JAVA nyelvi alapok

Adatbányászati technikák (VISZM185)

Dávid István
david@cs.bme.hu

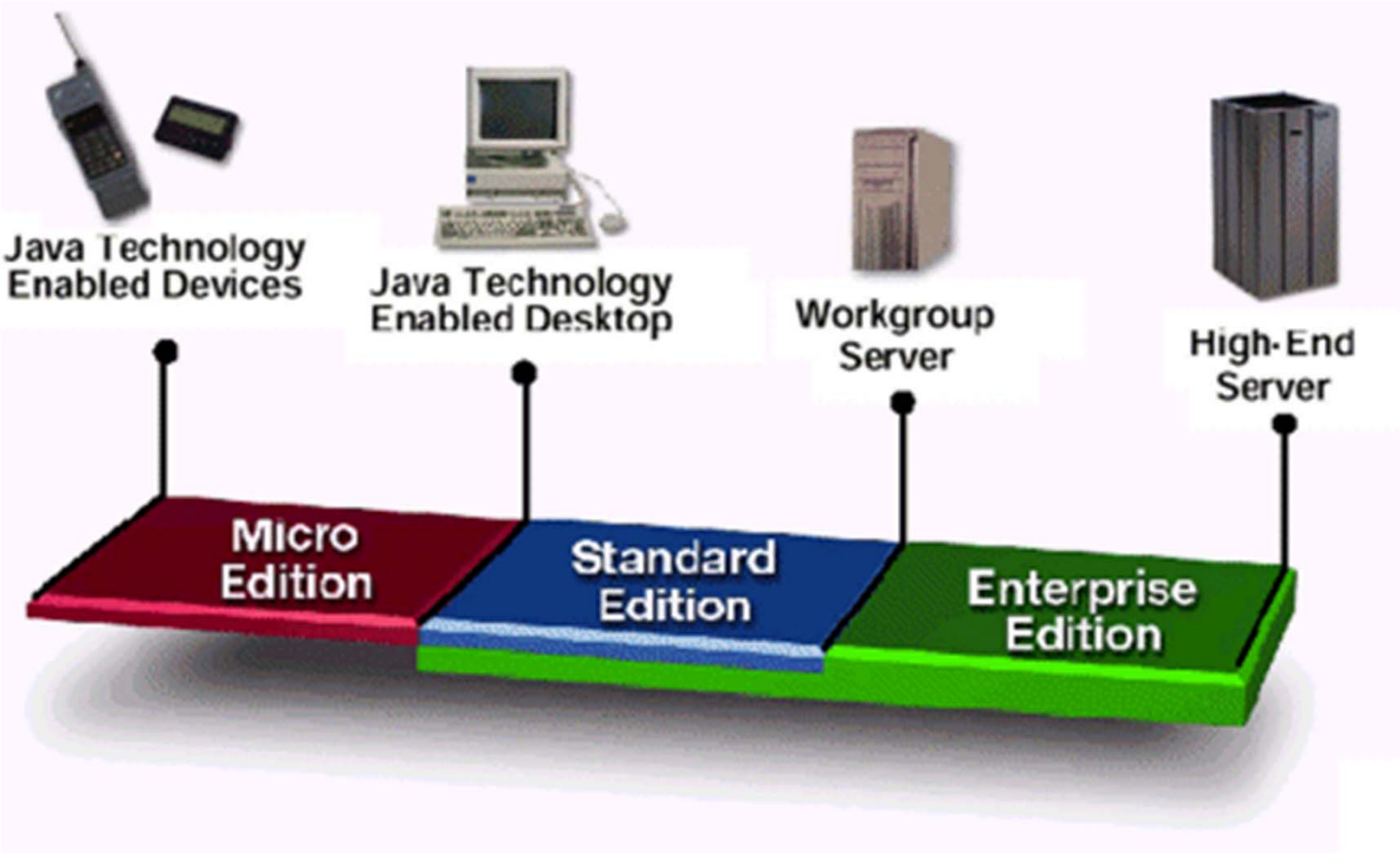
Bevezető

- Hol és miért van szükség a JAVÁra az adatbányászatban?
  - Programozott végrehajtás (imperatív, WF…)
  - Platform (JRE…)
  - Nagy adathalmazok kezelése (Cognos, Pentaho BI Suite…)
  - Integrációs feladatok (JCA, EAI…)
  - …
- Mi is a JAVA tulajdonképpen?
  - Platform,
  - nyelv,
  - valamint az ezeken implementált keretrendszerek összessége.
- Miért épp a JAVA?
  - Ingyenes és nyílt, szemben a .NET-tel
  - Eszközök széles spektrumán megtalálható (lásd később: platformok)

- **A Java, mint platform**
- A Java, mint nyelv
  - OO paradigmák
  - Tervezési minták
- Tooling
  - Eclipse, RCP

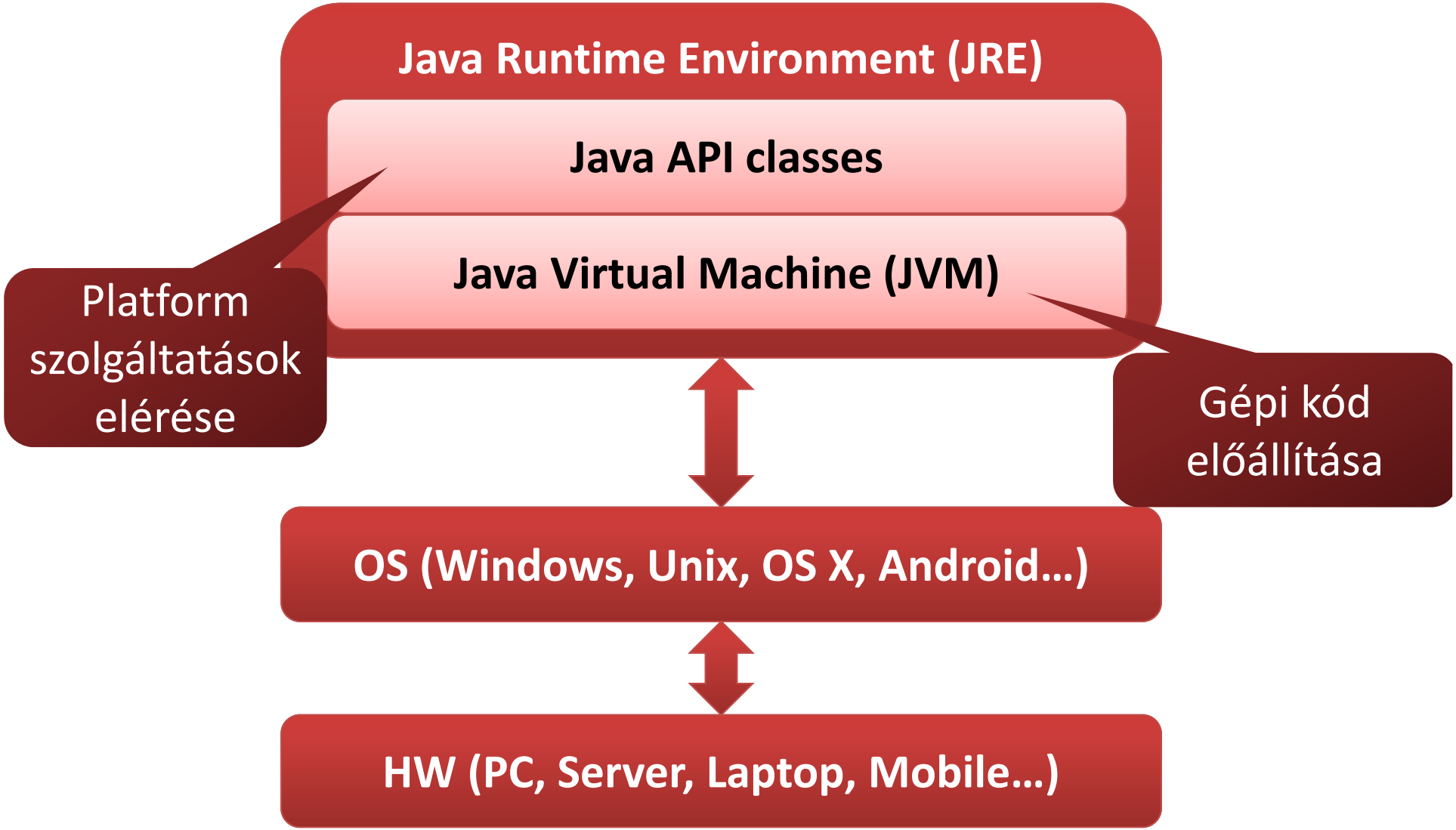
A Java, mint platform

- Java ME, Java SE, Java EE (JME, JSE, JEE)
- Korábban: ~~J2ME, J2SE, J2EE~~



Java Virtual Machine (JVM)

- Virtuális gép koncepció



API (Application programming interface)

- Cél
  - Megvalósított funkcionalitás kiajánlása
  - Anélkül, hogy a forrást ki kéne adni
- Publikus interfész a külvilág felé, de „black box” struktúra
- Megvalósítás
  - Megírt, lefordított forráskód JAR fájlba csomagolva

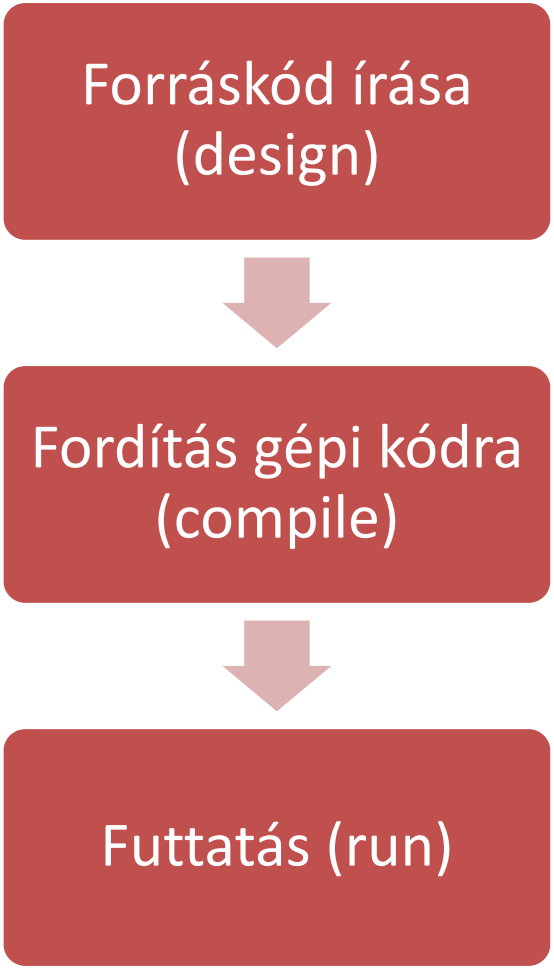
Java archive

- Honnan tudjuk, hogy milyen metódusok hívhatók?
  - Javadoc az API-hoz

- A Java, mint platform
- **A Java, mint nyelv**
  - **OO paradigmák**
  - **Tervezési minták**
- Tooling
  - Eclipse, RCP

A Java, mint nyelv

- Jellemzői
  - Objektum-orientált
  - Robosztus, biztonságos
  - Architektúra-független
  - Nagy teljesítmény
- Mi kell hozzá, ha futtatni akarom?
  - JRE (implements JVM)
- Mi kell hozzá, ha fejleszteni akarok?
  - JDK (includes JRE, compilers, etc)
- A fejlesztés folyamata
  - „design-time”, „compile-time”, „run-time”
  - run-time ≠ runtime



OO alapfogalmak

- Osztály
- Objektum
- Konstruktor
  - Destruktor, Garbage collector (GC)
- Metódus
  - Függvény
- Attribútumok (tagváltozók)
  - Access modifiers
- Statikus elemek (osztályok, metódusok, változók)

Osztály (Class)

- Egy konkrét fogalom absztrakciója, illetve annak modellje
  - Pl.: ház, fa, kutya, ember…
- Mivel jellemezhető?
  - Osztály azonosító (*class id*) – kötelező
  - Tulajdonságok (*properties*)
  - Az osztályon végrehajtható műveletek (*method*)
  - Absztrakciós szint
  - Kapcsolatok más osztályokkal
  - …

Encapsulation

```
class Ember {
  String nev;
  int kor;

  void setKor(int k){
     kor = k;
  }
}
```

Visszatérési érték.
(void = semmi)

Objektum (Object)

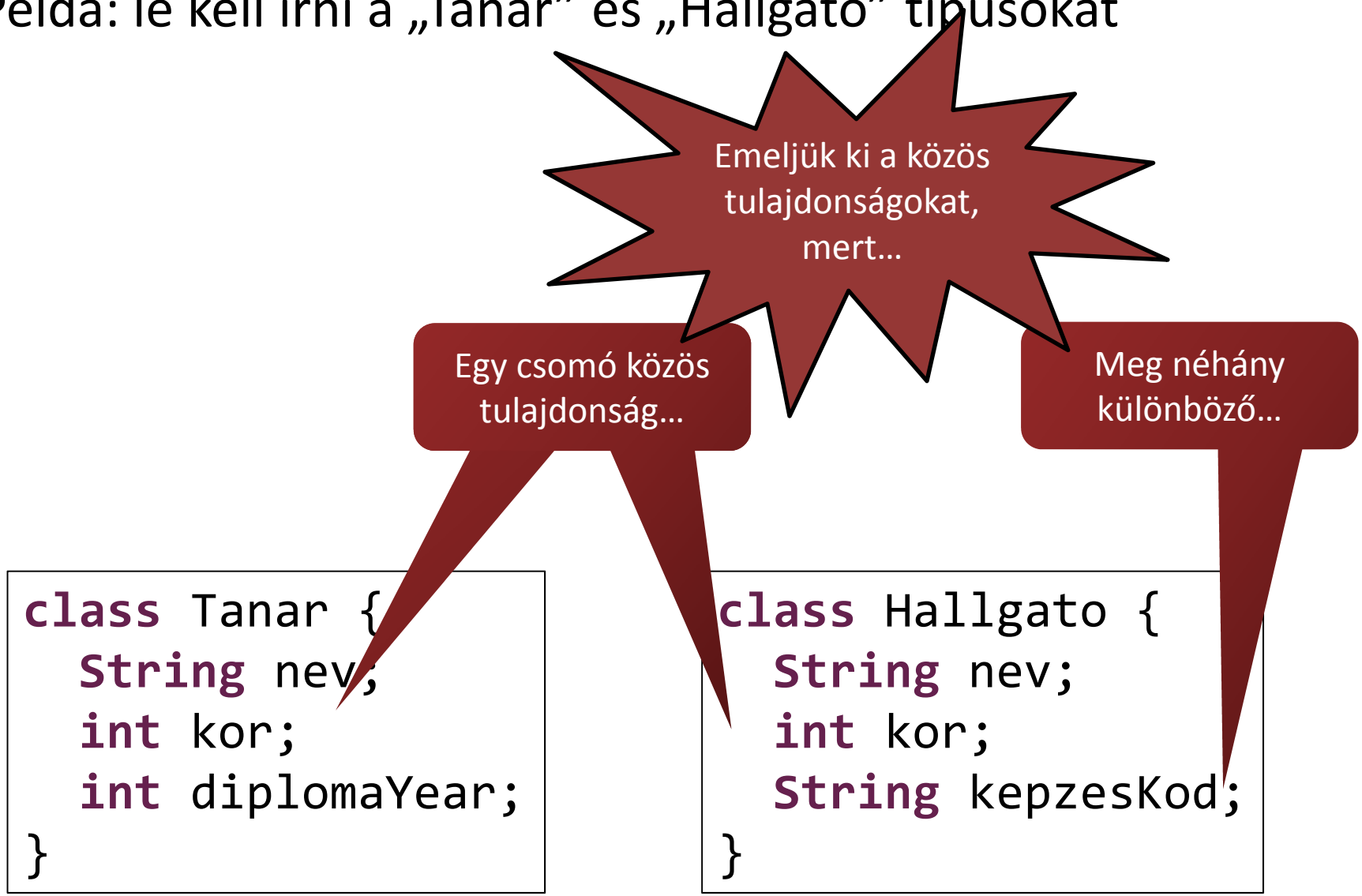
- Más néven: Példány (Instance)
- Minek a példánya?
  - Egy osztályé!
- Példa: Géza az Ember osztály példánya
- A példányosítás menete:
  - 0. ismerni kell az osztályt
  - 1. megadjuk a típust és a new kulcsszóval új példányt hozunk létre
  - 2. megadjuk a tulajdonságokat

```
class Ember {
  String nev;
  int kor;
}
```

```
Ember geza = new Ember();
geza.nev = „Kiss Géza”;
geza.kor = 20;
```

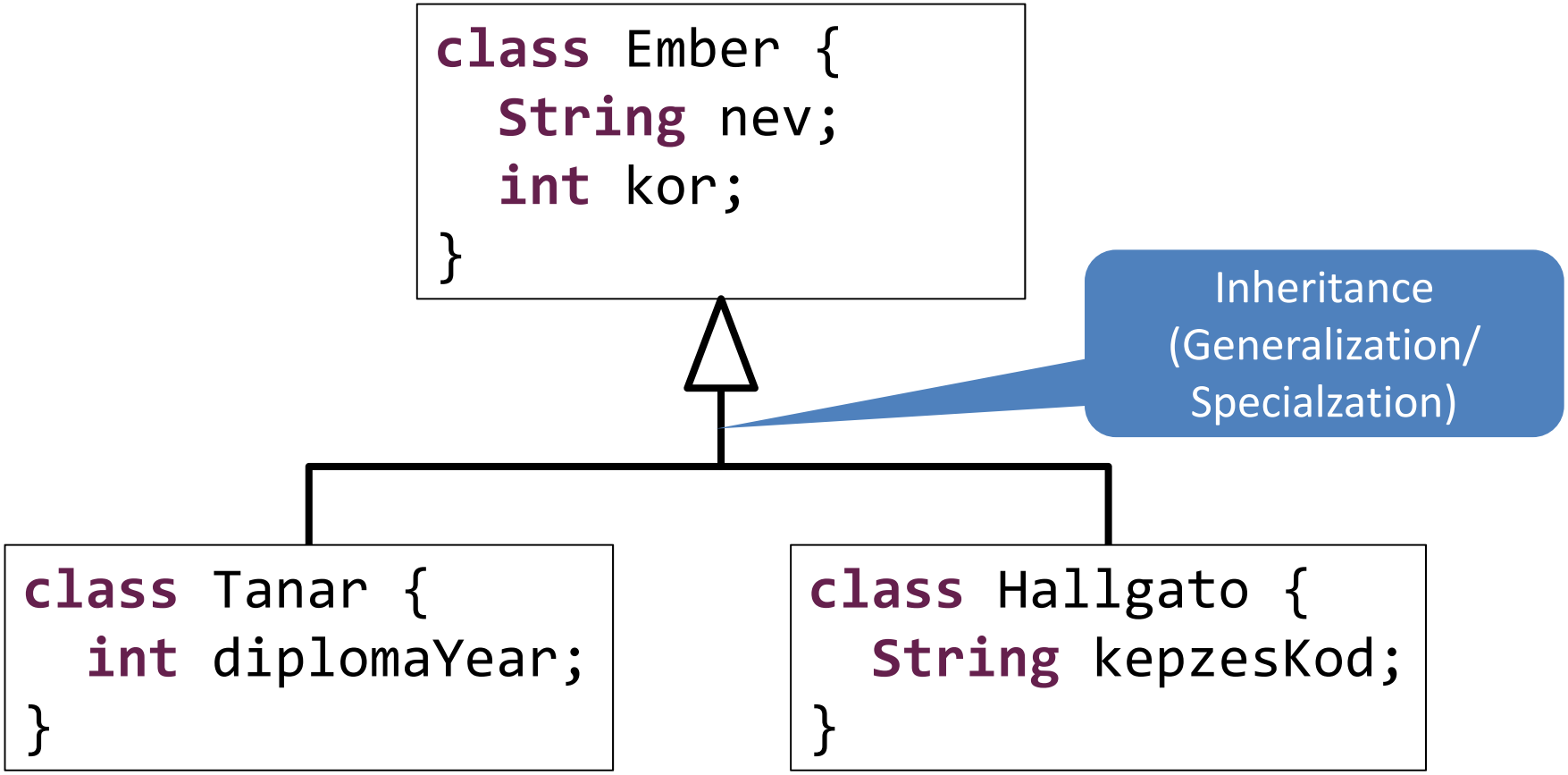
Kapcsolatok osztályok között

- Példa: le kell írni a „Tanár” és „Hallgató” típusokat



Kapcsolatok osztályok között

- Példa: le kell írni a „Tanár” és „Hallgató” típusokat



Kapcsolatok osztályok között

- Példa: le kell írni a „Tanár” és „Hallgató” típusokat

Kérdés: miből fogunk példányokat létrehozni?

Ebből NEM.

```
class Ember {
  String nev;
  int kor;
}
```

Inheritance (Generalization/ Specialzation)

```
class Tanar {
  int diplomaYear;
}
```

```
class Hallgato {
  String kepzesKod;
}
```

Absztrakt osztályok

Abstraction

- Nem példányosítható!
- Használjuk ha
  - Össze szeretnénk gyűjteni osztálytulajdonságok egy halmazát, de nem akarjuk, hogy példányosítsák az osztályt, mert nem egy értelmezett „üzleti koncepció”
  - Szeretnénk előírni az osztályra valamilyen végrehajtható viselkedést, de szeretnénk előírni ennek kötelező újradefiniálását minden leszármaztatott osztályban (abstract method)

```
abstract class Ember {
  String nev;
  int kor;
}
```

- Később megnézzük még egyszer

Osztály metódusok

- Az osztályon, mint struktúrán végrehajtható művelet

```
class Ember {
  String nev;
  int kor;

  void setKor(int k){
    kor = k;
  }
}
```

```
Ember geza = new Ember();
geza.nev = „Kiss Géza”;
geza.kor = 20;
geza.setKor(20);
```

ugyanaz a hatás!

Default konstruktor

- Mi történik itt?

```
Ember geza = new Ember();
```

- A new kulcsszó egy speciális osztálymetódust hív
- Konstruktor: a példány létrehozásáért felelős
- Szabályok
  - A neve mindig az osztály nevével azonos
  - Nincs visszatérési érték
- Default konstruktor
  - Ha nincs átadott paraméter
  - Nem kötelező definiálni (implicit definiált)
- Lehet írni más konstruktorokat is
  - Paraméterezhetők, stb

```
class Ember {
  String nev;
  int kor;

  Ember(){
  }
}
```

Összetettebb konstruktorok

```
Ember geza = new Ember();
geza.nev = „Kiss Géza”;
geza.kor = 20;
```

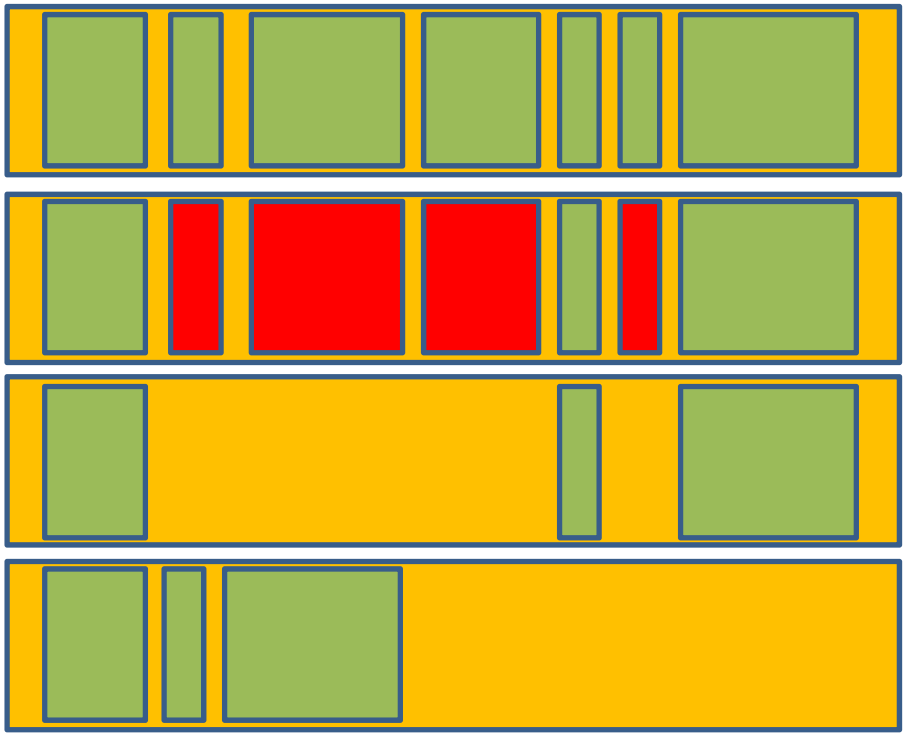
Polimorfizmus

```
Ember geza = new
    Ember(„Kiss Géza”, 20);
```

```
class Ember {
  String nev;
  int kor;

  Ember(){
  }

  Ember(String nev,
                int kor){
    this.nev = nev;
    this.kor = kor;
  }
}
```

Destruktorok, GC

- Mi lesz a nem használt objektumokkal?
- A heapen vannak, amíg…
- Destrukor
  - A konstruktor párja. Meg kell hívni és törli az objektumot.
  - **JAVÁban nincs!**
- Garbage Collector (GC)
- Ezzel tehát nem kell explicit törődni (többnyire)



Access modifiers

- Az objektumhoz való hozzáférés szabályható
- Tagváltozó és metódusszinten is
- **Public**: publikus elem, mindenki hozzáfér
- **Private**: csak az osztályon belül érhető el
- **Protected**: csak az osztályon belül és a leszármazottakból érhető el
- „Package protected” = no modifier

```
class Ember {
  String nev;
  int kor;
}
```

```
class Ember {
  public String nev;
  private int kor;
}
```

```
Ember geza = new Ember();
geza.nev = „Kiss Géza”;
geza.kor = 20;
```

Bean konvenciók

- Minden tagváltozó (attribute) private
- Minden tagváltozót getter és setter metódusokkal érünk el
- A getterek és setterek publikusak, neveik: getXXX és setXXX

```
class Ember {
  private String nev;

  public String getNev(){
    return this.nev;
  }
  public void setNev(String nev){
    this.nev = nev;
  }
}
```

```
Ember geza = new Ember();
geza.nev = „Kiss Géza”;
geza.setNev(„Kiss Géza”);
```

Statikus elemek

- Amit eddig láttunk: az osztályokat példányosítottuk
- Lehet példányosítás nélkül is osztályokkal dolgozni?
- Igen: static

```
class Greeting {
  static String sayHello(){
    return „Hello”;
  };
}
```

```
…
String hello = Greeting.sayHello();
…
```

Nem kell „new Greeting()”

- Van itt példányosítás? Ez mikor jó? Mikor szükséges?

Ciklusok: For, While, Foreach

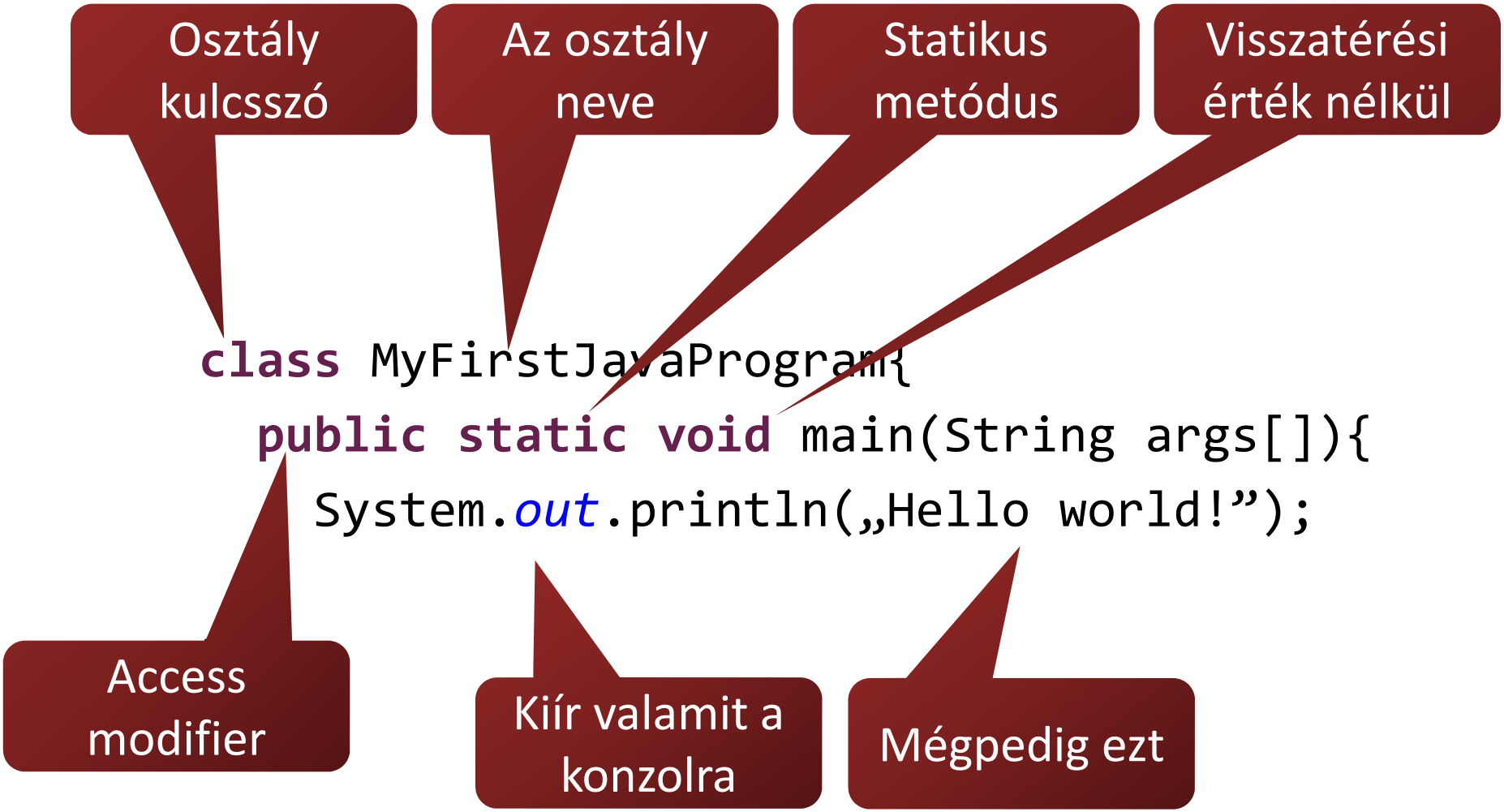
```
for (int i=0; i<10; i++){
  System.out.println(i);
}
```

```
int i = 0;
while (i<10){
  System.out.println(i);
  i++;
}
```

„while(true)”?

```
Collection<Ember> emberek = new ArrayList<Ember>();
emberek.add(Geza); emberek.add(Bela);
int sum;

for (Ember e : emberek){
  System.out.println(„I’m ” + e.getName());
  sum += e.getAge();
}
System.out.println(„AVG age:” + sum/emberek.size());
```

Hello world!


Osztály kulcsszó
Az osztály neve
Statikus metódus
Visszatérési érték nélkül
class MyFirstJavaProgram{
public static void main(String args[]){
System.out.println(„Hello world!”);
Access modifier
Kiír valamit a konzolra
Mégpedig ezt

Hello world! még egyszer

A végrehajtható szekvenciák osztályok metódusaiban vannak.

Nem minden osztályban van MAIN metódus, de egy programon belül mindig van egy valahol. Ugyanis itt „kezdődik” a végrehajtás.

```
class MyFirstJavaProgram{
  public static void main(String args[]){
    System.out.println(„Hello world!”);
  }
}
```

Ez a main metódus standard paraméterlistája.

- Kérdés: miért static a main metódus?

Hello advanced world!

```
class Student{
  public String name;

  public Student(String name){
    this.name = name;
  }
}

class MySecondJavaProgram{
  public static void main(String args[]){
    Student s = new Student(„Béla”);
    System.out.println(„Hello ” + s.name + „!”);
  }
}
```

Hello advanced world!

```
class Student{
  public String name;

  public Student(String name){
    this.name = name;
  }
}

class MySecondJavaProgram{
  public static void main(String args[]){
    Student s = new Student(„Béla”);
    System.out.println(„Hello ” + s.name + „!”);
  }
}
```

Nem minden osztályban van MAIN metódus, de egy programon belül mindig van egy valahol. Ugyanis itt „kezdődik” a végrehajtás.

Hello advanced world!

```
class Student{
  public String name;

  public Student(String name){
    this.name = name;
  }
}

class MySecondJavaProgram{
  public static void main(String
    Student s = new Student(„Béla
    System.out.println(„Hello ” +
  }
}
```

Az osztályok lefordulnak gépi kódra.

A runtime megkeresi a maint.

A main példányosít egyet a Student osztályból, Béla névvel.

Konzolra írás.

Összegezés

- Mindig lesz egy *static void main(String args[])* metódus
  - Itt „kezdődik” a program
- Osztályok segítségével írunk le összetett struktúrákat
  - Ez bármilyen absztrakt, vagy konkrét fogalom lehet
  - Osztály = struktúra + a rajta végezhető műveletek
- Általában a maint igyekszünk minimálizálni
  - „Kiszervezzük az üzleti logikát”
- Lehetőleg kövessük a bean konvenciókat

OO paradigmák

- Dynamic dispatch
- Öröklés (inheritance)
- Polimorfizmus (polymorphism)
- Egységbezárás (encapsulation)
- Absztrakció (abstraction)
- Open recursion

- **Interfész vs Absztrakt ősosztály**

Tömbök és generikusok

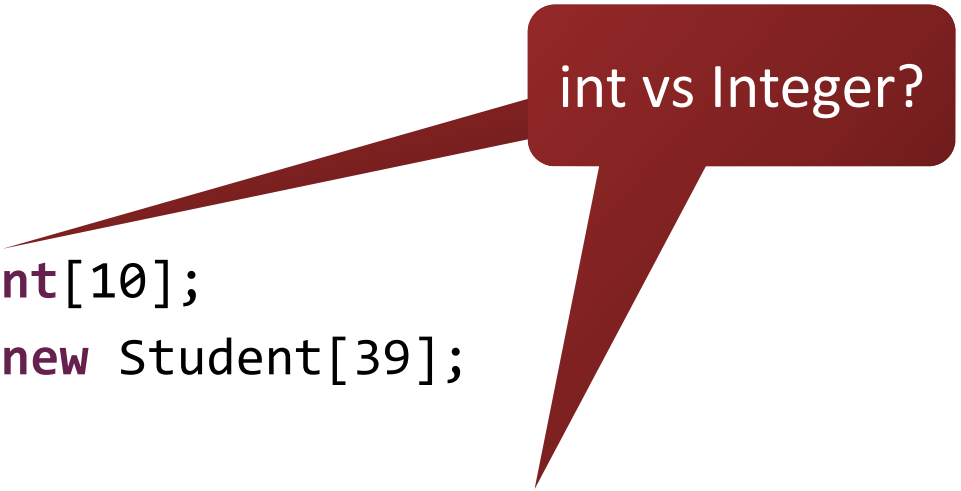
- Korábbi példa: *Ember* osztály és gyerekei

- Dinamikus struktúrák
  - Tömbök (Array)

```
int[] elements = new int[10];
Student[] hallgatok = new Student[39];
```

  - Generikusok (Generics)

```
List<Integer> elements = new ArrayList<Integer>();
List<Student> elements = new ArrayList<Student>();
```



Boxing

- int: primitív típus
  - Egy „nyers” 32, vagy 64 bites információegység
- Integer: objektum típus
  - Egy darab int tagváltozót tartalmaz
- Mikor melyiket?
  - int: jó performancia, + - * / végrehajtható rajta
  - Integer: szerializálható, RMI-kompatibilis, generikusokkal működik

```
int i = 6;
Integer j = new Integer(i);
int k = j.intValue();
```

Generikusok

- Melyik a jobb és miért?

```
List<Student> elements = new ArrayList<Student>();
Student[] hallgatok = new Student[39];
```

- Általános szabály:
  - A tömbök performancia szempontjából erősebbek
  - A generikusok könnyebben kezelhetőek (sok előre megírt kezelő metódus érhető el, mivel kihasználják az öröklési hierarchiát)
- További szempontok:
  - Kovariáns, kontravariáns és invariáns típusok
  - Típusbiztosság, szálbiztosság
- Komplexebb generikusok (nem ritka ez sem):

```
<T extends Annotation> T getAnnotation(Class<T> annotationType)
```

(Ha érdekel: http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/reflect/AnnotatedElement.html)

Tervezési minták

- *„Design patterns are recurring solutions to software design problems you find again and again in real-world application development.”* (http://www.dofactory.com/Patterns/Patterns.aspx)
- Létrehozási minták (Creational patterns)
  - **Singleton**, **Factory**, Abstract Factory, Builder, Prototype
- Strukturális minták (Structural patterns)
  - Composite, Facade, Bridge, Adapter, Proxy, Flyweight, Decorator
- Viselkedési minták (Behavioral patterns)
  - Observer, Mediator, Memento, Iterator, Strategy, Visitor

Tervezési minták

- Singleton
  - One instance of a class or one value accessible globally in an application.
- Factory
  - Provides an abstraction or an interface and lets subclass or implementing classes decide which class or method should be instantiated or called, based on the conditions or parameters given.
- AbstractFactory
  - Provides one level of interface higher than the factory pattern. It is used to return one of several factories.

Forrás: http://www.javacamp.org/designPattern/

Singleton

- One instance of a class or one value accessible globally in an application.

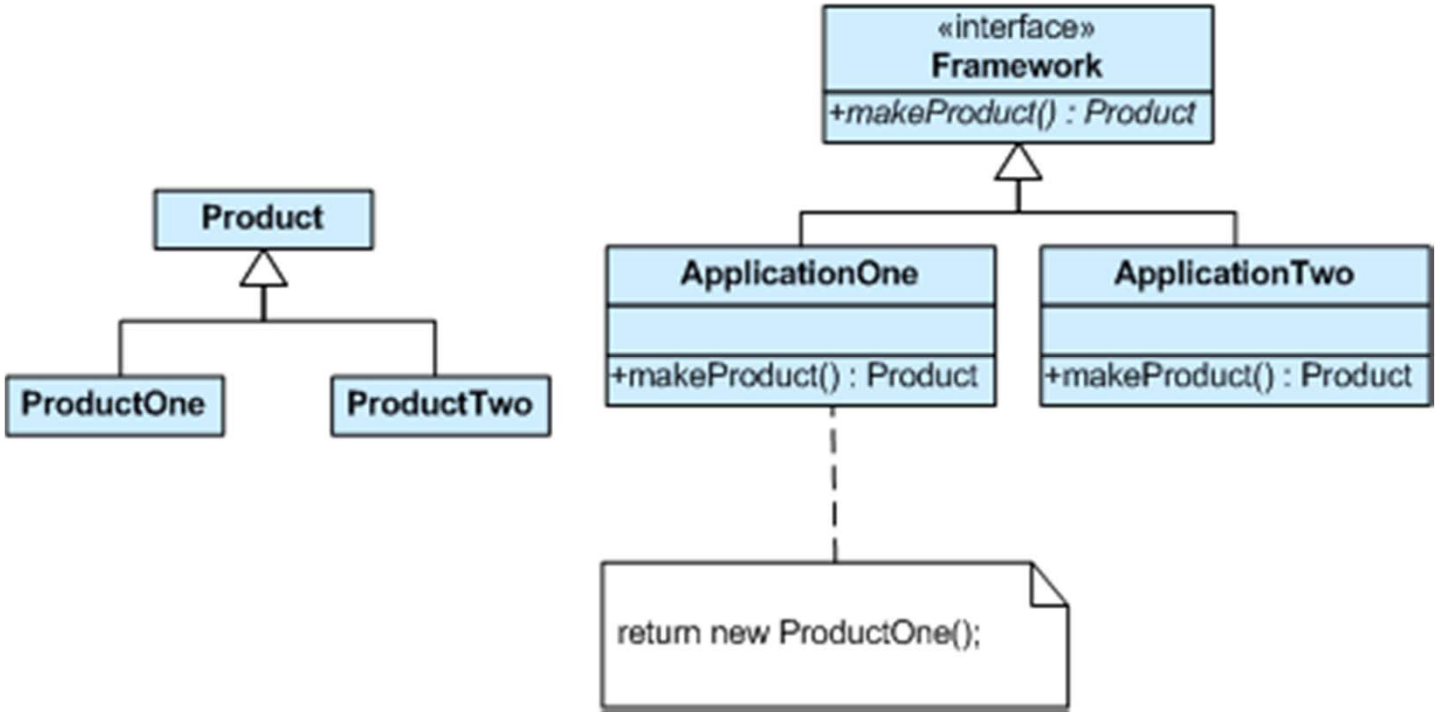
```
public class ClassicSingleton{
  private static ClassicSingleton instance = null;

  private ClassicSingleton(){};

  public static ClassicSingleton getInstance(){
    if(instance == null){
      instance = new ClassicSingleton();
    }
    return instance;
  }
}
```

Ehhez hozzá se férünk kívülről!

```
ClassicSingleton instance = ClassicSingleton.getInstance();
```

Factory

- Creates an instance of several derived classes.




```
Product p1 = ApplicationOne.makeProduct();
Product p2 = ApplicationTwo.makeProduct();
```

Nem kell tudnom, hogy a kód adott helyén milyen *Product*ra van szükségem – majd a Factory tudja.

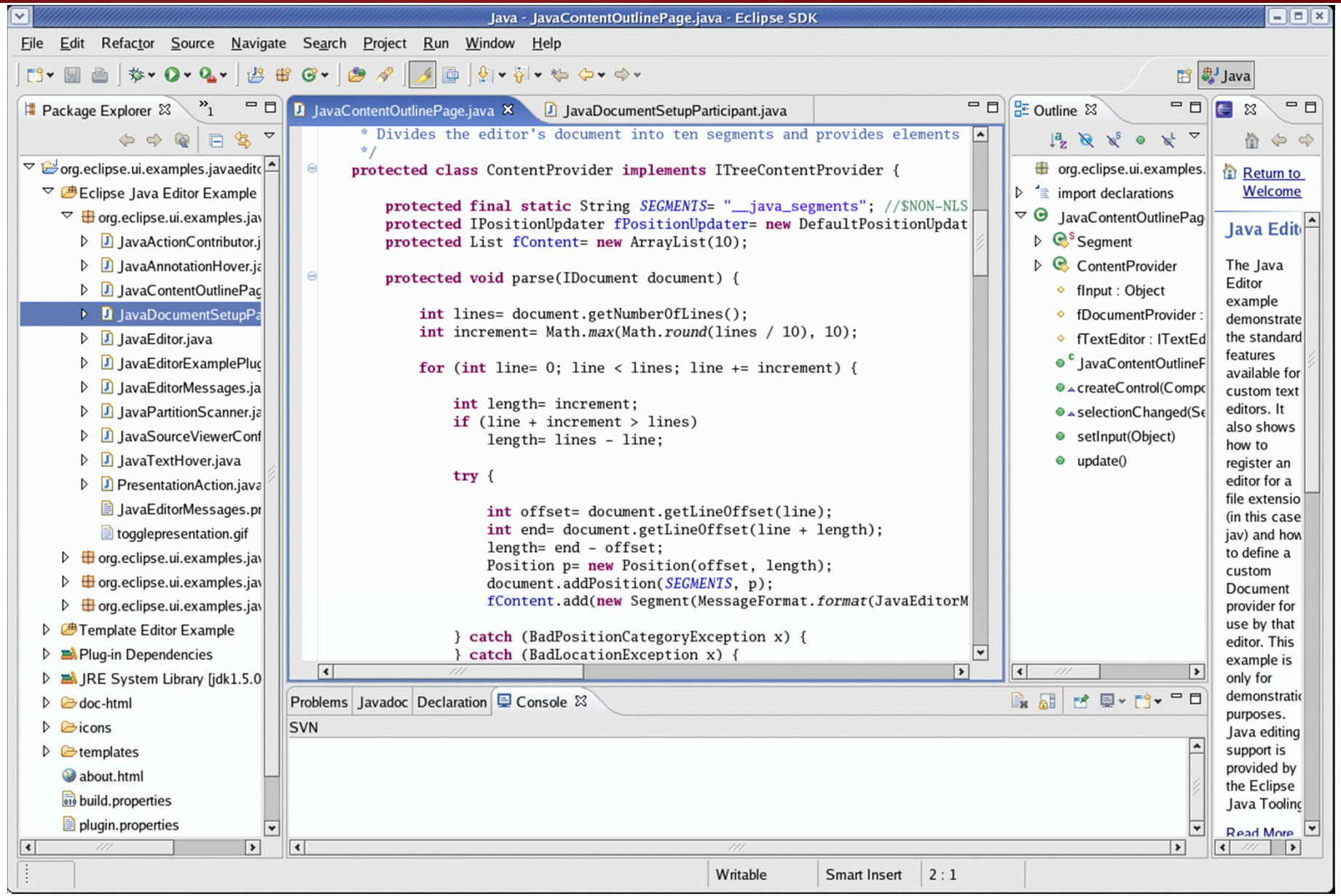
Tooling

- A Java, mint platform
- A Java, mint nyelv
  - OO paradigmák
  - Tervezési minták
- **Tooling**
  - **Eclipse, RCP**

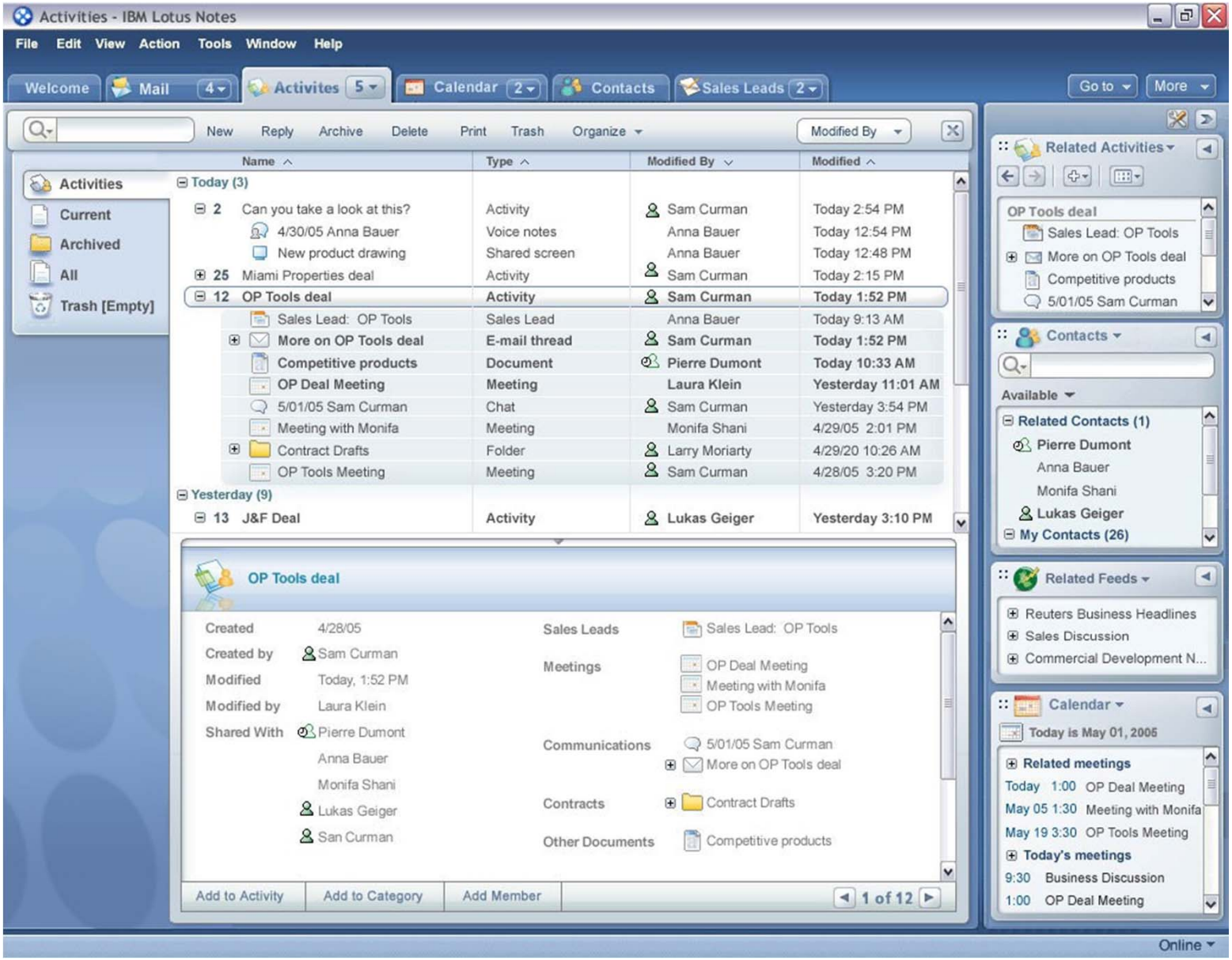
Eclipse

- Első megközelítésben: IDE
- Valójában: platform+tooling
- EMF, RCP, SWT, GMF, Mylyn, Equinox, OSGi, VIATRA…
- Knime, IBM Rational/WebSphere/Lotus Tools, Zend Studio…

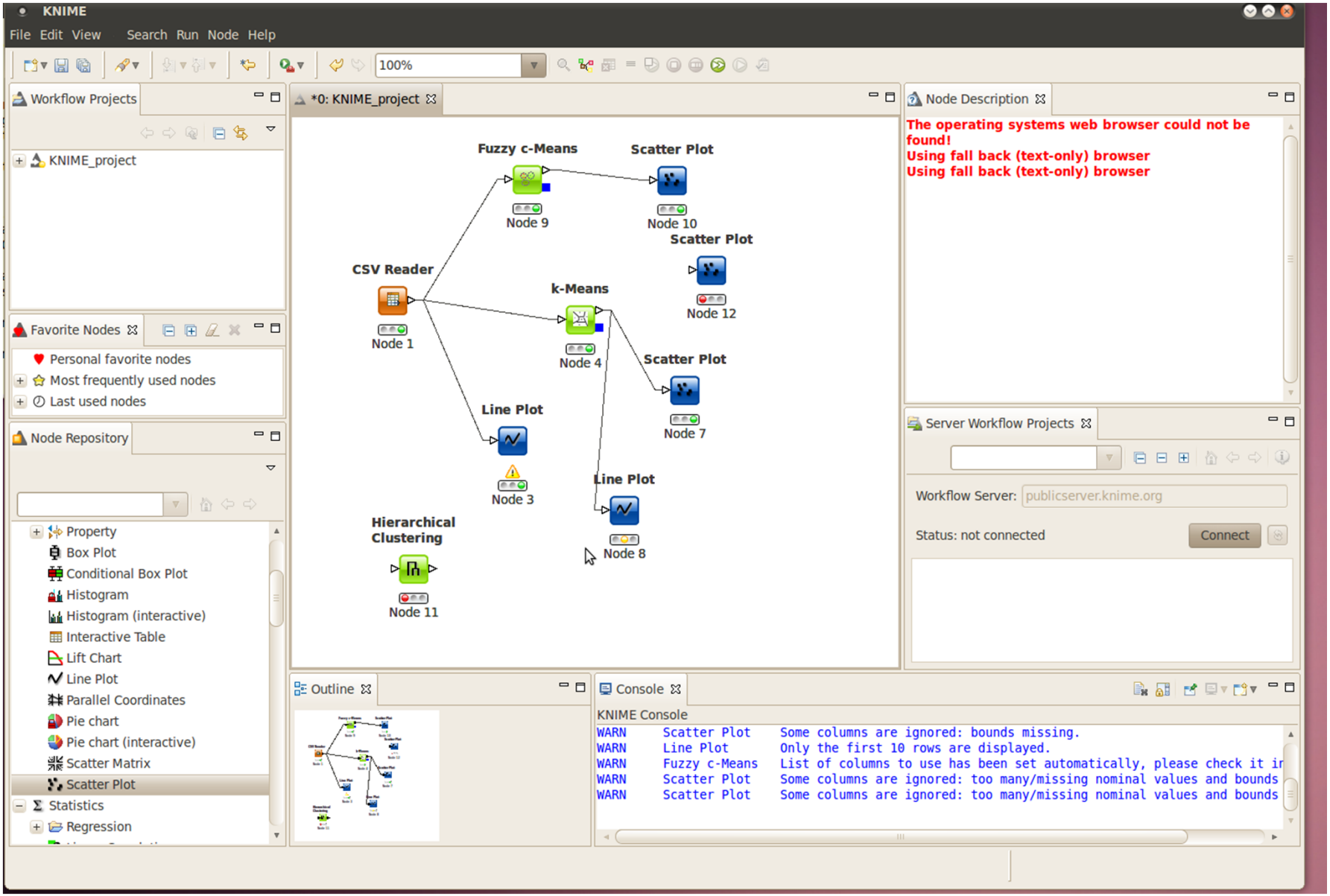
Eclipse IDE


Java - JavaContentOutlinePage.java - Eclipse SDK
File Edit Refactor Source Navigate Search Project Run Window Help
Java
Package Explorer
org.eclipse.ui.examples.javaedit
Eclipse Java Editor Example
org.eclipse.ui.examples.jav
JavaActionContributor.j
JavaAnnotationHover.ja
JavaContentOutlinePag
JavaDocumentSetupPa
JavaEditor.java
JavaEditorExamplePlug
JavaEditorMessages.ja
JavaPartitionScanner.ja
JavaSourceViewerConf
JavaTextHover.java
PresentationAction.java
JavaEditorMessages.pr
togglepresentation.gif
org.eclipse.ui.examples.jav
org.eclipse.ui.examples.jav
org.eclipse.ui.examples.jav
Template Editor Example
Plug-in Dependencies
JRE System Library [jdk1.5.0
doc-html
icons
templates
about.html
build.properties
plugin.properties
JavaContentOutlinePage.java
JavaDocumentSetupParticipant.java
* Divides the editor's document into ten segments and provides elements
*/
protected class ContentProvider implements ITreeContentProvider {
protected final static String SEGMENTS= "__java_segments"; //$NON-NLS
protected IPositionUpdater fPositionUpdater= new DefaultPositionUpdat
protected List fContent= new ArrayList(10);
protected void parse(IDocument document) {
int lines= document.getNumberOfLines();
int increment= Math.max(Math.round(lines / 10), 10);
for (int line= 0; line < lines; line += increment) {
int length= increment;
if (line + increment > lines)
length= lines - line;
try {
int offset= document.getLineOffset(line);
int end= document.getLineOffset(line + length);
length= end - offset;
Position p= new Position(offset, length);
document.addPosition(SEGMENTS, p);
fContent.add(new Segment(MessageFormat.format(JavaEditorM
} catch (BadPositionCategoryException x) {
} catch (BadLocationException x) {
Outline
org.eclipse.ui.examples.
import declarations
JavaContentOutlinePag
Segment
ContentProvider
fInput : Object
fDocumentProvider :
fTextEditor : ITextEd
JavaContentOutlineF
createControl(Compc
selectionChanged(Se
setInput(Object)
update()
Return to Welcome
Java Edit
The Java Editor example demonstrate the standard features available for custom text editors. It also shows how to register an editor for a file extensio (in this case jav) and how to define a custom Document provider for use by that editor. This example is only for demonstratic purposes. Java editing support is provided by the Eclipse Java Tooling
Read More
Problems Javadoc Declaration Console
SVN
Writable
Smart Insert
2 : 1

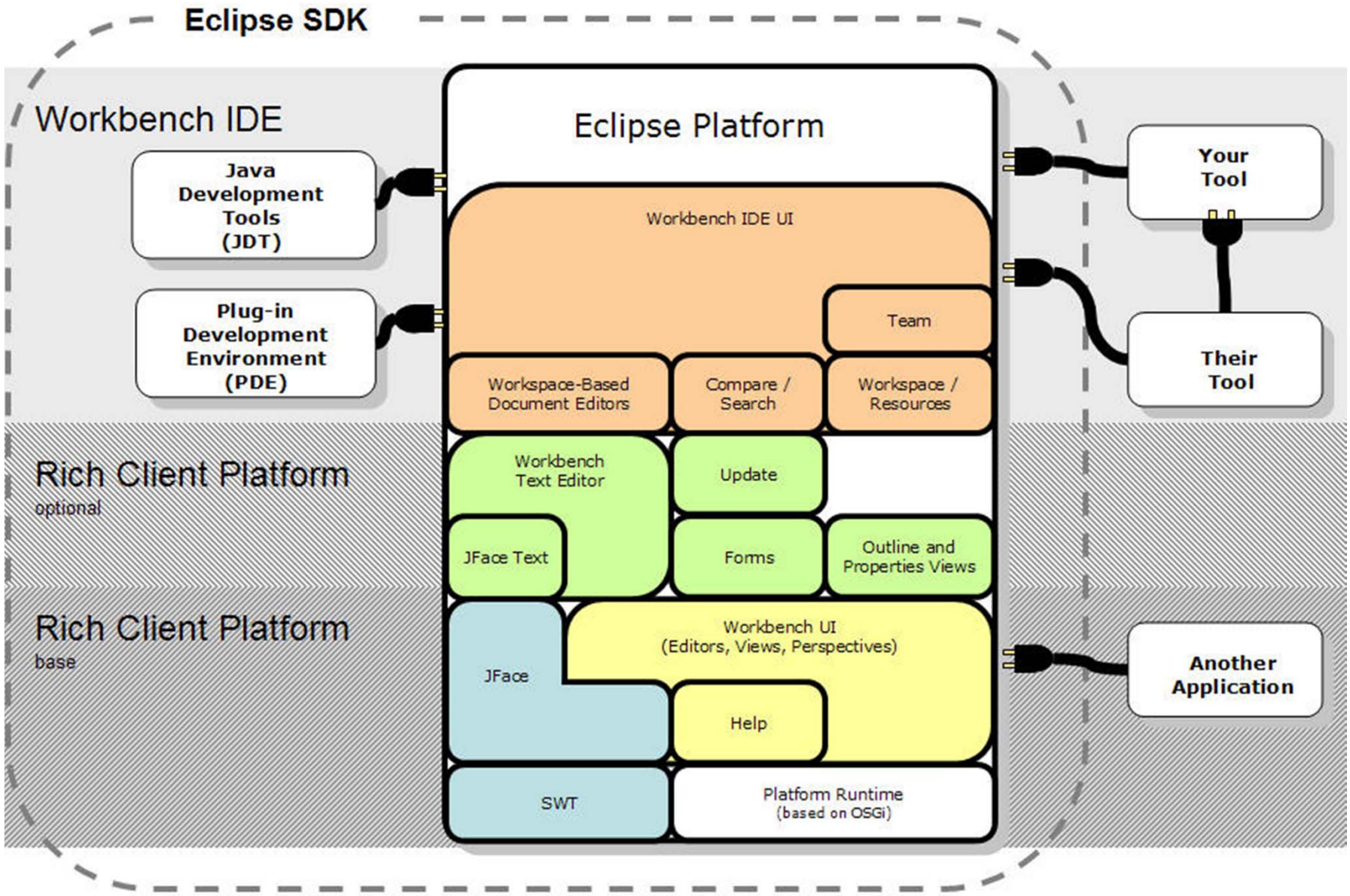
Ez is Eclipse (Lotus Notes 8)



És ez is (KNIME)

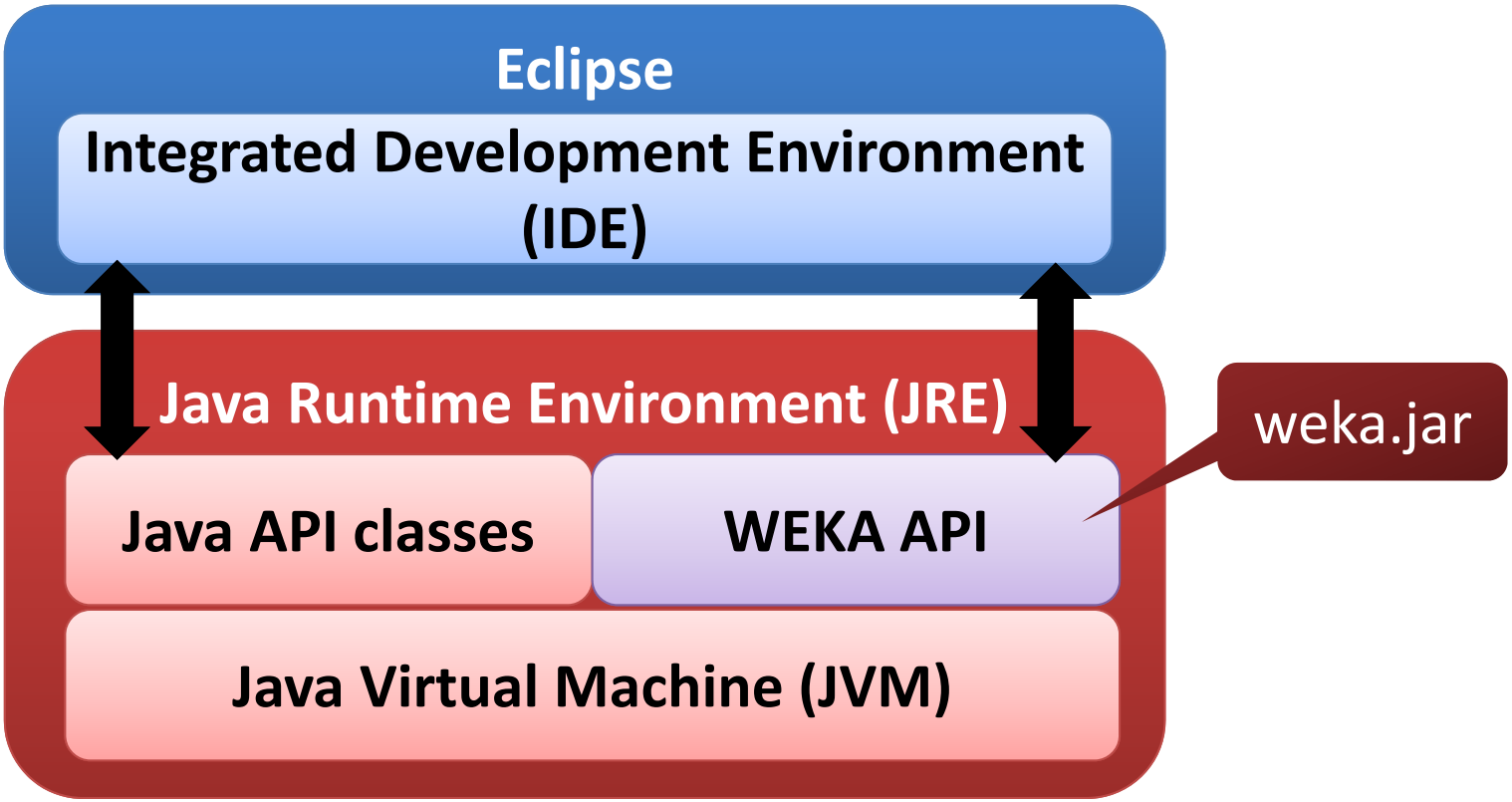


És ez van mögötte (Eclipse architektúra)



Labor környezet

- Amit a laborban használunk (és a házihoz is ajánlunk)



Keywords

- Class, instance, method, property, access modifier, static
- Interface vs. abstract base class
- OO paradigms: inheritance, polymorphism, abstraction
- Java language structures: arrays, generics
- Main method
- Loops: for, while, foreach
- JVM, JRE, JDK, Eclipse
- API, javadoc
- Design patterns: Singleton, Factory