AMAZING JAVA:

LEARN JAVA QUICKLY!

JAVA: LEARN JAVA QUICKLY

By

ANDREI BESEDIN

Table of Contents

Table of Contents
All Rights Reserved
Chapter 1. Primitive Types
Chapter 2. Java Output
Chapter 3. String
Chapter 4. Java IO
Chapter 5. Objects
Chapter 6. Access Modifiers
Chapter 7. Static
Chapter 8. Collections
Chapter 9. Interfaces
Chapter 10. Design Patterns
Other Books By Andrei Besedin

Chapter 1. Primitive Types

Primitive types are most basic data types that are used in Java. They are stored in variables. Variable is nothing but reserved memory. Size of memory depends of primitive type. All primitive types have their default value. Sizes are:

Primitive type	Size	Minimum	Maximum
Boolean	1-bit	-	-
char	16-bit	Unicode 0	Unicode 2^{16}- 1
byte	8-bit	-128	+127
short	16-bit	-2^{15}	$+2^{15} - 1$
int	32-bit	-2^{31}	$+2^{31} - 1$
long	64-bit	-2^{63}	$+2^{63} - 1$
float	32-bit	IEEE754	IEEE754
double	64-bit	IEEE754	IEEE754

Declaration and initialization of primitive types

```
int a;
int a = 1;
int a,b;
int a = 2; int b = 2;
final int a = 15;//constant
```

Declaration means to give variable type along with name: **int**a; So, our variable is type of integer and have name a;Initialization is process of adding value to primitive type, in this case: **int**a = 0; Before this int have default value. For boolean default value is false, for char is '\u0000', for rest is zero (0).

Conversation of primitive types

Implicit

```
int a = 1;
double b;
b = a;
```

Explicit

```
long a = 4l;
int b = a;//mistake
int c = (int)a;
```

Note: This is called cast operator. Operators will be covered next.

OPERATORS

ARITHMETIC OPERATORS

Main operators are: +, -, *, /, %

y = 5;

Operator	Result
x = y + 1	y = 6
x = y – 1	y = 4
x = y % 2	y = 1
x = ++y	x = 6; y = 6
x = y++	x = 5; y = 6
x = --y	x = 4; y = 4

x = 6

y = 3

Operator	Same as	Result
x = y	x = 3	
x + = y	x = x + y	x = 9
x - = y	x = x – y	x = 3
x * = y	x = x * y	x = 18
x / = y	x = x / y	x = 2
x % = y	x = x % y	x = 0

Relational Operators

Main operators are: <, >, <=, >=, ==, !=

x = 3
y = 2

Operator	Result
x < y	false
x > y	true
x <= y	false
x >= y	true
x == y	false
x != y	true

Arithmetic operators are commonly used, as you guess, in math operations. Modulo (%) can be used to check if number is even or odd for example. Relational operators are very simple. ==and !=are common used for comparisons. Operator + can be used to concatenate two Strings also.

Logical Operators

Logical operators

Main operators are: && (logical AND), || (logical OR), ! (logicalNOT)

&&false true || false true!

falsefalse falsefalse falsetruefalsetrue

true false true true true truetruefalse

x = 2
y = 4

Operator	Result

((x< 1) && (y > 3)) false
((x< 5) || (y = 5)) true
!(x > y) true

Chapter 2. Java Output

Class System is used for printing to console some text. Let`s try it on most famous example of your first Java program, Hello World! To do this we need to set up our Java project. First open Eclipse. Creating project in Eclipse is very simple File ->New ->Java Project. Name project as FirstJavaProject and then click Finish.
In our project press right click on src folder and create a new package by New ->Package. Name package as first.pack.

Note: Packages are used for better organization of Java classes. In this case we build our own classes, but there are many build-in classes which need some packages to import (java.io, java.util etc).
All operations in Java are made in **classes**. To create them, right click on package New ->Class and name it as JavaClass. Check public static void main (String[]args) and click Finish.

Note: public static void main (String[]args) is method which will be explained later. For now, it is enough to know that in main method are called all other methods in your Java application. This and every other code is supposed to be putinside main method.
System.***out***.println("Hello World!");
Our program should look like this:

```
package first.pack;

public class JavaClass {

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

    }

}
```

Note: There is a one little trick to print this faster. Type syso, hold ctrl and press space. Then press Enter.

Ok, now this need somehow to be started. To do that pressRun ->Run As ->Java Application.
expected: Hello World! will be printed to console

We learned how to print some massages to console. Let`s now use what we already covered and make some mix with primitive types. Create a new class called PrintPrimitiveTypes and add main method.
Add following lines inside main method:

```
int a = 2;
double b = 3.14;
System.out.println("1. Old value of b = " +b);
b = a;
System.out.println("2. New value of b = " +b);
```

Run application.
expected: 1. Old value of b = 3.14 will be printed to console.
2. New value of b = 2.0
Note: Try it with other operators. For example:

```
int c = 4;
b = c * 3.14;
System.out.println("3. New value of b = " +b);
```

As you can see here is used + operator to chain text with integer. Try to make new examples on your own to feel more comfortable with code writing.

Control Flow Statements

IF

if(condition)
action
else
    another action

Now we will do some simple example of IF usage. Make a new class ControFlow and add following lines:

```
int a = 10;
if(a< 11) {
    System.out.println("A is less than 10");
}
else {
    System.out.println("A is not less than 10");
}
```

expected: A is less than 10 will be printed to console.

Let`s make it a little more difficult. Just like in a real life, you can provide more than one condition for any situation. In this case, it is made by adding else if statement. Add following lines to current classand run it :

```
int b = 15;
if (b == 13)
    System.out.println("b == 13");
elseif ((b<=17) && (b>= 12))
    System.out.println("b is less than or equal to 17 and greater than or equal to 12");
else
    System.out.println("b is not any from this");
```

expected: b is less than or equal to 17 and greater than or equal to 12 will be printed to console.

Program first checks if b == 13, it is not, then checks another condition which

is true and print it to console. If this condition was also false, then last one sentence will be printed. Try to make another condition on your own to see output. You can see that statement works fine also without curly braces.
Now we got idea how to make a simple program in Java. Let`s solve this problem. Imagine that you had test in your school. You got some score and grade. Try to make program, using if statement, to check which grade you got based on your score.
Note: First you need to declare two variables score and grade, like this:

```
intgrade;
intscore = 85;
```

Try it yourself firstly, think about conditions. Here is solution:

```
if (score>= 95)
        grade = 10;
elseif (score>= 85)
        grade = 9;
elseif (score>= 75)
        grade = 8;
elseif (score>= 65)
        grade = 7;
elseif (score>= 55)
        grade = 6;
else
        grade = 5;
System.out.println("Grade is: " + grade);
```

expected: Grade is: 9 will be printed to console

Let`s now see one simple example of code:

```
if (i< 10)
        a = i * 100;
else
        a = i * 10;
```

Now look at this code:

```
a = i<10 ?i * 100 : i * 10;
```

This is same thing made intwo different ways. Like if/else statement this line

of code tells to compile to check if i < 10, if it is true a = i * 100. If it is false, then multiply i with 10 (i * 10).
Ifis ,with for loop, most common used statement so it would be good to have as much as it possible practice.

SWITCH

```
switch(a) {
case '1': action
        break;
case '2': action
        break;
case '3': action
        break;
default:
         action
}
```

By looking at example above, try to write a program that will print season of year based on condition. Here is a solution:

```
intseason = 2;
switch (season) {
case 1: System.out.println("Spring");
        break;
case 2: System.out.println("Summer");
        break;
case 3: System.out.println("Autumn");
        break;
case 4: System.out.println("Winter");
        break;
default: System.out.println("Unknown season");
        break;
}
```

expected: Summer will be printed to console

Note: There are three types of print statements:

System.*out*.print("Massage");

```
System.out.println("Massage");//add a new line
String str = "World";
System.out.printf ("Hello %s", str);//placeholder for a String (%s), also can be for char, int, boolean, float
```

FOR

```
for (initialization; condition; increment/decrement) {
body
}
```

Now we will do some basic iteration from 1 to 10 with for loop. Here is a code:

```
for(int i = 1;i<= 10;i++) {
        System.out.print(i + " ");//try println to see difference
}
```

expected: 1 2 3 4 5 6 7 8 9 10will be printed to console

Ok, but what if we want to iterate in reverse way. In first example is used incrementation, so in second we need decrementation. Think what else need to be changed, is condition still i <= 10 ? Here is solution of problem:

```
for(int i = 10;i>= 1;i--) {
        System.out.print(i + " ");
}
```

expected: 10 9 8 7 6 5 4 3 2 1will be printed to console

FOR EACH

For each loops are used to iterate through arrays, what will be covered later. This is simpler type of loop because arrays have already fixed length and there is no need for that condition (i <= 10). Here is some basic example:

```
double[] array = {1.6, 2.8, 4.54};
for (double ar : array){
        System.out.print(ar + " ");
}
```

expected: 1.6 2.8 4.54 will be printed to console

Nested loops

We will now make things a little more complicated by using nested loops. Nested loops are loops inside another loops. It can be confusing for someone who is beginner, but when you start to understand concept of them you will realize how powerful they could be. Let`s start with this one example:

```
for(inti = 0; i< 10; i++){
        if (i==7){
        break;
        }
        if (i == 2)
        continue;
        System.out.print(i + " ");
}
```

expected: 0 1 3 4 5 6 will be printed to console

Difference between break and continue is clear to resolve from this example.
break – when loop reaches to 7 it stops and prints everything until that condition.
continue – when condition is met it stops and start another iteration.
Next peace of code will make this more advanced. Look at this:

```
intsumEven=0;
intsumOdd=0;
intcounterEven=1;
intcounterOdd=1;
for(inti = 1;i<= 100;i++){
if(i % 2 == 0){
         if(counterEven == 3){
        sumEven+=i;
counterEven = 1;
}elsecounterEven++;
        }else{
        If(counterOdd == 5){
        sumOdd+=i;
        counterOdd = 1;
        }elsecounterOdd++;
```

```
        }
}
System.out.println("Sum of even numbers is: " + sumEven + " , sum of odd numbers is: " + sumOdd);
```

expected: Sum of even numbers is: 816, sum of odd numbers is: 540 will be printed to console

Program will loop through first 100 numbers, check which one is even or odd and based on that condition will sum them.

WHILE

```
while (condition) {
        body
}
```

while loop is looping until condition is false. Let`s see this on example:

```
intn = 15;
inti = 5;
while (i<n){
        System.out.print(i + " ");
         i++;
}
```

expected: 5 6 7 8 9 10 11 12 13 14 will be printed to console

So, it starts with 5 (**int**i = 5;) and goes through while loops every time until condition i < n (**int**n = 15;) is false. i++ means that every iteration through loop adds 1 to value of I (6, 7, 8..14).

DO WHILE

```
do{
        body
}while(condition);
```

Note: Difference between those two is that do while loop will execute

minimum one time. To figure this out follow this code:

```
int y = 0;
do {
        System.out.print(y++);
} while (y< 10);
```

expected: 0123456789 will be printed to console

It literally says do something, in this case print value of y incremented every other time, and then check condition. Iterate through loop until condition is false (until 9).

Chapter 3. String

String is an object that represent sequence of char values. There is a two was for creating String objects:

1. By String literal
   String str = "New String";
2. By new keyword
   String str1 = **new**String("New String");

Note: Difference will be deeper explained when we will discuss about memory in JVM.

Let`s now create a few Strings by literal that contain your name, surname and country where you live. So in your current project make new class called ClassStringwith main method in first.pack package.
For example:

```
String name = "Your name";//type your name
String surname = "Your surname";//type your surname
String country = "Your country";//type your country
```

Note: Additional information that you can see is called comment. Comments are used to type some massages about your code for better reviewing latter on or organization of code. Here is another type of comment:

```
/*
This is a comment.
*/
```

If one operand is String, whole expression is String. It will be explained in another lines of code:

```
inti = 1;
String a=  "Value: " + i;
System.out.println(a);
```

expected: Value: 1 will be printed to console

Second operand has converted to String(his String representation has made).

Methods Of Class String

- **equals**

We have two String literals that are needed to be compared if they are equal or not:

```
String a1= "This is some text";
String a2= "this is some text";
```

Based on progress that we made until now, it won`t be hard to recognize what this code works:

```
if(a1.equals(a2))
        System.out.println("Strings a1 and a2 are similar");
else
        System.out.println("Strings a1 and a2 are not similar");

if(a1.equalsIgnoreCase(a2))
        System.out.println("Strings a1 and a2 are similar");
else
        System.out.println("Strings a1 and a2 are not similar");
```

expected: Strings a1 and a2 are not similar will be printed to console
Strings a1 and a2 are similar

Try to make it different. Make another String literals. What if we have String in lowercase and want to convert it to uppercase? This method will help as:

```
System.out.println("this is text".toUpperCase());
```

expected: THIS IS TEXT will be printed to console
This could be done vice versa, with method .toLowerCase().

- **substring/startsWith**

```
String s = "This is some text";
String s1 = s.substring(0,5);
if(s.startsWith(s1))
```

```
        System.out.println("Starts with character from s1");
else
        System.out.println("Does not start");
```
expected: Starts with character from s1 will be printed to console
Let`s discuss about this example. Method substring(a, b) with two parameters will count from begging of String s, it counts from 0, until fifth character. Note that space is also counted. In this case result of this will be “This”. Next is startsWith(a) method which checks if String s starts with String s1 (“This”). If it is true it will be print first sentence, else second one.
Note: method substring(a) could be with one parameter.

```
String s2 = "Some text";
System.out.println(s2.substring(5));
```
expected:text will be printed to console

- **contains**

```
if(s.contains("text"))
        System.out.println("String s contains 'text'");
else
        System.out.println("Does not contain");
```
expected: String s contains 'text' will be printed to console

- **compareTo**

```
String name1 = "Ara";
String name2 = "Ana";
if(name1.compareTo(name2)>0)
        System.out.println("After");
elseif(name1.compareTo(name2)<0)
        System.out.println("Before");
else
        System.out.println("Same");
```
expected: After will be printed to console

This code will lexicographically compare String names.

- returns < 0 then the String calling the method is lexicographically

first

- returns == 0 then the two strings are lexicographically equivalent
- returns> 0 then the parameter passed to the compareTo method is lexicographically first.

ARRAYS

Array can be treated as a container whose holds a fixed number of values. That values must be of **same** type. Array have**fixed** length when is created.

- One dimensional arrays

```
inta[];
a = newint[5];
inta[] = newint[5];//same as previous
int[] a = newint[5];//also same
Declaration and initiation:
inta[] = {1, 2, 3, 4, 5};
```

- Two dimensional arrays

```
int[][] a = {{1, 2, 3}, {4, 5, 6}};
```

What if we want to print all values that are stored in some array, one or two dimensional. This could be done by using **for** and **for each** loops:

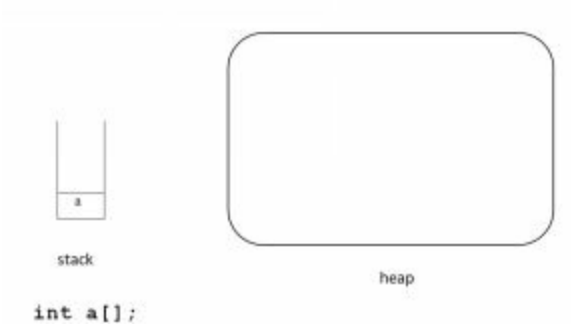
```
intarray[] = {1, 3, 5 ,7, 9};
for(inti = 0;i<array.length;i++) {
        System.out.print(array[i] + " ");
}
intarray2[] = {2, 4, 6, 8, 10};
for(inti : array2) {
        System.out.print(i + " ");
}

int[][] a = { {1, 2, 3}, {4, 5, 6} };
for (inti = 0; i<a.length; i++) {
for (intj = 0; j<a[i].length; j++) {
        System.out.println(a[i][j]);
         }
        System.out.println();
```

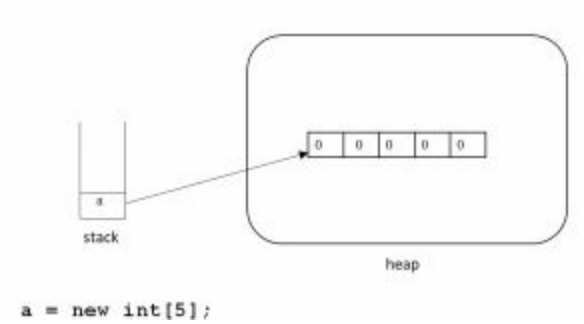
}
Loop go through each row and then again through each column in every row and print result.

Stack vs Heap

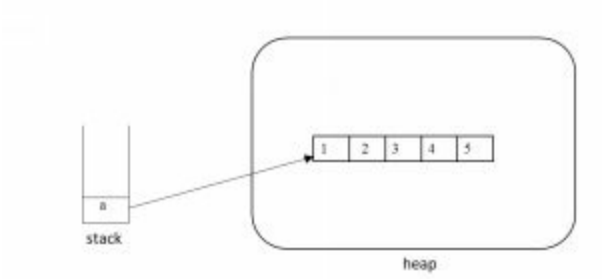
As just you could see creating primitive types, arrays and String is very easy. But, as just in other programs, every operation need some memory to be allocated. There are two types of memory in JVM (Java Virtual Machine), Stack and Heap. For example:




```
int a[];
```



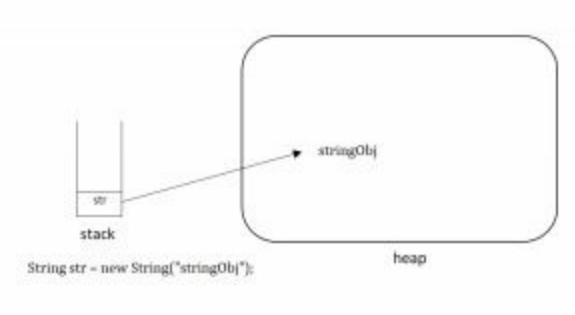

```
a = new int[5];
```



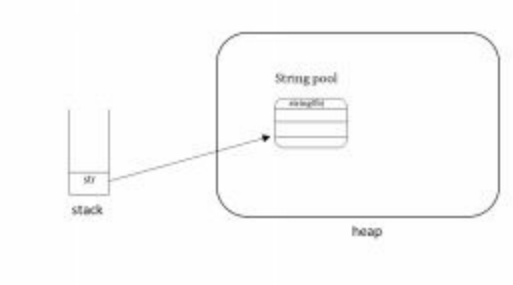

```
a[0]=1; a[1]=2; a[2]=3; a[3]=4; a[4]=5;
```

In first example we created an empty int, which is primitive type. We also could do it with double, float etc. result will be same.
In second picture you can see that we created an object with keyword **new**. From this we can conclude that on stack are stored primitive types and

references, on heap are stored objects.
In last one picture declared values are delegated to empty memory spaces on heap.



Similar thing happens with **String**. Reference of String object is stored on stack and object that is created is on heap. Here we have example of String pool, what we will explain next. For example, imagine that you created same String literal, like this:
String str1 = “stringObj”;



Note: Creating object with keyword **new** does not check String pool, it automatically creates a new String object. But, without keyword new, it creates new String in String pool and every other time it first check if there is String with same name. If that condition is true, it just return reference of existing object.

Chapter 4. Java IO

Buffered Reader

As we saw, System.out is **output** stream. Opposite of that is System.in,**input** stream. In new class JavaIO inside main method add following lines:

```
BufferedReader in = newBufferedReader(new
InputStreamReader(System.in));

System.out.print("Type your name: ");
String name = in.readLine();
System.out.println("Your name is: " + name);
System.out.print("Type your height: ");
intheight = Integer.parseInt(in.readLine());
System.out.println("Your height is: " + height);
```

There will be some errors. It says: “Unhandled exception type IOException”

Note: To fix this problem click on red alert and Add throws declaration. Now main method will throw exception if it occurs.
Run program.
expected: Type your name: will be printed to console
Type your height:

Also, we can see import section between package and class name. Java packages are used for better organization of java libraries and classes. In this case we can see that java.io package is imported from whose are derived methods inside our current class.

Wrapper Classes

Wrapper classes are used when something is typed that is **not**a String. Every primitive type has his own Wrapper class.

- int -> Integer
- long -> Long
- boolean -> Boolean

Wrapper classes are doing automatic boxing and unboxing (conversion

primitive types to objects and vice versa) using **XXXparseXXX** method. For example:

```
inti = Integer.parseInt("10");
longl = Long.parseLong("10");
```

SCANNER

Alternative for this is class **Scanner**, which can read both primitive types and Strings. Scanner is using delimiter to break input into tokens. Delimiter is whitespace by default. Try to run this code:

```
StringyourName;
intyourHeight;
Scanner sc = newScanner(System.in);

System.out.print("Enter your name: ");
yourName = sc.nextLine();
System.out.println("Your name is: " + yourName);
System.out.print("Enter your height: ");
yourHeight = sc.nextInt();
System.out.println("Your height is: " + yourHeight);
```

expected: Enter your name will be printed to console
Enter your height

Good practice is to close operations at the end of program using:
`sc.close();`

Try to make something different on your own. Use other primitive types, first for Buffered Reader and then for Scanner. Which method will be for float? Did you figured out that Buffered Reader and Scanner have different methods,
readLine() and nextLine()?

CHAPTER 5. OBJECTS

OOP

Object-oriented programming is a way of thinking for solving programming problems. Everything about this is connected with concept of **object**. Our mission is to somehow recognize relationships between objects(entities) from real world and "transfer" them to our code.
In Java, all starts with **classes**. Variable, methods and objects are placed inside classes. Objects are created by instantiating classes. Every entity has attributes and methods.
How we can group our data? Using arrays.

```
String[][] person = new String[1][3];
person[0][0] ="Name";
person[0][1] ="Surname";
person[0][3] ="Birthdate";
```

Using classes? Much more natural.

Let`s do some programming. In our current project make new package called second.pack. Inside that package add class called Car, but without main method. Our Car class will contain one attribute activate, and two methods start and stop.

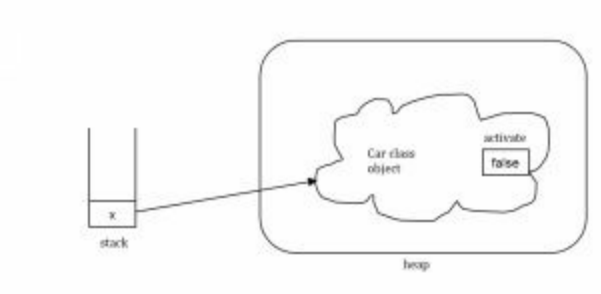
```
booleanactivate;
void start() {
        activate = true;
}
void stop() {
        activate = false;
}
```

We said that default value for boolean primitive type is **false**. Usage of start() and stop() methods is to change value of activatevariable.
Now we need another class for making this more interactive. In new class Test will be created Car object and called method.

Note: Just like it is mentioned, methods are supposed to be called inside main method. So, we need to place this code inside that method. Main method does not need to be implemented by default, it could be written just like any other method. Also, void keyword will be explained a little later.

```
Car bmw = newCar();//Car object created by keyword new
bmw.start();//reference of Car object used to call start() method
```

By calling start() method on our Car object value of activate attribute is set to true. But calling stop() method on same object value will be false. This program won`t print anything to console, don`t get confused.
Try it out. Make your own simple project. Imagine that you are in classroom for example. Make object of yourself, add attributes and methods. Methods could be based on condition if you are passed exam or not.



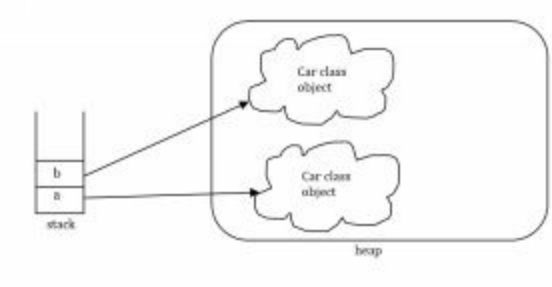
It is possible to initiate reference which doesn`t point to any object. Reference still exists on stack, but there is no new object created on heap.**null**is not object, You cannot instantiate or create variables of this type though. Method invocation on a **null**results in a NullPointerException.

```
Car opel = null;
```

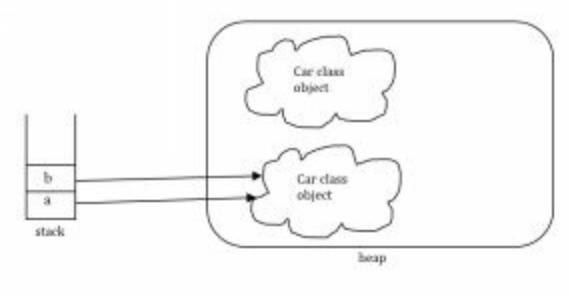
What if we have this situation:

```
Car a = newCar();
Car b = newCar();
```

Note: Objects are supposed to be created by using keyword **new**.



b = a;



Operator= is only coping a **reference**, not object.

CONSTRUCTORS

Constructor is used to **initialize** object. Constructor is automatically called by creation of object and must have same name as class. There is a two type of constructors:

- Default constructor (no-arg constructor)

For our coding we need new class Studentwith main method. Inside that method create new Student object like this:
Student stud = **new**Student();

Constructors are special methods, so we are creating them outside main method:

```
Student() {
}
```

Note: If default constructor is not created, compiler will do that by default. To be sure that constructor is called let`s leave some massage inside him and run program:

```
Student() {
        System.out.println("Student is created!");
}
```

expected: Student is created! will be printed to console

- Parameterized constructor

For this purpose, we need class Professor (without main method) and class Faculty (with main method). Class Professor will contain attributes and method along with parameterized constructor:

```
String name;
intage;
String city;
```

```
Professor(String n, inta, String c) {
        this.name = n;
        this.age = a;
        this.city = c;
}
void print() {
        System.out.println(name + " " + age + " " + city);
}
```

Note: **this** keyword means that we are referring to current object. Current object is object in class which we are using.

Now we want to make this functional. Do you remember when method and constructor are called? By creation of object, for sure. So, let`s do that in Faculty class and run program:

```
Professor prof = newProfessor("Ivan", 35, "Moscow");
prof.print();
```

expected: Ivan 35 Moscow will be printed to console

On this example is clearly shown usage of parameterized constructor. Try to delete parameters from Professor object. It will make some error: The constructor Professor() is undefined

METHODS

```
modifierreturn_type name (parameters) {
        body
}
```

return

We already covered methods and learned how to write most of them. But, not all methods are the same. For this we need Area class with main method. First method with **return** keyword will be explained:

```
publicstaticint countArea(intwidth, intheight) {
```

```
        returnwidth * height;
}
```

public static – modifier, what will be explained a bit latter
int – return type

This method need to be called somehow. In this case, we are aiming to print out result of width * height. To do that, println will help us, so this code will be put inside main method:

System.***out***.println(*countArea*(3, 5));
expected: 15 will be printed to console

VOID

As we saw, methods that have return keyword return some value. That value could be later on used for some operation. But what if we don`t want to return anything. Just to print any massage or something like that. How to make that method? We need class VoidMethods with main method implemented. Our method will check input and based on that condition will print relevant result. Try to make it on your own, it you got stuck see this code:

```
staticvoid seasonOfYear(String s) {
        if(s == "Spring") {
        System.out.println("It is spring!");
        } elseif (s == "Summer") {
        System.out.println("It is summer!");
        } elseif (s == "Autumn") {
        System.out.println("It is autumn!");
        } elseif (s == "Winter") {
        System.out.println("It is winter!");
        } else {
        System.out.println("You made mistake, sorry");
        }
}
```

It is not hard to figure out that we have String parameter which is used based

on input when method is called. So to do that, inside our main method we need to call seasonOfYear(String s) method:
seasonOfYear("Summer");
expected: Summer will be printed to console

Chapter 6. Access Modifiers

public – visible for all classes
protected – visible within package and subclasses.
private – visible only within class
default (friendly) – when there is no modifier, it is treated as default and visible within package

From this we can conclude that our method, for example, that is public is accessible in whole project. But what is with other modifiers? How to handle them? Now we will see one example with private modifier.

First add two classes PrivateClass and TestClass (with main method). In first class we are creating one simple method that will be called:

```
privatevoidmethodOne() {
        System.out.println("Method is called!");
}
```

In second class to call this method we first need object. In this case we will create object of PrivateClass class, just like this:

```
PrivateClass pc = newPrivateClass();
pc.methodOne();
```

It complains: The method methodOne() from the type PrivateClass is not visible

This is just what we said. Our method is private, so it is visible(accessible) only within class. We can`t reach that method from another class. To solve this click on error and change visibility of ‘methodOne()’ to ‘package’. Our method should now looks like this:

```
void methodOne() {
        System.out.println("Method is called!");
}
```

This is **default** modifier. What will happen if we change modifier to **public**?

```
publicvoid methodOne() {
	System.out.println("Method is called!");
}
```

Nothing. Public methods are accessible from whole out project. Change now modifier to **protected**.

```
protectedvoid methodOne() {
	System.out.println("Method is called!");
}
```

Same again. Protected method is accessible within package.

THIS

In one of our previous section, Constructors, if you remember this keyword was used. We had a few attributes like name, age and city and based on them constructor of class Professor with input parameters was constructed. **this**.name = n;this is a reference to the **current object** — the object whose method or constructor is being called. So this refers to name attribute (attributes name, age and city belong to Professor prof = **new**Professor();, static fields belong to class what we will discuss). Parameters in constructor String n, **int**a, String creserve some memory space and they need to be exactly type like attributes (String, int).
Primitive types are not only that could be parameters in constructors or methods. It is also possible to place **object** as parameter. We need class Person and TestPerson (with main). In our first class we will create constructor with parameters, attributes of person and method which will print our attributes. Let`s see example:

```
String name;
intage;
Person(String n, inta) {
        this.name = n;
        this.age = a;
}
void printMyName(Person p) {
        String myNameAge = p.name + " " + p.age;
        System.out.println("My name and age are: " + myNameAge);
}
```

As it is mentioned, object is passed as **argument**. Our method refers to name and age attributes using **p** reference of Person object. myNameAge is String whose stores that value and later on is printed to console.

```
Person person = newPerson("Name", 27);
person.printMyName(person);
```
expected: My name and age are: Name 27

To call our method we need object with parameters just like in Person class. This code is supposed to be inside TestPerson class (in main method).

Chapter 7. Static

All attributes and methods that we covered belong to some object, except one. Do you remember seasonOfYear() method? What was difference? Of course method has static modifier, but what does it means? Is there any object whose is used for call of this method? No. So, static fields and methods belong to class, not an object.
If there is a need for a variable to be common to all object of some java class, then it is supposed to be **static variables**(class variables).All instances share the same copy of the variable. Static variables are initialized only once.
Like static variables, **static methods** belong to class, not and object. Static method is called by class name. It is time so see some examples of static.
Create StaticClass which will include this fields:

```
intval = 3;
staticint val2 = 4;

staticvoid changeValue() {
        val2 = 6;
        System.out.println(val2);
}
staticvoid print() {
        System.out.println("Static method calling!");
}
```

So, in our main method we will call static methods:

```
StaticClass.print();
StaticClass.changeValue();
```

expected: Static method calling! Will be printed to console
6

Let`s now change *val2*to valinside changeValue() method. What will happen? It is complaining: Cannot make a static reference to the non-static field val. It is because static method can only access static fields. There is also another way for calling static methods, result will be same:

print();
changeValue();
Now when we covered everything that is needed, our famous main method can be explained in total.
public – method is visible in whole project.
static – method belong to class. Reason why main is static is because it can be ambiguity, compile doesn`t know which constructor to call.
void – method doesn`t return any value.
Ok, let`s see now one very interesting example:

```
static{
        System.out.println("static");
}
publicstaticvoid main(String[] args) {
         System.out.println("main");
}
```

expected: static will be printed to console
main

How this happens? Aren`t we said that methods are executing in main method? This is called **static block**. Static block will be executed when a class is first loaded into the JVM, even before main method.
Output of this program will be random number from 0 to 1:

```
System.out.println("Random number: " + Math.random());
```

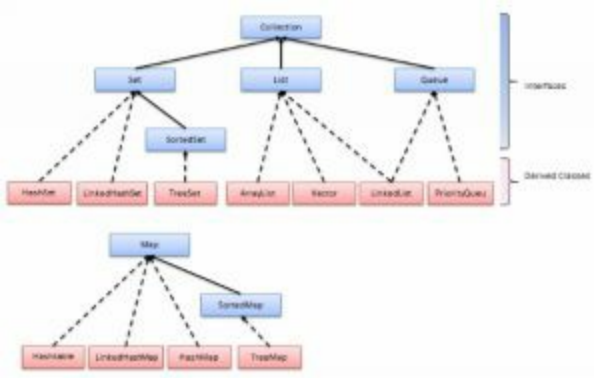
But, how is this working? Just like System.*out*.println();
java.lang.System
java.land.Math
Both classes have their static methods that can be called. They don`t need object so it is simplified. So, methods that we already commonly used, **out()**, is static method from class System, on which class name is called. It is same with **random()** method from Math class. Try to use other methods like this:

```
inta = 4;
intb = 2;
```

```
double c = 2.4;
System.out.println("Minimum is: " + Math.min(a, b));
System.out.println("Maximum is: " + Math.max(a, b));
System.out.println("Pow of c is: " + Math.round(c));
```

CHAPTER 8. COLLECTIONS

Collections are used to store and manipulate the group of objects. Here is shown hierarchy of Collection classes and interfaces. We will cover ones that are commonly used.



Arrays have one disadvantage. Ones they are create, it is impossible to change their size. This problem are solving Collections.

List– list of objects which is ordered. Objects are accessible by index. It is possible to have two same objects.

Queue – is ordered just like List, difference is that new elements are added at the end and removed from the beginning from the queue.

Set–all elements are unique.

Map–is not type of Collection interface. Elements are mapped by key/value.

Most of this Collections are located in java.util package.

Common **methods** for all Collections are:

- Adding elements
- Removing elements
- Iteration through elements

Once we cover a few Collections it would be easier to look at JavaDoc and figure out how rest of them works. For now we will focus on List and Map, it is biggest chance to have need for them. Queue and Set are not that usually used, but from experience with this both it would be easy to handle them. Also, don`t get confused, interfaces are covered in one of next sections.

- ArrayList

ArrayList represents dynamic Collection. Methods for manipulating elements

are:

- **add()** – for adding element to ArrayList
- **remove()** – for removing elements from ArrayList
- **get()** – for getting elements from ArrayList

It is time to see some example of ArrayList. So, let`s create a new package third.pack and class Array inside.

```
ArrayList<Integer>ar = new ArrayList<Integer>();
ar.add(1);
ar.add(2);
ar.add(3);
System.out.println("Size of ArrayList is: " + ar.size());
ar.remove(0);
intnum = ar.get(0);
System.out.println(num);
System.out.println("New size of ArrayList is: " + ar.size());
```

expected: Size of ArrayList is: 3 will be printed to console
2
New size of ArrayList is: 2

So, in our code is used everything that is mentioned about ArrayList. First it is declared. We said that arrays have one big disadvantage, they are not supposed to change they size. But as you can see, our ArrayList is dynamic, three elements are added(Size of ArrayList is: 3). Compiler count elements from 0, so 1 = 0, 2 = 1, 3 = 2. It is important to remember that because of manipulation of elements. Another operation is removing first element from our array. Variable num holds, by get(0) method, first element and println print it to console(2). Also, size is not anymore 3, it I s now 2.

What if we have bigger ArrayList and we want to print all values? For loop is solution, of course:

```
for (intcounter = 0; counter<ar.size(); counter++) {
        System.out.println(ar.get(counter));
}//for loop
for (Integer n : ar) {
       System.out.println(n);
}//for each loop, same result
```

- **HashMap**

As you could see, Map is not derived from Collection interface and it

behaves a bit different. Elements are added in Map with theirkey, which need to be **unique**. Advantage of this approach is that is fast to find elements by their key. Methods for manipulations are:

- put() – key and value are put to HashMap.
- get() - value is founded based on key, if there is no values null will be returned.

Make new class Map and start on another example:

```
HashMap<Integer, String>hm = new HashMap<Integer, String>();
hm.put(1, "numOne");
hm.put(2, "numTwo");
hm.put(3, "numThree");
System.out.println("Print me value of key 1: " + hm.get(1));
```

expected: Print me value of key 1: numOne will be printed to console

Map is not ordered Collection, that means it does not return the keys and values in the same order in which they have been inserted to the HashMap. HashMap allows one null key and any number of null values.
Now let`s discus about something. In our examples of both, ArrayList and HashMap, we saw somothing like this: <Integer> ,<Integer, String>. When any Collection contains this classes, they are called **Generics**. What does it mean? Well, try to add this code in Map class:

```
hm.put("four", "numFour");
```

Compailer complains. It is because we declared our keys as integers(Integer wrapper class), "four"is String. Same is with ArrayList:

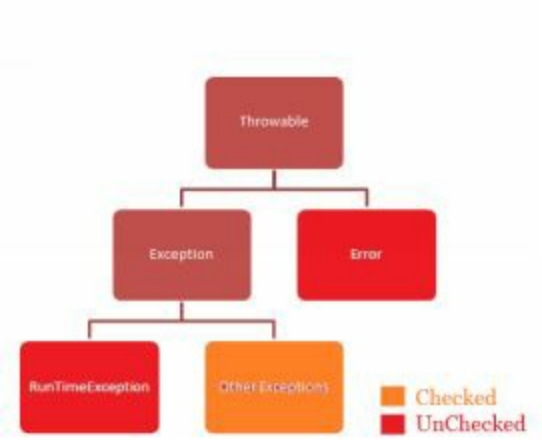
```
ar.add("four");
```

In our ArrayList only integers are allowed.
Enum is special type of collection. It is collection of constants. Enum will be useful for our later work, for now this is enough to know.

```
publicenum Person {
        name,
        age,
        city
}
```

EXCEPTION HANDLING

Handling exceptions when they occur is very important part of Java programing. Imagine that you don`t have mechanism for dealing with run time errors. What will happen? Who knows. But for sure most of projects would fail. So, let`s get start on this.
would fail. So, let`s get start on this.



Classification of exceptions is based on compile-time checking of exceptions.
Note: **compile-time**– time while you type your code. For example: if you remember cases with ArrayList and Map when we had problems because compailer complained.

run-time – time when you run your program/project.

Let`s on one common example how **try/catch block** works:

```
try {
        intnum = 2 / 0;
    } catch (ArithmeticException e) {
System.out.println("Error: Don't divide a number by zero");
}
System.out.println("Out of try/catch");
```

expected: Error: Don't divide a number by zero will be printed to console
Out of try/catch

Try to instead of 0 type 1. Error won`t occurs. In “real life” handling exceptions is much complex than this. But for beginning it is important to figure out difference between exceptions and why we need to catch them.
Unchecked exceptions: don`t need to be caught because they can be removed

while coding. Examples: NullPointerException. IndexOutOfBoundsException, ArithmeticException.
Checked exceptions: need to be caught by try/catch block. Example: FileNotFoundException, SQLException.
Errors: indicate some serious problem. Example: OutOfMemoryError. We shouldn`t try to deal with them.

ENCAPSULATION

Encapsulation is term that is connected with access modifiers. Imagine that you have some class with attributes. Everything can work well, but problem may occur in context of insecurity of your data. It is because someone can to reach to your class and make changes. So, to make this impossible and to have full control of your data inside class, **getters** and **setters** will help you. To see on example, first make new fourth.pack package and Person class inside. Our Person would have name and age attributes, which are declared as private.

```
private String name;
privateintage;
```

Now we need to open Source ->Generate Getters and Setters
Final move is to check name and age fields and press OK.
expected: Getters and Setters will be generated
So, as you can see, out attributes remain **private** (accessible only within class) and methods (getters and setters) that we generated are **public**. We made our class safer.

```
public String getName() {
        returnname;
}
publicvoid setName(String name) {
        this.name = name;
}
```

Look at getName() method. If we need our name for some reason we will call this method, which return name. Our attributes is untouched. Same is with setName(String name) method. It changes values of our name by referring to it with this keyword.

Abstract Classes

Do you remember that we mentioned classes like Professor and Student? They are related entities and have some common attributes like name, age and city in our example. So, why to always declare that attributes again and again? Isn`t would be easier to have some base class that contains all data, attributes and methods, and to share to all classes whose need them? Of course it would be. Class that do it is called **abstract class**. To make previous class Professor abstract, add abstract declaration to it`s name:

```
publicabstractclass Person
```

Abstract class can`t be instantiated. Also, if there is at least one abstract method in class, class is automatically abstract. **Abstract method** is method without implementation and declared abstract:

```
abstractvoidrun();
```

Note: Make parameterized constructor for class Person that includes mentioned attributes and generate getters and setters. Also don`t forget to add run() method.

Now we have our first abstract class. Another step is to make class that will use fields from Person class. Let it be class Student, again. To make Person class useable, we need **extends** declaration:

```
publicclass Student extends Person
```

Now our class Student extends (shares) all fields with Person, but also we can add some specific attributes for Student. Beside that, constructor will looks like this now:

```
privateint[] grades;
Student(String name, intage) {
        super(name, age);
        this.grades = newint[5];
}
```

Something new is here for us. **super** keyword means that our constructor refers to it`s superclass. So, we extended Person class, which has name and age attributes. That attributes aren`t declared again in class. Also, you can now recognize that there is an array which takes int values and size is 5.

Note: you can see that all the time compiler is complaining: The type Student

must implement the inherited abstract method Person.run()
That is because we need also to implement run() method from our superclass (Person).

```
@Override
void run() {
        System.out.println("Student runs!");
}
```

Note: @Overridedeclaration means that method is derived from super class. One thing that is very important is that class can only extends **one** abstract class.

CHAPTER 9. INTERFACES

Interfaces are **not** classes. They only contain methods and attributes whose class that implements that interfaces need to have. You can recognize first difference from abstract classes, interfaces are implementing by keyword **implements**, not extending. All methods are implicit public and all attributes are implicit public static final. Let`s now make out first interface by right click on current package New ->Interface and name it House. We will add following lines:

```
String bed = "My bed";
void sleep();
```

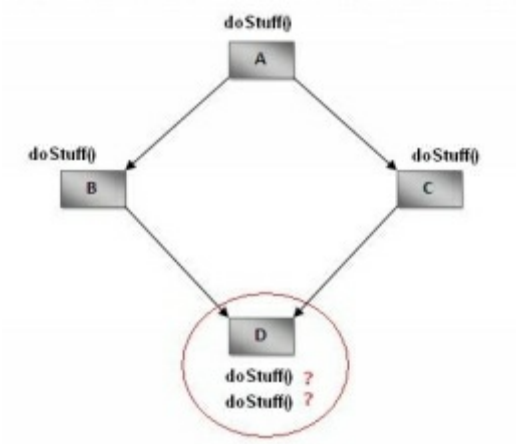
So, as you can see our method sleep() don`t have implementation. That is because we don`t need to know how method must be implemented at this moment. To understand better, imagine that you have interface Vehiclewith methods like start() and stop(), and classes which implement that interface like Car, Bicycle and Airplane. At the moment of creation of Vehicle interface you only know that you would have some different type in future, that we mentioned, but you actually don`t know how they work because all work different. Think in that way about interfaces.

But, from Java 8, interfaces are changed. Now you can have **default** method:

```
defaultvoid watchTV() {
        System.out.println("I am wathcing TV");
}
```

As you can guess, this means that now interfaces are more similar to abstract classes. One thing that is very important is that you can implement **multiple** interfaces, rather that only one class.

Both, abstract classes and interfaces use **inheritance**. Inheritance is type of **code reusing**. This is very useful because in development you will be in situations where there are some classes with some methods and attributes that you may need. When class is extended and interface implemented, subclass need to override **all** their methods. Why we told that only one class can be extended?



Class A is super class and class D extends both class B and class C which have method with same signaturedoStuf() but they have different implementation. Which one should compiler choose? Situations like this are reason why creators of Java left out multiple inheritance. This is called a **Diamond problem**.

Note: Inheritance is is – a relationship, Student is a Person; Car is a Vehicle.

Association vs Aggregation vs Composition

When two objects are related to each other they are in **association**. Both composition and aggregation are types of association.

COMPOSITION

Inheritance is not only type of code reusing. Composition is another one, whose is has – a relationship between classes. With inheritance, composition is main technique for code reusing. Composition is relationship where part can`t exists without a whole.

```
publicclass Thought {

}

publicclass Brain {
        Thought thought = newThought();
}
```

Brain has a thought. Thought can`t exists without brain. Composition has a **stronger** relationship then aggregation. Code reusing is here implemented by using object reference, thought, to use class which you need. Composition is special case of aggregation, it is more restricted.

AGGREGATION

Aggregation is **weaker** relationship then composition. In aggregation one part can exists without main object. For example: Bed and Pillow. Bed have many pillows and without pillow bed is functional. Also, without bed, pillow can exists for another purpose.

```
publicclass Pillow {

}
publicclass Bed {
public List<Pillow>pillows = new ArrayList<Pillow>();

}
```

Overriding vs Overloading

Overriding

Overriding is term that we already saw in our examples. If in subclass is method that have same name and same input parameters like method in super class, then we have **overriding**. Usage of this is to change implementation in our subclasses. Do you remember example with vehicles? When method start() is derived to subclass Car or Airplane, implementation is different. Car and Airplane have totally different engines.

```
voidstart() {
        System.out.println("Starting car engine");
}

voidstart() {
        System.out.println("Starting airplane engine");
}
```

Note: Maybe it would be little confusing when it says implementation but inside method body is only println. This are only simple examples, in real development there will be some logic and things like that.

Overloading

In overriding section we said that derived methods have same input parameters. If you didn`t figured out input parameters are inside curly braces of method declaration like: (String n, **int**a). That is not case with **overloading**. Methods are overloaded when they have same signature but different input parameters, of course in same class. We will explain this on constructor overloading. For example, make class StudentClass and add attributes name, age and diploma:

```
private String name;
privateintage;
private String diploma;

StudentClass(String n, inta) {
        this.name = n;
```

```
        this.age = a;
}
StudentClass(String n, inta, String d) {
        this.name = n;
        this.age = a;
        this.diploma = d;
}
```

Constructors are overloaded. But what is usage of this? Imagine that you have class Faculty that manages with students. Not all students are same. Until finishing studies student don`t have diploma, so if you need some operations with students like this constructor without diploma parameter will be used. On another hand, if you want to store students that are finished studies to some document, constructor with diploma parameter will be useful. Beside constructors, methods also could be overloaded:

```
void doSomething(String a){
        System.out.println("Print a: " + a);
}

void doSomething(String a, String b){
        System.out.println("Print both a and b: " + a + b);
}
```

Both overriding and overloading are types of **polymorphism**. Polymorphism means that one method can do different things based on used **object**. Polymorphism in Java have two types:

- Compile time polymorphism

As we already sad, compile time is time while you type your code and compiler automatically recognized if you have some errors. So, compile time is decided in that time. Perfect example of this type of polymorphism is overloading. Compile time polymorphism is also called **static binding**.

- Runtime polymorphism

Overriding is type of runtime polymorphism. Unlike static binding, this type of polymorphism is called **dynamic binding**. From name you can guess that an overridden method is recognized at runtime.Upcasting is when reference variable of parent class refers to the object of child class we. JVM decides

about overriding methods, not compiler.
Note: JVM (Java Virtual Machine) is an abstract machine that enables you to run a Java program on computer.

Subheading 1

Chapter Text

Sub Subheading

Chapter Text

Chapter 10. Design Patterns

Singleton Pattern

Singleton pattern is one of most commonly used design patterns. He belongs tocreational pattern category. Pattern **restrict** object creations and ensures that only one instance of class exists. Imagine that you only need one garage at home, only one HR at company, only one captain at team. That are examples from real life in whose you will need singleton pattern. For implementation of pattern we need some conditions:

- private constructor - We didn`t explained in constructor section, but in short private constructor doesn`t give class to be instantiated from another class
- private static instance – only instance of the class
- public static method – method which returns instance of class

There are two types of making class singleton:

- Lazy initialization

To better figure out what exactly means this three conditions, let`s see out code below:

```
private SingletonOne(){
        }

privatestatic SingletonOne instance;

publicstatic SingletonOne getInstance(){
        if(instance == null){
        instance = new SingletonOne();
        }
         returninstance;
}
```

Creation of Class object is derived to getInstance() method. As public static method have global access and there is no need for object for calling, static method is called by class name. This type of singleton pattern is called lazy

because instance of Class is not created until getInstance() method is called. getInstance() method checks if there is already instance, is it is false (**null**) method creates new one (**new** CLass())and return it.With lazy initialization you crate instance only when its needed and not when the class is loaded. So you escape the unnecessary object creation.

- Eager initialization

```
privatestaticfinal SingletonTwo instance = new SingletonTwo();

private SingletonTwo(){}

publicstatic SingletonTwo getInstance(){
        returninstance;
}
```

Unlike lazy initialization, eager initialization is done when class is loaded. So, difference is that in first approach you get instance of singleton class only when you actually need it.

MVC Pattern

MVC pattern is maybe most common used pattern in making Java applications. As name tells, this is acronym of Model – View – Controller

- Model – represents data or POJO classes. POJO is just declaration which must match some conditions like: public default no-arg constructor, private attributes and getters and setters.
- View – represents view of mentioned attributes.
- Controller – controller behaves between both model and view, it updates view based on changes from model.

For our explanation we need classes below:

```
publicclass Doctor {
private String name;
privateintage;

Doctor() {}

public String getName() {
        returnname;
}
publicvoid setName(String name) {
        this.name = name;
}
publicint getAge() {
        returnage;
}

publicvoid setAge(intage) {
        this.age = age;
}
}
```

Class Doctor represents our **model**. If you remember, default no-arg constructor will be made by default by compiler, but we did it just to don`t make confusion.

```
publicclass ShowDoctor {
publicvoid showDoctor(String name, intage) {
```

```
        System.out.println("Doctor with name: " + name + " and age: " + age);
        }
}
```

This class represent String presentation, **view**, of our Doctor with attributes name and age.

```
publicclass Controller {
private Doctor doctor;
private ShowDoctor showD;
Controller(Doctor d, ShowDoctor s) {
        this.doctor = d;
        this.showD = s;
}

public Doctor getDoctor() {
        returndoctor;
}

publicvoid setDoctor(Doctor doctor) {
        this.doctor = doctor;
}
public ShowDoctor getShowD() {
        returnshowD;
}
publicvoid setShowD(ShowDoctor showD) {
        this.showD = showD;
}
void shomMeDoctor() {
        showD.showDoctor(doctor.getName(), doctor.getAge());
}
}
```

As you can see, our **controller** takes references from both model and view classes. Now we need one class to manage with all this classes.

```
publicclass Program {
Doctor doctor = setDoctorAttributes();
ShowDoctor show = newShowDoctor();
Controller controller = newController(doctor, show);
privatestatic Doctor setDoctorAttributes(){
```

```
    Doctor doc = newDoctor();
doc.setName("Bob");
doc.setAge(50);
returndoc;
}
void printDoctor(){
        controller.shomMeDoctor();
}
publicstaticvoid main(String[] args){
        Program program = newProgram();
        program.printDoctor();
         }
}
```

Program is class with main method, you remember that main method is used for calling other methods. Our whole program would not be possible to start without it, so we needed to create object of current class. We provided references of Doctor, ShowDoctor and Controller to current class. Association is here used for code reusing. Because we don`t have doctor, method setDoctorAttributes() is made to set values. It is done by using doc reference and setName() and setAge() method, in other words setters. This method is derived to doctor reference. We don't create new doctor by newDoctor();Output is: Doctor with name: Bob and age: 50

PROJECT

Everything that we covered until now have his own usage and it is very important part of Java programing. But they are only parts of one big "story". Best way for learning something, Java also, is to use it on real problems, in this case in project. In this last section, we will try to make from all (almost all) that we learned, one functional project. Based on this you will have clearerpicture how and when to use some features of Java. Idea is to make a project **Hospital**.
Characteristics of Hospital are:

- Doctor (name, surname, specialty) have many Patient (name, surname, medical record number)
- Patient can have only one Doctor
- Doctor assigns laboratory examination to Patient
- Examination have date and time
- Types of examinations:
  - ¬ Blood sugar level (upper value, lower value, pulse)
  - ¬ Blood pressure (value, last time meal)

Make script that simulates next:

1. Create Doctor "Bob Williams"
2. Create Patient "Fill Kane"
3. Patient "Fill" chooses "Bob" to be his Doctor
4. Doctor "Bob" calls Patient "Fill" for blood sugar level examination
5. Doctor "Bob" calls Patient "Fill" for blood pressure examination
6. Examinations have examinationID and date values
7. Show results for both examinations

Start to think about project with entities, Doctor and Patient, are they have something similar? Do we maybe need abstract class? Is this also case with examinations? How to derive only one doctor to patient? Singleton pattern? Let`s start making our project to better figure out all of this.
In Eclipse create a new project Hospital. Our entities will be placed inside model package. Every hospital has entities like Doctor and Patient, our also will. First, we will make abstract class Person:

```
protected String nin;
protected String name;
protected String surname;

public Person(String n, String name, String surname){
        this.nin = n;
        this.name = name;
        this.surname = surname;
}
```

Generate getters and setters for nin, name and age attributes also.nin stands for National Identity Number.Next class is **Doctor**:

```
publicclass Doctor extends Person{

private String specialty;
public Doctor(String n, String name, String surname, String spec){
        super(n, name, surname);
        this.specialty = spec;
         }
}
```

Also, generate getters and setters for specialty attribute. **Patient** class:

```
publicclass Patient extends Person{
privateintmedicalRecordNumber;
public Patient(String n,String name, String surname, intmedRecNum){
        super(n, name, surname);
        this.medicalRecordNumber = medRecNum;
        }
}
```

Getters and setters for medicalRecordNumber.

Our next job to do is to assign doctor to patient. We have condition, our patient can have only one doctor. You can guess, singleton pattern need to be implemented. Use lazy initialization in Hospital class:

```
private Hospital(){
}
privatestatic Hospital instance;
publicstatic Hospital getInstance(){
        if(instance == null){
        instance = new Hospital();
```

```
        }
        return instance;
}
```
Doctor and patients need to be stored somewhere. We will use HashMap for that. Add this to Hospital class:
```
private Map<String, Person>persons = new HashMap<String, Person>();
```
Our Hospital also needs methods for manipulationg with persons.
```
private Person TryGetPerson(String nin){
        returnpersons.get(nin);
}
public Doctor TryGetDoctor(String nin){
        Doctor doct;
        doct = (Doctor)TryGetPerson(nin);
        returndoct;
}
public Patient TryGetPatient(String nin){
        Patient patient;
        patient = (Patient)TryGetPerson(nin);
        returnpatient;
}
```
Method TryGetPerson() returns person with his nin. To decide if that person is doctor or patient, we need TryGetDoctor() and TryGetPatient() methods.
Note: Look at script. At 7. it says we need to show results. For this purpose, we will use log4j, new for us but very simple feature. Download jar file from this site: http://www.java2s.com/Code/Jar/l/Downloadlog4jjar.htm
Extract file and add jar file to project by Right click ->Properties ->Java Build Path ->Add External JARs ->log4j
Below class name Hospital, add our Logger:
```
static Logger log = Logger.getLogger(Hospital.class);
```

Now is order to make methods which actually create doctor and patient:
```
publicvoid createDoctor(String nin, String name, String surname, String spec){
        if(TryGetDoctor(nin) == null&& !persons.containsKey(nin)){
        Doctor doctor = newDoctor(nin, name, surname, spec);
        persons.put(nin, doctor);
        log.info("Doctor " + name + " " + surname + " with speciality " + spec
```

```
+ "is created!");
        }else{
        log.info("Doctor " + name + " " + surname + " already exists!");
        }
}
publicvoid createPatient(String nin, String name, String surname,
intmedRecNum){
        if(TryGetPatient(nin) == null&& !persons.containsKey(nin)){
        Patient patient = newPatient(nin, name, surname, medRecNum);
        persons.put(nin, patient);
        log.info("Patient " + name + " " + surname + " with medical record " +
medRecNum + " is created!");
        }else{
        log.info("Patient " + name + " " + surname + " already exists!");
        }
}
```

Both methods are doing same thing. HashMap takes input by key/value. In this case nin is key, so in our methods we are checking if person doesn`t exists. If it is true, then we are creating a new one. If it is false we are printing some massage. containsKey() method isn`t covered, it checks if map contains a mapping for the specified key.

Next step is to assign patient to doctor. First we need some method in Patient class for assigning to doctor. Method is similar like this two above:

```
private Doctor assign;
publicvoid assignDoctor(Doctor doc){
        if(Hospital.getInstance().TryGetDoctor(doc.JMBG) != null){
        assign = doc;
         log.info("Patient is assigned to " + doc.getName() + " "+
doc.getSurname());
        }else{
         log.info("Doctor " + doc.getName() + " " + doc.getSurname() + "
doesn`t exists!");
        }
}
```

Our method is using reference of Doctor class to assign type of Doctor to itself.

Note: Also add logger to Patient class.

```
static Logger log = Logger.getLogger(Hospital.class);
```

In beginning it sad that doctor can have many patients, so we also need HashMap for that patients, inside Doctor class of course:

```
private Map<String, Patient>patients = new HashMap<String, Patient>();
```

This map again need method for putting patients inside. Let`s create her:

```
publicvoid addPatient(Patient patient){
        if(Hospital.getInstance().TryGetPatient(patient.nin) != null){
        patients.put(nin, patient);
        }
}
```

Class Hospital now need method to implement assignDoctor() method:

```
publicvoidassignDoctorToPatient(Doctor doctor, Patient patient){
        patient.assignDoctor(doctor);
}
```

Now is time to move on examinations. It is sad that each examination have ID and date as common attributes, so in this case we also can use concept of abstract classes. LaboratoryExaminationwill be our abstract class:

```
publicabstractclass LaboratoryExamination {

        publicintexaminationID;
        public Date date;
        public LaboratoryExamination(intexam, Date date){
        this.examinationID = exam;
        this.date = date;
        }
}
```

Don`t forget to generate getters and setters, also import java.util.Date class. We are not discussed about this, but compiler will complain to import classes like Map, HashMap and Hospital to Doctor and Patient classes. Just do it.

BloodSugarLevel class:

```
publicclass BloodSugarLevel extends LaboratoryExamination{

privatedoublevalue;
private Date lastMealTime;
```

```
public BloodSugarLevel(intexam, Date date, doublevalue, Date last){
        super(exam, date);
        this.value = value;
        this.lastMealTime = last;
}
public BloodSugarLevel(intexam, Date date){
        super(exam, date);
         }
}
```

BloodPressure class:

```
publicclass BloodPressure extends LaboratoryExamination{
privateintupperValue;
privateintlowervalue;
privateintpulse;
public BloodPressure(intexam, Date date, intupper, intlower, intpulse){
        super(exam, date);
        this.upperValue = upper;
        this.lowervalue = lower;
        this.pulse = pulse;
        }

public BloodPressure(intexam, Date date){
        super(exam, date);
         }
}
```

Again, getters and setters!

Before we make last one methods for examination manipulations, we need class that will hold references from Patient, Doctor and LaboratoryExamination classes. You will see latter why, so create PatientDoctorExamination class:

```
private Doctor doctor;
private Patient patient;
private LaboratoryExamination labExam;
publicPatientDoctorExamination(Doctor doc, Patient pat,
LaboratoryExamination ladExamination){
        this.doctor = doc;
        this.patient = pat;
```

```
        this.labExam = ladExamination;
}
```
As always, getters and setters need to be generated in this class.
Let`s finish with our Hospital class. Like with persons, we need HashMap for examinations:
```
private Map<Integer, PatientDoctorExamination>examinations = new HashMap<Integer, PatientDoctorExamination>();
```

Here you can see usage of previous PatientDoctorExamination class. Also, we need something that will store our examinations. For this, let`s use enum in Hospital class:
```
publicenum laboratoryExaminations{
        bloodPressure,
        bloodSugarLevel,
}
```

Ok, but as always, we need something to do with our data, examinations. Method need to put our examination to HashMap based on input. One more thing that we covered earlier will be used, switch, so add method to Hospital class:
```
privateintexamID;
publicint ScheduleExamination(Doctor doctor, Patient patient, Date date, laboratoryExaminations examinationsLab){
        LaboratoryExamination examination = null;
        switch(examinationsLab){
        casebloodPressure:
        examination = new BloodPressure(examID, date);
        break;
        casebloodSugarLevel:
        examination = new BloodSugarLevel(examID, date);
        break;
        }
        examinations.put(examID, new PatientDoctorExamination(doctor, patient, examination));
        examID++;
        returnexamination.examinationID;
}
```

Last one thing that we need to do in Hospital class to make it functional is to somehow make methods that are finally making examinations. This will be shown is code below:

```
publicvoid FillBloodSugarLevelExamination(intexaminationID,
doublevalue, Date date){
        PatientDoctorExamination patientDoctorExamination = null;
        patientDoctorExamination = examinations.get(examinationID);
        if(patientDoctorExamination != null){
        if(patientDoctorExamination.getLabExam() instanceof
BloodSugarLevel){
        BloodSugarLevel bloodSugarLevel =
(BloodSugarLevel)patientDoctorExamination.getLabExam();
        bloodSugarLevel.setValue(value);
        bloodSugarLevel.setLastMealTime(date);

log.info("Examination is successfully done! \n" +"=======\n"+ "value: " +
value + ", last meal time: " + date);
        }else
        log.info("Selected examination is not for blood sugar level!");
        }else
        log.info("Selected examination does not exist!");
}
publicvoid FillBloodPressureExamination(intexaminationID, intupperValue,
intlowerValue, intpulse){
        PatientDoctorExamination patientDoctorExamination = null;
        patientDoctorExamination = examinations.get(examinationID);
        if(patientDoctorExamination != null){
        if(patientDoctorExamination.getLabExam() instanceof BloodPressure)
{
        BloodPressure bloodPressure =
(BloodPressure)patientDoctorExamination.getLabExam();
        bloodPressure.setUpperValue(upperValue);
        bloodPressure.setLowervalue(lowerValue);
        bloodPressure.setPulse(pulse);
log.info("Examination is successfully done! \n" +"=======\n"+"upper
value: " + upperValue + ", lower value: " + lowerValue + ", pulse: " + pulse);
```

```
        }else
        log.info("Selected examination is not for blood pressure!");
        }else
        log.info("Selected examination does not exist!");
}
```

Now when all characteristics and conditions are satisfied, only thing that is left to do is to create class with main method to see output of our program. Hope you didn`t forgot that all classes, except Hospital, were supposed to be in model package. Hospital and ProgramHospitalclass, which will be made now, are placed in program package. While creation of this class, add main method, and following lines inside:

```
org.apache.log4j.BasicConfigurator.configure();
Hospital.getInstance().createDoctor("1234", "Bob", "Williams", "surgeon");
Hospital.getInstance().createPatient("2345", "Fill", "Kane", 25);
Doctor doctor = Hospital.getInstance().TryGetDoctor("1234");
Patient patient = Hospital.getInstance().TryGetPatient("2345");
```

Note: Don`t get confused about first line of code, it`s just configuring our log4j.

As you can see, when class is singleton, methods that are inside her are only available to be called through getInstance() static method. First, we created doctor and patient, so they are in HashMap now. Next step is where we stored our doctor and patient to doctor and patient references. Finally, we assigned them to each other. And last part is to delegate examinations to patient:

```
intexamination1 = Hospital.getInstance().ScheduleExamination(doctor,
patient, new Date(), laboratoryExaminations.bloodSugarLevel);

intexamination2 = Hospital.getInstance().ScheduleExamination(doctor,
patient, new Date(), laboratoryExaminations.bloodPressure);


Hospital.getInstance().FillBloodSugarLevelExamination(examination1, 12
new Date());

Hospital.getInstance().FillBloodPressureExamination(examination2, 120, 90,
32);
```

Previously you can clearly recognize what this methods do. They fill schedule examinations.
Run program.

Thank You But Can I Ask you for a Favor?

Let me say thank you for downloading and reading my book. This would be all about Java in this book. Hope you enjoyed it but you need to keep on learning to be perfect!

I'd really appreciate it if you would post a short review on Amazon.

https://www.amazon.com/Amazing-JAVA-Learn-Quickly-ebook/dp/B0737762M8/ref=zg_bs_8624232011_f_23?_encoding=UTF8&psc=1&refRID=W8F9B0G980WACSXY1G8P

I read all the reviews personally so that I can continually write what you need. Thanks.