandatory part of every Java program.

Java Identifiers:

All Java components require names. Names used for classes, variables and methods are called **identifiers**.

Remember

In Java, there are several points to remember about identifiers. They are as follows:

All identifiers should begin with a letter (**A** to **Z** or **a** to B), currency character ($) or an underscore (_).

After the first character identifiers can have any combination of characters. “Camel notation” like *ThisIsMyNewClass* is recommended.

A **key word** cannot be used as an identifier.

Most importantly identifiers are case sensitive.

Examples of legal identifiers: *age, $salary, _value, __1_value*

Examples of illegal identifiers: *123abc, -salary*

Java Keywords

In Java, there are the following keywords. These keywords must not be used as identifiers.

abstract	assert	boolean	break
byte	case	catch	char
class	const	continue	default
do	double	else	enum
extends	final	finally	float
for	goto	if	implements
import	instanceof	int	interface
long	native	new	package
private	protected	public	return

short	static	strictfp	super
switch	synchronized	this	throw
throws	transient	try	void
volatile	while		

Java Types

What is a Type

In computer memory, all data items are expressed using ones and zeros:

```
0110 1100
```

But this can have a lot of meanings. It can be a program instruction. It can be a number. It can be a character. It can be a pixel in an image etc. Working with the numbers is quite different than working with texts. And with that again different than with pictures. Therefore, the computer needs to know exactly what the data means.

We tell him it using **types**. If data item represents a number, then we say that its type is a number. The character is said to be of type character. And so on.

Two kinds of types

The Java programming language is statically-typed, which means that all variables must first be declared before they can be used.

This involves stating the variable's type and name, as you've already seen.

At the beginning it is necessary to say that the types can be divided into two categories: **primitive types** and **referential types**.

Primitive Data Types

One example of primitive data type shows figure.

The Java programming language is statically-typed, which means that all variables must first be declared before they can be used. This involves stating the variable's type and name, as you've already seen:

```
int price = 123456;
```

Doing so tells your program that a field named "price" exists, holds numerical data, and has an initial value of "123456". A variable's data type determines the values it may contain, plus the operations that may be performed on it. In addition to *int*, the Java programming language supports seven other primitive data types. A primitive type is predefined by the language and is named by a reserved keyword. Primitive values do not share state with other primitive values.

The eight primitive data types supported by the Java programming language are:

type	description	size	min. value	max. value

byte	integer	8 bits	-128	+127
short	integer	16 bits	-32768	+32767
int	integer	32 bits	-2147483648	+2147483647
long	integer	64 bits	-922337203…	+922337203…
float	floating point num.	32 bits	-3.40282e+38	+3.40282e+38
double	floating point num.	64 bits	-1.797…e+308	+1.797…e+308
char	UNICODE character	16 bits	/u0000	/uFFFF
boolean	boolean value	1 bit	-	-

- **byte**: The byte data type is an 8-bit signed two's complement integer. It has a minimum value of -128 and a maximum value of 127 (inclusive). The byte data type can be useful for saving memory in large arrays, where the memory savings actually matters. They can also be used in place of int where their limits help to clarify your code; the fact that a variable's range is limited can serve as a form of documentation.

- **short**: The short data type is a 16-bit signed two's complement integer. It has a minimum value of -32 768 and a maximum value of 32 767 (inclusive). As with byte, the same guidelines

apply: you can use a short to save memory in large arrays, in situations where the memory savings actually matters.

•**int**: By default, the *int* data type is a 32-bit signed two's complement integer, which has a minimum value of -2^{31} and a maximum value of $2^{31}-1$. In Java SE 8 and later, you can use the int data type to represent an unsigned 32-bit integer, which has a minimum value of 0 and a maximum value of $2^{32}-1$. Use the *Integer* class to use *int* data type as an unsigned integer. See the section The Number Classes for more information. Static methods like compareUnsigned, divideUnsigned etc. have been added to the Integer class to support the arithmetic operations for unsigned integers.

- **long**: The long data type is a 64-bit two's complement integer. The signed long has a minimum value of -2^{63} and a maximum value of $2^{63}-1$. In Java SE 8 and later, you can use the long data type to represent an unsigned 64-bit long, which has a minimum value of 0 and a maximum value of $2^{64}-1$. Use this data type when you need a range of values wider than those provided by int. The Long class also contains methods like compareUnsigned, divideUnsigned etc. to support arithmetic operations for unsigned long.

- **float**: The float data type is a single-precision 32-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in the Floating-Point Types, Formats,

and Values section of the Java Language Specification. As with the recommendations for byte and short, use a float (instead of double) if you need to save memory in large arrays of floating point numbers. This data type should never be used for precise values, such as currency. For that, you will need to use the java.math.BigDecimal class instead. Numbers and Strings covers BigDecimal and other useful classes provided by the Java platform.

•**double**: The double data type is a double-precision 64-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in the Floating-Point Types, Formats, and Values section of the Java Language Specification. For

decimal values, this data type is generally the default choice. As mentioned above, this data type should never be used for precise values, such as currency.

•**boolean**: The boolean data type has only two possible values: true and false. Use this data type for simple flags that track true/false conditions. This data type represents one bit of information, but its "size" isn't something that's precisely defined.

•**char**: The char data type is a single 16-bit Unicode character. It has a minimum value of '\u0000' (or 0) and a maximum value of '\uffff' (or 65 535 inclusive).

In addition to the eight primitive data types listed above, the Java programming language also provides special support for **character strings** via the `java.lang.String` class.

Enclosing your character string within double quotes will automatically create a new String object; for example

```
String s = "this is a string";
```

String objects are immutable, which means that once created, **their values cannot be changed**. The String class is not technically a primitive data type, but considering the special support given to it by the language, you'll probably tend to think of it as such.

Literals

Primitive types are special data types built into the language; they are not objects created from a class. It's possible to assign a literal to a variable of a primitive type:

```
boolean result = true;
char capitalC = 'C';
byte b = 100;
short s = 10000;
int i = 100000;
```

An integer literal is of type long if it ends with the letter L or l; otherwise it is of type int. It is recommended that you use the upper case letter L because the lower case letter l is hard to distinguish from the digit 1.

Values of the integral types byte, short, int, and long can be created from int literals. Values of type long that exceed the range of int can be created from long literals. Integer literals can be expressed by these number systems:

- Decimal: Base 10, whose digits consists of the numbers 0 through 9; this is the number system you use every day
- Hexadecimal: Base 16, whose digits consist of the numbers 0 through 9 and the letters A through F
- Binary: Base 2, whose digits consists of the numbers 0 and 1 (you can create binary literals in Java SE 7 and later)

For general-purpose programming, the decimal system is likely to be the only number system you'll ever use. However, if you need to use another number system, the following example shows the correct syntax. The **prefix 0x indicates hexadecimal** and **0b indicates binary**:

```
// The number 26, in decimal
int decVal = 26;

//  The number 26, in hexadecimal
int hexVal = 0x1a;

// The number 26, in binary
int binVal = 0b11010;
```

A floating-point literal is of type float if it ends with the letter F or f; otherwise its type is double and it can optionally end with the letter D or d.

The floating point types (float and double) can also be expressed using E or e (for scientific notation), F or f (32-bit float literal) and D or d (64-bit double literal; this is the default and by convention is omitted).

```
double d1 = 123.4;
// same value as d1, but in scientific
notation
double d2 = 1.234e2;
float f1  = 123.4f;
```

Literals of types char and String may contain any Unicode (UTF-16) characters. If your editor and file system allow it, you can use such characters directly in your code. If not, you can use a "Unicode escape" such as '\u0108' (capital C with circumflex), or e.g. "S\u00ED Se\u00F1or" (Sí Seňor in Spanish).

Always use 'single quotes' for *char* and "double quotes" for *Strings*.

Unicode escape sequences may be used elsewhere in a program (such as in field names, for example), not just in char or String literals.

The Java programming language also supports a few special escape sequences for char and String literals: **\b** (backspace), **\t**

(tab), **\n** (line feed), **\f** (form feed), **\r** (carriage return), **\"** (double quote), **\'** (single quote), and **** (backslash).

There's also a special **null** literal that can be used as a value for any reference type; null may be assigned to any variable, except variables of primitive types. There's little you can do with a null value beyond testing for its presence. Therefore, null is often used in programs as a marker to indicate that some object is unavailable.

In Java SE 7 and later, any number of underscore characters (_) can appear anywhere between digits in a numerical literal. This feature enables you, for example. to separate groups of digits in numeric literals, which can improve the readability of your code.

For instance, if your code contains numbers with many digits, you can use an underscore character to separate digits in groups of three, similar to how you would use a punctuation mark like a comma, or a space, as a separator.

The following example shows other ways you can use the underscore in numeric literals:

```
long creditCardNumber =
1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
float pi =  3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
```

```
byte nibbles = 0b0010_0101;
long bytes =
0b11010010_01101001_10010100_10010010;
```

The referential data types

All data types that are not primitive, are referential. Example of referential data types are classes, all types of arrays etc.

One example of referential data type shows figure.

Referential data types are stored in memory in two places. When you declare a type, the compiler creates space in memory and assigns it a name. We say that we have declared a variable. Interestingly, the length of the place does not matter how many

bytes occupies a reference type; place has such a length that it could be the memory address

In the second step, which is called initialization, creates the space for the variable itself. This place has the size as needed to fit all data belonging to a variable. The address of this place (or reference) is inserted into a named location, which originated in the declaration

Both operations, declaration and initialization, can be combined into a single command, as shown in the figure.

This way of working with reference types has advantages and disadvantages:

The **advantage** is that the data size can be changed dynamically at runtime, as needed.

The disadvantage is that **each reference type must be initialized before we can use it**. It is not enough to declare it like this

```
int [] thisIsBad;
```

because there is still not created any place for data and that `thisIsBad` is empty. Any attempt to work with `thisIsBad` without initialization fails.

Java Modifiers:

Like other languages, it is possible to modify classes, methods, etc., by using **modifiers**. There are two categories of modifiers:

- **Access Modifiers:** default, public , protected, private
- **Non-access Modifiers:** final, abstract, strictfp

Java Variables

We will see following kinds of variables in Java:

- **Instance Variables** (Non-static variables). Instance variables reside and are valid only in its own instance. Instance variables are not accessible in basic class. Results that

instance variables with same name, in different instances can have distinct values.

- Class Variables (**Static Variables**) are defined in class and their modifier is **static**. Static variables reside in class. There is only one copy and it is shared between all instances. As a result, if one instance changes value in a static variable, changed value reflects in all instances.
- **Local Variables** are variables that are created on-the-fly, only for purposes of one command or one block. Example:

```
for (int i=0; i<7; i++)…
```

Comments in Java

Java supports single-line and multi-line comments very similar to C and C++. All characters available inside any comment are ignored by Java compiler.

```
public class MyFirstJavaProgram{

   /* This is my first java program.
      This will not print 'Hello World'
   as the output.
      This is an example of multi-line
   comment. */

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

```
        // This is a single line comment
   /* This is a single line comment.*/
        System.out.println("Hello World");
    }
}
```

Using Blank Lines:

A line containing only whitespaces (space, TAB, CR/LF), possibly with a comment, is known as a blank line, and Java totally ignores it.

Java Arrays

Arrays are objects that store multiple variables of the same type. An array itself is an object on the *heap*. We will look into how to declare, construct and initialize in the upcoming chapters.

Java Enums:

Enums were introduced in java 5.0. Enums restrict a variable to have one a few predefined values. The values in this enumerated list are called enums.

With the use of enums it is possible to reduce the number of bugs in your code.

For example, if we consider an application for a fresh juice shop, it would be possible to restrict the glass size to small, medium and large. This would make sure that it would not allow anyone to order any size other than the small, medium or large. Moreover, it is much more illustrative than using numeric constants (such as 0 = small, medium = 1, etc..)

Example:

```
class FreshJuice {

// here I will prepare data
```

```
// type FreshJuiceSize
enum FreshJuiceSize{SMALL,MEDIUM,LARGE}

// I will create a variable “size”
// of this data type
FreshJuiceSize size;
}

public class TestClass{

public static void main(String []args){
System.out.println("Starting:");

// I will create
// juice = FreshJuice class instance
```

```
FreshJuice juice = new FreshJuice();

// FreshJuice class contains
// variable “size”
// and also
// constant MEDIUM
juice.size =
FreshJuice.FreshJuiceSize.MEDIUM;

System.out.println("Size: " +
juice.size);
System.out.println("Finished OK.");
}
}
```

Above example will produce the following result:

```
Size: MEDIUM
```

Note: enums can be declared as their own or inside a class.

Methods, variables, constructors can be defined inside enums as well.

Java Basic Operators

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups:

- Arithmetic Operators
- Relational Operators
- Bitwise Operators
- Logical Operators
- Assignment Operators
- Misc Operators

The Arithmetic Operators:

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Assume integer variable A holds 10 and variable B holds 20, then:

Operator	Description	Example
+	Addition - Adds values on either side of the operator	A + B gives 30
-	Subtraction - Subtracts right hand operand from	A - B gives -10

	left hand operand	
*	Multiplication - Multiplies values on either side of the operator	A * B gives 200
/	Division - Divides left hand operand by right hand operand	B / A gives 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	B % A gives 0
++	Increment - Increases the value of operand by 1	B++ gives 21
--	Decrement - Decreases the	B-- gives 19

value of operand by 1

The Relational Operators:

There are following relational operators supported by Java language

Assume variable A holds 10 and variable B holds 20, then:

Opera-tor	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not truenení pravda.

!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less	(A <= B) is

	than or equal to the value of right operand, if yes then condition becomes true.	true.

Beware that equity is tested by double == so that it doesn't mix up with assignment into variable which is done by single = . It is very common mistake.

If we want to negate some expression, we enclose it with brackets and we place exclamation mark before the brackets.

If we want to execute more than one command we must insert the command into a block consisting of curly brackets.

Beware!

We compare strings with equals() method, not with == operator used with numbers!

That is so because a String is referential data type. Condition `"Text1" == "Text2"` is incorrect, we must type condition like `"Text1".equals("Text2")`.

The Bitwise Operators:

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.

Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60; and b = 13; now in binary format they will be as follows:

```
  a = 0011 1100
  b = 0000 1101
-----------------
a&b = 0000 1100
a|b = 0011 1101
a^b = 0011 0001
~a  = 1100 0011
```

The following table lists the bitwise operators.

Assume integer variable A holds 60 and variable B holds 13 then:

Oper	Description	Example

a-tor		
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) gives 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) gives 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) gives 49 which is 0011 0001
~	Binary Ones Complement Operator is	(~A) gives -61 which is 1100 0011 in 2's complement form

	unary and has the effect of 'flipping' bits.	due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 gives 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 gives 15 which is 1111

>>>	Shift right, zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	A >>>2 gives 15 which is 0000 1111

The Logical Operators:

The following table lists the logical operators.

Assume Boolean variables A holds true and variable B holds false, then:

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

The Assignment Operators

There are following assignment operators supported by Java language:

Opera-tor	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A

-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
***=**	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and	C %= A is equivalent

	assign the result to left operand	to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment	C |= 2 is

operator	same as C = C | 2

Misc Operators

There are few other operators supported by Java Language.

Conditional Operator (? :):

Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide which value should be assigned to the variable. The operator is written as:

```
variable x = (expression) ? value if true : value if false
```

Following is the example:

```
public class Test {

   public static void main(String args[]){
      int a , b;
      a = 10;
      b = (a == 1) ? 20: 30;
      System.out.println("Value of b is : "
     +  b );

      b = (a == 10) ? 20: 30;
      System.out.println("Value of b is : "
     + b );
```

```
    }
}
```

This would produce the following result:

```
Value of b is : 30
Value of b is : 20
```

instanceof Operator:

This operator is used only for object referential variables. The operator checks whether the object is of a particular type(class type or interface type). instanceof operator is wriiten as:

```
( Referential variable ) instanceof
(class/interface type)
```

If the object referred by the variable on the left side of the operator passes the IS-IT? check for the class/interface type on

the right side, then the result will be true. Following is the example:

```
public class Test {

   public static void main(String args[]){
      String name = "James";
   // following will return true
   // since name is type of String
      boolean result = name
     instanceof String;
      System.out.println( result );
   }
}
```

This would produce the following result:

```
true
```

This operator will still return true if the object being compared is the **assignment compatible** with the type on the right. Following is one more example:

```
class Vehicle {}

public class Car extends Vehicle {
   public static void main(String args[]){
      Vehicle a = new Car();
      boolean result =  a instanceof Car;
      System.out.println( result );
   }
}
```

This would produce the following result:

```
true
```

Precedence of Java Operators:

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

For example, x = 7 + 3 * 2; here x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within

an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	() [] . (dot operator)	Left to right
Unary	++ - - ! ~	**Right to left**
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	>> >>> <<	Left to right
Relational	> >= < <=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right

Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	**Right to left**
Assignment	= += -= *= /= %=>> = <<= &= ^= |=	**Right to left**
Comma	,	Left to right

Java Operators Usage Examples

The following simple example program demonstrates the arithmetic operators. Copy and paste the following Java program in Test.java file and compile and run this program:

```
public class Test {

  public static void main(String args[]) {
     int a = 10;
     int b = 20;
     int c = 25;
     int d = 25;
     System.out.println("a + b = "
     + (a + b) );
     System.out.println("a - b = "
     + (a - b) );
     System.out.println("a * b = "
     + (a * b) );
     System.out.println("b / a = "
     + (b / a) );
```

```
        System.out.println("b % a = "
        + (b % a) );
        System.out.println("c % a = "
        + (c % a) );
        System.out.println("a++   = "
        +  (a++) );
        System.out.println("b--   = "
        +  (a--) );
        // Check the difference in d++ and ++d
        System.out.println("d++   = "
        +  (d++) );
        System.out.println("++d   = "
        +  (++d) );
    }
}
```

This would produce the following result:

```
a + b = 30
a - b = -10
a * b = 200
b / a = 2
b % a = 0
c % a = 5
a++   = 10
b--   = 11
d++   = 25
++d   = 27
```

Java - Relational Operators Example

The following simple example program demonstrates the relational operators. Copy and paste the following Java program in Test.java file and compile and run this program. :

```
public class Test {

  public static void main(String args[]) {
     int a = 10;
     int b = 20;
     System.out.println("a == b = "
     + (a == b) );
     System.out.println("a != b = "
     + (a != b) );
     System.out.println("a > b = "
```

```
        + (a > b) );
        System.out.println("a < b = "
        + (a < b) );
        System.out.println("b >= a = "
        + (b >= a) );
        System.out.println("b <= a = "
        + (b <= a) );
    }
}
```

This would produce the following result:

```
a == b = false
a != b = true
a > b = false
a < b = true
b >= a = true
```

```
b <= a = false
```

Java Bitwise Operators Example

The following simple example program demonstrates the bitwise operators. Copy and paste the following Java program in Test.java file and compile and run this program:

```
public class Test {

  public static void main(String args[]) {
     int a = 60;/* 60 = 0011 1100 */
     int b = 13;/* 13 = 0000 1101 */
     int c = 0;
```

```
c = a & b;         /* 12 = 0000 1100 */
System.out.println("a & b = " + c );

c = a | b;         /* 61 = 0011 1101 */
System.out.println("a | b = " + c );

c = a ^ b;         /* 49 = 0011 0001 */
System.out.println("a ^ b = " + c );

c = ~a;            /*-61 = 1100 0011 */
System.out.println("~a = " + c );

c = a << 2;     /* 240 = 1111 0000 */
System.out.println("a << 2 = " + c );
```

```
      c = a >> 2;      /* 215 = 1111 */
      System.out.println("a >> 2  = " + c );

      c = a >>> 2;      /* 215 = 0000 1111 */
      System.out.println("a >>> 2 = " + c );
   }
}
```

This would produce the following result:

```
a & b = 12
a | b = 61
a ^ b = 49
~a = -61
a << 2 = 240
a >> 15
a >>> 15
```

Java Logical Operators Example

It is possible to combine conditions, with help of two basic operators:

Operator	Significance
&&	and at the same time
||	or

A priority can be adjusted with brackets as in the example below:

```
Scanner sc = new Scanner(System.in,
   "Windows-1250");
```

```
   System.out.println(""Input an integer
number
   between 10-20 or 30-40:");
int a = Integer.parseInt(sc.nextLine());
if (((a >= 10) && (a <= 20))
   || ((a >=30) && (a <= 40)))
   System.out.println(
      "Your input is correct");
else
         System.out.println(
      "Wrong input");
```

The following simple example program also demonstrates the logical operators. Copy and paste the following Java program in *Test.java* file and compile and run this program:

```
public class Test {

  public static void main(String args[]) {
     boolean a = true;
     boolean b = false;

     System.out.println("a && b = "
     + (a&&b));

     System.out.println("a || b = "
     + (a||b) );

     System.out.println("!(a && b) = "
     + !(a && b));
  }
```

```
}
```

This would produce the following result:

```
a && b = false
a || b = true
!(a && b) = true
```

Java Assignment Operators Example

The following simple example program demonstrates the assignment operators. Copy and paste the following Java program in Test.java file and compile and run this program:

```
public class Test {

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

```
        int a = 10;
        int b = 20;
        int c = 0;

        c = a + b;
        System.out.println("c = a + b = " + c
);

        c += a ;
        System.out.println("c += a  = " + c );

        c -= a ;
        System.out.println("c -= a = " + c );

        c *= a ;
```

```
System.out.println("c *= a = " + c );

a = 10;
c = 15;
c /= a ;
System.out.println("c /= a = " + c );

a = 10;
c = 15;
c %= a ;
System.out.println("c %= a  = " + c );

c <<= 2 ;
System.out.println("c <<= 2 = " + c );
```

```
        c >>= 2 ;
        System.out.println("c >>= 2 = " + c );

        c >>= 2 ;
        System.out.println("c >>= a = "+ c );

        c &= a ;
        System.out.println("c &= 2 = "+ c );

        c ^= a ;
        System.out.println("c ^= a = "+ c );

        c |= a ;
        System.out.println("c |= a = "+ c );
    }
```

```
}
```

This would produce the following result:

```
c = a + b = 30
c += a  = 40
c -= a = 30
c *= a = 300
c /= a = 1
c %= a  = 5
c <<= 2 = 20
c >>= 2 = 5
c >>= 2 = 1
c &= a  = 0
c ^= a   = 10
c |= a   = 10
```

Java Decision Making

There are two types of decision making statements in Java. They are:

- **if** statements
- **switch** statements

The if Statement:

An *if* statement consists of a Boolean expression followed by one or more statements.

Syntax:

The syntax of an *if* statement is:

```
if(Boolean_expression)
{
   //Statements will execute
   //if the Boolean expression is true
}
```

If the Boolean expression evaluates to true then the block of code inside the if statement will be executed. If not the first set of code after the end of the if statement (after the closing curly brace) will be executed.

Example:

```
public class Test {

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

```
      int x = 10;

      if( x < 20 ){
         System.out.print
     ("This is if statement");
      }
   }
}
```

This would produce the following result:

```
This is if statement
```

The if...else Statement:

An *if* statement can be followed by an optional *else* statement, which executes when the Boolean expression is false.

Incorrect solution

Let's first see the example of incorrect solution. Our objective is to calculate a square root. In the following example don't forget to import *java.util.Scanner*, so that the program knows the Scanner class! Aslo don't forget to import Math class, which contaings mathematical functions including *sqrt()*, the square root function.

```
import java.lang.Math;
Scanner sc = new Scanner(System.in,
      "Windows-1250");
   System.out.println("Input some number and
"
   +"I will calculate a square root "
```

```
   +"from it:");
int a = Integer.parseInt(sc.nextLine());
if (a > 0){
        System.out.println(
     "You inputed number greaer than 0, "
     +" it is possible to make "+
     "a square root from it.");
        double o = Math.sqrt(a);
        System.out.println("Square root of
number  "
     + a + " is " + o);
}
System.out.println("Děkuji za zadání");
```

We can make sure that the algorithm is incorrect. We should treat all the variants for a correct function, i.e. like so:

```
if (a > 0){
          System.out.println("Input some "+
   "number and I will calculate "+
   "a square root from it:");
       double o = Math.sqrt(a);
          System.out.println("Square root "+
   "of number  " + a + " is " + o);
}
if (a < 0){
          System.out.println("Square root of
"+
   "a negative number doesn't exist!");
   }
System.out.println("Thank you for your
input.");
```

The code can be markedly simplified by using **else** keyword, which will execute the next statement or block of statements in case the condition becomes false. A code becomes much more synoptical and we don't have to devise opposite condition, which can be quite difficul, especially for combined condition.

Syntax:

The syntax of an if...else is:

```
if(Boolean_expression){
   //Executes when the Boolean
   // expression is true
}else{
   //Executes when the Boolean
   //expression is false
```

```
}
```

In case of multiple statements there would be another block of curly brackets **{ }** behing *else*.

Example:

```
public class Test {

   public static void main(String args[]){
      int x = 30;

      if( x < 20 ){
         System.out.print("This is "+
     "if statement");
      }else{
         System.out.print("This is ELSE");
```

```
        }
    }
}
```

This would produce the following result:

```
This is ELSE
```

The *else* statement is also used in case we need to mainpulate with variable from a condition inside of the statement, so we can't ask for it again.

The program itself remembers that condition didn't become true and will jump to the *else* section. Let's do an example for a demonstration. Let's have a number a with value 0 or 1, and we want to swap the values (if it's 1 it will become 0 and vice versa).

Try it.

This is how we would write it in a naive way:

```
int a = 0; // first we will assign 0 to a
if (a == 0) // if a==0 we assign a=1
        a = 1;
if (a == 1) // if a==1, we assign a=0
        a = 0;
System.out.println(a);
```

but that doesn't work, of course.

At the beginning there is zero in **a**, firs condition will become true and **a** will be assinged one.

But that will also meet the second condition.

If we swap the conditions we will have the same problem with one.

We better use **else**:

```
int a = 0; // first we will assign 0 to a

if (a == 0) // if a is 0 we assign 1
        a = 1;
else // if a is 1, we assign 0
        a = 0;

System.out.println(a);
```

The if...else if...else Statement:

An *if* statement can be followed by an optional *else if...else* statement, which is very useful to test complex conditions.

When using *if , else if, else* statements there are few points to keep in mind.

- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the final *else*.

- Once an else if succeeds, none of the remaining else if's or else's will be tested.

Syntax:

The syntax of an *if...else* is:

```
if(Boolean_expression1){
   //Executes when the Boolean
   //expression1 is true
}else if(Boolean_expression2){
   //Executes when the Boolean
   //expression2 is true
}else if(Boolean_expression3){
   //Executes when the Boolean
   //expression3 is true
```

```
}else {
   //Executes when the none
   //of the above condition is true.
}
```

Example:

```
public class Test {

   public static void main(String args[]){
      int x = 30;

      if( x == 10 ){
         System.out.print(
     "Value of X is
      10");
      }else if( x == 20 ){
```

```
         System.out.print(
      "Value of X is
       20");
       }else if( x == 30 ){
         System.out.print("Value of X is
       30");
       }else{
         System.out.print("This is else
      statement");
       }
   }
}
```

This would produce the following result:

```
Value of X is 30
```

Nested if...else Statement:

It is always legal to *nest if-else* statements which means you can use one if or else if statement inside another if or else if statement.

Syntax:

The syntax for a nested *if...else* is as follows:

```
if(Boolean_expression1){
   //Executes when the Boolean expression1
is true
   if(Boolean_expression2){
      //Executes when the Boolean
expression2 is true
   }
```

```
}
```

You can nest *else if...else* in the similar way as we have nested *if* statement.

Example:

```
public class Test {

   public static void main(String args[]){
      int x = 30;
      int y = 10;

      if( x == 30 ){
         if( y == 10 ){
             System.out.print("X = 30 a Y =
10");
```

```
            }
        }
    }
}
```

This would produce the following result:

```
X = 30 a Y = 10
```

The switch Statement:

Switch is a construct taken from C language (just like a majority of Java's grammar). It makes it possible to simplify the wtitting of multiple conditions.

A *switch* statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Syntax:

The syntax of switch statement:

```
switch(výraz){
    case value :
       //Příkazy
       break; //volitelně
    case value :
       //Příkazy
       break; //volitelně
    //Můžete mít libovolný
    // počet příkazů
    default : //Volitelně
       //Příkazy
}
```

The following rules apply to a *switch* statement:

- The variable used in a *switch* statement can only be a byte, short, int, or char.
- You can have any number of *case* statements within a *switch*. Each *case* is followed by the value to be compared to and a colon.
- The value for a *case* must be the same data type as the variable in the *switch* and it must be a constant or a literal.
- When the variable being switched on is equal to a *case*, the statements following that case will execute until a *break* statement is reached.

- When a *break* statement is reached, the *switch* terminates, and the flow of control jumps to the next line following the *switch* statement.

Not every case needs to contain a break. If no break appears, the flow of control will *fall through* to subsequent cases until a break is reached or until *switch* block is finished.

A *switch* statement can have an optional default case, which must appear at the end of the *switch*. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Example:

```
public class Test {

   public static void main(String args[]){
      //char grade = args[0].charAt(0);
      char grade = 'C';

      switch(grade)
      {
         case 'A' :

System.out.println("Excellent!");
            break;
         case 'B' :
         case 'C' :
```

```
            System.out.println("Well
done");
            break;
         case 'D' :
            System.out.println("You
passed");
         case 'F' :
            System.out.println(
      "Better try again");
            break;
         default :
            System.out.println(
      "Invalid grade");
      }
      System.out.println(
```

```
            "Your grade is " + grade);
    }
}
```

Compile and run above program using various command line arguments. This would produce the following result:

```
Well done
Your grade is C
```

Example:

Let's take a look at one more example. Suppose a calculator that takes 2 numbers and computes all 4 operations. But this time we want to choose which operation we want to perform:

```
System.out.println("Welcome to
calculator");
```

```
System.out.println("Input first number:");
float a = Float.parseFloat(sc.nextLine());
System.out.print("Input second number:");
float b = Float.parseFloat(sc.nextLine());
System.out.println("Choose your
operation:");
System.out.println("1 - addition");
System.out.println("2 - subtraction");
System.out.println("3 - multiplication");
System.out.println("4 - division");
int choice =
Integer.parseInt(sc.nextLine());
float result = 0;
switch (choice)
{
```

```
            case 1: result = a + b;
            break;
            case 2: result = a - b;
            break;
            case 3: result = a * b;
            break;
            case 4: result = a / b;
            break;
}
if ((choice > 0) && (choice < 5))
            System.out.printf("Result: %f",
      result);
else
            System.out.println("Invalid
choice");
```

Notice that we declared a variable *result* at the beginning only, that is the only way for us to assign it later on. If we declared it at each assignement Java wouldn’t compile the code and it would throw and variable redeclaration error!

It is also important to assing some default value to the result, here zero, otherwise error would occure – we are trying to list a variable that hasn’t been iniciated definitely.

This program would in this case work the same without else, but why ask further now that we already have a result.

Java Loops - for, while and do...while

There may be a situation when we need to execute a block of code several number of times, and is often referred to as a loop.

A word cycle gives you already clue that something is going to repeat. If we want to do something 100 times in a program, we don’t want to write the same code over and over 100 times, but we will insert it into cycle.

Java has very flexible three looping mechanisms. You can use one of the following three loops:

- **while** Loop
- **do...while** Loop
- **for** Loop
- As of Java 5, the *enhanced for loop* was introduced. This is mainly used for Arrays.

The while Loop:

A while loop is a control structure that allows you to repeat a task a certain number of times.

Syntax:

The syntax of a while loop is:

```
while(Boolean_expression)
```

```
{
    //Statements
}
```

We can demonstrate a while cycle by a flowchart like so.

When executing, if the *boolean_expression* result is true, then the actions inside the loop will be executed. This will continue as long as the expression result is true.

Here, key point of the *while* loop is that the loop might not ever run. When the expression is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example:

```
public class Test {

   public static void main(String args[]) {
      int x = 10;

      while( x < 20 ) {
```

System.out.print("value of x

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
```

```
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

Another example

We can improve our calculator example by repeating the calculation for so long until we input the “end”.

We will adjust beginning like so:

```
Scanner sc = new Scanner(System.in,
      "Windows-1250");
System.out.println("Welcome to
calculator.");
String pokracovat = "yes";
```

```
while (pokracovat.equals("yes"))
{
….. here the original code of the
calculator will continue …..
```

and we will change the end like so::

```
System.out.println("Do you wish to input
another example? [yes/no]");
        continue = sc.nextLine();
}  // the end of while cycle

    System.out.println("Thank you for
using "+
    "the calculator.");
```

The do...while Loop:

A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Syntax:

The syntax of a do...while loop is:

```
do
{
    //Statements
}while(Boolean_expression);
```

We can demonstrate a do..while loop by a flowchart like so.

Notice that the Boolean expression appears at the end of the loop, so the statements in the loop execute once before the Boolean is tested.

If the Boolean expression is true, the flow of control jumps back up to do, and the statements in the loop execute again. This process repeats until the Boolean expression is false.

Good advice

Good advice: Always write the **while** word on the same line as closing bracket **}**. If you wrote it on a separate line, you may start to wonder after some time, why is there a while cycle with no body and no statements.

Example:

```
public class Test {

   public static void main(String args[]){
      int x = 10;

      do{
```

System.out.print("value of x

value of x

A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

A for loop is useful when you know how many times a task is to be repeated.

Syntax:

The syntax of a for loop is:

```
for(initialization; Boolean_expression;
update)
{
    //Statements
}
```

Here is the flow chart of a for loop:

The initialization step is executed first, and only once. This step allows you to declare and initialize any loop control variables.

You are not required to put a statement here, as long as a semicolon appears.

Next, the Boolean expression is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement past the for loop.

After the body of the for loop executes, the flow of control jumps back up to the update statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the Boolean expression.

The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then update step, then Boolean expression). After the Boolean expression is false, the for loop terminates.

Example:

```
public class Test {

   public static void main(String args[]) {

      for(int x = 10; x < 20; x = x+1) {
         System.out.print("value of x : " + x );
         System.out.print("\n");
      }
```

```
    }
}
```

This would produce the following result:

value of xCycle will run 10 times, at the beginning there is 10 in x variable, cycle outpust the value and increments x variable by one. Then runs in the same fashion with different numbers.

As soon as there is twenty in 3, the condition `x<20` becomes false and the cycle ends.

Same rules as for conditions apply for skipping the curly brackets. They needn't to be there in this case, because the cycle executes one statement only.

Nested *for* Cycles

If you want to nest two or multiple cycles, nothing is in your way:

```
   System.out.println("Easy multiplication"
   +" using two nested cycles:");
for (int j = 1; j <= 10; j++)
{
        for (int i = 1; i <= 10; i++)

System.out.printf("%d*%d=%d",
          i, j, i*j);
        System.out.println();  //new line
}
```

A multiplication table will be the output.

Another *for* example

We will write a program that will be capable of calculation any (integral) value of an integer.

We will get a^n by multiplying n-1 times number **a** with number **a**.

```
Scanner sc = new Scanner(System.in,
"Windows-1250");

System.out.println("Powerer");
System.out.println("==========");
System.out.println("Input basis powers: ");
int a = Integer.parseInt(sc.nextLine());
System.out.println("Input exponent: ");
int n = Integer.parseInt(sc.nextLine());
```

```
int vysledek = a;
for (int i = 0; i < (n - 1); i++)
        result = result * a;

System.out.printf("Result: %d", result);
System.out.println("Thank you for using
Powerer.");
```

Deterrent example:

// this code is wrong

```
for (int i = 1; i <= 10; i++)
        i = 1;
```

We can see that the program is stuck.

Enhanced *for* loop in Java

As of Java 5, the enhanced for loop was introduced. This is mainly used for arrays.

Syntax:

The syntax of enhanced for loop is:

```
for(declaration : expression)
{
   //Statements
}
```

Declaration: The newly declared block variable, which is of a type compatible with the elements of the array you are

accessing. The variable will be available within the *for* block and its value would be the same as the current array element.

Expression: This evaluates to the array you need to loop through. The expression can be an array variable or method call that returns an array.

Example:

```
public class Test {
   public static void main(String args[]){
      int [] numbers = {10,20,30,40,50};
      for(int x : numbers ){
         System.out.print( x );
         System.out.print(",");
      }
```

```
        System.out.print("\n");
        String [] names =
       {"James", "Larry", "Tom", "Lacy"};
        for( String name : names ) {
            System.out.print( name );
            System.out.print(",");
        }
    }
}
```

This would produce the following result:

```
10,20,30,40,50,
James,Larry,Tom,Lacy,
```

The *break* Keyword

The *break* keyword is used to stop the entire loop. The *break* keyword must be used inside any loop or a *switch* statement.

The *break* keyword will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax:

The syntax of a *break* is a single statement inside any loop:

```
break;
```

Example:

```
public class Test {

```

```
   public static void main(String args[]) {
      int [] numbers = {10, 20, 30, 40,
50};

      for(int x : numbers ) {
         if( x == 30 ) {
        break;
         }
         System.out.print( x );
         System.out.print("\n");
      }
   }
}
```

This would produce the following result:

```
10
```

The *continue* Keyword:

The *continue* keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loop.

In a for loop, the *continue* keyword causes flow of control to immediately jump to the update statement.

In a while loop or do/while loop, flow of control immediately jumps to the Boolean expression.

Syntax:

The syntax of a *continue* is a single statement inside any loop:

```
continue;
```

Example:

```
public class Test {

   public static void main(String args[]) {
      int [] numbers = {10, 20, 30, 40,
50};

      for(int x : numbers ) {
         if( x == 30 ) {
         continue;
         }
         System.out.print( x );
         System.out.print("\n");
      }
```

```
    }
}
```

This would produce the following result:

```
10
20
40
50
```

Java - Numbers Class

Normally, when we work with Numbers, we use **primitive data types** such as byte, int, long, double, etc.

Example:

```
int i = 5000;
```

```
float gpa = 13.65;
byte mask = 0xaf;
```

However, in development, we come across situations where we need to use objects instead of primitive data types. In order to achieve this Java provides **wrapper classes** for each primitive data type.

All the wrapper classes (Integer, Long, Byte, Double, Float, Short) are subclasses of the abstract class Number.

This wrapping is taken care of by the compiler, the process is called boxing. So when a primitive is used when an object is required, the compiler **boxes** the primitive type in its wrapper

class. Similarly, the compiler **unboxes** the object to a primitive as well. The **Number** is part of the *java.lang* package.

Here is an example of boxing and unboxing:

```
public class Test{

   public static void main(String args[]){
   // boxes int to an Integer object
         Integer x = 5;

   // unboxes the Integer to a int
         x =  x + 10;
   System.out.println(x);
    }
}
```

This would produce the following result:

```
15
```

When x is assigned integer values, the compiler boxes the integer because x is integer objects. Later, x is unboxed so that they can be added as integers.

Number Methods:

Here is the list of the instance methods that all the subclasses of the Number class implement:

SN	Methods with Description
1	**xxxValue()** Converts the value of *this* Number object to the xxx data type and returned it.

2	**compareTo()** Compares *this* Number object to the argument.
3	**equals()** Determines whether *this* number object is equal to the argument.
4	**valueOf()** Returns an Integer object holding the value of the specified primitive.
5	**toString()** Returns a String object representing the value of specified int or Integer.
6	**parseInt()** This method is used to get the primitive data type of a certain String.

7	**abs()** Returns the absolute value of the argument.
8	**ceil()** Returns the smallest integer that is greater than or equal to the argument. Returned as a double.
9	**floor()** Returns the largest integer that is less than or equal to the argument. Returned as a double.
10	**rint()** Returns the integer that is closest in value to the argument. Returned as a double.
11	**round()** Returns the closest long or int, as indicated by the method's return type, to the argument.

12	**min()** Returns the smaller of the two arguments.
13	**max()** Returns the larger of the two arguments.
14	**exp()** Returns the base of the natural logarithms, e, to the power of the argument.
15	**log()** Returns the natural logarithm of the argument.
16	**pow()** Returns the value of the first argument raised to the power of the second argument.
17	**sqrt()** Returns the square root of the argument.

18	**sin()** Returns the sine of the specified double value.
19	**cos()** Returns the cosine of the specified double value.
20	**tan()** Returns the tangent of the specified double value.
21	**asin()** Returns the arcsine of the specified double value.
22	**acos()** Returns the arccosine of the specified double value.
23	**atan()** Returns the arctangent of the specified double value.
24	**atan2()** Converts rectangular coordinates (x, y) to polar coordinate

	(r, theta) and returns theta.
25	**toDegrees()** Converts the argument to degrees
26	**toRadians()** Converts the argument to radians.
27	**random()** Returns a random number.

Java - Character Class

Normally, when we work with characters, we use primitive data types char.

Example:

```
char ch = 'a';

// Unicode for uppercase Greek omega
character
char uniChar = '\u039A';

// an array of chars
char[] charArray ={'a', 'b', 'c', 'd',
'e'};
```

However in development, we come across situations where we need to use objects instead of primitive data types. In order to achieve this, Java provides wrapper class **Character** for primitive data type char.

The Character class offers a number of useful class (i.e., static) methods for manipulating characters. You can create a Character object with the Character constructor:

```
Character ch = new Character('a');
```

The Java compiler will also create a Character object for you under some circumstances. For example, if you pass a primitive char into a method that expects an object, the compiler automatically converts the char to a Character for you. This

feature is called autoboxing or unboxing, if the conversion goes the other way.

Example:

```
// Here following primitive char 'a'
// is boxed into the Character object c
Character ch = 'a';

// Here primitive 'x' is boxed for method
// test, return is unboxed to char 'c'

char c = test('x');
```

Escape Sequences:

A character preceded by a backslash (\) is an *escape sequence* and has special meaning to the compiler.

The newline character (\n) has been used frequently in this lecture in System.out.println() statements to advance to the next line after the string is printed.

Following table shows the Java escape sequences:

Escape Sequence	Description
\t	Inserts a tab in the text at this point.
\b	Inserts a backspace in the text at this point.
\n	Inserts a newline in the text at this point.

\r	Inserts a carriage return in the text at this point.
\f	Inserts a form feed in the text at this point.
\'	Inserts a single quote character in the text at this point.
\"	Inserts a double quote character in the text at this point.
****	Inserts a backslash character in the text at this point.

When an escape sequence is encountered in a print statement, the compiler interprets it accordingly.

Example:

If you want to put quotes within quotes you must use the escape sequence, \", on the interior quotes:

```
public class Test {

   public static void main(String args[]) {
      System.out.println(
   "She said \"Hello!\" to me.");
   }
}
```

This would produce the following result:

```
She said "Hello!" to me.
```

Character Methods

Here is the list of some important instance methods that all the subclasses of the Character class implement:

SN	Methods with Description
1	**isLetter()** Determines whether the specified char value is a letter.
2	**isDigit()** Determines whether the specified char value is a digit.
3	**isWhitespace()** Determines whether the specified char value is white space.
4	**isUpperCase()** Determines whether the specified char value is uppercase.
5	**isLowerCase()**

	Determines whether the specified char value is lowercase.
6	**toUpperCase()** Returns the uppercase form of the specified char value.
7	**toLowerCase()** Returns the lowercase form of the specified char value.
8	**toString()** Returns a String object representing the specified character valuethat is, a one-character string.

Java - String Class

Strings, which are widely used in Java programming, are a sequence of characters. In the Java programming language, strings are **objects**.

The Java platform provides the String class to create and manipulate strings.

Creating Strings:

The most direct way to create a string is to write:

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

Whenever it encounters a string literal in your code, the compiler creates a String object with its value in this case, "Hello world!'.

As with any other object, you can create String objects by using the new keyword and a constructor. The String class has eleven constructors that allow you to provide the initial value of the string using different sources, such as an array of characters.

```
public class StringDemo{

   public static void main(String args[]){
      char[] helloArray =
     { 'h', 'e', 'l', 'l', 'o', '.'};
      String helloString =
```

```
      new String(helloArray);
        System.out.println( helloString );
    }
}
```

This would produce the following result:

```
hello.
```

Note: *The String class is immutable, so that once it is created a String object cannot be changed. If there is a necessity to make a lot of modifications to Strings of characters, then you should use String Buffer & String Builder Classes.*

String Length:

Methods used to obtain information about an object are known as accessor methods. One accessor method that you can use

with strings is the *length()* method, which returns the number of characters contained in the string object.

After the following two lines of code have been executed, len equals 17:

```
public class StringDemo {

   public static void main(String args[]) {
      String palindrome =
     "Dot saw I was Tod";
      int len = palindrome.length();
      System.out.println
     ("String Length is : " + len);
   }
}
```

This would produce the following result:

```
String Length is : 17
```

Concatenating Strings:

The String class includes a method for concatenating two strings:

```
string1.concat(string2);
```

This returns a new string that is *string1* with *string2* added to it at the end. You can also use the *concat()* method with string literals, as in:

```
"My name is ".concat("Zara");
```

Strings are more commonly concatenated with the **+** operator, as in:

```
"Hello," + " world" + "!"
```

which results in:

```
"Hello, world!"
```

Let us look at the following example:

```
public class StringDemo {

   public static void main(String args[]) {
      String string1 = "saw I was ";
      System.out.println("Dot "
   + string1 + "Tod");
   }
}
```

This would produce the following result:

```
Dot saw I was Tod
```

Creating Format Strings:

You have *printf()* and *format()* methods to print output with formatted numbers. The String class has an equivalent class method, *format()*, that returns a String object rather than a PrintStream object.

Using String's static *format()* method allows you to create a formatted string that you can reuse, as opposed to a one-time *print()* statement. For example, instead of:

```
System.out.printf(
  "The value of the float variable is " +
  "%f, while the value of the integer " +
  "variable is %d, and the string " +
```

```
  "is %s", floatVar, intVar, stringVar);
```

you can write:

```
String fs;
fs = String.format(
  "The value of the float variable is " +
  "%f, while the value of the integer " +
  "variable is %d, and the string " +
  "is %s", floatVar, intVar, stringVar);
System.out.println(fs);
System.out.println(fs);
System.out.println(fs);
```

Special Characters and Escaping

We can see that a string can contain special characters that are preceeded by backslash “\”. It is especially the character /n that causes new line anywhere in the text, and /t where tabelator is needed. Let’s try it out.

```
System.out.println("First line\nSecond line");
```

Character “\” denotes some special character sequence in a string and it is furthermore used to write a unicode characters, such as “\uxxxx”, where xxxx is a character code.

We may encouter a problem in a moment we try to type “\” itself, we must escape it.

```
System.out.println("This is standard backslash: \\");
```

We may escape for example quote in the same fashion, so that Java doesn’t interpret it as an end of a string.

```
System.out.println("This is quote: \"");
```

Console and array inputs in window applications can naturally escape themselves so that user cannot input \n etc. In code it is legal and the programmer must keep it in mind.

String Methods

Here is the list of methods supported by String class:

SN	Methods with Description
1	**char charAt(int index)** Returns the character at the specified index.
2	**int compareTo(Object o)** Compares this String to another Object.
3	**int compareTo(String anotherString)** Compares two strings lexicographically.
4	**int compareToIgnoreCase(String str)** Compares two strings lexicographically, ignoring case differences.
5	**String concat(String str)** Concatenates the specified string to the end of this string.
6	**boolean contentEquals(StringBuffer sb)** Returns true if and only if this String represents the same

	sequence of characters as the specified StringBuffer.
7	**static String copyValueOf(char[] data)** Returns a String that represents the character sequence in the array specified.
8	**static String copyValueOf(char[] data, int offset, int count)** Returns a String that represents the character sequence in the array specified, count characters from given offset.
9	**boolean endsWith(String suffix)** Tests if this string ends with the specified suffix.
10	**boolean equals(Object anObject)** Compares this string to the specified object.
11	**boolean equalsIgnoreCase(String anotherString)** Compares this String to another String, ignoring case considerations.

12	**byte[] getBytes()** Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.
13	**byte[] getBytes(String charsetName)** Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array.
14	**void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)** Copies characters from this string into the destination character array.
15	**int hashCode()** Returns a hash code for this string.
16	**int indexOf(int ch)**

	Returns the index within this string of the first occurrence of the specified character.
17	**int indexOf(int ch, int fromIndex)** Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.
18	**int indexOf(String str)** Returns the index within this string of the first occurrence of the specified substring.
19	**int indexOf(String str, int fromIndex)** Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
20	**String intern()** Returns a canonical representation for the string object.

21	**int lastIndexOf(int ch)** Returns the index within this string of the last occurrence of the specified character.
22	**int lastIndexOf(int ch, int fromIndex)** Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.
23	**int lastIndexOf(String str)** Returns the index within this string of the rightmost occurrence of the specified substring.
24	**int lastIndexOf(String str, int fromIndex)** Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.

25	**int length()** Returns the length of this string.
26	**boolean matches(String regex)** Tells whether or not this string matches the given regular expression.
27	**boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)** Tests if two string regions are equal.
28	**boolean regionMatches(int toffset, String other, int ooffset, int len)** Tests if two string regions are equal.
29	**String replace(char oldChar, char newChar)** Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.

30	**String replaceAll(String regex, String replacement** Replaces each substring of this string that matches the given regular expression with the given replacement.
31	**String replaceFirst(String regex, String replacement)** Replaces the first substring of this string that matches the given regular expression with the given replacement.
32	**String[] split(String regex)** Splits this string around matches of the given regular expression.
33	**String[] split(String regex, int limit)** Splits this string around matches of the given regular expression.
34	**boolean startsWith(String prefix)** Tests if this string starts with the specified prefix.

35	**boolean startsWith(String prefix, int toffset)** Tests if this string starts with the specified prefix beginning a specified index.
36	**CharSequence subSequence(int beginIndex, int endIndex)** Returns a new character sequence that is a subsequence of this sequence.
37	**String substring(int beginIndex)** Returns a new string that is a substring of this string.
38	**String substring(int beginIndex, int endIndex)** Returns a new string that is a substring of this string.
39	**char[] toCharArray()** Converts this string to a new character array.
40	**String toLowerCase()** Converts all of the characters in this String to lower case

	using the rules of the default locale.
41	**String toLowerCase(Locale locale)** Converts all of the characters in this String to lower case using the rules of the given Locale.
42	**String toString()** This object (which is already a string!) is itself returned.
43	**String toUpperCase()** Converts all of the characters in this String to upper case using the rules of the default locale.
44	**String toUpperCase(Locale locale)** Converts all of the characters in this String to upper case using the rules of the given Locale.
45	**String trim()** Returns a copy of the string, with leading and trailing

	whitespace omitted.
46	**static String valueOf(primitive data type x)** Returns the string representation of the passed data type argument.

Java - String Buffer & String Builder Classes

The **StringBuffer** and **StringBuilder** classes are used when there is a necessity to make a lot of modifications to Strings of characters.

Unlike Strings objects of type *StringBuffer* and *StringBuilder* can be modified over and over again without leaving behind a lot of new unused objects.

The *StringBuilder* class was introduced as of Java 5 and the main difference between the *StringBuffer* and *StringBuilder* is that *StringBuilders* methods are not thread safe (=not synchronised).

It is recommended to use **StringBuilder** whenever possible because it is faster than StringBuffer. However if thread safety is necessary the best option is StringBuffer objects.

Example:

```
public class Test{
```

```
    public static void main(String args[]){
       StringBuffer sBuffer =
      new StringBuffer(" test");
       sBuffer.append(" String Buffer");
       System.ou.println(sBuffer);
   }
}
```

This would produce the following result:

```
test String Buffer
```

StringBuffer Methods

Here is the list of important methods supported by StringBuffer class:

SN	Methods with Description
1	**public StringBuffer append(String s)** Updates the value of the object that invoked the method. The method takes boolean, char, int, long, Strings etc.
2	**public StringBuffer reverse()** The method reverses the value of the StringBuffer object that invoked the method.
3	**public delete(int start, int end)** Deletes the string starting from start index until end index.
4	**public insert(int offset, int i)** This method inserts an string s at the position mentioned by offset.
5	**replace(int start, int end, String str)** This method replaces the characters in a substring of this

StringBuffer with characters in the specified String.

Here is the list of other methods (Except set methods) which are very similar to String class:

SN	Methods with Description
1	**int capacity()** Returns the current capacity of the String buffer.
2	**char charAt(int index)** The specified character of the sequence currently represented by the string buffer, as indicated by the index argument, is returned.
3	**void ensureCapacity(int minimumCapacity)** Ensures that the capacity of the buffer is at least equal to

	the specified minimum.
4	**void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)** Characters are copied from this string buffer into the destination character array dst.
5	**int indexOf(String str)** Returns the index within this string of the first occurrence of the specified substring.
6	**int indexOf(String str, int fromIndex)** Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
7	**int lastIndexOf(String str)** Returns the index within this string of the rightmost occurrence of the specified substring.

8	**int lastIndexOf(String str, int fromIndex)** Returns the index within this string of the last occurrence of the specified substring.
9	**int length()** Returns the length (character count) of this string buffer.
10	**void setCharAt(int index, char ch)** The character at the specified index of this string buffer is set to ch.
11	**void setLength(int newLength)** Sets the length of this String buffer.
12	**CharSequence subSequence(int start, int end)** Returns a new character sequence that is a subsequence of this sequence.
13	**String substring(int start)**

	Returns a new String that contains a subsequence of characters currently contained in this StringBuffer.The substring begins at the specified index and extends to the end of the StringBuffer.
14	**String substring(int start, int end)** Returns a new String that contains a subsequence of characters currently contained in this StringBuffer.
15	**String toString()** Converts to a string representing the data in this string buffer.

Java – Arrays

In General

An array is used to store a large number of variables of the same type.

We can visualize an array as a line of boxes, each containg a single element. The boxes are numbered, first has an index of 0.

Programming languages vary greatly at how they work with arrays.

In some languages (especially old fashioned, compiled ones) it wasn't possible to create an array with dynamic size (ie. to change its length according to some variable) at runtime.

This was bypassed by so called pointers, specific data structures, which often lead to errors with manual memory management and to unstable program (ie, in C++).

Conversely, some interpreted languages allow not only to declare a dynamic length array, but even to change this length on existing array (i.e. PHP).

We know that Java is a virtual machine, something between a compiler and interpreter.

So we can declare an array with a size that will become known during a runtime, **but we cannot change the size of an existing array.**

Array is simple. It is fast to work with, because the elements are lined up in a buffer, they all occupy equal space and they are accessed quickly.

Many Java's internal functions there fore work with array in some fashion or return an array. It is a key structure.

Arrays in JAVA

Java provides a data structure, the **array**, which stores a fixed-size sequential collection of elements of the same type. An array

is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as *number0*, *number1*, ..., and *number99*, you declare one array variable such as *numbers* and use *numbers[0]*, *numbers[1]*, and ..., *numbers[99]* to represent individual variables.

Declaring Array Variables

To use an array in a program, you must declare a variable to reference the array, and you must specify the type of array the variable can reference. Here is the syntax for declaring an array variable:

```
dataType[] arrayRefVar;
// preferred way.
```

or

```
dataType arrayRefVar[];
// works, but not preferred way.
```

Note: *The style dataType[ ] arrayRefVar is preferred. The style dataType arrayRefVar[ ] comes from the C/C++ language and was adopted in Java to accommodate C/C++ programmers.*

Example:

The following code snippets are examples of this syntax:

```
double[] myList;
// preferred way.
```

or

```
double myList[];
// works, but not preferred way.
```

Creating Arrays

You can create an array by using the new operator with the following syntax:

```
arrayRefVar = new dataType[arraySize];
```

The above statement does two things:

- It creates an array using *new dataType[arraySize];*
- It assigns the reference of the newly created array to the variable *arrayRefVar*.

Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below:

```
dataType[] arrayRefVar = new
dataType[arraySize];
```

Alternatively you can create arrays as follows:

```
dataType[] arrayRefVar = {value0, value1,
..., valuek};
```

The array elements are accessed through the **index**. Array indices are 0-based; that is, they start from 0 to **arrayRefVar.length-1**.

Example:

Following statement declares an array variable, myList, creates an array of 10 elements of double type and assigns its reference to myList:

```
double[] myList = new double[10];
```

Figure represents array *myList*. Here, *myList* holds ten double values and the indices are from 0 to 9.

Processing Arrays

We can access elements of an array using brackets, so if we want to save under first available index (therefore index 0) a number 1, we will write it down like so:

```
int[] array = new int[10];
array [0] = 1;
```

When processing array elements, we often use either *for* loop or *foreach* loop because all of the elements in an array are of the same type and the size of the array is known.

Example:

Here is a complete example of showing how to create, initialize and process arrays:

```
public class TestArray {

   public static void main(String[] args) {
      double[] myList = {1.9,2.9,3.4,3.5};

      // Print all the array elements
      for (int i=0;i<myList.length;i++) {
         System.out.println(myList[i]
     + " ");
      }
```

```
    // Summing all elements
    double total = 0;
    for (int i=0;i<myList.length;i++) {
      total += myList[i];
    }
    System.out.println("Total is "
    + total);

    // Finding the largest element
    double max = myList[0];
    for (int i=1;i<myList.length;i++) {
      if (myList[i]>max) max=myList[i];
    }
    System.out.println("Max is " + max);
  }
```

```
}
```

This would produce the following result:

```
1.9
2.9
3.4
3.5
Total is 11.7
Max is 3.5
```

Manual array brakes

An array can also be filled “manually”, we don’t have to type in index by index. We can use curly brackets and seprarate the items by commas:

```
String[] simpsons = {"Homer", "Marge",
"Bart", "Lisa", "Meggie"};
```

Numbered and other arrays can also be filled “manually”.

The foreach Loops:

JDK 1.5 introduced a new for loop known as foreach loop or enhanced for loop, which enables you to traverse the complete array sequentially without using an index variable.

Example:

The following code displays all the elements in the array myList:

```
public class TestArray {

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

```
      double[] myList = {1.9,2.9,3.4,3.5};

      // Print all the array elements
      for (double element: myList) {
         System.out.println(element);
      }
   }
}
```

This would produce the following result:

```
1.9
2.9
3.4
3.5
```

Passing Arrays to Methods

Just as you can pass primitive type values to methods, you can also pass arrays to methods. For example, the following method displays the elements in an int array:

```
public static void printArray(int[] array)
{
  for (int i = 0; i < array.length; i++) {
    System.out.print(array[i] + " ");
  }
}
```

You can invoke it by passing an array as well. For example, the following statement invokes the printArray method to display 3, 1, 2, 6, 4, and 2:

```
printArray(new int[]{3, 1, 2, 6, 4, 2});
```

Returning an Array from a Method

A method may also return an array. For example, the method shown below returns an array that is the reversal of another array:

```
public static int[] reverse(int[] list) {
  int[] result = new int[list.length];
```

```
  for (int i=0,j=result.length-1;
       i<list.length;i++,j--) {
    result[j] = list[i];
  }
  return result;
}
```

The Arrays Class

Similarly to primitive types, arrays also have their wrapping class **Arrays**. The *java.util.Arrays* class contains various static methods for sorting and searching arrays, comparing arrays, and filling array elements. These methods are overloaded for all primitive types.

SN	Methods with Description
1	**public static int binarySearch(Object[] a, Object key)** Searches the specified array of Object (Byte, Int , double, etc.) for the specified value using the binary search algorithm. The array must be sorted prior to making this call. This returns index of the search key, if it is contained in the list; otherwise, (-(insertion point + 1).
2	**public static boolean equals(long[] a, long[] a2)** Returns true if the two specified arrays of longs are equal to one another. Two arrays are considered equal if both arrays contain the same number of elements, and all corresponding pairs of elements in the two arrays are equal. This returns true if the two arrays are equal. Same method could be used by all other primitive data types (Byte, short,

	Int, etc.)
3	**public static void fill(int[] a, int val)** Assigns the specified int value to each element of the specified array of ints. Same method could be used by all other primitive data types (Byte, short, Int etc.)
4	**public static void sort(Object[] a)** Sorts the specified array of objects into ascending order, according to the natural ordering of its elements. Same method could be used by all other primitive data types (Byte, short, Int, etc.)

Example

We will allow a user to input name x and then we will check whether he belongs to the Simpson family.

```
Scanner sc = new Scanner(System.in,
   "Windows-1250");

String[] simpsons =
   {"Homer", "Marge", "Bart", "Lisa",
"Meggie"};

System.out.println(
```

```
   "Input a name
   from a Simpson family): ");
String x = sc.nextLine();

Arrays.sort(simpsons);

int position = Arrays.binarySearch(
   simpsons, x);

if (position >= 0)
        System.out.println(
    "It is a Simpson!");
else
        System.out.println(
    "It is not a Simpson!");
```

Arrays with variable length

We have already mentioned that we can define an array length during a runtime. Let's try it in an example where we will calculate an average from variable amount of numbers.

By that we naturally mean that we can input any (positive) amount of numbers, and once we inputed it we cannot change it any more.

Example

```
Scanner sc = new Scanner(System.in,
   "Windows-1250");
```

```
System.out.println("
   Hellow, I will calculate your GPA.
   How many grades will you input?");

int number =
   Integer.parseInt
      (sc.nextLine());

int[]numbers
   = new int[number];

for (int i=0;i<number; i++){
    System.out.printf("Input %d. number: ",
    i + 1);
```

```
        numbers [i] =
Integer.parseInt(sc.nextLine());
}

// calculation of average
int sum = 0;
for (int i: numbers)
        sum += i;
float average = sum / (float)
numbers.length;

System.out.printf("Your GPA is: %f",
average);
```

This example could have been written even without using an array, but how about if we wanted to calculate a median? Or list

all the numbers backwards? That wouldn't be possible without using an array.

In this fashion we have original values still in array and we can work easily and indefinitely.

Beware!

At average calculation notice that during division before one of the operand there is stated **(float)**.

We state by it that we want to devide not integrally.

Otherwise the mathematic oparations proceed as standard numerict types match. So by deviding 2/3 we get the result 1,

while during 3 / 2.0F we get the result 1.5. The principle is the same at this point.

Simple examples

We will demonstrate several simple examples to excercise strings and arrays.

Example - substring

It will return a substring from a given position till the string end. We can give another parameter, which is a string length.

```
System.out.println("Kdo se směje naposled, ten je admin.".substring(13, 21));
```

Output:

Example – CompareTo

Allowes you to compare two strings alphabetically. Returns -1 if string alfabetically preceeds the one in parameter, 0 if both strings are equal and +1 if the string in parameter alphabetically preceeds given string.

```
System.out.println("acacia".compareTo("morn
ingstar"));
```

Output:

```
-1
```

Examle – Character Coding, ASCII table

Character Coding

In era of MS DOS operating system there was basically no way to register text but in a form of ASCII table.

Individual characters were saved as numbers of byte type, so with range of values from 0 to 255.

ASCII table had also 255 characters and to each ASCII code (numerical code) one character was assigned.

The main advantage was that the characters are saved one after one in a table, alphabetically. Ie. on position 97 we find “a”, 98 “b” and like.With numbers it is similar.

Diacritical characters are unfortunately broken.

Main drawback is that all the characters of all national alphabets simply didn't fit into the table.

That is why unicode (UTF8) encoding is used where are the characters represented in a slightely different way.

But we still have possibility to work with ASCII values of individual characters in Java.

Numeric value of characters

Let's now try to convert a characters into it's ASCII value and vice versa, to create a given character from it's ASCII value.

We need a **type conversion** for the task.

Type conversion, ie **(int)c** means that the compilator is supposed to view **c**, despite it being a number, as a character.

Similarly, **(char)i** is a character.

```
char c;
// character
int i;
// ordinary (ASCII)
// character value

// We will convert a character
// into it's ASCII value
c = 'a';
i = (int)c;
System.out.printf(
```

```
"We converted the %c character into it's
ASCII value. %d\n", c, i);

// We will convert ASCII value
// into a character
i = 98;
c = (char)i;
System.out.printf(
"We converted an ASCII value of %d into a
character %c", i, c);
```

Example - Caesar Cipher

We will create a simple program for a text encryption. If you ever heard of Ceasar cipher, that is what we are going to program.

Text encryption is based on shifting an alphabetic character by certain fixed amount of characters. Ie the word “hi” with the shift of 1 will be translated to “ij”. We will allow the user to choose a shift.

We will need variables for original text, encrypted text and shift. We will aslo need a loop iterating over individual characters and encrypted message output.

We will hardcode the message into the code so that we don’t have to rewrite it during every execution of a program. Who wants to is welcome to input the message text each time using **sc.nextLine()**.

The cipher is not national character based, doesn’t consider spaces and national characters.

We will ban the national characters and we will suppose that the user won’t input it. In an ideal scenario we would remove the national characters before encryption, along with anything other than characters.

The program scaffolding will look like this:

```
// variable inicialiation
String s =
"blackholesareeverywherewheregoddividedbyzero
";
```

```
System.out.printf("Original message: %s\n",
s);
String message = "";
int shift = 1;

// Loop iterating over individual
characters

for (char c : s.toCharArray())
{
   // processing

}

// output
```

```
System.out.printf("Encrypted message
: %s\n", message
);
```

Now we will move to the inner loop. We will convert c character into it's ASCII value (or ordinary value), we will increment this value by the shift and we will convert it back to the character. We will append the character to the end of the message.

```
int i = (int)c;
i += shift;
char character = (char)i;
message += character;
```

There is one weak link. Let's try to input higher shift or type a word "zebra".We can see that can "overflow" after into ASCII values of other characters.

We will create a loop of characters so that shift would go smoothly from "z" to "z" and so on. We will do it using nothing more than % operator (modulus), remainder after division.

The code will now look like this:

```
int i = (int)c;
i = (shift+i) mod 26;
char character = (char)i;
message += character;
```

Example – Split, Morse Code

Let's first prepare a scaffolding for our program. We will need 2 strins with message, one in Morse code, second empty for now. We will save our results in it. Furthermore we will need some exemplar of characters, just like in the example with vowels. And we of course need an exemplar of characters and characters in Morse code. We can save the numbers into a simple String since they are just a single character. Morse code characters consist of multiple characters, we must input them into an array. Our program structure should look like this:

```
// A string we want to decode
```

```
String s = ".. ... .-.. .- -. -.. ... ---
..-. -";
System.out.printf("Original message: %s\n",
s);
// string with decoded message
String message = "";

// exemplar arrays
String alphabeticalCharacters =
"abcdefghijklmnopqrstuvwxyz";
String[]morseCodeCharacters = {".-", "-
...", "-.-.", "-..", ".", "..-.", "--.",
"....",
```

```
".." , ".---", "-.-", ".-..", "--", "-.", "-
--", ".--.", "--.-", ".-.", "...", "-",
"..-",
"...-", ".--", "-..-", "-.--", "--.."};
```

You can add other characters such as numbers and punctuation marks later on. We will skip them here. Now we will split a string using **split()** method into an array of strings, containing individual morse code characters. We will split by the space character. We will iterate through the array with **foreach** loop later on.

```
// will break a string into Morse code
characters
String[]characters = s.split(" ");
```

```
// iteration over Morse code characters
for (String morseCodeCharacter :
characters)
{

}
```

In an ideal case scenario we should get to grips with cases when user inputs ie. multiple white spaces (users love doing that). *Split()* will then create one more string in an array that will be blank. We should detect and ignore it in the loop, but we will not worry about it in a tutorial.

We will try to search aplphabetically for a Morse character in a *morseCodeCharacters* array. We will be interested in it's index,

because when we look into the same index of *alphabeticalCharacters* array, we will find a corresponding character there. The reason being that both arrays are sorted alphabetically, of course. We will insert the following code int the loop body:

```
char alphabeticalCharacter = '?';

int index = -1;
for (int i = 0; i <
morseCodeCharacters.length; i++)
{
        if (morseCodeCharacters
[i].equals(morseCodeCharacter))
                index = i;
```

```
}

if (index >= 0) // character found
        alphabeticalCharacter =
alphabeticalCharacters.charAt(index);
message += alphabeticalCharacter;
```

The code will at first save ‘?’ into an alphabetical character, because the situation can arise that we don’t have the character in our set. In a following step we must search for it’s index. Java array unfortunatelly doesn’t have *indexOf()* method and I don’t want to consider more complicated data structures yet. So we will code the search of index ourselves, it is quite simple.

First we will set index to -1 since we don’t know if the array contains given String (Morse character). In the following step we will compare individual Strings to our string (character). We already know that we must use equals() method to compare two strings. If the strings match, we will save the current index.

If we found the character (index >= 0), we will assign alphabeticalCharacter from alphabetical characters under this index. Finally we will adjoin to the message the += operator that replaces message = message + alphabeticalCharacter.

And finally we will output the message:

```
System.out.printf("Decoded message: %s\n",
message);
```

Class Math

Basic mathematical functions are contained in class Math in Java. We will import it by an instruction

```
import java.lang.Math;
```

Class provides two basic constants: PI and E. Pi is understandably the number π (3.1415...) and E is Euler's number, or the base of natural logarithm (2.7182...). It is probably clear how to work with the class, but to make sure we will output the constants into a console:

```
System.out.printf("Pi: %f \ne: %f",
Math.PI, Math.E);
```

We can see that we do all the calls on Math class. There is nothing interesting about the code besides using the special character \n in a string which calls a new line.

min, max

Both functions thake two numbers of any data type as a parameter. Function min() will return the smaller one, function max() the larger one.

round, ceil, floor

All three functions deal with rounding. Round() takes real number and rounded double number in the same way we know

from school (from 0.5 upwards, rest downwards). Ceil() will always round upwards and floor() always downwards.

We will need Round() ofter, I used other functions in practice ie. for detection of number of pages in commentary list in a guest book. If we have 33 replies and 10 listed on a page, they will take up 3.3 pages. We must round the result upwards because there will be 4 real pages.

abs a signum

Both methods take any number of any type as a parameter; abs() will return it's absolute value and singum() will return -1, 0 or 1 (for negative number, zero and positive number) depending on sign.

sin, cos, tan

Classical goniometric functions. They take as a parameter an angle of type double, which are considered in radians, not in degrees. For conversion from degrees to radians we will multiply the degrees by * (Math.PI/180).

acos, asin, atan

Classical cytrometric (arcus) functions once again will return given angle. The parameter will be a value in Double, output the ange in radians (aslo Double). If we wish to have an angle in degrees we divide radians by / (180 / Math.PI).

pow a sqrt

Pow takes in two parameters of Double type, first is base of power, and the second is exponent. So if we wanted to calculate say 2^3, the code would be following:

```
System.out.println(Math.pow(2, 3));
```

Sqrt is an abbreviation of square root from a given double number. Both functions return double number.

exp, log, log10

Exp returns Euler's number exponentiated to the given exponent. Log returns natural logarithm of a given number. Log10 returns decadic logarithm of a given number.

Arbitrary extraction is obviously missing in the list of methods. But we can calculate it from the Math functions at our disposal.

We know that 3 extraction from 8 = 8^(1/3). So we can type:

```
System.out.println(Math.pow(8, (1.0/3.0));
```

Beware! It is very important to type at least one number containing decimal point, otherwise Java will assume integral devision and the result will be 80 = 1 in this case.

Recursion

What is recursion?

Method calling itself is called *recursive*. So the recursion is when a method calls itself. Key component of recursive method is an instruction that calls itself.

What is recursion good for?

Recursion can be easily used for calculation of factorial. If you don't know what factorial is don't worry, we will explain it here. Factorial of N number is a multiplication of all integers between 1 and N. Factorial of 5 is 1 * 2 * 3 * 4 * 5 = 120. So the factorial of 5 is 120.

Without a recursion you would probably solve the following example using loop. Let's take a look how it can be solved without recursion first:

main class Program.java

```
public class Program {
    public static void main(String args[]){
        // I will create a new Factorial
object
        Factorial f = new Factorial ();
        // I will list factorial of number
5
        System.out.println(f. factorial
(5));
    }
```

```
}
```

Třída Factorial.java

```
public class Factorial {
    // I will create a public method that
returns Integer.
    public int factorial (int n){
        N is number we want to get
factorial of.
        int i, result = 1;
        // multiplication of all numbers
between 1 and N
        for(i = 1; i <= n;i++)
            result = result * i;
        // I will return factorial of
number N
```

```
        return result;
    }
}
```

As you probably noticed the public factorial method in Factorial class returned multiplication of all numbers from 1 to 5 (or N). Now let's do the same example over using recursion.

Main class Program.java is the same as before

```
public class Program {

    public static void main(String args[]){
        Factorial f = new Factorial();
        System.out.println(f.factorial(5));
    }
}
```

Class Factorial.java in which recursion is used.

```
public class Factorial {

    public int factorial(int n){
        int result;
        if(n == 1)
            return 1;
        result = factorial(n-1) * n;
        return result;
    }
}
```

As you can see, the method calls itself `result = factorial (n-1) * n;` and therefore it's true to say that

it is recursive. It is much easier than if we used the cycle. When calling the factorial of the argument one method returns one. Otherwise return the product *factorial(n-1) - n*. To evaluate this expression method is called *factorial()* with a value n-1. This process is repeated until n is equal to 1 and call methods begin to return. For example, when calling calculating the factorial of the number 2 cause the first method call *factorial()* performing a second call with an argument of 1. This call returns a value of 1, which is then multiplied by 2. The answer is 2.

Fibonacci sequence

Another showcase of recursion is Fibonacci sequence. It's complexity is O(2n).

```
public class Program {

    public static void main(String args[]){
        Fibonacci fib = new Fibonacci();

System.out.println(fib.fibonaci(5));
    }
}
```

Fibonacci.java

```
public class Fibonacci {

    public int fibonaci(int n)
    {
        if (n == 1) return 1;
```

```
        else return fibonaci(n-1) +
fibonaci(n-2);
    }
}
```

The program works, but it is very slow. Just a side note< another solution exist with complexity of o(2) exists.

```
public class Fibonacci {
    private List<Integer> list = new
ArrayList<Integer>();
    public int fibonaci(int n)
    {   list.add(n);
        if (n == 1) return 1;
        if(!list.contains(n-1)) return
fibonaci(n - 1);
```

```
        if(!list.contains(n-2)) return
fibonaci(n - 2);
        return 0;
    }
}
```

The meat of it was creating a list a saving partial results in it. But because we are not yet familiar with construct

```
List<Integer>
```

we will not deal with it yet, we will find out about the details in a chapter about collections.

Content