Prepare for Technical Interviews

[1 Basic](#h.iq1h0wodhig5)

[1.1 Mi az enkapszuláció? (TO BE APPROVED)](#h.mxdq5uq5c62p)

[1.2 Öröklődés(TO BE APPROVED)](#h.1p2pogyj4ul6)

[1.3 Mi az polimorfizmus?(TO BE APPROVED)](#h.wwu8wb708bj1)

[1.4 Különbség interfész és az absztrakt osztály között?(TO BE APPROVED)](#h.5cdlqb2f5e2l)

[1.5 Lehet-e protected láthatóságú metódus egy interfészen?](#h.5pvt7cab11c5)

[1.6 Lehet-e olyan, hogy egy absztrakt osztálynak nincsen absztrakt metódusa?](#h.u1d0or1a6m37)

[1.7 Lehet interfészen futtatható kód?(TO BE APPROVED)](#h.456t5bsxjnxa)

[1.8 Mire jó a generic?(TO BE APPROVED)](#h.l8tg6mihexsz)

[1.9 Mire jók a tervezési minták?(TO BE APPROVED)](#h.j5lu0063r2zl)

[1.10 Singleton pattern vs static metódusokat tartalmazó osztály (Static class)?](#h.skska9ukvbb)

[1.11 Hol fordulhat elő a final kulcsszó?(TO BE APPROVED)](#h.r19u99h9mpwx)

[1.12 Mire jó, hogy final egy paraméter?](#h.p4nmse3pqj9p)

[1.13 Static szó hány helyen lehet?](#h.yrsorhcr9d22)

[1.14 Transient (TO BE APPROVED)](#h.9udqjswijlht)

[1.15 Volatile](#h.55gtf4oxe0jx)

[1.16 Autoboxing](#h.giemzi63rzwk)

[1.17 Mi az immutable java-ban(TO BE APPROVED)](#h.32nagz6nwb9a)

[1.18 Tervezési minták javaban(TO BE APPROVED)](#h.ixnh6w51v3vk)

[1.19 String kezelés javaban(TO BE APPROVED)](#h.vbsvr9ni7kah)

[1.20 Különbség a StringBuffer és StringBuilder között](#h.43kqqxwqpy7u)

[1.21 Equals és hashcode(TO BE APPROVED)](#h.t9gdfep9qhpx)

[1.22 Checked vs. unchecked exceptions (TO BE APPROVED)](#h.2ji5tskqakbm)

[1.23 Java 8 új feature-ei(TO BE APPROVED)](#h.2d171aso0aqy)

[1.24 Paraméter átadás javaban(TO BE APPROVED)](#h.66ho2l3300xj)

[1.25 Annotáció hivatalos definíciója](#h.1d1939now46t)

[1.26 Dependency injection(TO BE APPROVED)](#h.f56u37rhk4r)

[1.27 Dependency injection vs Dependency inversion (TO BE APPROVED)](#h.agcfiewyffq1)

[1.28 Szál megállítás(TO BE APPROVED)](#h.4xckxe6ifx2q)

[1.29 Egyik szál a másik megállítása után induljon el(TO BE APPROVED)](#h.6x44a4mx1u8)

[1.30 Javaban, hogyan kaphatunk out of memory exceptiont](#h.7im1964n6se1)

[1.31 Hogy lehet java-ban memory leak(TO BE APPROVED)](#h.sx96xfvegbkf)

[1.32 Garbage collector, hogyan működik?(TO BE APPROVED)](#h.w7jk5p7fbzhh)

[1.33 Hogyan találnánk meg egy memory leaket?](#h.7c4zjc86vjwu)

[1.34 Hogyan működik a generáció alapú garbage collection?](#h.vvpdl0e7tpii)

[1.35 Set és Lista közötti különbség(TO BE APPROVED)](#h.szdoye9rjuno)

[1.35.1 TreeSet](#h.4lto7jm66pgs)

[1.35.2 LinkedHashSet](#h.evqi8ki9vfqu)

[1.36 Mikor használnálj LinkedList-et és mikor ArrayList-et?](#h.4tcmpmj029or)

[1.37 Mire jó a permGen memória terület?(TO BE APPROVED)](#h.5zsai5q601ne)

[1.38 JIT?(TO BE APPROVED)](#h.2fx6syctswfv)

[1.39 Referencia típusok java-ban?(TO BE APPROVED)](#h.8ujl59342hdz)

[1.40 XSD és DTD(TO BE APPROVED)](#h.clcc9rnm2aii)

[1.41 XPath(TO BE APPROVED)](#h.a9uia5l33c4t)

[1.42 XSL, XSLT(TO BE APPROVED)](#h.hmgx0vm6ao49)

[1.43 DOM és SAX közötti különbség (TO BE APPROVED)](#h.uthudgk60uv3)

[1.44 SAX vs TAX(TO BE APPROVED)](#h.impnsi3kephc)

[1.45 Mi az a session?(TO BE APPROVED)](#h.buenr7nv7s0z)

[1.46 Mi az a cookie?(TO BE APPROVED)](#h.96fwigcv2yzz)

[1.47 Mi az AJAX(TO BE APPROVED)](#h.wi6peffn39qd)

[1.48 Mire jó az AJAX(TO BE APPROVED)](#h.byzoanb22nuv)

[1.49 Írunk-e scriptleteket JSP-ben(TO BE APPROVED)](#h.fdtf1nf2pzer)

[1.50 Attribútum scope-ok servletek és jsp-k esetén(TO BE APPROVED)](#h.365g2mc7bnvc)

[1.51 UNIT teszt mire jó?(TODO)](#h.zfxyyzp7ahqg)

[1.52 STUB vs SPY vs MOCK (TO BE APPROVED)](#h.tdvpizqhthsq)

[1.53 MOCK(TODO)](#h.q7jq0cd1wpi5)

[1.54 SPY(TODO)](#h.ab3ijfsby141)

[1.55 STUB(TODO)](#h.ab3ijfsby141)

[1.56 SPRING? (Lagalább négy feature)(TO BE APPROVED)](#h.yfbxrqcjikl2)

[1.57 Open-source framework-ök? (Minden open source amit használtunk) JUnit, TestNG](#h.sp3lkefkelg6)

[1.58 Final vs Finally vs Finalized(TO BE APPROVED)](#h.rtxwr12vc2ax)

[2 Conceptual](#h.ycipptra6id5)

[2.1 SOLID(in details) - Very important! (TO BE APPROVED)](#h.id9v1g6qwm99)

[2.2 DRY (TO BE APPROVED) - Don’t repeat yourself](#h.ru40as6cd51z)

[2.3 GRASP](#h.n28z3oj2o1fa)

[2.4 KISS - Keep it simple, stupid](#h.9f5llvshpfuz)

[2.5 YAGNI (TO BE APPROVED) - You ain't gonna need it](#h.kcd9023vurqs)

[2.6 What is the advantage of dependency injection? (TO BE APPROVED)](#h.e1phrp8z6mmp)

[2.7 LoD - Law of Demeter (TO BE APPROVED)](#h.qme21m8hu187)

[3 GIT](#h.umwlbjpp4ejg)

[3.1 What does Git rebase do? (TO BE APPROVED)](#h.webq9wvxyxw7)

[3.2 What does Git reset do? (TO BE APPROVED)](#h.khuurpna1irr)

[4 JAVA](#h.9njk1ho8qk6d)

[4.1 How can you serialize objects? (TO BE APPROVED)](#h.dv3f8ym74orn)

[4.2 How can you compare two objects? (TO BE APPROVED)](#h.iyz5k6m57c9u)

[4.3 Access modifiers (TO BE APPROVED)](#h.cf3gs0bkslnr)

[4.4 Protected vs default](#h.jy2cf4j6a2rg)

[4.5 What is stored on the stack? (TO BE APPROVED)](#h.jov5hrvwnmtv)

[5 Multithread](#h.tpk35ioj1ihb)

[5.1 What concurrent collections do you know?](#h.aoc2j1rkn3uo)

[5.2 What does putIfAbsent (ConcurrentHashMap) method do? (TO BE APPROVED)](#h.pyf8wflyuohm)

[5.3 Increment a number thread-safely (synchronized methods, volatile, AtomicInteger)](#h.n86ki9xq7zit)

[5.4 What may be the consequences of the lack of synchronization?(TO BE APPROVED)](#h.pm0a2hwrxc4z)

[5.5 Unexpected exceptions (ConcurrentModificationException)](#h.20195ak9ddxp)

[5.6 What is the difference between synchronizing a method or the content of the method?](#h.qyys3fwvze2s)

[5.7 How does ConcurentHashMap work?(TO BE APPROVED)](#h.ugvpez9zmedu)

[6 Spring](#h.uqlh5aq6jj6i)

[6.1 How can you make a factory in Spring?(TO BE APPROVED)](#h.4z07w3ftfw1g)

[6.2 How can you manipulate a Spring bean during creation?(TO BE APPROVED)](#h.7yfh89sqj86y)

[6.3 Spring bean lifecycle(TO BE APPROVED)](#h.9w2z2vb6gcpx)

[7 JPA](#h.23m64llprxwd)

[7.1 JPA ManyToMany/OneToMany relationships.(TO BE APPROVED)](#h.5m91vkqorl3t)

[7.2 Difference between detached and unmanaged entities(TO BE APPROVED)](#h.8v5wxok0wfpq)

[7.3 Caching with JPA (TO BE APPROVED)](#h.izx2n4x9hl7p)

[7.3.0.1 JPA Level 1 Cache](#h.u1cvpate3g5i)

[7.3.0.2 JPA Level 2 Cache](#h.nxqvw6ncg78k)

[7.4 Tell if one can bypass JPA caching mechanism by using JPQL (TO BE APPROVED)](#h.4sqlvbhqk3qq)

[8 JMS](#h.r48k7ca0o6i)

[8.1 JMS queue (TO BE APPROVED)](#h.sk6dxwgu9qqu)

[8.2 JMS topic (TO BE APPROVED)](#h.fxe2z66l8qvp)

[8.3 JMS mdb (TO BE APPROVED)](#h.h2bsaqn9hjmi)

[9 SQL (ejtsd: szíkvel)](#h.650ug2z81qz)

[9.1 Basic SQL (TODO)](#h.x7ecm5wig1or)

[9.2 SQL: Explain difference between where and having (TO BE APPROVED)](#h.5fqpf3zeqm5e)

[9.3 Synchronize / lock / reentrant lock in details (TO BE APPROVED)](#h.djmgf6ab3erw)

# 1 Basic

## 1.1 Mi az enkapszuláció? (TO BE APPROVED)

Az összetartozó adattagok és metódusok egységbe zárása

It means the encapsulation of the object’s state and the relating state-operations (i.e.: behavior).

(Attila)

Encapsulation refers to the bundling of data with the methods that operate on that data. [[source]](http://en.wikipedia.org/wiki/Encapsulation_(object-oriented_programming)#General_definition)

Encapsulation can be described as a protective barrier that prevents the code and data being randomly accessed by other code defined outside the class. Access to the data and code is tightly controlled by an interface.

The main benefit of encapsulation is the ability to modify our implemented code without breaking the code of others who use our code. With this feature Encapsulation gives maintainability, flexibility and extensibility to our code.

## 1.2 Öröklődés(TO BE APPROVED)

Osztályok vagy interfészek közti reláció ami engedi a kód újrahasznosítását. A gyermek osztály megörökli a szülő attribútumait, metódusait.

Definition: Inheritance is a relation between classes or interfaces, which allows code reusability. When a new class is derived from an existing class, it inherits attributes (fields) and functionality (methods) from that so-called superclass. In addition, new members can be added, or the inherited members can be overridden.

At the top of the class-hierarchy, Object is the most general of all classes. That is, classes near the bottom of hierarchy provide more specialized behaviour.

We can talk about single and multiple inheritance depending on the number of allowed entities to inherit from.

Classes can inherit from one superclass, but can implement multiple interfaces.

Interfaces can inherit from one superinterface and can’t implement other interfaces???. **JAVA 8-ban lehet**

(Attila + Safi)

As succinctly described [here](http://www.linuxtopia.org/online_books/programming_books/thinking_in_java/TIJ309_006.htm), overriding private methods in Java is invalid because a parent class's private methods are "automatically final, and hidden from the derived class". You can define yours though.

ÜRES INTERFACEK-ET LEHET ÖRÖKÖLNI BÁRMENNYIT, DE NINCS ÉRTELME. TÖBB NEMÜRES INTERFACE ESETÉN A GUI HIBÁT JELEZ. (diamond of death)

|  |  |
| --- | --- |
|  | **Multiple inheritance of *implementations* is not allowed. Components can inherit multiple interfaces, though.**  **Inheriting multiple interfaces isn't problematic, since you're simply defining new method signatures to be implemented. It's the inheritance of multiple copies of functionality that is traditionally viewed as causing problems, or at the very least, confusion** |

## 1.3 Mi az polimorfizmus?(TO BE APPROVED)

Egy objektum azon képessége amitől több formát is felvehet, (metódus override, overload)

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object. Do the same thing with different methods.

* **Method Override** is when you redefine a method (same name, same parameters, same return type) that has already been defined in a parent class. (@Override)
* **Method Overload** is when you define two methods with the same name in the same class, distinguished by their signatures (aka name+parameters).
* The **Generic Programming** paradigm is an approach to software decomposition whereby fundamental requirements on types are abstracted from across concrete examples of algorithms and data structures and formalised as [concepts](http://en.wikipedia.org/wiki/Concept_(generic_programming)).

(Névtelen iguána + Safi)

## 1.4 Különbség interfész és az absztrakt osztály között?(TO BE APPROVED)

|  |  |
| --- | --- |
| **Interface** | **Abstract class** |
| “should do” | “is a” |
| multiple implementation | single extension |
| all methods are public | methods can have any access modifier |
| no method implementation (In java8 there is) | possible method implementations |
| [Java 8 introduces default and static methods to interfaces](https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html)  ([these methods must be implemented in the interface](https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html)) | |

Runnable code: static init ~~block~~ is possible

|  |
| --- |
|  |
| methods  (and constants (public final static)),  [but is not advised](http://en.wikipedia.org/wiki/Constant_interface)) | methods, constructors, constants, variables |

## 1.5 Lehet-e protected láthatóságú metódus egy interfészen?

There can be only public methods on an interface. Every method is by default public, its not a must to declare it public.

## 1.6 Lehet-e olyan, hogy egy absztrakt osztálynak nincsen absztrakt metódusa?

Yes. Declaring a class as abstract only means that it cannot be instantiated on its own.

Declaring a method abstract means that subclasses have to provide an implementation for that method. Every abstract method must be in abstract classes, but its not required for an abstract class to have abstract methods.

## 1.7 Lehet interfészen futtatható kód?(TO BE APPROVED)

Yes, e.g we can use static initialization or an inner class (was used before static methods) or enum in an interface. (Java8 adds default and static methods as well)

## 1.8 Mire jó a generic?(TO BE APPROVED)

It reduces the code repetition because it allows us to run the same code with different type of objects. (It came with JDK 5.0) We can use generic at class, interface, method, constructor declaration with type variables.

## 1.9 Mire jók a tervezési minták?(TO BE APPROVED)

In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

Main types: creational, structural, behavioral. ()

## 1.10 Singleton pattern vs static metódusokat tartalmazó osztály (Static class)?

1. Static class: methods are bound on compile time, therefore it provides better **performance**
2. Singleton: methods can be **overridden** while static methods cannot
3. Singleton: it is easier to **test**, because static methods are hard to **mock**
4. Singleton: it can contain fields, so it can be stateful (????)

(E.Ádám)

## 1.11 Hol fordulhat elő a final kulcsszó?(TO BE APPROVED)

1. Final member variable (field): must be initialized at declaration time or inside the constructor
2. Final local variable: can only be assigned once (they can be accessed from anonymous inner classes)
3. Final method: it cannot be overridden
4. Final class: no inheritance from this class
5. Final method parameter: the reference on the parameter cannot be changed(note that if you have a final object, you can still change the attributes of the object)

## 1.12 Mire jó, hogy final egy paraméter?

It ensures that you can’t change the object reference in the parameter. It doesn’t give you any guarantee about changing the object itself.

## 1.13 Static szó hány helyen lehet?

Static keyword may appear in the following contexts:

* **field**: defines a *class variable* that is common across all class instances and can be accessed without instantiating the class
* **method**: defines a *class method* that can be invoked without instantiating the class. Only class variables are accessible from a class method.
* **class**: defines a *static nested class* which can be accessed without instantiating the enclosing class. Only the enclosing class’ static members are accessible from a static nested class. In effect, a static nested class is behaviorally a top-level class that has been nested in another top-level class for packaging convenience.
* **block**: defines a *static initialization block* used to initialize class variables (static fields). The runtime system guarantees that static initialization blocks are called in the order that they appear in the source code.
* **import**: The *static import* construct allows static members (fields and methods) defined in a class to be used in Java code without specifying the class in which the member is defined. It helps avoiding [Constant Interface Antipattern](http://en.wikipedia.org/wiki/Constant_interface)

(Attila + András)

## 1.14 Transient (TO BE APPROVED)

Non serializable variables.

Variables may be marked transient to indicate that they are not part of the persistent state of an object.

(E.Ádám)

Variables marked transient are skipped during serialization. (Krisz)

## 1.15 Volatile

A field may be declared volatile, in which case the Java Memory Model ensures that all threads see a consistent value for the variable. Writing to a volatile field happens before and synchronizes with every subsequent read of that field.

(E.Ádám + András)

## 1.16 Autoboxing

Automatic conversion that the Java compiler makes between the primitive types and their corresponding object **wrapper** classes. For example, converting an int to an Integer, a double to a Double, and so on. If the conversion goes the other way, this is called *unboxing*.

<https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html>

## 1.17 Mi az immutable java-ban(TO BE APPROVED)

An object is considered immutable if its state cannot change after it is constructed.

Since they cannot change state, they cannot be corrupted by thread interference or observed in an inconsistent state.

In practice:

* Fields are private and final
* No set methods and no methods that can change the state of the object

(E.Ádám)

* Class is final too. (Krisz)

## 1.18 Tervezési minták javaban(TO BE APPROVED)

There are lots of Design patterns already implemented in Java, the following table shows quite a few of these:

|  |  |
| --- | --- |
| Builder | StringBuilder, Calendar.Builder |
| Factory Method | Object.toString(), Iterable.iterator() |
| Static Factory Method | Calendar.getInstance(), NumberFormat.getInstance(), Charset.forName() |
| Singleton | Runtime, GraphicsEnvironment |
|  | |
| Adapter | Arrays.asList(), HashSet, InputStreamReader/Writer |
| Bridge | DriverManager |
| Decorator | Collections.synchronizedSet(...), ObjectInputStream, FileInputStream |
| Facade | JOptionPane, SLF4J |
| Proxy | RMI, Proxy |
|  | |
| Command | Runnable, Callable |
| Iterator | Iterable, Iterator |
| Observer | Observer, Observable |
| Strategy | Comparator, BufferStrategy |
| Template Method | HttpServlet, OutputStream |

## 1.19 String kezelés javaban(TO BE APPROVED)

String is immutable. It uses a pool that contains the once used strings and if you use one a second time it will use the string from the pool. You can use the intern method to add strings to this pool.

String literals that get automatically interned / added to the String pool. Strings that are created by an application are not interned ... unless your application explicitly calls String.intern().

## 1.20 Különbség a StringBuffer és StringBuilder között

StringBuffer is synchronized.

String buffers are safe for use by multiple threads. The methods are synchronized where necessary. For this reason, its slower, consider using StringBuilder when thread safety is not an issue.

## 1.21 Equals és hashcode(TO BE APPROVED)

If 2 objects are equal, so is their hashcodes, however if 2 objects have the same hashcode they might not be equal.

Formally: If obj1 and obj2 are two objects, then the contract is:

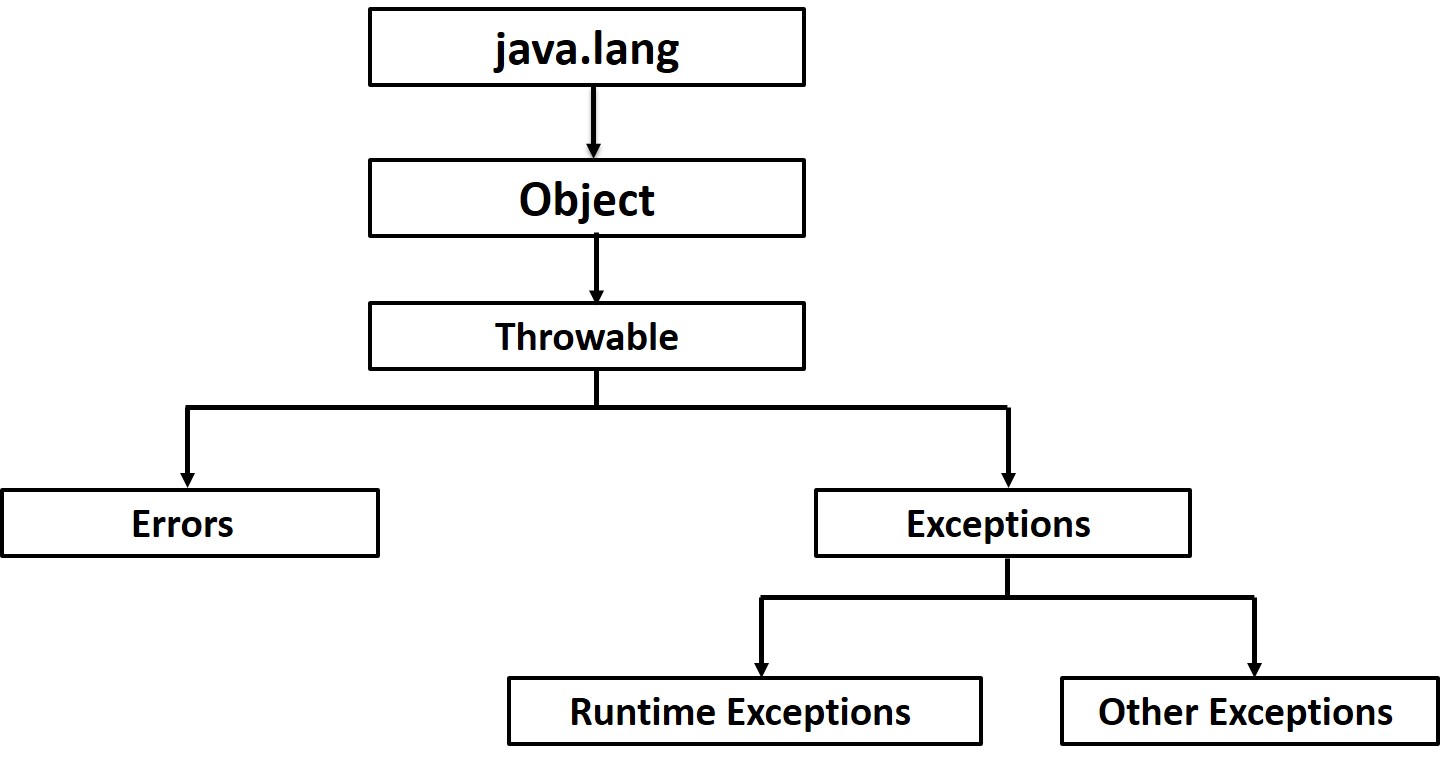
* whenever obj1.equals(obj2) **->** obj1.hashCode() == obj2.hashCode() must **always** met, but
* whenever obj1.hashCode() == obj2.hashCode() **->** obj1.equals(obj2) is **not** necessarily true

So the consequence is: each time you override the equals() method, you **must** override hashCode() as well to guarantee their contract.

Reference: [link](http://www.programcreek.com/2011/07/java-equals-and-hashcode-contract/)

(kiegészítés: Attila)

## 1.22 Checked vs. unchecked exceptions (TO BE APPROVED)



Checked exceptions:

* must be caught,
* they are mainly related to business logic,
* technically: each subclass of Exception class, except RuntimeException and its subclasses

Unchecked exceptions:

* can be caught,
* they indicate the possibility of programmer mistakes,
* technically: subclasses of RuntimeException class and subclasses of Error class

(Attila, unchecked javítva: Isti)

## 1.23 Java 8 új feature-ei(TO BE APPROVED)

<http://www.java8.org/>

**Interface updates:** **default** and **static methods**, that must be implemented in the interface. (must still be public)

**Lambda expressions:** [Lambda expressions and Stream API](https://vimeo.com/98919321)

Lambda expressions let you express instances of single-method interfaces (referred to as functional interfaces) more compactly. (instead of using anonymous classes)

The main syntax of a lambda expression is “parameters -> body”. The compiler can usually use the context of the lambda expression to determine the functional interface being used and the types of the parameters. There are four important rules to the syntax:

* Declaring the types of the parameters is optional.
* Using parentheses around the parameter is optional if you have only one parameter.
* Using curly braces is optional (unless you need multiple statements).
* The “return” keyword is optional if you have a single expression that returns a value.

**Method references:** easy-to-read lambda expressions for methods that already have a name.

| **Type** | **Example** | **Syntax** |
| --- | --- | --- |
| 1. Reference to a static method | ContainingClass::staticMethodName | Class::staticMethodName |
| 2. Reference to a constructor | ClassName::new | ClassName::new |
| 3. Reference to an instance method of an arbitrary object of a particular type | ContainingType::methodName | Class::instanceMethodName |
| 4. Reference to an instance method of a particular object | containingObject::instanceMethodName | object::instanceMethodName |

**Stream API:** [Programming with Streams in Java 8](https://vimeo.com/124034512)

[**Optional**](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html)**:** erm.. read the linked stuff...

[**MetaSpace**](https://blogs.oracle.com/poonam/entry/about_g1_garbage_collector_permanent)instead of PermGen: The main difference between these two is that MetaSpace is allocated from native OS memory not from the JVM’s. This means that MetaSpace has the ability to auto increase its size if needed upto -XX:MaxMetaspaceSize or, if that is not set, OS heap size.

[**Date-Time API**](http://docs.oracle.com/javase/8/docs/technotes/guides/datetime/index.html)**:** LocalTime,LocalDate,LocalDateTime stb… (finally immutable)

**Annotations**: Annotations can also be applied to the use of types. examples:

1 *// Class instance creation:*  
 2 new @Interned RocketShip();  
 3   
 4 *// Type cast:*  
 5 notNullString = (@NonNull String) str;  
 6   
 7 *// implements clause:*  
 8 class ImmutableSet<T> implements  
 9 @Readonly Set<@Readonly T> { ... }  
10   
11 *// Thrown exception declaration:*  
12 void launchRocket() throws  
13 @Critical FireException { ... }

(Safi)

## 1.24 Paraméter átadás javaban(TO BE APPROVED)

Java is always *pass-by-value*. The difficult thing to understand is that Java passes objects (instances of non-primitive classes) as references and those *references* are passed by value.

## 1.25 Annotáció hivatalos definíciója

Annotations, a form of metadata, provide data about a program that is not part of the program itself. Annotations have no direct effect on the operation of the code they annotate.

Annotations have a number of uses, among them:

* **Information for the compiler** — Annotations can be used by the compiler to detect errors or suppress warnings.
* **Compile-time and deployment-time processing** — Software tools can process annotation information to generate code, XML files, and so forth.
* **Runtime processing** — Some annotations are available to be examined at runtime.

## 1.26 Dependency injection(TO BE APPROVED)

Dependency injection is the passing of a dependency (a service) to a dependent object (a client). The service is made part of the client's state. With dependency injection, the client which uses a module or service doesn't need to know all its details, and typically the module can be replaced by another one of similar characteristics without altering the client.

## 1.27 Dependency injection vs Dependency inversion (TO BE APPROVED)

Dependency injection is an alternative/option to solve the problem of Dependency inversion. Other alternative: Service locator (JNDI)

(Attila)

## 1.28 Szál megállítás(TO BE APPROVED)

The proper way to stop a thread is to introduce a flag variable, which informs the thread that it need to terminate. Don’t use the deprecated methods on Thread like destroy(), stop(), suspend(). The interrupt mechanism is implemented using an internal flag known as the *interrupt status*. Invoking Thread.interrupt sets this flag. When a thread checks for an interrupt by invoking the static method Thread.interrupted, interrupt status is cleared. The non-static isInterrupted method, which is used by one thread to query the interrupt status of another, does not change the interrupt status flag.

By convention, any method that exits by throwing an InterruptedException clears interrupt status when it does so. However, it's always possible that interrupt status will immediately be set again, by another thread invoking interrupt.

## 1.29 Egyik szál a másik megállítása után induljon el(TO BE APPROVED)

You use the join() method on a thread for this functionality. The join() method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.

## 1.30 Javaban, hogyan kaphatunk out of memory exception-t

It's not an exception; it's an error: java.lang.OutOfMemoryError.

Thrown when the Java Virtual Machine cannot allocate an object because it is out of memory, and no more memory could be made available by the garbage collector. OutOfMemoryError objects may be constructed by the virtual machine.

Example:

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

double[] d;

try {

d = new double[Integer.MAX\_VALUE];

System.out.println("1");

} catch (final OutOfMemoryError e) {

System.out.println("2");

d = null;

}

System.out.println("3");

d = new double[Integer.MAX\_VALUE];

System.out.println("4");

}

prints the following:

2

3

Exception in thread "main" java.lang.OutOfMemoryError: Requested array size exceeds VM limit

## 1.31 Hogy lehet java-ban memory leak(TO BE APPROVED)

The most common memory leak reasons in java are

* mutable static fields: they are never garbage-collected
* complex circular references
* local variables in threads: they are only garbage-collected after the thread dies, and every thread instance has its own instance of that variable
* wrong implementation of equals/hashcode: if you don’t fulfill the contract between them, its possible to add an object to a collection and never find it again, but as the collection has its reference, it cannot be garbage-collected
* Java Native Interface calls: any java object created by native method will be referenced until the native method returns.

## 1.32 Garbage collector, hogyan működik?(TO BE APPROVED)

All objects are allocated on the heap area managed by the JVM. Every item that the developer uses is treated this way, including class objects, static variables, and even the code itself. As long as an object is being referenced, the JVM considers it alive. Once an object is no longer referenced and therefore is not reachable by the application code, the garbage collector removes it and reclaims the unused memory. Every object tree must have one or more root objects. As long as the application can reach those roots, the whole tree is reachable.

There are four kinds of GC roots in Java:

**Local variables** are kept alive by the stack of a thread. This is not a real object virtual reference and thus is not visible. For all intents and purposes, local variables are GC roots.

**Active Java threads** are always considered live objects and are therefore GC roots. This is especially important for thread local variables.

**Static variables** are referenced by their classes. This fact makes them de facto GC roots. **Classes** themselves can be garbage-collected, which would remove all referenced static variables. This is of special importance when we use application servers, [OSGi containers](http://www.wikipedia.com/osgi) or class loaders in general.

**JNI References** are Java objects that the native code has created as part of a JNI call. Objects thus created are treated specially because the JVM does not know if it is being referenced by the native code or not.

To determine which objects are no longer in use, the JVM intermittently runs what is very aptly called a mark-and-sweep algorithm. Its a two step process:

1. The algorithm traverses all object references, starting with the GC roots, and marks every object found as alive.
2. All of the heap memory that is not occupied by marked objects is reclaimed. It is simply marked as free, essentially swept free of unused objects.

Garbage collection is intended to remove the cause for classic memory leaks: unreachable-but-not-deleted objects in memory. However, this works only for memory leaks in the original sense. It's possible to have unused objects that are still reachable by an application because the developer simply forgot to dereference them. Such objects cannot be garbage-collected.

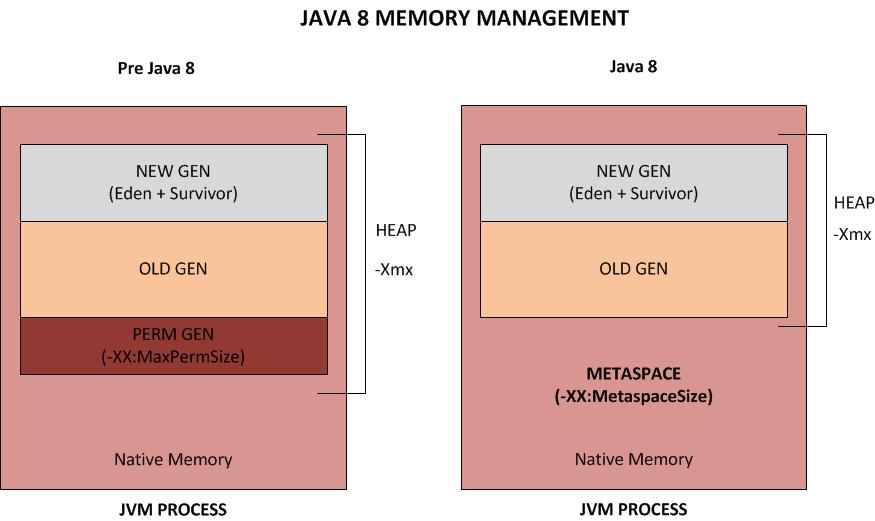
## 1.33 Hogyan találnánk meg egy memory leaket?

By using jconsole, jvisualvm or a profiler tool.

## 1.34 Hogyan működik a generáció alapú garbage collection?

* **Weak generational hypothesis:**
  + Most objects soon become unreachable
  + References from old objects to young objects only exists in small numbers
* **Types of generations:**
  + **Young**
    - Most of the newly created objects are located here.
    - When objects disappear from this area, we say a "**minor GC**" has occurred.
    - Parts of the Young-generation: Eden, Survivor1, Survivor2
  + **Old**
    - Objects that did not become unreachable and survived from the young generation are copied here.
    - Generally larger than the young generation
    - GC occurs less frequently than in the young generation
    - When objects disappear from the old generation, we say a "**major GC**" (or a "**full GC**") has occurred
  + **Permanent**
    - Also called the "**method area**" and it stores classes or interned character strings
    - **Not** for objects that survived from the old generation
    - GC that took place here is still counted as a **major GC**

(E.Ádám) <http://www.cubrid.org/blog/dev-platform/understanding-java-garbage-collection/>



## 1.35 Set és Lista közötti különbség(TO BE APPROVED)

|  |  |  |
| --- | --- | --- |
|  | **Set** | **List** |
| Ordered | Depends on implementation (TreeSet, LinkedHashSet) | Yes |
| Element Duplication | No | Yes |
| Positional Access | No | Yes |

### [1.35.1 TreeSet](http://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html)

* Elements are ordered using their [natural ordering](http://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html), or by a [Comparator](http://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html) provided at set creation time, depending on which constructor is used.

### [1.35.2 LinkedHashSet](http://docs.oracle.com/javase/7/docs/api/java/util/LinkedHashSet.html)

* Iteration ordering
* Order is *not* affected if an element is *re-inserted* ( *just runs the contains() method*)

## 1.36 Mikor használnál LinkedList-et és mikor ArrayList-et?

If there is a requirement of **frequent addition** and deletion in application then **LinkedList** is a better choice.

**Reason:** LinkedList’s each element maintains two pointers (addresses) which points to the both neighbor elements in the list. Hence removal only requires change in the pointer location in the two neighbor nodes (elements) of the node which is going to be removed. While In ArrayList all the elements need to be shifted to fill out the space created by removed element.

If there are less add and remove operations and more **search** operations requirement, **ArrayList** would be your best bet.

**Reason:** ArrayList maintains index based system for its elements as it uses array data structure implicitly which makes it faster for searching an element in the list. On the other side LinkedList implements doubly linked list which requires the traversal through all the elements for searching an element.

## 1.37 Mire jó a permGen memória terület?(TO BE APPROVED)

~~JVM Permanent Generation is where most current JVM store class files (some of which are loaded when your app starts, some dynamically as your app runs). For healthy apps the PermGen usage can grow a bit - but should then stop growing at some point, if it does grow too much - you will see an out of memory exception in your application which mentions Permgen space (at which point your application will be restarted).~~

The permanent generation is the area of heap that holds all the reflective data of the virtual machine itself, such as class and method objects (also called “method area” in The Java Virtual Machine Specification). This area also holds internalized strings (aka. String pool).

Since Java8 permGen was replaced by [MetaSpace](https://blogs.oracle.com/poonam/entry/about_g1_garbage_collector_permanent). The main difference between these two is that MetaSpace is allocated from native OS memory not the JVM’s. This means that MetaSpace has the ability to auto increase its size if needed upto -XX:MaxMetaspaceSize or, if that is not set, OS memory size. (Safi)

## 1.38 JIT?(TO BE APPROVED)

JIT stands for *Just-In-Time compilation* which allows the Java application to start and run while the code that is generated is not highly optimized for the platform. JIT comes into use whenever a Java method is called, and it compiles the bytecode of that method into native machine code, thereby compiling it “just in time” to execute. After a method is compiled, the JVM calls that method’s compiled code directly instead of trying to interpret it, which makes the running of the application fast. The JVM later optimizes the code during the application’s run.

## 1.39 Referencia típusok java-ban?(TO BE APPROVED)

A *reference object* encapsulates a reference to some other object so that the reference itself may be examined and manipulated like any other object. Three types of reference objects are provided, each weaker than the last: *soft*, *weak*, and *phantom*. Each type corresponds to a different level of reachability, as defined below. Soft references are for implementing memory-sensitive caches, weak references are for implementing canonicalizing mappings that do not prevent their keys (or values) from being reclaimed, and phantom references are for scheduling pre-mortem cleanup actions in a more flexible way than is possible with the Java finalization mechanism.

Going from strongest to weakest, the different levels of reachability reflect the life cycle of an object. They are operationally defined as follows:

* An object is ***strongly*** *reachable* if it can be reached by some thread without traversing any reference objects. A newly-created object is strongly reachable by the thread that created it.
* An object is ***softly*** *reachable* if it is not strongly reachable but can be reached by traversing a soft reference. (*SoftReference<T>)*
* An object is ***weakly*** *reachable* if it is neither strongly nor softly reachable but can be reached by traversing a weak reference. When the weak references to a weakly-reachable object are cleared, the object becomes eligible for finalization. (*WeakReference<T>*)
* An object is ***phantom*** *reachable* if it is neither strongly, softly, nor weakly reachable, it has been finalized, and some phantom reference refers to it. (*PhantomReference<T>*)
* Finally, an object is ***unreachable***, and therefore eligible for reclamation, when it is not reachable in any of the above ways.

Source: [java.lang.ref](https://docs.oracle.com/javase/7/docs/api/java/lang/ref/package-summary.html)

If the garbage collector determines that an object is softly reachable, it **may** clear atomically all soft references to the object, in the case that it finds that memory is running low, or at its own discretion. But if the garbage collector determines that an object is weakly reachable, it **will** clear atomically all weak references to the object. This is the major difference between weak and soft references.

| **Soft vs Weak vs Phantom References** | | | | |
| --- | --- | --- | --- | --- |
| **Type** | **Purpose** | **Use** | **When GCed** | **Implementing Class** |
| Strong Reference | An ordinary reference. Keeps objects alive as long as they are referenced. | normal reference. | Any object not pointed to can be reclaimed. | default |
| Soft Reference | Keeps objects alive provided there’s enough memory. | to keep objects alive even after clients have removed their references (memory-sensitive caches), in case clients start asking for them again by key. | After a first gc pass, the JVM decides it still needs to reclaim more space. | java.lang.ref.SoftReference |
| Weak Reference | Keeps objects alive only while they’re in use (reachable) by clients. | Containers that automatically delete objects no longer in use. | After gc determines the object is only weakly reachable | java.lang.ref.WeakReference java.util.WeakHashMap |
| Phantom Reference | Lets you clean up after finalization but before the space is reclaimed (replaces or augments the use of finalize()) | Special clean up processing | After finalization. | java.lang.ref.PhantomReference |

## 1.40 XSD és DTD(TO BE APPROVED)

XSD stands for *XML Schema Definition*, which offers facilities for describing the structure and constraining the contents of XML documents, including those which exploit the XML Namespace facility. The schema language, which is itself represented in an XML vocabulary and uses namespaces, substantially reconstructs and considerably extends the capabilities found in XML *Document Type Definition*s (DTDs).

DTD is a set of markup declarations that provide a grammar for a class of documents.

A Document Type Definition (DTD) defines the legal building blocks of an XML document. It defines the document structure with a list of legal elements and attributes. A DTD can be declared inline inside an XML document, or as an external reference.

1.41 The critical difference between DTDs and XML Schema is that XML Schema utilize an XML-based syntax, whereas DTDs have a unique syntax held over from SGML DTDs. XML schemas define datatypes for elements and attributes while DTD doesn't support datatypes. XML schemas allow support for namespaces while DTD does not. XML schemas define number and order of child elements, while DTD does not. XML schemas can be manipulated on your own with XML DOM but it is not possible in case of DTD.

## 1.41 XPath(TO BE APPROVED)

XPath is a language for addressing parts of an XML document, designed to be used by both XSLT and XPointer.

**XPath**, the **XML Path Language**, is a [query language](http://en.wikipedia.org/wiki/Query_language) for selecting [nodes](http://en.wikipedia.org/wiki/Node_%28computer_science%29) from an [XML](http://en.wikipedia.org/wiki/XML) document. In addition, XPath may be used to compute values (e.g., [strings](http://en.wikipedia.org/wiki/String_%28computer_science%29), numbers, or [Boolean](http://en.wikipedia.org/wiki/Boolean_datatype) values) from the content of an XML document. The XPath language is based on a tree representation of the XML document, and provides the ability to navigate around the tree, selecting nodes by a variety of criteria.

## 1.42 XSL, XSLT(TO BE APPROVED)

XSL is a family of recommendations for defining XML document transformation and presentation. It consists of three parts:

**XSL Transformations (XSLT):** a language for transforming XML;

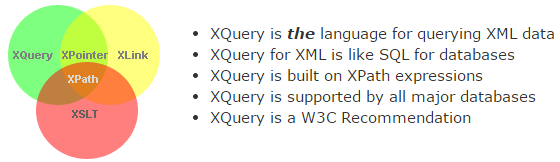
**The XML Path Language (XPath):** an expression language used by XSLT (and many other languages) to access or refer to parts of an XML document;

**XSL Formatting Objects (XSL-FO):** an XML vocabulary for specifying formatting semantics.

**XSLT (Extensible Stylesheet Language Transformations)** is a language for transforming XML documents into other XML documents, or other formats such as HTML for web pages, plain text or into XSL Formatting Objects, which may subsequently be converted to other formats, such as PDF PostScript and PNG.

The original document is not changed; rather, a new document is created based on the content of an existing one. Typically, input documents are XML files, but anything from which the processor can build an XQuery and XPath Data Model can be used, for example relational database tables, or geographical information systems.

XSLT is a Turing-complete language, meaning it can specify any computation that can be performed by a computer.

(Attila: below is just to see relationships between the many X… stuff. Ref.: [link](http://www.w3schools.com/xquery/))

## 1.43 DOM és SAX közötti különbség (TO BE APPROVED)

Both DOM and SAX parser are extensively used to read and parse XML file in java.

|  |  |  |
| --- | --- | --- |
|  | **Simple API for XML** | **Document Object Model** |
| **Processing method** | event-based | tree-based |
| **Can update data** | no | yes |
| **Navigation in document** | serial | direct |
| **Document is loaded in memory** | partly | entirely |
| **Runtime performance** | faster | Slower (with huge XML-s) |

## 1.44 SAX vs StAX(TO BE APPROVED)

Both SAX and StAX are stream / event oriented XML parsers, but there is a subtle difference in how they work. SAX uses a "push" model, and StAX uses a "pull" model.

The SAX push model means that it is the SAX parser that calls your handler, not your handler that calls the SAX parser. The SAX parser thus "pushes" events into your handler.

The StAX pull model means that it is your "handler" class that calls the parser, not the other way around. Thus your handler class controls when the parser is to move on to the next event in the input. In other words, your handler "pulls" the XML events out of the parser. Additionally, you can stop the parsing at any point.

SAX does not have support for writing XML, while StAX has.

## 1.45 Mi az a session?(TO BE APPROVED)

The HTTP session provides a way to identify a user across more than one page request or visit to a Web site and to store information about that user.

## 1.46 Mi az a cookie?(TO BE APPROVED)

**HTTP cookies** are key/value pairs used by websites to store stateful information in the browser.

**Server side cookies** are known as "**sessions**". The website in this case stores a single cookie on the browser containing a unique Session Identifier and stores the stateful information on the server.

## 1.47 Mi az AJAX(TO BE APPROVED)

AJAX = Asynchronous JavaScript and XML.

AJAX is for exchanging data with a server, and updating parts of a web page - without reloading the whole page.

AJAX applications are browser- and platform-independent!

Krisz:

Ajax is a **group** **of** interrelated [Web development](http://en.wikipedia.org/wiki/Web_development) **techniques** used on the [**client-side**](http://en.wikipedia.org/wiki/Client-side) to create [**asynchronous**](http://en.wikipedia.org/wiki/Asynchronous_I/O) [Web applications](http://en.wikipedia.org/wiki/Web_application). (Forrás: [wiki](http://en.wikipedia.org/wiki/Ajax_%28programming%29))

## 1.48 Mire jó az AJAX(TO BE APPROVED)

AJAX is about updating parts of a web page, without reloading the whole page.

AJAX is a technique for creating fast and dynamic web pages.

AJAX allows web pages to be updated asynchronously by exchanging small amounts of data with the server behind the scenes. This means that it is possible to update parts of a web page, without reloading the whole page. Classic web pages, (which do not use AJAX) must reload the entire page if the content should change.

AJAX is based on internet standards, and uses a combination of:

* XMLHttpRequest object (to exchange data asynchronously with a server)
* JavaScript/DOM (to display/interact with the information)
* CSS (to style the data)
* XML (often used as the format for transferring data)

Example:

[Google Suggest](http://www.google.com/) is using AJAX to create a very dynamic web interface: When you start typing in Google's search box, a JavaScript sends the letters off to a server and the server returns a list of suggestions.

## 1.49 Írunk-e scriptleteket JSP-ben(TO BE APPROVED)

It is highly discouraged. (Because our hands will be broken.)

Main disadvantages of *scriptlets* are:

* **Reusability:** you can't reuse scriptlets.
* **Replaceability:** you can't make scriptlets abstract.
* **OO-ability:** you can't make use of inheritance/composition.
* **Debuggability:** if scriptlet throws an exception halfway, all you get is a blank page.
* **Testability:** scriptlets are not unit-testable.
* **Maintainability:** more time is needed to maintain mingled/cluttered/duplicated code logic.

## 1.50 Attribútum scope-ok servletek és jsp-k esetén(TO BE APPROVED)

**page**: Objects with *page scope* are accessible only within the page in which they're created. The data is valid only during the processing of the current response; once the response is sent back to the browser, the data is no longer valid. If the request is forwarded to another page or the browser makes another request as a result of a redirect, the data is also lost.

**request**: Objects with *request scope* are accessible from pages processing the same request in which they were created. Once the container has processed the request, the data is released. Even if the request is forwarded to another page, the data is still available though not if a redirect is required.

**session**: Objects with *session scope* are accessible from pages processing requests that are in the same session as the one in which they were created. A *session* is the time users spend using the application, which ends when they close their browser, when they go to another Web site, or when the application designer wants (after a logout, for instance). So, for example, when users log in, their username could be stored in the session and displayed on every page they access. This data lasts until they leave the Web site or log out.

**application**: Objects with *application scope* are accessible from JSP pages that reside in the same application. This creates a global object that's available to all pages.

Application scope uses a single namespace, which means all your pages should be careful not to duplicate the names of application scope objects or change the values when they're likely to be read by another page (this is called *thread safety*). Application scope variables are typically created and populated when an application starts and then used as read-only for the rest of the application.

## 1.51 UNIT teszt mire jó?(TODO)

Target of unit testing is always a particular class. Focus of unit testing is to test the behaviour of that class’s functions **isolatedly** from other classes. Interactions with **other** classes are out of the scope of unit testing.

The three mandatory stages of a unit test function:

* setup (~ required)
* execution (~ when)
* verification (~ then)

(Attila)

In [computer programming](http://en.wikipedia.org/wiki/Computer_programming), **unit testing** is a [software testing](http://en.wikipedia.org/wiki/Software_testing) method by which individual units of [source code](http://en.wikipedia.org/wiki/Source_code), sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use. Intuitively, one can view a unit as the smallest testable part of an application. In [procedural programming](http://en.wikipedia.org/wiki/Procedural_programming), a unit could be an entire module, but it is more commonly an individual function or procedure. In [object-oriented programming](http://en.wikipedia.org/wiki/Object-oriented_programming), a unit is often an entire interface, such as a class, but could be an individual method. Unit tests are short code fragments created by programmers or occasionally by [white box testers](http://en.wikipedia.org/wiki/White-box_testing) during the development process. It forms the basis for **component testing**.

(Krisz)

## 1.52 STUB vs SPY vs MOCK (TO BE APPROVED)

Dummy: Does nothing, and we don’t care.

Stub: Dummy+ Returns “canned” responses.

Spy: Stub + maintains an inner state (makes assertions on state)

Fake: Stub/Spy + works like a collaborator (makes assertions on state)

Mock: MAKES ASSERTIONS ON BEHAVIOUR (Fake/Spy/Stub+verify method)

|  |  |  |  |
| --- | --- | --- | --- |
|  | **STUB** | **SPY** | **MOCK** |
| Can fail test | NO | YES | YES |
| Test with indirect input (usable in Unit Test) | YES | NO (Real Objects) | YES |
| Records interactions | NO | YES | YES |
| Unit Test makes assertions on | SUT\* | SUT\* | mock |

*\*SUT = System Under Test*

(Attila: A nice article: <https://adamcod.es/2014/05/15/test-doubles-mock-vs-stub.html>)

## 1.53 MOCK(TODO)

In object-oriented programming, **mock objects** are simulated objects that mimic the behavior of real objects in controlled ways. A programmer typically creates a mock object to test the behavior of some other object, therefore useful when a real object is impractical or impossible to incorporate into a unit test. Mock objects have the same interface as the real objects they mimic, allowing a client object to remain unaware of whether it is using a real object or a mock object. Many available mock object frameworks allow the programmer to specify which, and in what order, methods will be invoked on a mock object and what parameters will be passed to them, as well as what values will be returned. Thus, the behavior of a complex object such as a network socket can be mimicked by a mock object, allowing the programmer to discover whether the object being tested responds appropriately to the wide variety of states such mock objects may be in.

## 1.54 SPY(TODO)

up

## 1.55 STUB(TODO)

up

## 1.56 SPRING? (Lagalább négy feature)(TO BE APPROVED)

* Spring Core: Inversion of Control, Dependency Injection, Bean management, i18n, event propagation, resource loading
* Spring AOP: Aspect Oriented Programming, AspectJ integration
* Spring Data: JDBC, Hibernate, JPA integration
* Spring MVC: Model-View-Controller, REST web services
* Spring Test: unit and integration testing w/ JUnit, TestNG integration, mocking

## 1.57 Open-source framework-ök? (Minden open source amit használtunk) JUnit, TestNG

* EasyMock, Mockito
* SLF4J, Logback, Log4j
* Spring Core, AOP, Data, MVC, Test
* standard Java frameworks
* (Git, Maven)
* <https://github.com/akullpp/awesome-java>

## 1.58 Final vs Finally vs Finalized(TO BE APPROVED)

* Final denotes that something cannot be changed. There can be final variables, methods, classes, method parameters.
* Finally is used in the exception handling try-catch block for defining code parts that executed always, regardless of the outcome of the try block.
* Finalize method is called by the JVM before garbage collecting the object. As garbage collection is not ensured to run in a specific time, its rarely overridden.

# 2 Conceptual

## 2.1 SOLID(in details) - Very important! (TO BE APPROVED)

* **SRP - Single responsibility principle** : Every class should have a single responsibility, and that responsibility should be entirely encapsulated by the class. The class should have one, and only one, reason to change. #YOLO
  + 1, Make the class more robust. (no anemic model please)
  + 2, Avoid coupled responsibilities.
* **OCP - Open/closed principle** : Software entities should be open for extension, but closed for modification. Reasons:
  + Changing the behaviour of a class has an impact on the clients that use this class. (Fixing bugs is ok.)
  + Write code in a fashion that adding a new functionality would involve minimal changes to existing code.

Solution: Use abstractions: abstractions are fixed and represent an unbounded group of possible behaviours.

* **LSP - Liskov substitution principle** : Derived types must be completely substitutable for their base types. Consume any implementation without changing the correctness of the system.

Conditions:

* + Method signature
  + Precondition cannot be strengthened in a subtype
  + Postconditions cannot be weakened in a subtype
  + Invariants of the supertype must be preserved in a subtype
  + History constraint
* **ISP - Interface segregation principle** : Clients should not be forced to depend on methods they do not use, favour [Role interfaces](http://martinfowler.com/bliki/RoleInterface.html) over [Header interfaces](http://martinfowler.com/bliki/HeaderInterface.html).
  + Keep a system decoupled and thus easier to refactor, change, and redeploy.
  + When clients are forced to depend upon interfaces that they don’t use, then those clients are subject to changes to those interfaces.
* **DIP - Dependency inversion principle** : High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend upon details. Details should depend upon abstractions.
  + Define abstract interfaces for low level components
  + Resolve dependencies upon object construction (or use an IoC container)

## 2.2 DRY (TO BE APPROVED) - Don’t repeat yourself

A basic strategy for reducing complexity to manageable units is to divide a system into pieces. The DRY principle states that these small pieces may only occur exactly once in your entire system.

## 2.3 GRASP

**General Responsibility Assignment Software Patterns** (or **Principles**), abbreviated **GRASP**, consists of guidelines for assigning responsibility to classes and objects in [object-oriented design](http://en.wikipedia.org/wiki/Object-oriented_design).

The different patterns and principles used in GRASP are: Controller, Creator, Indirection, Information Expert, High Cohesion, Low [Coupling](http://en.wikipedia.org/wiki/Coupling_(computer_science)), [Polymorphism](http://en.wikipedia.org/wiki/Polymorphism_(object-oriented_programming)), Protected Variations, and Pure Fabrication. All these patterns answer some [software](http://en.wikipedia.org/wiki/Software) problem, and in almost every case these problems are common to almost every [software development](http://en.wikipedia.org/wiki/Software_development) project.

These pattern and principles are further explained [here](http://en.wikipedia.org/wiki/GRASP_%28object-oriented_design%29).

(Kiegészítés: Krisz)

## 2.4 KISS - Keep it simple, stupid

The KISS principle states that most systems work best if they are kept simple rather than made complicated; therefore simplicity should be a key goal in design and unnecessary complexity should be avoided.

## 2.5 YAGNI (TO BE APPROVED) - You ain't gonna need it

Should not add functionality until it is not necessary.

Instead spend time on testing.

## 2.6 What is the advantage of dependency injection? (TO BE APPROVED)

* Because dependency injection doesn't require any change in code behavior it can be applied to legacy code as arefactoring. The result is clients that are more independent and that are easier to unit test in isolation using stubs or mock objects that simulate other objects not under test. This ease of testing is often the first benefit noticed when using dependency injection.
* Dependency injection allows a client to remove all knowledge of a concrete implementation that it needs to use. This helps isolate the client from the impact of design changes and defects. It promotes reusability, testability and maintainability.
* Dependency injection can be used to externalize a system's configuration details into configuration files allowing the system to be reconfigured without recompilation. Separate configurations can be written for different situations that require different implementations of components. This includes, but is not limited to, testing.
* Reduction of boilerplate code in the application objects since all work to initialize or set up dependencies is handled by a provider component.
* Dependency injection allows concurrent or independent development. Two developers can independently develop classes that use each other, while only needing to know the interface the classes will communicate through. Plugins are often developed by third party shops that never even talk to the developers who created the product that uses the plugins.
* Dependency Injection decreases coupling between a class and its dependency.

## 2.7 LoD - Law of Demeter (TO BE APPROVED)

It is a specific case of loose coupling.

* Each unit should have only limited knowledge about other units: only units "closely" related to the current unit.
* Each unit should only talk to its friends; don't talk to strangers.
* Only talk to your immediate friends.

The fundamental notion is that a given object should assume as little as possible about the structure or properties of anything else (including its subcomponents).

# 3 GIT

## 3.1 What does Git rebase do? (TO BE APPROVED)

Forward-port local commits to the updated upstream head

'git rebase' [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>]

[<upstream> [<branch>]]

'git rebase' [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>]

--root [<branch>]

'git rebase' --continue | --skip | --abort | --edit-todo

## 3.2 What does Git reset do? (TO BE APPROVED)

Reset current HEAD to the specified state

'git reset' [-q] [<tree-ish>] [--] <paths>...

'git reset' (--patch | -p) [<tree-ish>] [--] [<paths>...]

'git reset' [--soft | --mixed [-N] | --hard | --merge | --keep] [-q] [<commit>]

# 4 JAVA

## 4.1 How can you serialize objects? (TO BE APPROVED)

To *serialize* an object means to convert its state to a byte stream so that the byte stream can be reverted back into a copy of the object. A Java object is *serializable* if its class or any of its superclasses implements either the java.io.Serializable interface. *Deserialization* is the process of converting the serialized form of an object back into a copy of the object.

Classes **ObjectInputStream** and **ObjectOutputStream** are high-level streams that contain the methods for serializing and deserializing an object, like void writeObject(Object x) and Object readObject(). To be serialized successfully, the class’ fields must also be Serializable or marked with the transient keyword. transient fields are not going to be serialized.

Classes that require special handling during the serialization and deserialization process must implement special methods with these exact signatures:

private void readObject(java.io.ObjectInputStream stream)  
 throws IOException, ClassNotFoundException;  
 private void writeObject(java.io.ObjectOutputStream stream)  
 throws IOException  
 private void readObjectNoData()  
 throws ObjectStreamException;

Subclasses of Objects that are not serializable can be serializable. In this case the non-serializable class must have a no-arg constructor to allow its fields to be initialized. In this case it is the responsibility of the subclass to save and restore the state of the non-serializable class. It is frequently the case that the fields of that class are accessible (public, package, or protected) or that there are get and set methods that can be used to restore the state. If the subclass has access to the state of the superclass it can implement writeObject and readObject to save and restore that state.

public class A {

public A(String some) {};

private A() {} //as protected or public everything would work

}

public class B extends A implements Serializable {

public B() {

super("");

}

//these doesn't really matter

//private void writeObject(java.io.ObjectOutputStream out) throws IOException { }

//private void readObject(java.io.ObjectInputStream in) throws IOException, ClassNotFoundException { }

}

public class BSerializer {

public static void main(String ... args) throws Exception {

B b = new B();

ByteArrayOutputStream baos = new ByteArrayOutputStream();

ObjectOutputStream oos = new ObjectOutputStream(baos);

oos.writeObject(b);

ByteArrayInputStream bais = new ByteArrayInputStream(baos.toByteArray());

ObjectInputStream ois = new ObjectInputStream(bais);

B deserialized = (B) ois.readObject(); //InvalidClassException

}

}

## 4.2 How can you compare two objects? (TO BE APPROVED)

**== operator (reference equality)**Used to determine whether two object references point to the same object.

**overriding boolean equals(Object obj) method (value equality)**To be able to compare two Java objects of the same class theequals method must be overridden and implemented by the class. The implementor decides which values must be equal to consider two objects to be equal.   
When overriding the equals method, int hashCode() **must** also be overridden to follow the general contract of hashCode to return the same integer for equal objects. Failure to do so renders hash tables practically useless with this class.

**implementing the Comparable<T> interface (natural ordering)**

To be able to sort objects from the same class, the class must implement the Comparable interface. This imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class' natural ordering, and the class' int compareTo(T obj) method is referred to as its natural comparison method.

**creating a Comparator<T> implementation (custom ordering)**

A Comparator implementation can be used to provide different ordering besides the class’ natural ordering with the int compare(T obj1, T obj2) method. Such as descending order instead of ascending, or comparing clients by their address instead of their last name.

## 4.3 Access modifiers (TO BE APPROVED)

A class may be declared with the modifier public, in which case that class is visible to all classes everywhere. If a class has no modifier (the default, also known as *package-private*), it is visible only within its own package (packages are named groups of related classes — you will learn about them in a later lesson.)

At the member level, you can also use the public modifier or no modifier (*package-private*) just as with top-level classes, and with the same meaning. For members, there are two additional access modifiers: private and protected. The private modifier specifies that the member can only be accessed in its own class. The protected modifier specifies that the member can only be accessed within its own package (as with *package-private*) and, in addition, by a subclass of its class in another package.

The following table shows the access to members permitted by each modifier.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Modifier** | **Class** | **Package** | **Subclass** | **World** |
| public | Y | Y | Y | Y |
| Protected(m) | Y | Y | Y | N |
| *no modifier(p-p)* | Y | Y | N | N |
| Private(m) | Y | N | N | N |

## 4.4 Protected vs default

The protected modifier specifies that the member can only be accessed within its own package (as with *package-private*) and, in addition, by a subclass of its class in another package.

## 4.5 What is stored on the stack? (TO BE APPROVED)

The basic difference between stack and heap is the life cycle of the values.

Stack values only exist within the scope of the function they are created in. Once it returns, they are discarded.

Heap values however exist on the heap. They are created at some point in time, and destructed at another (either by GC or manually, depending on the language/runtime).

**Java only stores primitives on the stack.** This keeps the stack small and helps keeping individual stack frames small, thus allowing more nested calls.

**Objects are created on the heap, and only references (which in turn are primitives) are passed around on the stack.** ([Source](http://programmers.stackexchange.com/questions/65281/stack-and-heap-memory-in-java))

# 5 Multithread

## 5.1 What concurrent collections do you know?

* **BlockingQueue**: FIFO data structure, that blocks or times out when you attempt to add to a full queue, or remove from an empty one.
* **ConcurrentMap**: its the subinterface of java.util.Map. It has atomic operations to remove or to replace a key-value pair only if the key is present, or to add a key-value pair only if the key is absent.
* **CopyOnWriteArrayList**: provides better concurrency than synchronized List by allowing multiple concurrent reader and replacing the whole list on write operation.
* **SynchronizedXXXX – synch version of collections**

## 5.2 What does putIfAbsent (ConcurrentHashMap) method do? (TO BE APPROVED)

If the specified key is not already associated with a value, associate it with the given value. This is equivalent to

if (!map.containsKey(key))  
 return map.put(key, value);  
 else  
 return map.get(key);

except that the action is performed atomically (so it is thread-safe). ([Source](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentMap.html))

## 5.3 Increment a number thread-safely (synchronized methods, volatile, AtomicInteger)

* **Volatile variable**: Concurrent modifications are visible for other threads, but volatile does NOT provide any kind of synchronization and thus it is not threadsafe. Even simple cases of race conditions can occur (e.g.: use of the ++ operator).
* **Synchronized methods**: Mutable state gets guarded by a lock (intristic lock in this case). Concurrent modifications are visible for other threads and simple usage is threadsafe, though while using compound actions, race conditions can arise. Subsequent calls are synchronized, but between them the state of the object can change e.g.:   
  if (object.getInt() < 10) {   
   object.increment();  
  }  
  In cases like this, additional synchronization solves the problem, but still, its open for mistakes.
* **AtomicInteger**: A threadsafe class designed for multithreaded programs. It also contains methods for compound actions like getAndIncrement(), incrementAndGet() and getAndSet() among many others.

(Krisz)

## 5.4 What may be the consequences of the lack of synchronization?(TO BE APPROVED)

To cut this question short:

* Corrupted Data

## 5.5 Unexpected exceptions (ConcurrentModificationException)

* Inconsistency
* Phantom read
* Phantom update

(Krisz)

## 5.6 What is the difference between synchronizing a method or the content of the method?

* Synchronizing a method uses the methods ~~object~~  class (intristic lock) to lock the methods content. The more methods are synchronized this way, the less concurrent those objects can be used (Note the class is going to be threadsafe, only it’s concurrent usefulness decreases). The synchronized keyword can be removed from methods using reflection(!).
* Synchronizing only a methods content allows the programmer to use different locks for different methods, ensuring thread safety while maximising concurrent usage. While this sounds great, there are some conditions to this. (state guarded by different locks have to be independent).

Additional info:

public synchronized void doesNothing() {

}

is equal to this:

public void doesNothing() {

synchronized(this) {

}

}

## 5.7 How does ConcurentHashMap work?(TO BE APPROVED)

ConcurrentHashMap allows concurrent access to the map. HashTable too offers synchronized access to map, but your entire map is locked to perform any operation.

The logic behind ConcurrentHashMap is that not your entire table is getting locked, but only a portion[segment]. Each segment manages its own HashTable. Locking is applied only for updates. In case of of retrievals, it allows full concurrency.

The ConcurrentHashMap offers new method putIfAbsent() which is similar to put except the value will not be overridden if the key exists.

olvasmány: <https://ria101.wordpress.com/2011/12/12/concurrenthashmap-avoid-a-common-misuse/>

# 6 Spring

## 6.1 How can you make a factory in Spring?(TO BE APPROVED)

You must implement the FactoryBean interface, and register your custom factory bean as a bean.

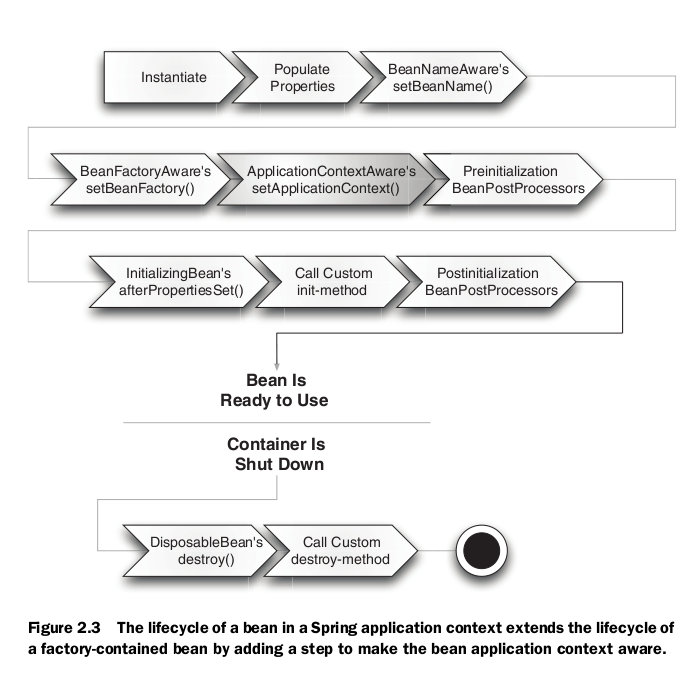
## 6.2 How can you manipulate a Spring bean during creation?(TO BE APPROVED)

Through BeanPostProcessors or BeanFactoryPostProcessors.

* A BeanPostProcessor gives you a chance to process an instance of a bean created by the IoC container after its instantiation and then again after the initialization lifecycle event has occurred on the instance.
* A BeanFactoryPostProcessor lets you modify the actual bean definition instead of the instance as it's created.

## 6.3 Spring bean lifecycle(TO BE APPROVED)

When a bean is instantiated, it may be required to perform some initialization to get it into a usable state. Similarly, when the bean is no longer required and is removed from the container, some cleanup may be required. There is lists of the activities that take place behind the scenes between the time of bean Instantiation and its destruction.



**Instantiate** - first Spring instantiate the bean

**Populate Properties** - Spring Inject the bean’s properties

**BeanNameAware’s** - if the bean implements BeanNameAware, spring passes the bean’s id to setBeanName() method

**BeanFactoryAware** - if Bean implements BeanFactoryAware, spring passes the beanfactory to setBeanFactory() method. E.g. Bean:

@Component  
public MyBean implements BeanFactoryAware {  
 private BeanFactory beanFactory;  
  
 @Override  
 public void setBeanFactory(final BeanFactory beanFactory) {  
 this.beanFactory = beanFactory;  
 }  
  
 public void myMethod() {  
 //I can now use beanFactory here  
 }  
}

**PreInitialization** - If there are any classes implementing BeanPostProcessor, Spring calls its postProcessBeforeInitialization(Object bean, String beanName)

**InitializingBeans** - If the bean implements IntializingBean, its afterPropertySet() method is called. If the bean has init method declaration, the specified initialization method is Called.

**PostInitialization** - If there are any classes implementing BeanPostProcessor, Spring calls its postProcessAfterInItialization(Object bean, String beanName)

**Ready to use** - Now the bean is ready to be used by the application.

**DisposableBean** - If the bean implements DisposableBean, Spring will call its destroy() method.

**Call Custom destroy-method** - If a custom destroy() method is defined, the specified method is called.

# 7 JPA

## 7.1 JPA ManyToMany/OneToMany relationships.(TO BE APPROVED)

**OneToMany**

The most common relationship used when creating relational databases. A row in a table can be associated with one or (likely) more rows in another table. An example of a one-to-many relationship is a single order has many items on that order. And since relationships (can?) work both ways it is not uncommon to hear reference to many-to-one-relationships as well.

**ManyToMany**

When one or more rows in a table are associated with one or more rows in another table. An example of a many-to-many relationship: a customer can purchase many different products and a product can be purchased by many different customers.

One difference between a OneToMany and a ManyToMany relationship in JPA is that a ManyToMany needs a join table to store the relationship, whereas a OneToMany can either use a join table, or a foreign key in target object's table referencing the source object table's primary key.

The JoinTable is defined using the [@JoinTable](https://java.sun.com/javaee/5/docs/api/javax/persistence/JoinTable.html) annotation.

**OneToOne**

A row in a table is associated to one and only one row in another table. An example of a one-to-one relationship: a person can have one social security number and a social security number can only be assigned to one person.

In most cases there is no need for a one-to-one relationship as the contents of the two tables can be combined into one table.

## 7.2 Difference between detached and unmanaged entities(TO BE APPROVED)

A persistence context is a set of managed entity object instances. Persistence contexts are managed by an entity manager. The entity manager tracks all entity objects within a persistence context for changes and updates made, and flushes these changes to the database using the flush() or commit() functions.

When an entity is detached, it is unmanaged. Any state changes to an entity that is detached are not tracked by the entity manager.

## 7.3 Caching with JPA (TO BE APPROVED)

#### **7.3.0.1 JPA Level 1 Cache**

A JPA entity manager uses persistence context to manage entities. The persistence context associated with the respective entity managers acts as the first-level cache. At any point within a persistence context, an entity manager will have only *one* instance of the object mapped to a particular row in the database. When another user in a different persistence context refers to the same entity object, JPA uses the scope of the persistence context to resolve the situation.

#### **7.3.0.2 JPA Level 2 Cache**

Level 2 cache was introduced in the JPA 2.0 release. JPA provides a Cache API for basic cache operations, while level 2 cache shares the state of an entity -- which is accessed with the help of the entity manager factory -- across various persistence contexts. Level 2 cache underlies the persistence context, which is highly transparent to the application.

To enable or disable caching on a class, you can use the @Cacheable annotation or the cacheable XML attribute. When Level 2 cache is enabled, the persistence provider will look for the entities in the persistence context first. If it does not find them there, the persistence provider will look in the Level 2 cache next instead of sending a query to the database.

## 7.4 Tell if one can bypass JPA caching mechanism by using JPQL (TO BE APPROVED)

The DML operations are directly executed against the database, therefore bypassing the first level cache. JPQL queries should automatically invalidate the 2nd level cache, but this may depend on the JPA provider.

# 8 JMS

Messaging is a method of communication between software components or applications. A messaging system is a peer-to-peer facility: A messaging client can send messages to, and receive messages from, any other client. Each client connects to a messaging agent that provides facilities for creating, sending, receiving, and reading messages.

Messaging enables distributed communication that is **loosely coupled**. A component sends a message to a destination, and the recipient can retrieve the message from the destination. However, the sender and the receiver do not have to be available at the same time in order to communicate. In fact, the sender does not need to know anything about the receiver; nor does the receiver need to know anything about the sender. The sender and the receiver need to know only which message format and which destination to use. In this respect, messaging differs from tightly coupled technologies, such as Remote Method Invocation (RMI), which require an application to know a remote application’s methods.

The JMS API enables communication that is not only loosely coupled but also:

* **Asynchronous**: A JMS provider can deliver messages to a client as they arrive; a client does not have to request messages in order to receive them.
* **Reliable**: The JMS API can ensure that a message is delivered once and only once. Lower levels of reliability are available for applications that can afford to miss messages or to receive duplicate messages.

## 8.1 JMS queue (TO BE APPROVED)

A staging area that contains messages that have been sent and are waiting to be read (by only one consumer). Contrary to what the name *queue* suggests, messages don't have to be received in the order in which they were sent. A JMS queue only guarantees that each message is processed only once.

* Point-to-point model
* Only one consumer gets the message
* A JMS queue only guarantees that each message is processed only once.
* The Queue knows who the consumer or the JMS client is. The destination is known.
* The JMS client (the consumer) does not have to be active or connected to the queue all the time to receive or read the message.
* Every message successfully processed is acknowledged by the consumer

## 8.2 JMS topic (TO BE APPROVED)

A distribution mechanism for publishing messages that are delivered to multiple subscribers.

* Publish/subscribe model
* Multiple clients subscribe to the message
* There is no guarantee messages have to be delivered in the order sent
* There is no guarantees that each message is processed only once. -- As this can be sensed from the model
* The Topic, have multiple subscribers and there is a chance that the topic does not know all the subscribers. The destination is unknown.
* The subscriber / JMS client needs to the active when the messages are produced by the producer, unless the subscription was a durable subscription.
* No, Every message successfully processed is not acknowledged by the consumer/subscriber.

## 8.3 JMS mdb (TO BE APPROVED)

A **message-driven bean** is an enterprise bean that allows Java EE applications to process messages asynchronously. This type of bean normally acts as a JMS message listener, which is similar to an event listener but receives JMS messages instead of events. The messages can be sent by any Java EE component (an application client, another enterprise bean, or a web component) or by a JMS application or system that does not use Java EE technology. Message-driven beans can process JMS messages or other kinds of messages.

# 9 SQL (ejtsd: szíkvel)

## 9.1 Basic SQL (TODO)

SQL stands for *Structured Query Language* designed for managing data held in *Relational Database Management Systems* (RDBMS). SQL consists of:

* *Data Definition Language* (DDL), e.g.: CREATE, RENAME, ALTER, TRUNCATE, DROP
* *Data Manipulation Language* (DML), e.g.: INSERT, UPDATE, DELETE, MERGE
* *Data Query Language* (DQL): SELECT
* *Data Control Language* (DCL), e.g.: GRANT, REVOKE

(András + Attila)

## 9.2 SQL: Explain difference between where and having (TO BE APPROVED)

WHERE is used to create conditions to table data, while HAVING is used to create conditions on aggregated data ([aggregate functions](http://en.wikipedia.org/wiki/Aggregate_function) include: sum, max, min, count, avg...).

As a rule of thumb, use WHERE before GROUP BY and HAVING after GROUP BY. It is a rather primitive rule, but it is useful in more than 90% of the cases.

(Safi)

Example (from a forum application known by all of us):

Table def:

CREATE TABLE `comment` (

`id` int(11) NOT NULL,

`comment` varchar(1000) NOT NULL,

`userId` int(11) NOT NULL,

`date` datetime NOT NULL,

`topicId` int(11) NOT NULL,

PRIMARY KEY (`id`)

)

SQL with a “WHERE” clause:

(Selects the comments made by a specified user)

select \* from comment where userId = 1;

SQL with a “HAVING” clause:

(Selects users whose average length of comments is less than 5 (lots of short, useless comments))

select userId, avg(LENGTH(comment)) as avgCommentLength from forum.comment group by userId having avgCommentLength < 5;

(Krisz)

## 9.3 Synchronize / lock / reentrant lock in details (TO BE APPROVED)

Synchronization is built around an internal entity known as the *intrinsic lock* or *monitor lock*. Intrinsic locks play a role in both aspects of synchronization: enforcing exclusive access to an object's state and establishing happens-before relationships that are essential to visibility.

When a thread invokes a *synchronized method*, it automatically acquires the intrinsic lock for that method's object and releases it when the method returns. The lock release occurs even if the return was caused by an uncaught exception.

Another way to create synchronized code is with *synchronized statements*. Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock:

public void addName(String name) {  
 synchronized(this) {  
 lastName = name;  
 nameCount++;  
 }  
 nameList.add(name);  
}

Recall that a thread cannot acquire a lock owned by another thread. But a thread *can* acquire a lock that it already owns. Allowing a thread to acquire the same lock more than once enables *reentrant synchronization*. This describes a situation where synchronized code, directly or indirectly, invokes a method that also contains synchronized code, and both sets of code use the same lock. Without reentrant synchronization, synchronized code would have to take many additional precautions to avoid having a thread cause itself to block.

**SOAP vs REST**

REST operates through a solitary, consistent interface to access named resources. It’s most commonly used when you’re exposing a public API over the Internet. SOAP, on the other hand, exposes components of application logic as services rather than data.

**DEFENSIVE PROGRAMMING**

**BEAN FACTORY vs FACTORY BEAN**

tl;dr A FactoryBean is an interface that you, as a developer, implements when writing factory classes and you want the object created by the factory to be managed as a bean by Spring, while a BeanFactory on the other hand, represents the Spring IoC container, it contains the managed beans and provides access to retrieving them. It is part of the core of the framework which implements the base functionality of an inversion of control container.

**Factory bean is a spring interface:**public interface FactoryBean<T> {

T getObject() throws Exception;

Class<T> getObjectType();

boolean isSingleton();

} //tells spring that the created object won’t be the type of the class, but the type of getObjectType();

**Bean factory is:**

**ClassPathXmlApplicationContext** as an example of **BeanFactory**.

The ClassPathXmlApplicationContext is one of the ways you can initiate Springs IoC container. You supply it with the configuration(s), it does the necessary object resolution and dependency injection and returns to you a container from which you can retrieve a fully resolved bean.