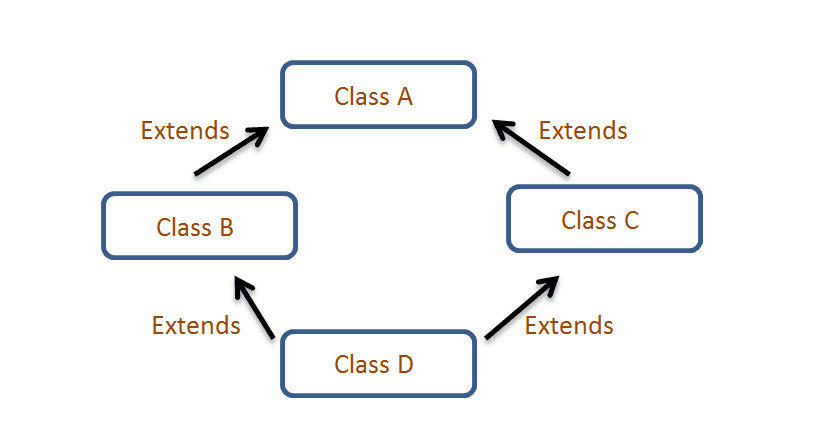
**Abstraction:** Öröklődés

Gyémánt öröklődés probléma



**Abstract class-t nem lehet new-val pédányosítani:**

abstract class Animal {

public abstract void animalSound();

public void sleep() {

System.out.println("Zzz");

}

}

Animal myObj = new Animal(); // will generate an error

**Helyette származtatni tudjuk:**

// Abstract class

abstract class Animal {

// Abstract method (does not have a body)

public abstract void animalSound();

// Regular method

public void sleep() {

System.out.println("Zzz");

}

}

// Subclass (inherit from Animal)

class Pig extends Animal {

public void animalSound() {

// The body of animalSound() is provided here

System.out.println("The pig says: wee wee");

}

}

class Main {

public static void main(String[] args) {

Pig myPig = new Pig(); // Create a Pig object

myPig.animalSound();

myPig.sleep();

}

}

**Encapsulation:** egységbezárás (privát fieldek és public getter setter - külvilág elől a logika elrejtése)

Lehető legszűkebb hozzáférés

**Polymorphism:** Többalakúság. Egy típuselméleti fogalom, amely szerint egy ősosztály típusú változó hivatkozhat ugyanazon közös ősosztályból származó (vagy ugyanazon interfészt megvalósító) osztályok példányaira. A polimorfizmus lehet statikus és dinamikus. (a) statikus polimorfizmus: metódusok túlterhelése, függvénysablonok, osztálysablonok. Satikus, fordításidejű kötés. (b) dinamikus polimorfizmus: metódusok felülírása. Dinamikus, futásidejű kötés

**Inheritance:**

* **alosztály** (gyermek) - az osztály, amely egy másik osztálytól örökli
* **szuperosztály** (szülő) - az osztály, amelyből öröklődik

Ha megpróbál hozzáférni egy finalosztályhoz, a Java hibát generál:

* final class Vehicle {
* ...
* }
* class Car extends Vehicle {
* ...
* }

**Konstruktor:** Az a művelet, amely inicializálja az objektumot. Automatikusan hívódik. Egy osztályhoz annyiféle konstruktort készítünk, ahányféle képpen lehetővé tesszük a példányok inicializálását .

Annotációk:

@AllArgsConstructor

@NoArgsConstructor

**Asszociáció:**

× Két vagy több osztály objektumainak valamilyen relációval történő összekapcsolása

× Lehet reflexív, azaz azonos osztályú objektumok összekapcsolása is lehetséges

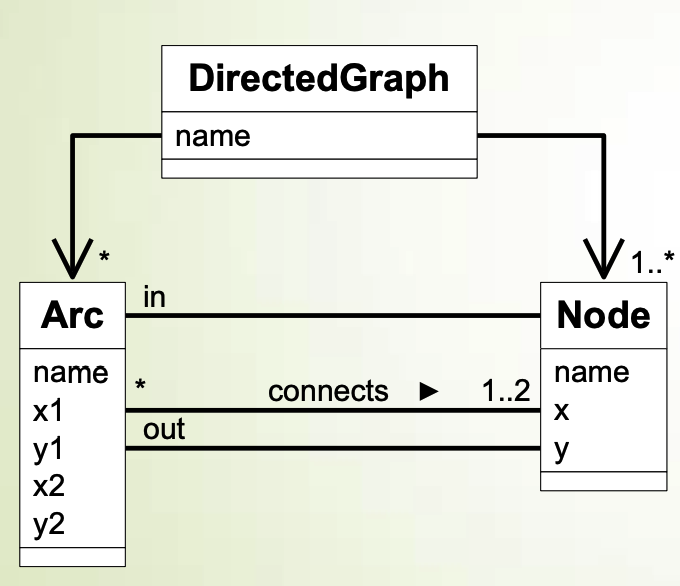
× Az asszociációhoz társulhat annak neve, azonosítója

× Lehet iránya: az aktív objektumtól a passzív objektum felé mutat

× Az összekapcsolt objektumoknak lehet multiplicitása, szerepe, és az összekapcsoláshoz minősítő is társulhat

× Navigálhatóság is megadható, amellyel kifejezhető, hogy az osztályok objektumai ismerik-e egymást

× A navigálhatóság elhagyása esetén kölcsönös elérhetőséget tételezünk fel

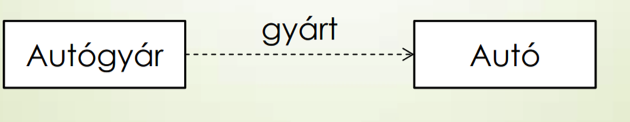


Függőség:

× Egy osztály függ egy másiktól, ha a független osztályunk paraméterként vagy lokális változóként megjelenik egy, a függő osztály metódusánál

× Az asszociációhoz képest fontos eltérés, hogy ott a függő osztály attribútumaként jelenik meg a független osztálynak a példánya

× Függőségnél a kapcsolat néha annyira gyenge, hogy a függőséget generáló osztályhoz tagváltozó egyáltalán nem is jelenik meg a függő osztályban

****

**Aggregáció:**

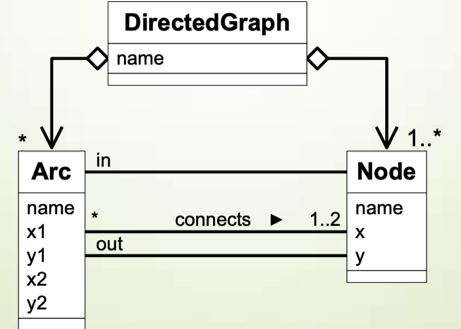
× Speciális asszociáció × Az általános asszociációnál erősebb kapcsolat, pl.:

× Egész és annak részei,

× Felépítmény és annak komponensei

× Azt fejezi ki, hogy az egyik osztály objektumai részét képezik egy másik osztály objektumainak

× Az aggregáció tranzitív, antiszimmetrikus, de nem lehet reflexív

****

**Kompozíció:**

× Speciális aggregáció

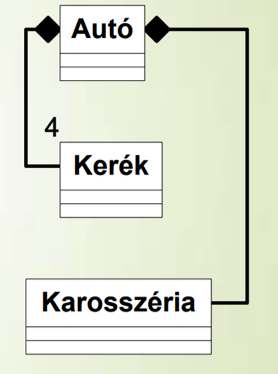
× Azt fejezi ki, hogy az egyik osztály objektumai a másik osztály objektumait fizikailag tartalmazzák

× A kompozíciós kapcsolat és az attribútum jellegű kapcsolat jelentése ugyanaz, csupán a diagramokban jelenik meg másképp

× Egy komponens objektum legfeljebb egy gazdaobjektumhoz tartozhat

× Egy gazdaobjektumnak tetszőleges számú komponense lehet

× A gazdaobjektum és annak komponensei azonos életciklusúak, azaz egyszerre jönnek létre, és egyszerre szűnnek meg

****

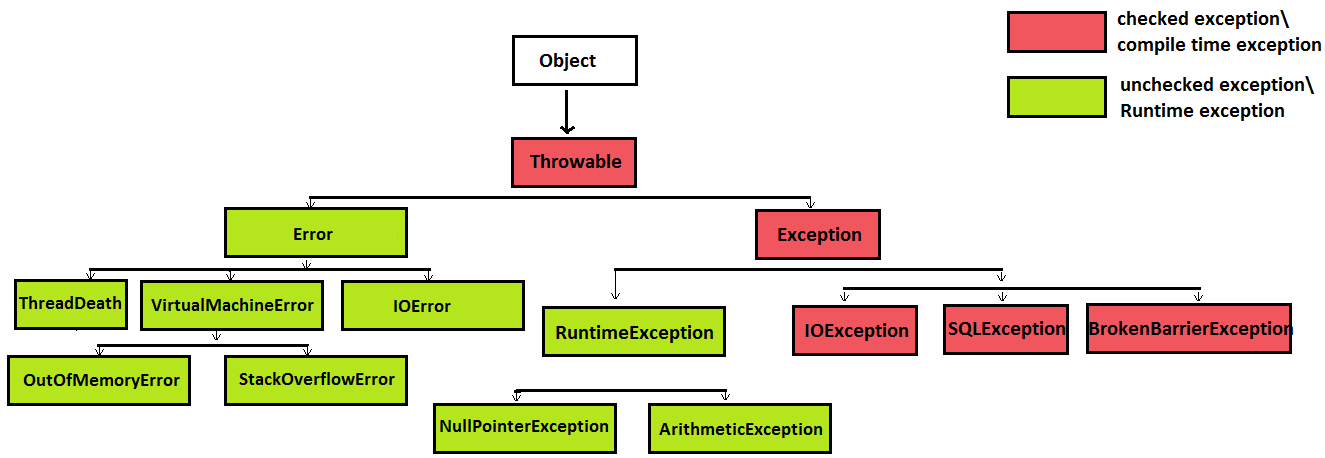
**Interface:** https://okt.inf.szte.hu/prog1/gyakorlat/eloadas/Java/interface/

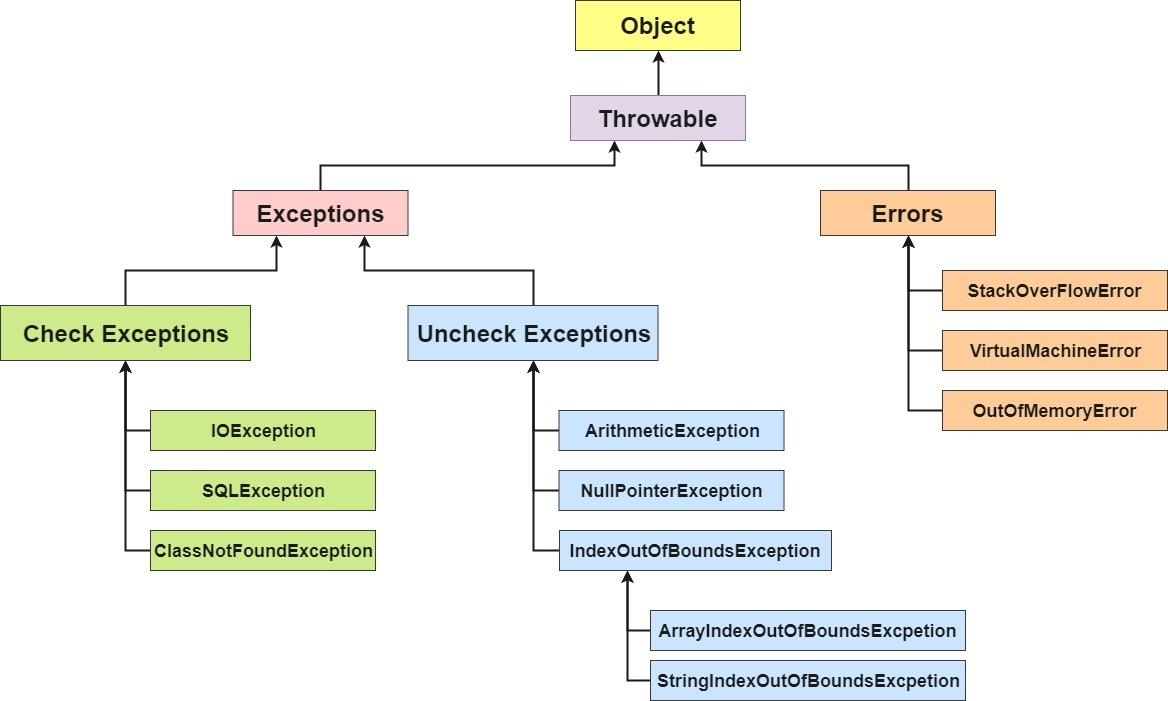
**-----------------------------------------------------**

**Hiba/Hibakezelés**

Linter: IntelliJ-be beépített fordító, ami miatt felismeri a hibákat, pl. ;} stb

CSAK az Uncheck Exception-t kezeljük! -> el tudod kapni





**Különbség a Java ellenőrzött és nem ellenőrzött kivételei között**

Összefoglalva, a különbség a bejelölt és a nem ellenőrzött kivételek között a következő:

* Az ellenőrzött kivételeket a rendszer fordítási időben fogja el, míg a futásidejű vagy ellenőrizetlen kivételt, ahogyan az szerepel, futási időben.
* A bejelölt kivételt vagy újradobással, vagy try catchblokkal kell kezelni, míg a nem bejelölt kivételt nem szükséges kezelni.
* A futásidejű kivétel programozási hiba, és végzetes, míg az ellenőrzött kivétel kivételes feltétel a kód logikájában, és visszaállítható vagy újra megpróbálható.

## Difference Between Checked and Unchecked Exceptions in Java

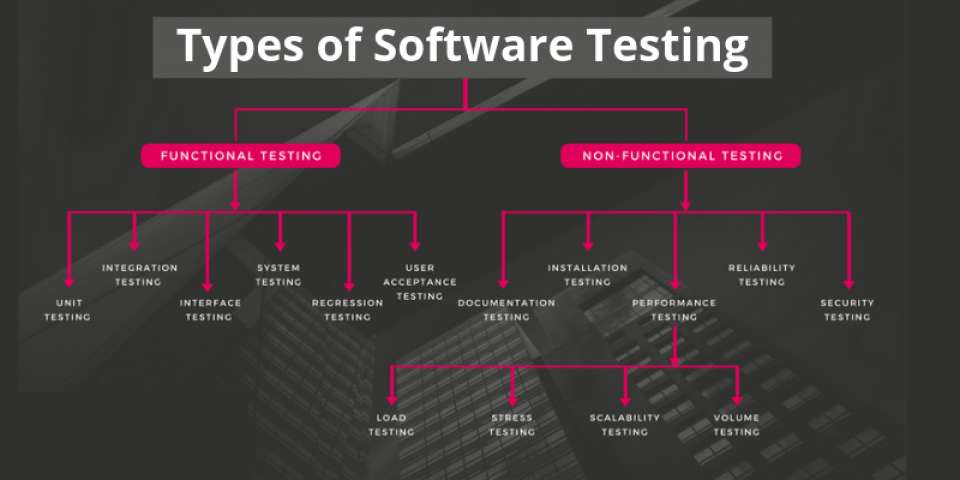
To summarize, the difference between a checked and unchecked exception is:

* A checked exception is caught at compile time whereas a runtime or unchecked exception is, as it states, at runtime.
* A checked exception must be handled either by re-throwing or with a try catch block, whereas an unchecked isn’t required to be handled.
* A runtime exception is a programming error and is fatal whereas a checked exception is an exception condition within your code’s logic and can be recovered or re-tried from.

**-----------------------------------------------------**

**Tesztelés**

**Miért szeretjük?**



van pl. Stress test - szoftver robosztussága, nem mindenhol használják

Unit test esetén: nekünk a mock jó!!! de van fake és van stub

Mock az adott metódust igazolja vissza (függetlenít a függőségtől)

**Fake** objects actually have working implementations, but usually take some shortcut which makes them not suitable for production

**Stubs** provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test. Stubs may also record information about calls, such as an email gateway stub that remembers the messages it 'sent', or maybe only how many messages it 'sent'.

**Mocks** are what we are talking about here: objects pre-programmed with expectations which form a specification of the calls they are expected to receive.

Mockito mock vagy MockMvs

Unit-teszt minimum 80% elvárás ált.

**-----------------------------------------------------**

**Inversion of Control**

The **Inversion-of-Control (IoC)** pattern, is about providing any kind of callback (which controls reaction), instead of acting ourself directly (in other words, inversion and/or redirecting control to external handler/controller). The **Dependency-Injection (DI)** pattern is a more specific version of IoC pattern, and is all about removing dependencies from your code.

Every DI implementation can be considered IoC, but one should not call it IoC, because implementing Dependency-Injection is harder than callback (Don't lower your product's worth by using general term "IoC" instead).

For DI example, say your application has a text-editor component, and you want to provide spell checking. Your standard code would look something like this:

public class TextEditor {

private SpellChecker checker;

public TextEditor() {

this.checker = new SpellChecker();

}

}

What we've done here creates a dependency between the TextEditor and the SpellChecker. In an IoC scenario we would instead do something like this:

public class TextEditor {

private IocSpellChecker checker;

public TextEditor(IocSpellChecker checker) {

this.checker = checker;

}

}

In the first code example we are instantiating SpellChecker (this.checker = new SpellChecker();), which means the TextEditor class directly depends on the SpellChecker class.

In the second code example we are creating an abstraction by having the SpellChecker dependency class in TextEditor's constructor signature (not initializing dependency in class). This allows us to call the dependency then pass it to the TextEditor class like so:

SpellChecker sc = new SpellChecker(); // dependency

TextEditor textEditor = new TextEditor(sc);

Now the client creating the TextEditor class has control over which SpellChecker implementation to use because we're injecting the dependency into the TextEditor signature.

To understanding the concept, Inversion of Control (IoC) or Dependency Inversion Principle (DIP) involves two activities: abstraction, and inversion. Dependency Injection (DI) is just one of the few of the inversion methods.

To read more about this you can read my blog [Here](http://kusnaditjung.blogspot.co.uk/2016/05/dependency-inversion-principle-dip.html)

1. What is it?

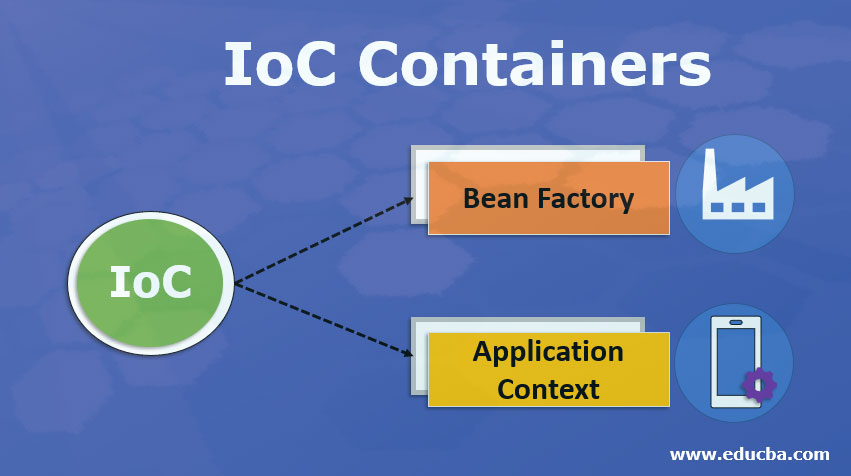
It is a practice where you let the actual behavior come from outside of the boundary (Class in Object Oriented Programming). The boundary entity only knows the abstraction (e.g interface, abstract class, delegate in Object Oriented Programming) of it.

1. What problems does it solve?

In term of programming, IoC try to solve monolithic code by making it modular, decoupling various parts of it, and make it unit-testable.

1. When is it appropriate and when not?

It is appropriate most of the time, unless you have situation where you just want monolithic code (e.g very simple program)



# Spring – IoC Container

The Spring framework can be considered as a collection of sub-frameworks, also referred to as layers, such as Spring AOP, Spring ORM, Spring Web Flow, and Spring Web MVC. You can use any of these modules separately while constructing a Web application. The modules may also be grouped together to provide better functionalities in a web application.

Prior to penetrating down to Spring to container do remember that Spring provides two types of Containers namely as follows:

1. BeanFactory Container
2. ApplicationContext Container

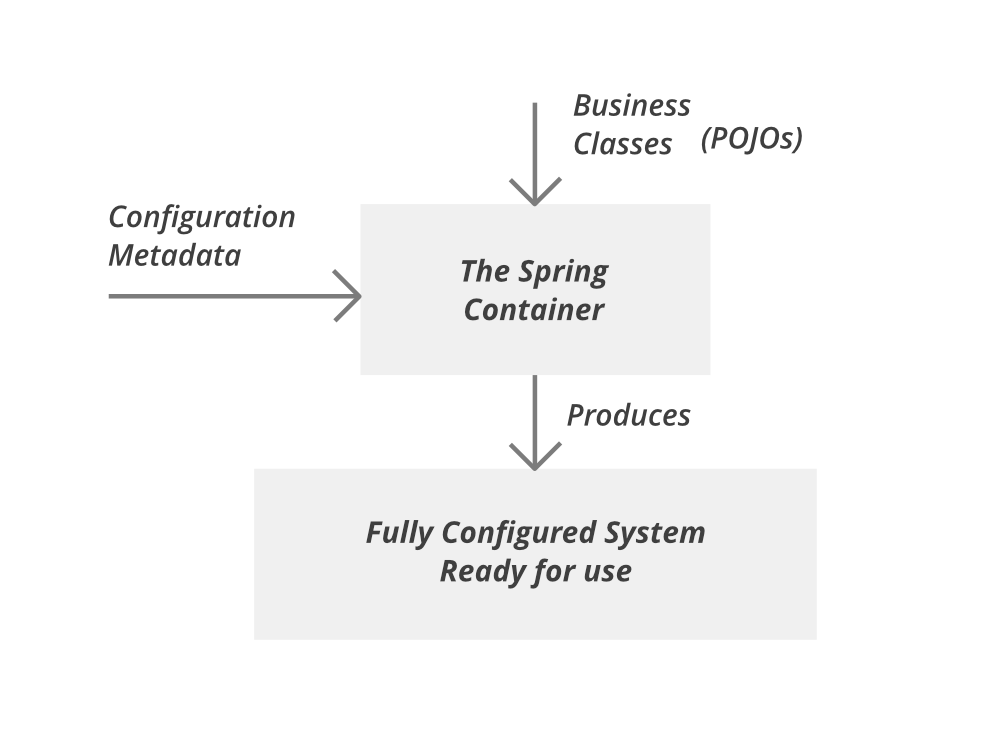
The features of the Spring framework such as IoC, AOP, and transaction management, make it unique among the list of frameworks. Some of the most important features of the Spring framework are as follows:

1. **IoC container**
2. Data Access Framework
3. Spring MVC
4. Transaction Management
5. Spring Web Services
6. JDBC abstraction layer
7. Spring TestContext framework

Spring IoC Container is the core of Spring Framework. It creates the objects, configures and assembles their dependencies, manages their entire life cycle. The Container uses Dependency Injection(DI) to manage the components that make up the application. It gets the information about the objects from a configuration file(XML) or Java Code or Java Annotations and [Java POJO class](https://www.geeksforgeeks.org/pojo-vs-java-beans/). These objects are called Beans. Since the Controlling of Java objects and their lifecycle is not done by the developers, hence the name **Inversion Of Control**.

***Note:****Spring IoC generally directly refers to a core container that uses the DI/DC pattern to implicitly provide an object reference in a class during runtime. The IoC container contains assembler code that handles the configuration management of application objects.*

The following diagram depicts how the Container makes use of Configuration metadata and Java [POJO classes](https://www.geeksforgeeks.org/pojo-vs-java-beans/) to manage beans.



So finally let us discuss out some major differences between BeanFactory vs ApplicationContext in order to get a clear cutaway understanding of the spring IoC container which is as shown below in a tabular format below as follows:

| Feature | BeanFactory | ApplicationContext |
| --- | --- | --- |
| Annotation Support | No | Yes |
| Bean Instantiation/Wiring | Yes | Yes |
| Internationalization | No | Yes |
| Enterprise Services | No | Yes |
| ApplicationEvent publication | No | Yes |
| Automatic BeanPostProcessor  registration | No | Yes |
| Loading Mechanism | Lazy loading | Aggressive loading |
| Automatic BeanFactoryPostProcessor  registration | No | Yes |

**POJO classes**

POJO stands for Plain Old Java Object. It is an ordinary Java object, not bound by any special restriction other than those forced by the Java Language Specification and not requiring any classpath. POJOs are used for increasing the readability and re-usability of a program. POJOs have gained the most acceptance because they are easy to write and understand. They were introduced in EJB 3.0 by Sun microsystems.

**A POJO should not:**

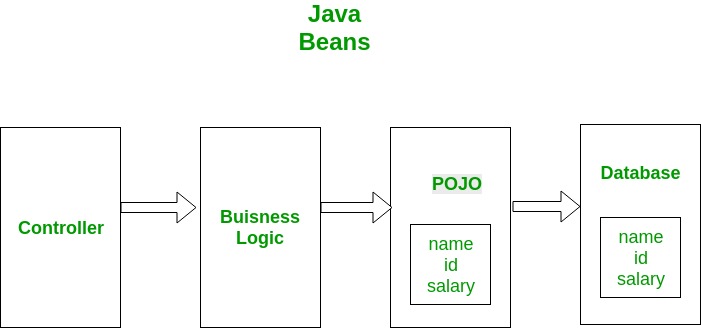
1. Extend prespecified classes, Ex: public class GFG extends javax.servlet.http.HttpServlet { … } is **not** a POJO class.
2. Implement prespecified interfaces, Ex: public class Bar implements javax.ejb.EntityBean { … } is **not** a POJO class.
3. Contain prespecified annotations, Ex: @javax.persistence.Entity public class Baz { … } is **not** a POJO class.

POJOs basically define an entity. Like in your program, if you want an Employee class, then you can create a POJO as follows:

* Java

|  |
| --- |
| // Employee POJO class to represent entity Employee  **public** **class** Employee  {      // default field      String name;        // public field  **public** String id;        // private salary  **private** **double** salary;        //arg-constructor to initialize fields  **public** Employee(String name, String id,  **double** salary)      {  **this**.name = name;  **this**.id = id;  **this**.salary = salary;      }        // getter method for name  **public** String getName()      {  **return** name;      }        // getter method for id  **public** String getId()      {  **return** id;      }        // getter method for salary  **public** Double getSalary()      {  **return** salary;      }  } |

The above example is a well-defined example of the POJO class. As you can see, there is no restriction on access-modifiers of fields. They can be private, default, protected, or public. It is also not necessary to include any constructor in it.  
POJO is an object which encapsulates Business Logic. The following image shows a working example of the POJO class. Controllers interact with your business logic which in turn interact with POJO to access the database. In this example, a database entity is represented by POJO. This POJO has the same members as the database entity.



**Java Beans**

**Beans are special type of Pojos. There are some restrictions on POJO to be a bean.**

1. All JavaBeans are POJOs but not all POJOs are JavaBeans.
2. Serializable i.e. they should implement Serializable interface. Still, some POJOs who don’t implement a Serializable interface are called POJOs because Serializable is a marker interface and therefore not of many burdens.
3. Fields should be private. This is to provide complete control on fields.
4. Fields should have getters or setters or both.
5. A no-arg constructor should be there in a bean.
6. Fields are accessed only by constructor or getter setters.

Getters and Setters have some special names depending on field name. For example, if field name is someProperty then its**getter** preferably will be:

public "returnType" getSomeProperty()

{

return someProperty;

}

and **setter**will be

public void setSomePRoperty(someProperty)

{

this.someProperty=someProperty;

}

Visibility of getters and setters is generally public. Getters and setters provide the complete restriction on fields. e.g. consider below the property,

Integer age;

If you set visibility of age to the public, then any object can use this. Suppose you want that age can’t be 0. In that case, you can’t have control. Any object can set it 0. But by using the setter method, you have control. You can have a condition in your setter method. Similarly, for the getter method if you want that if your age is 0 then it should return null, you can achieve this by using the getter method as in the following example:

* Java

|  |
| --- |
| // Java program to illustrate JavaBeans  **class** Bean  {      // private field property  **private** Integer property;      Bean()      {          // No-arg constructor      }        // setter method for property  **public** **void** setProperty(Integer property)      {  **if** (property == 0)          {              // if property is 0 return  **return**;          }  **this**.property=property;      }        // getter method for property  **public** Integer getProperty()      {  **if** (property == 0)          {              // if property is 0 return null  **return** **null**;          }  **return** property;      }  }    // Class to test above bean  **public** **class** GFG  {  **public** **static** **void** main(String[] args)      {          Bean bean = **new** Bean();            bean.setProperty(0);          System.out.println("After setting to 0: " +                                   bean.getProperty());            bean.setProperty(5);          System.out.println("After setting to valid" +                        " value: " + bean.getProperty());      }  } |

Output:-

After setting to 0: null

After setting to valid value: 5

**POJO vs Java Bean**

| POJO | Java Bean |
| --- | --- |
| It doesn’t have special restrictions other than those forced by Java language. | It is a special POJO which have some restrictions. |
| It doesn’t provide much control on members. | It provides complete control on members. |
| It can implement Serializable interface. | It should implement serializable interface. |
| Fields can be accessed by their names. | Fields are accessed only by getters and setters. |
| Fields can have any visibility. | Fields have only private visibility. |
| There may/may-not be a no-arg constructor. | It must have a no-arg constructor. |
| It is used when you don’t want to give restriction on your members and give user complete access of your entity | It is used when you want to provide user your entity but only some part of your entity. |

**Conclusion**

POJO classes and Beans both are used to define java objects to increase their readability and reusability. POJOs don’t have other restrictions while beans are special POJOs with some restrictions.

This article is contributed by **Vishal Garg**. If you like GeeksforGeeks and would like to contribute, you can also write an article using [write.geeksforgeeks.org](http://www.write.geeksforgeeks.org/) or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.  
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**Bean Factory:**

Az ami előállítja és inicializálja a bean-eket

We can configure the BeanFactory with XML. Let's create a file bean factory-example.xml:

<bean id="employee" class="com.baeldung.beanfactory.Employee">

<constructor-arg name="name" value="Hello! My name is Java"/>

<constructor-arg name="age" value="18"/>

</bean>

<alias name="employee" alias="empalias"/>Copy

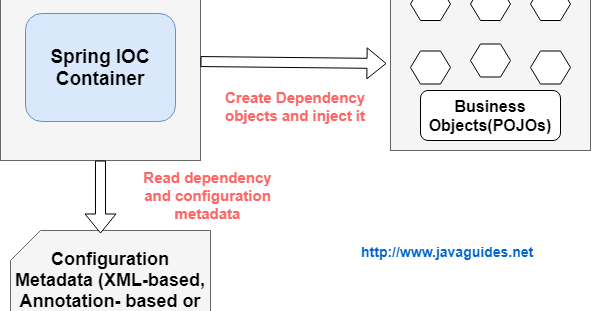
Note that we've also created an alias for the employee bean.

**Application Context:**

**Dependency Injection:**

A számítógép-programozásban a **dependency injection** egy technika, aminek lényege, hogy egy objektum más objektumok függőségeit elégíti ki.

A függőséget felhasználó objektum szolgáltatást nyújt, az injekció pedig ennek a függőségnek az átadása a kliens részére.



annotációval, vagy xml konfigurációval

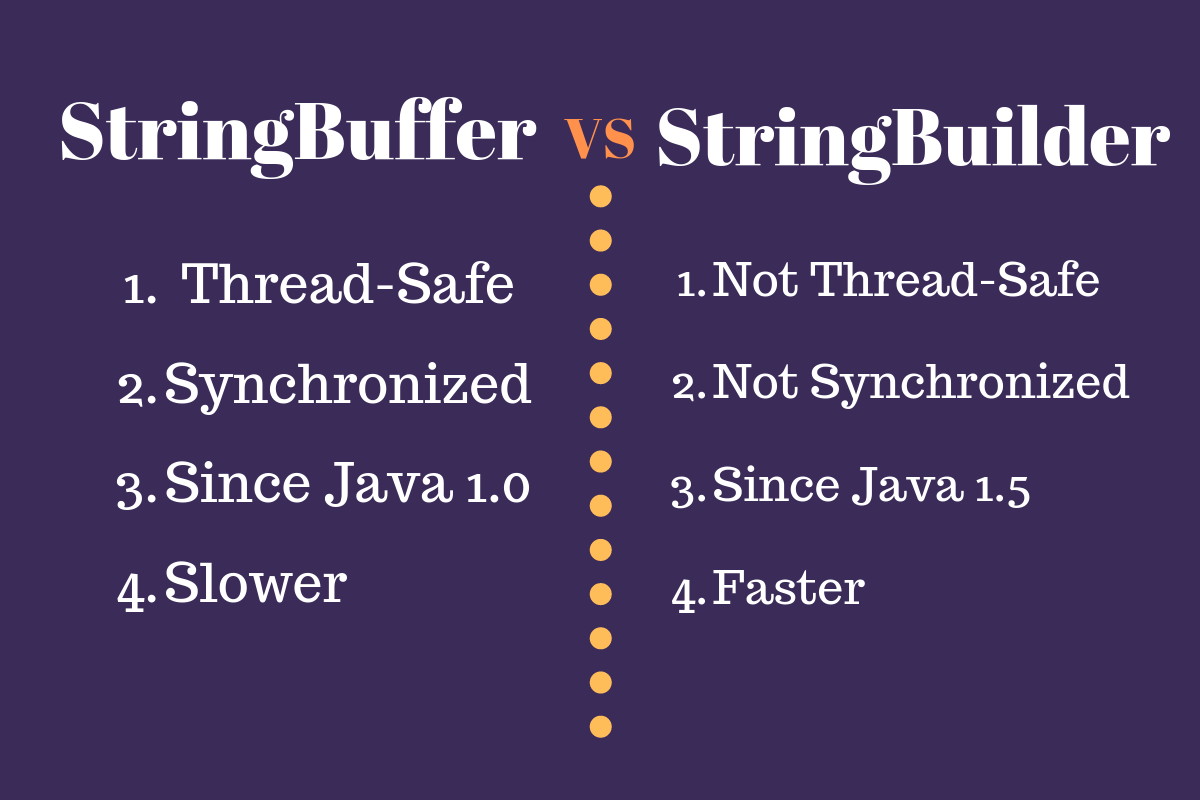
dependency injection - singleton instance (példány)

Singleton – a program futásakor egy darab példányt hoz létre!

@Qualifier annotation engedi, hogy ahol megjelölöd, annyi helyre külön singleton példányt hozzon létre

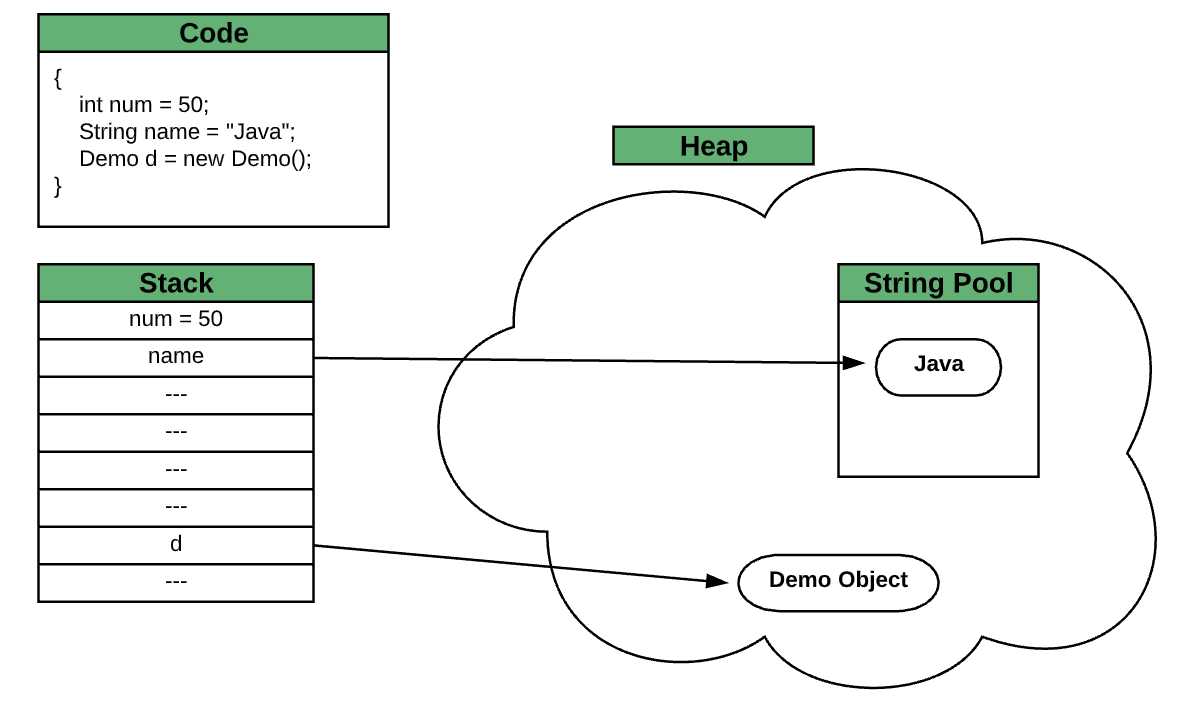
**-----------------------------------------------------**

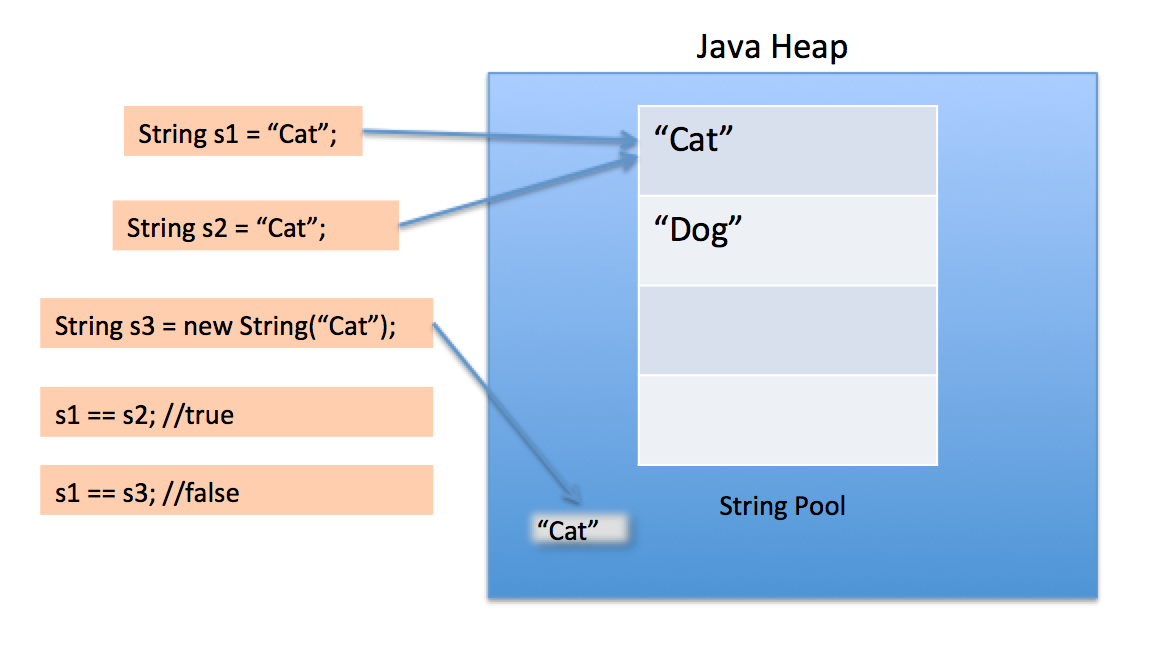
**String – különlegessége**

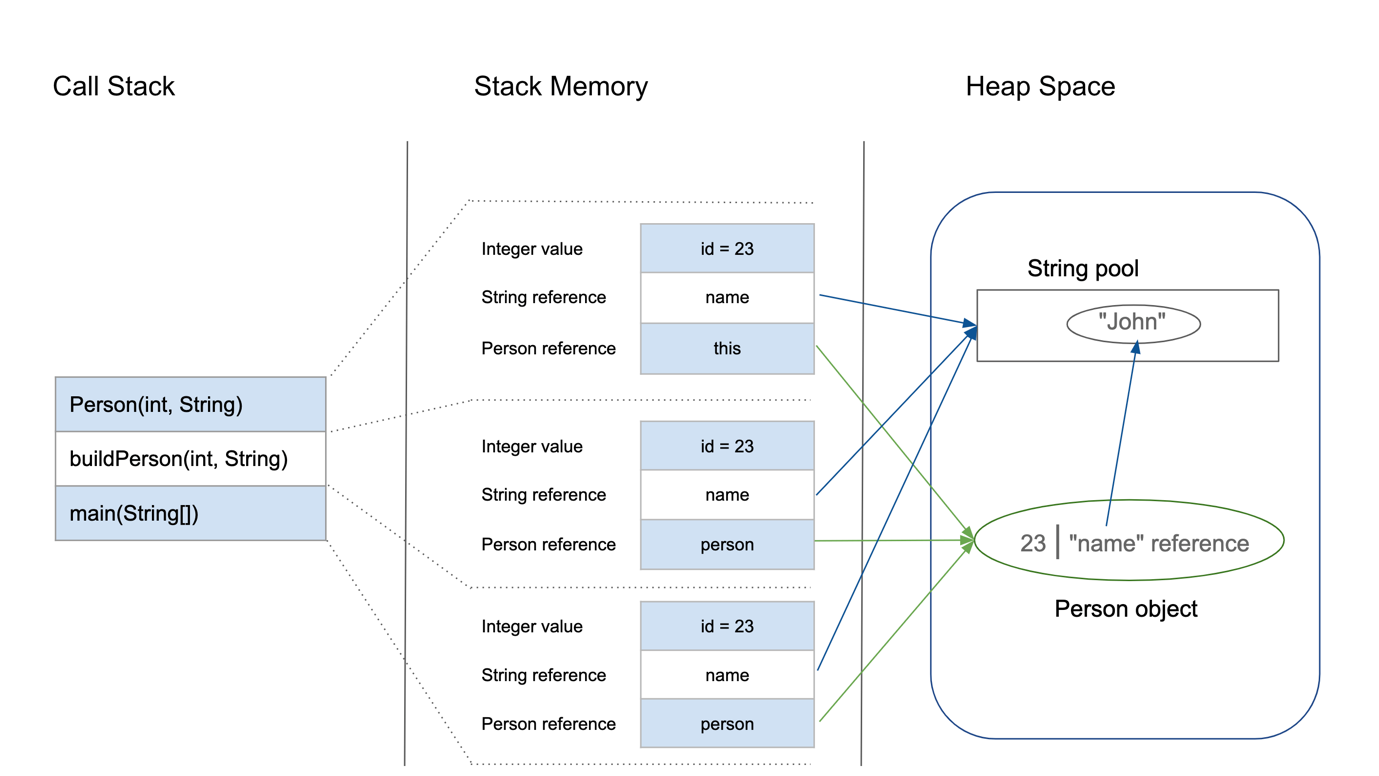


**-----------------------------------------------------**

**Heap és Stack memória (Object vs. Class)**







**-----------------------------------------------------**

**Verzió kezelő/ követő eszközök**

* GitHub
* Bitbucket
* Gitlab

**Git parancsok (Gitbash)**

* git branch
* git status
* git commit -m "this is my commit"
* git add . (ez hozzáadja a módosítást, de abban a könyvtár struktúrában)
* git add -a (ez pedig bármilyen könyvtárig mélyen mindent ami alatta van hozzáad)
* git pull (ha lehúzod)
* git push (feltolod)
* git clone (repo klónozás)

**-----------------------------------------------------**

**Docker**

A Docker egy számítógépes program, amely operációs rendszer szintű virtualizációt végez. A Docker konténerek futtatására szolgál. A konténerek egymástól elkülönülnek, és saját alkalmazást, eszközöket, könyvtárakat és konfigurációs fájlokat kötnek össze, jól meghatározott csatornákon keresztül kommunikálva egymással.