Systems for Design and Implementation

2015-2016

Course 6

Contents

Java XML API

- Two types:
  - Streaming: SAX, StAX
  - Document Object Model: DOM
- **`javax.xml.parsers:`**
  - **`SAXParserFactory`**, **`DocumentBuilderFactory`**, and **`TransformerFactory`**
- **`org.w3c.dom`**: Defines the Document class (a DOM) as well as classes for all the components of a DOM
- **`org.xml.sax`**: Defines the basic SAX APIs
- **`javax.xml.transform`**: Defines the XSLT APIs that let you transform XML into other forms
- **`javax.xml.stream`**: Contains the basic StAX API

Java XML API

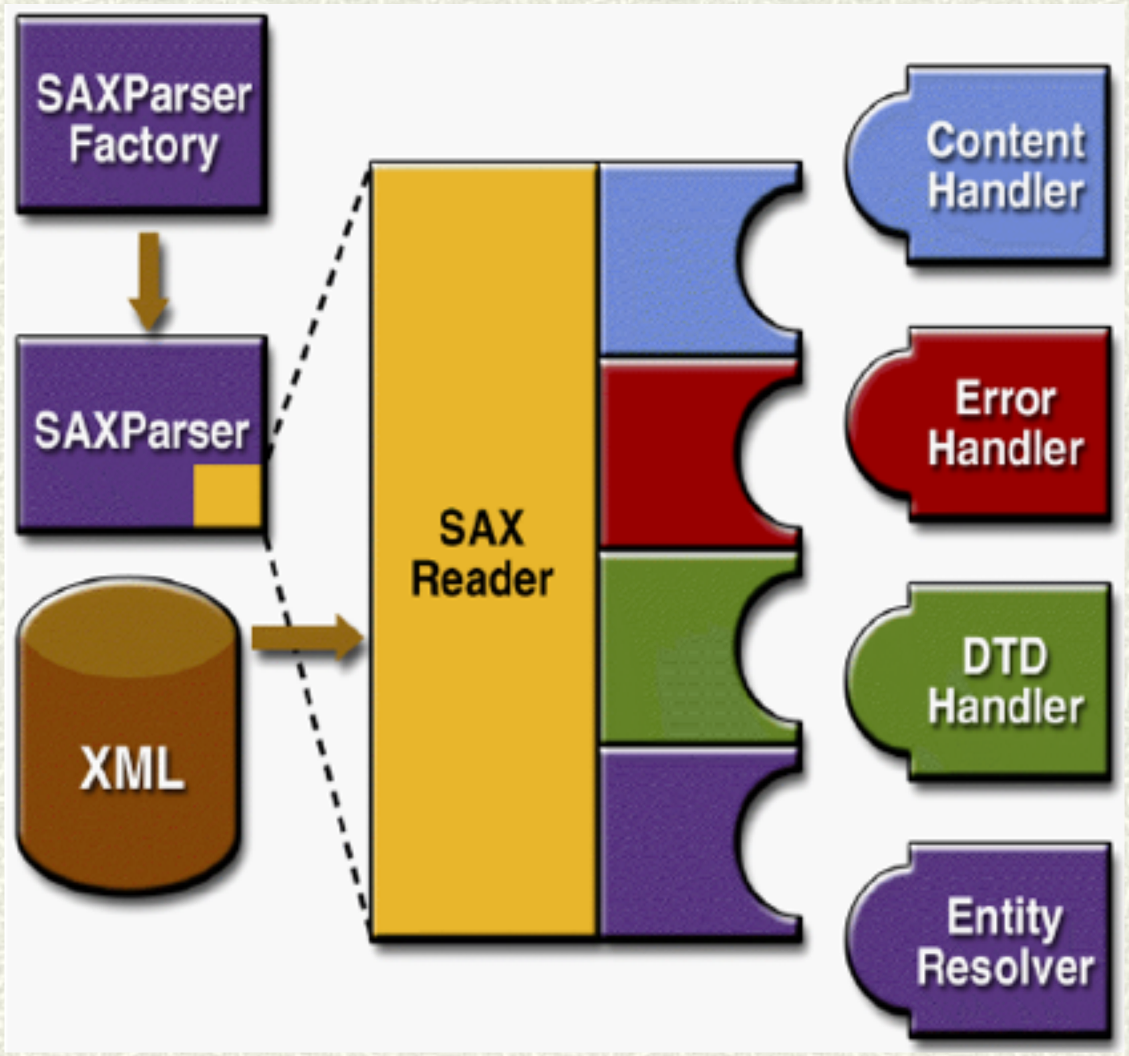
Feature	**StAX**	**SAX**	**DOM**
API Type	Pull, streaming	Push, streaming	In memory tree
Ease of Use	High	Medium	High
XPath Capability	No	No	Yes
CPU and Memory Efficiency	Good	Good	Varies
Forward Only	Yes	Yes	No
Read XML	Yes	Yes	Yes
Write XML	Yes	No	Yes
Create, Read, Update, Delete	No	No	Yes

SAX API

- Package `org.xml.sax`

`Attributes`	Interface for a list of XML attributes
`ContentHandler`	Receive notification of the logical content of a document.
`DTDHandler`	Receive notification of basic DTD-related events.
`EntityResolver`	Basic interface for resolving entities.
`ErrorHandler`	Basic interface for SAX error handlers.
`XMLReader`	Interface for reading an XML document using callbacks.
`SAXException`	Encapsulate a general SAX error or warning.
`SAXNotRecognizedException`	Exception class for an unrecognized identifier.
`SAXNotSupportedException`	Exception class for an unsupported operation.
`SAXParseException`	Encapsulate an XML parse error or warning.

SAX


SAXParser Factory
SAXParser
XML
SAX Reader
Content Handler
Error Handler
DTD Handler
Entity Resolver

ContentHandler

- **`startDocument`**() Receive notification of the beginning of a document.
- **`endDocument`**() Receive notification of the end of a document.
- **`startElement`**(String uri, String localName, String qName, Attributes atts) Receive notification of the beginning of an element.
- **`endElement`**(String uri, String localName, String qName) Receive notification of the end of an element.
- **`characters`**(char[] ch, int start, int length) Receive notification of character data.

- **`ignorableWhitespace`**(char[] ch, int start, int length) Receive notification of ignorable whitespace in element content.
- **`processingInstruction`**(String target, String data) Receive notification of a processing instruction.
- **`startPrefixMapping`**(String prefix, String uri) Begin the scope of a prefix-URI Namespace mapping.
- **`endPrefixMapping`**(String prefix) End the scope of a prefix-URI mapping.
- Others

ErrorHandler

- `error(SAXParseException exception)` Receive notification of a recoverable error.
- `fatalError(SAXParseException exception)` Receive notification of a non-recoverable error.
- `warning(SAXParseException exception)` Receive notification of a warning.

Parsing an XML using SAX

- Implement the ContentHandler interface, or extend DefaultHandler class

```
class MyHandler extends DefaultHandler{
      public void startElement(…){…}
      public void endElement(…){ …}
      public void characters(…){…}
}
```

- Create and obtain the parser:

```
 MyHandler handler = new MyHandler();
 SAXParserFactory factory = SAXParserFactory.newInstance();
 try {
      SAXParser saxParser = factory.newSAXParser();
      saxParser.parse( new File("file.xml"), handler );
 } catch (Throwable t) { …}
```

- Obtain the result from the handler.

Parsing an XML using SAX

- Validating parser:
  - DTD

```
MyHandler handler = new MyHandler();
SAXParserFactory factory = SAXParserFactory.newInstance();
factory.setValidating(true);
try {
    SAXParser saxParser = factory.newSAXParser();
    saxParser.parse( new File("file.xml"), handler );
} catch (Throwable t) { …}
```

  - XML Schema : properties must be set

```
static final String JAXP_SCHEMA_LANGUAGE ="http://java.sun.com/xml/
   jaxp/properties/schemaLanguage";
static final String W3C_XML_SCHEMA="http://www.w3.org/2001/
   XMLSchema";
    SAXParserFactory factory = SAXParserFactory.newInstance();
    factory.setNamespaceAware(true);
    factory.setValidating(true);
    saxParser.setProperty(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);
```

DOM Java API

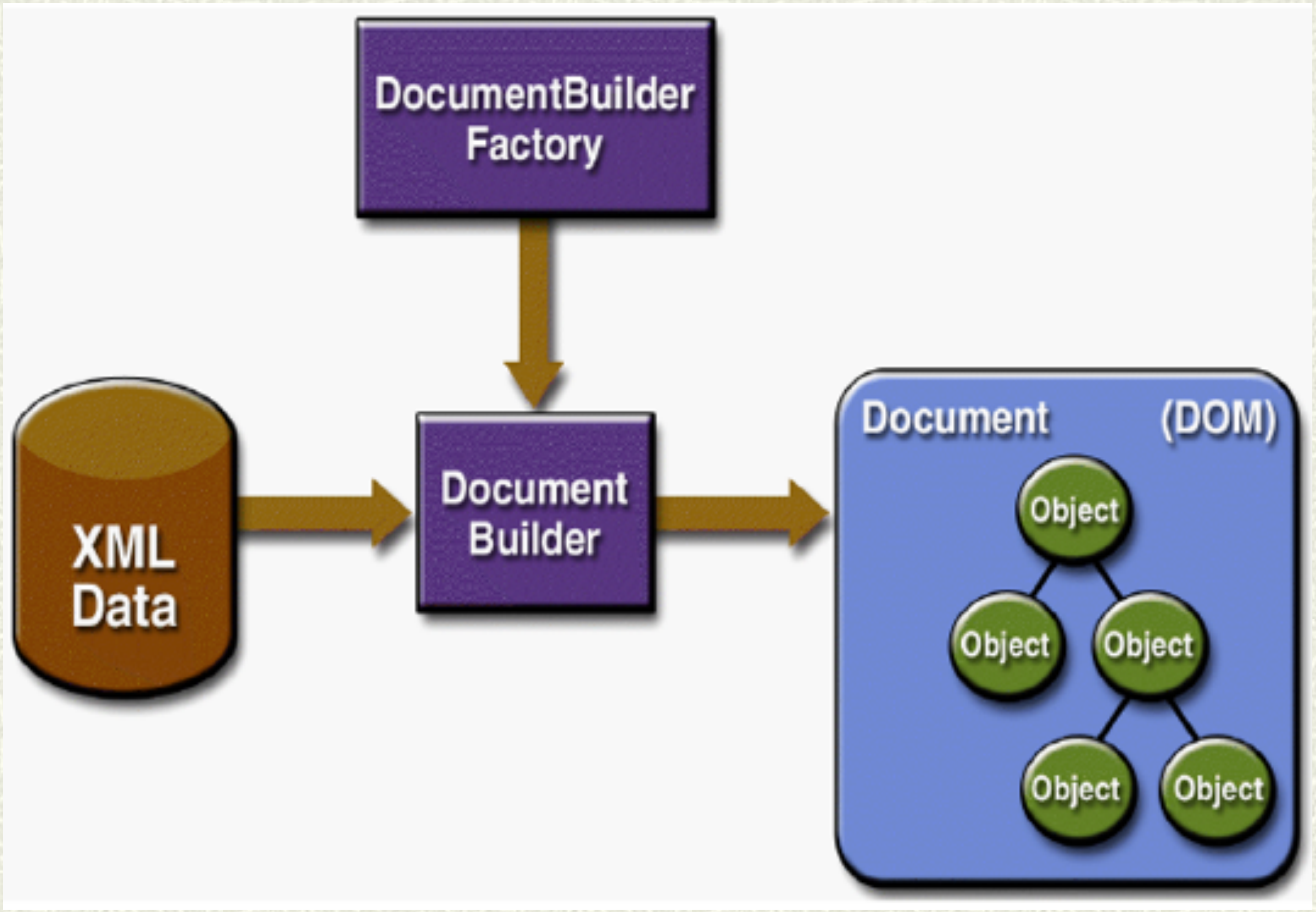
- Package `org.w3c.dom`
- It contains interfaces for node types:
  - `Attr, CDATASection, Comment, Document, Element, Entity, Node, NodeList, Text`
- Instantiating a DOM Parser:

```
DocumentBuilderFactory factory =DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
document = builder.parse( new File("person.xml") );
```

- Validation

```
DocumentBuilderFactory factory =DocumentBuilderFactory.newInstance();
factory.setValidating(true);
factory.setNamespaceAware(true);
```

DOM


DocumentBuilder Factory
XML Data
Document Builder
Document
(DOM)
Object
Object
Object
Object
Object

Document Interface

- Represents the entire XML document.
- It contains the factory methods to create elements, text nodes, attributes, comments, etc.
- Methods
  - `createAttribute (String attrName)`
  - `createElement(String tagName)`
  - `createTextNode(String data)`
  - `getElementById(String elementId)`
  - `getElementsByTagName(String tagname) : NodeList`
- The Node interface is used for iterating and for creating the tree:
  - `getAttributes(), getChildNodes(), getFirstChild(), getNextSibling(), getNodeName()`
  - `appendChild(…), removeChild(…), replaceChild(…)`

StAX API

- It exposes methods for iterative, event-based processing of XML documents.
- XML documents are treated as a filtered series of events, and infoset states can be stored in a procedural fashion.
- It is bidirectional, enabling both reading and writing of XML documents.
- It contains two distinct API sets: a cursor API and an iterator API.
- Cursor API:
  - It represents a cursor with which you can walk an XML document from beginning to end.
  - This cursor can point to one thing at a time, and always moves forward, never backward, usually one infoset element at a time.
- Iterator API:
  - It represents an XML document stream as a set of discrete event objects.
  - These events are pulled by the application and provided by the parser in the order in which they are read in the source XML document.

StAX - Cursor API

- Interfaces: `XMLStreamReader` and `XMLStreamWriter`
- `XMLStreamReader` includes accessor methods for all possible information retrievable from the XML document: document encoding, element names, attributes, namespaces, text nodes, start tags, comments, processing instructions, etc.
- Methods such as `getText` and `getName` get the data at the current cursor location.

```
public interface XMLStreamReader {
    public int next() throws XMLStreamException;
    public boolean hasNext() throws XMLStreamException;

    public String getText();
    public String getLocalName();
    public String getNamespaceURI();
    // ... other methods
}
```

StAX - Cursor API

- `XMLStreamWriter` provides methods that correspond to `StartElement` and `EndElement` event types.

```
public interface XMLStreamWriter {
     public void writeStartElement(String localName) throws
    XMLStreamException;
     public void writeEndElement() throws XMLStreamException;
     public void writeCharacters(String text) throws XMLStreamException;
     // ... other methods
}
```

StAX - Iterator API

- It represents an XML document stream as a set of discrete event objects.
- These events are pulled by the application and provided by the parser in the order in which they are read in the source XML document.
- The base iterator interface is called `XMLEvent`, and there are subinterfaces for each event type: `StartDocument`, `EndDocument`, `StartElement`, `EndElement`, `Characters`, `Comment`, etc.
- `XMLEventReader` interface contains five methods:
  - most important `nextEvent`, which returns the next event in an XML stream.

```
public interface XMLEventReader extends Iterator {
    public XMLEvent nextEvent() throws XMLStreamException;
    public boolean hasNext();
    public XMLEvent peek() throws XMLStreamException;
    // ...
}
```

StAX - Factory classes

- `XMLInputFactory`, `XMLOutputFactory`, and `XMLEventFactory` are used to define and configure implementation instances of XML stream reader, stream writer, and event classes.

```
XMLInputFactory f = XMLInputFactory.newInstance();
InputStream fileInputStream =
   ClassLoader.getSystemResourceAsStream(filename);
XMLStreamReader r = f.createXMLStreamReader(fileInputStream);
while(r.hasNext()) {
    r.next();
//…
}

XMLOutputFactory output = XMLOutputFactory.newInstance();
XMLStreamWriter writer = output.createXMLStreamWriter( ... );
writer.writeStartDocument();
//…
writer.flush();
```

XML Serialization

- Java beans can be serialized as XML documents.
- A Java bean is a Java object that is serializable, has a default constructor, and allows access to properties using getter and setter methods.

```
public class Person implements java.io.Serializable {
     private String name;
     private int age;
   public Person() {    }
   public String getName() { return name; }
   public void setName(String name) {this.name = name; }
   public int getAge() {    return age;   }
   public void setAge(int age) {this.age = age;}
}
```

- XMLEncoder, XMLDecoder `(package java.beans)`

XML Serialization

- Writing a java bean:

```
Person aPerson=new aPerson(“Ana”,23);
FileOutputStream fos = new FileOutputStream(“person.xml");
XMLEncoder xenc = new XMLEncoder(fos);
xenc.writeObject(aPerson);
```

- Reading a bean

```
 FileInputStream fos = new FileInputStream("foo.xml");
 XMLDecoder xdec = new XMLDecoder(fis);
 Person bPerson = (Person) xdec.readObject();
```

Runtime Type Information (RTTI)

- RTTI allows a developer to discover and use type information while a program is running:
  - Traditional RTTI: assumes that all the types are available at compile time.
  - *Reflection* is a mechanism that allows the discovery and the usage of a class information only at runtime.

C# Reflection and Metadata

- A C# program compiles into an assembly that includes metadata, compiled code, and resources. Inspecting the metadata and compiled code at runtime is called *reflection*.
- The compiled code in an assembly contains almost all of the content of the original source code.
- Some information is lost, such as local variable names, comments, and preprocessor statements.
- The System.Reflection namespace contains the reflection API.
- In C#, it is possible at runtime to dynamically create new metadata and executable instructions in IL (Intermediate Language) using the classes in the System.Reflection.Emit namespace.

Assemblies in C#

- An assembly is the basic unit of deployment in .NET and is also the container for all types.
- An assembly contains compiled types with their IL code, runtime resources, and information to assist with versioning, security, and referencing other assemblies.
- An assembly contains four kinds of things:
  - *An assembly manifest*: provides information to the .NET runtime: the assembly's name, version, requested permissions, and other assemblies that it references.
  - *An application manifest*: provides information to the operating system, such as how the assembly should be deployed and whether it requires special permissions.
  - *Compiled types*: The compiled IL code and metadata of the types defined within the assembly.
  - *Resources*: Nonexecutable data embedded within the assembly, such as images and localizable text
- Only the assembly manifest is mandatory, although an assembly nearly always contains compiled types.
- Assemblies are structured similarly whether they are executables (.exe) or libraries (.dll). The main difference with an executable is that it defines an entry point.

Assembly Class

- The `Assembly` class in `System.Reflection` namespace provides methods for accessing assembly metadata at runtime.
- There are a number of ways to obtain an assembly object:
  - Using a `Type's Assembly` property:

    ```
    Assembly a = typeof (Program).Assembly;
    ```
  - Calling one of Assembly's static methods:
    - `GetExecutingAssembly`: returns the assembly of the type that defines the currently executing function
    - `GetCallingAssembly:` does the same as `GetExecutingAssembly`, but for the function that called the currently executing function
    - `GetEntryAssembly`: Returns the assembly defining the application's original entry method

Assembly Class

- Methods:
  - `CreateInstance`
  - `Load`: loads an assembly (overloaded).
  - `LoadFrom`: loads an assembly (overloaded).
  - `LoadFile`: loads the contents of an assembly file (overloaded).
  - `ReflectionOnlyLoad`: loads an assembly into the reflection-only context, where it can be examined but not executed.
  - `GetName`(): `AssemblyName` (name, version)
- Properties:
  - `FullName`: gets the display name of the assembly
  - `CodeBase`: gets the location of the assembly as specified originally
  - `EntryPoint`: gets the entry point of this assembly

Type class

- Represents type declarations: class types, interface types, array types, value types, enumeration types, type parameters, generic type definitions, and open or closed constructed generic types.
- It represents the metadata for a type.
- **`Type`** is contained in the **`System`** namespace.
- Obtaining an instance of **`Type`**:
  - Using **`GetType`** method on any object:

    ```
    Type t1 = "Ana are mere.".GetType( );      // Type obtained at runtime
    ```
  - Using **`typeof`** operator:

    ```
    Type t2 = typeof (String);            // Type obtained at compile time
    ```
  - **`typeof`** can be used to obtain array types and generic types:

    ```
    Type t3 = typeof (DateTime[]);             // 1-d Array type
    Type t4 = typeof (DateTime[,]);            // 2-d Array type
    Type t5 = typeof (Dictionary<int,int>); // Closed generic type
    Type t6 = typeof (Dictionary<,>);          // Open generic type
    ```
  - A type can also be retrieved by its name:

```
Type t = Assembly.GetExecutingAssembly(  ).GetType ("MyClass");
```

Type Class

- A `System.Type` instance contains the entire metadata for the type—and the assembly in which it was defined.
- `System.Type` is abstract, so the typeof operator returns a subclass of `Type`. The subclass that the CLR uses is internal to mscorlib and is called `RuntimeType`.
- A type has `Namespace`, `Name`, and `FullName` properties. In most cases, `FullName` is a composition of the former two:

```
Type t = typeof (System.Text.StringBuilder);
Console.WriteLine (t.Namespace);        // System.Text
Console.WriteLine (t.Name);             // StringBuilder
Console.WriteLine (t.FullName);         // System.Text.StringBuilder
```

- There are two exceptions to this rule: nested types and closed generic types:
  - With nested types, the containing type appears only in FullName.
  - Generic type names are suffixed with the ' symbol, followed by the number of type parameters.

Base Type and Interfaces

•`Type` exposes a `BaseType` property:

```
Type base1 = typeof (System.String).BaseType;
Type base2 = typeof (System.IO.FileStream).BaseType;
Console.WriteLine (base1.Name);      // Object
Console.WriteLine (base2.Name);      // Stream
```

•The `GetInterfaces` method returns the interfaces that a type implements:

```
foreach (Type iType in typeof (Int16).GetInterfaces())
  Console.WriteLine (iType.Name);
// IComparable, IFormattable, IConvertible, IComparable’1,
IEquatable’1
```

Equivalents of is operator

- Reflection provides three dynamic equivalents to C#'s static is operator:
  - **`IsInstanceOfType`** : Accepts a type and instance
  - **`IsAssignableFrom`**: Accepts two types
  - **`IsSubclassOf`**: the same as **`IsAssignableFrom,`** but excludes interfaces.

```
object obj  =23;
Type target = typeof (IFormattable);
bool isTrue   = obj is IFormattable;               // Static C# operator
bool alsoTrue = target.IsInstanceOfType (obj);   // Dynamic equivalent

Type target = typeof (IComparable);
Type source = typeof (string);
Console.WriteLine (target.IsAssignableFrom (source));         // True
```

Instantiating Types

- The static `Activator.CreateInstance` method dynamically instantiates an object from its Type.
- `CreateInstance` accepts a `Type` and optional arguments that get passed to the constructor:
  - `CreateInstance(Type):Object`
  - `CreateInstance(assembly:String, typename:String):Object`
  - `CreateInstance(Type, args:Object[]):Object`

```
int i = (int) Activator.CreateInstance (typeof (int));
DateTime dt = (DateTime) Activator.CreateInstance (typeof (DateTime),
    2000, 1, 1);
```

- A `MissingMethodException` is thrown if the runtime cannot find a suitable constructor.

Remark:

Dynamic instantiation adds a few microseconds to the time needed to construct the object.

Reflecting Members

- The `GetMembers` method returns the members of a type.

```
MemberInfo[] members = typeof (Person).GetMembers(  );
foreach (MemberInfo m in members)
      Console.WriteLine (m);
```

- A member can be: properties, methods, fields, events, etc.
- `GetMembers()` returns all the public members of the current Type.
- `GetMembers(BindingFlags)` searches for the members defined for the current Type, using the specified binding constraints.
- BindingFlags (enum) specifies flags that control binding and the way in which the search for members and types is conducted by reflection:
  - DeclaredOnly, Instance, Static, Public, NonPublic
  - InvokeMethod, CreateInstance, SetField, GetField, GetProperty, SetProperty
  - Etc.
- `InvokeMember(name:String, invokeAttr:BindingFlags, binder:Binder, target:Object, args:Object[]):Object`
- If a constructor is invoked, the `name` and the `target` are null.
- `Binder` is almost always null.

MemberInfo Class

- Obtains information about the attributes of a member and provides access to member metadata.
- The **`MemberInfo`** class is the abstract base class for classes used to obtain information about all members of a class (constructors, events, fields, methods, and properties).
- This class introduces the basic functionality that all members provide.
- Important properties:
  - **`DeclaringType`**: gets the class that declares this member.
  - **`MemberType`**: gets a **`MemberTypes`** value indicating the type of the member — method, constructor, event, etc.
  - **`Name`**: gets the name of the current member.
- Enum **`MemberTypes`**: **`Constructor`**, **`Event`**, **`Field`**, **`Method`**, **`Property`**, …

FieldInfo class

- Discovers the attributes of a field and provides access to field metadata.
- The FieldInfo class does not have a public constructor.
- FieldInfo objects are obtained by calling either the GetFields or GetField method of a Type object.

```
using System;
using System.Reflection;
public class AClass{
    public int myField1 = 0;
    protected string myField2 = null;
    public static void Main()    {
        FieldInfo[] fields;
        Type myType = typeof(AClass);
       // Get the type and fields of AClass.
       fields = myType.GetFields(BindingFlags.NonPublic |
   BindingFlags.Instance | BindingFlags.Public);
        Console.WriteLine("Nr of fields {0}", fields.Length);
   }}
```

FieldInfo class

- Methods:
  - `GetType` Gets the `Type` of the field.
  - `GetValue` returns the value of a field supported by a given object.
  - `SetValue` sets the value of the field for the given object to the given value.
  - Etc.
- Properties
  - `IsPrivate`
  - `IsPublic`
  - `IsStatic`
  - etc

ConstructorInfo class

- Discovers the attributes of a class constructor and provides access to constructor metadata.
- Objects are created by calling `Invoke` on a `ConstructorInfo` returned by either the `GetConstructors` or `GetConstructor` method of a `Type` object.

```
public class AClass{
    public AClass(){}
    public AClass(int i){}
}
//…
Type myType = typeof(AClass);
Type[] types = new Type[1];
types[0] = typeof(int);
// Get the constructor that takes an integer as a parameter.
 ConstructorInfo constructorInfoObj = myType.GetConstructor(types);
// Gets all public constructors
ConstructorInfo[] cons=myType.GetConstructors();
```

ConstructorInfo Class

- Methods
  - `Invoke(Object[])` invokes the constructor reflected by the instance that has the specified parameters
  - `Invoke(BindingFlags, Binder, Object[], CultureInfo)` invokes the constructor reflected by this ConstructorInfo with the specified arguments, under the constraints of the specified Binder.
- Property: `MemberType`

```
Example:
Type myType = typeof(AClass);
ConstructorInfo cons=myType.getConstructor(new Type[0]{});
Object obj=cons.Invoke(new Object[0]{})
```

MethodInfo class

- Discovers the attributes of a method and provides access to method metadata.
- Instances of MethodInfo are obtained by calling the `GetMethods()` or `GetMethod()` method of a `Type` object or of an object that derives from `Type`.

```
public class AClass{
    public AClass(){}
    public AClass(int i){}
   public int methodA(){...}
   public void methodB(double d){...}
   public void methodB(int a){...}
}
//…
Type myType = typeof(AClass);
// Gets all public methods of current type.
 MethodInfo[] methods= myType.GetMethods();
// Gets a method by its name
MethodInfo amethod=myType.GetMethod("methodA");
//Gets a method by its name and paramters types
MethodInfo bmethod=myType.GetMethod("methodB", new Type[]
   {typeof(int)});
```

MethodInfo class

- Methods:
  - `Invoke(Object, Object[]):Object` invokes the method represented by the current instance, using the specified parameters.
  - `Invoke(Object, BindingFlags, Binder, Object[], CultureInfo):Object` invokes the reflected method or constructor with the given parameters.
- Properties:
  - `IsAbstract`
  - `IsConstructor`
  - `IsFinal`
  - `IsGenericMethod`
  - `IsPublic, IsPrivate`
  - `IsVirtual`
  - Etc..

PropertyInfo class

- Discovers the attributes of a property and provides access to property metadata.
- Properties are logically the same as fields.
- An instance of this class can be obtained using getProperties(), or getProperty methods from the corresponding Type class.

```
public class AClass{
   private String name;
    public String Name{
        get{return name;}
        set{name=value;}
   }}
Type myType = typeof(AClass);
// Gets all public properties of current type.
 PropertyInfo[] properties= myType.GetProperties();
```

PropertyInfo class

- Methods:
  - `GetAccessors`: returns an array of the get and set accessors on this property.
  - `GetGetMethod`: Returns a `MethodInfo` representing the get accessor for this property.
  - `GetSetMethod`: Returns a `MethodInfo` representing the set accessor for this property.
  - `GetValue`: Returns the value of the property.
  - `SetValue`: Sets the property value for the given object to the given value.
- Properties:
  - `CanRead`
  - `CanWrite`
  - `PropertyType`
  - `Name`

C# Examples

- Determines and prints all the properties of an object that are both set and get.
- Construct an object of type Book, invokes some method and property.

Reflection in Java

- The entry point for all reflection operations is `java.lang.Class`.
- Almost none of the classes in `java.lang.reflect` package have public constructors.
- To get to these classes, it is necessary to invoke appropriate methods on `Class`.
- A Class object can be obtained in several ways:
  - Using getClass() method (it cannot be used for primitive types):

```
Class cl="Ana are mere".getClass();
boolean ok;
Class cb=ok.getClass();      //compile time error
```

  - Using .class syntax:

```
Class c = boolean.class;
Class cpw = java.io.PrintWriter.class;
Class cint3=int[][][].class;
```

  - Using the static method `Class.forName()`. It cannot be used for primitive types:

```
    Class c = Class.forName("java.lang.String");
Class cStringArray = Class.forName("[[Ljava.lang.String;");  //String[][].class
```

  - `TYPE` field for primitive type wrappers: each of the primitive types and void has a wrapper class in `java.lang` that is used for boxing of primitive types to reference types. Each wrapper class contains a field named **TYPE** which is equal to the `Class` for the primitive type being wrapped:

```
Class c = Double.TYPE;
Class c = Void.TYPE;
```

Class

- There are several Reflection APIs which return classes but these may only be accessed if a Class instance has already been obtained either directly or indirectly:
  - `getSuperclass():Class` Returns the super class for the given class.
  - `getClasses():Class[]` Returns all the public classes, interfaces, and enums that are members of the class including inherited members.
  - `getDeclaredClasses():Class[]` Returns all of the classes interfaces, and enums that are explicitly declared in this class.
  - `getEnclosingClass():Class` Returns the immediately enclosing class of the class.
- There are two categories of methods provided in Class for accessing fields, methods, and constructors:
  - methods which enumerate these members
  - methods which search for particular members.
- Also there are distinct methods for:
  - accessing members declared directly on the class
  - methods that search the superinterfaces and superclasses for inherited members.
- `getConstructors() :Constructor[]        //public constructors`
- `getConstructor(Class... parameterTypes):Constructor<T>;`
- `getDeclaredConstructors(): Constructor[]`
- `getDeclaredField(String name):Field`
- `getDeclaredFields():Field[]`
- `getDeclaredMethod(String name, Class... parameterTypes) :Method`
- `getDeclaredMethods(): Method[]`

Class methods

- `getField(…)/getFields`
- `getMethod(…)/getMethods`
- `getInterfaces():Class[]`
- `getModifiers():int`
- `getName():String`
- `getPackage():Package`
- `isAssignableFrom(Class)`
- `isInstance(Object)`
- `isInterface():boolean`
- `isPrimitive`
- `isArray`
- `newInstance():T` Creates a new instance of the class represented by this Class object. The default constructor is called. It may throw InstantiationException when the constructor is not accesible or there is no default constructor.

Remark: It does not have an invoke method (InvokeMember from C#).

Modifier class

- The Modifier class provides static methods and constants to decode class and member access modifiers.
- The sets of modifiers are represented as integers with distinct bit positions representing different modifiers.
- Fields (static, int):
  - ABSTRACT
  - FINAL
  - INTERFACE
  - PRIVATE, PROTECTED, PUBLIC
  - STATIC
  - TRANSIENT, VOLATILE
- `Modifier.isPublic(aMethod.getModifiers());`

Member interface

- Member is an interface that reflects identifying information about a single member (a field or a method) or a constructor.
- Two public static fields: `PUBLIC`, `DECLARED`
- Four methods:
  - `getDeclaringClass(): Class` Returns the Class object representing the class or interface that declares the member or constructor represented by this Member.
  - `getModifiers():int` Returns the Java language modifiers for the member or constructor represented by this Member, as an integer.
  - `getName(): String` Returns the simple name of the underlying member or constructor represented by this Member.
  - `isSynthetic(): boolean` Returns true if this member was introduced by the compiler; returns false otherwise.

AccessibleObject Class

- The `AccessibleObject` class is the base class for `Field`, `Method` and `Constructor` objects.
- It provides the ability to flag a reflected object as suppressing default Java language access control checks when it is used.
- The access checks--for `public`, default (`package`) access, `protected`, and `private` members--are performed when `Fields`, `Methods` or `Constructors` are used to set or get fields, to invoke methods, or to create and initialize new instances of classes, respectively.
- Setting the accessible flag in a reflected object permits, for example, invoking private methods or modifying private fields:
  - `setAccessible(boolean flag)`
  - `isAccessible() :boolean`

Constructor Class

- The `java.lang.reflect.Constructor` class provides a way to obtain information about the name, modifiers, parameters, and list of throwable exceptions of a certain constructor.
- A `Constructor` instance is obtained using the `Class` instance:
  - `getDeclaredConstructor()`
  - `getConstructor()`
  - `getDeclaredConstructors()`
  - `getConstructors()`
- Methods:
  - `getExceptionTypes()`
  - `getParameterTypes()`
  - `newInstance(Object... initargs):T` //creates a new instance

Constructor class

`Class.newInstance()` versus `Constructor.newInstance()`

- `Class.newInstance()` can only invoke the zero-argument constructor, while `Constructor.newInstance()` may invoke any constructor, regardless of the number of parameters.
- `Class.newInstance()` throws any exception thrown by the constructor, regardless of whether it is checked or unchecked. `InvocationTargetException`.
- `Class.newInstance()` requires that the constructor be visible. `Constructor.newInstance()` may invoke private constructors under certain circumstances.

Field Class

- A `Field` provides information about, and dynamic access to, a single field of a class or an interface. The reflected field may be a class (static) field or an instance field.
- An instance of type `Field` can be obtained using the following methods from `Class`:
  - `getDeclaredField()`
  - `getField()`
  - `getDeclaredFields()`
  - `getFields()`
- Methods:
  - `get(Object obj): Object` Returns the value of the field represented by this Field, on the specified object.
  - `getBoolean(Object obj) :boolean`
  - `getByte(Object):byte`
  - `getChar(Object):char`
  - `getType(): Class<?>` Returns a `Class` object that identifies the declared type for the field represented by this `Field` object.
  - `set(Object obj, Object value)` Sets the field represented by this Field object on the specified object argument to the specified new value.
  - `setBoolean(Object, boolean)`
  - Etc.

Method Class

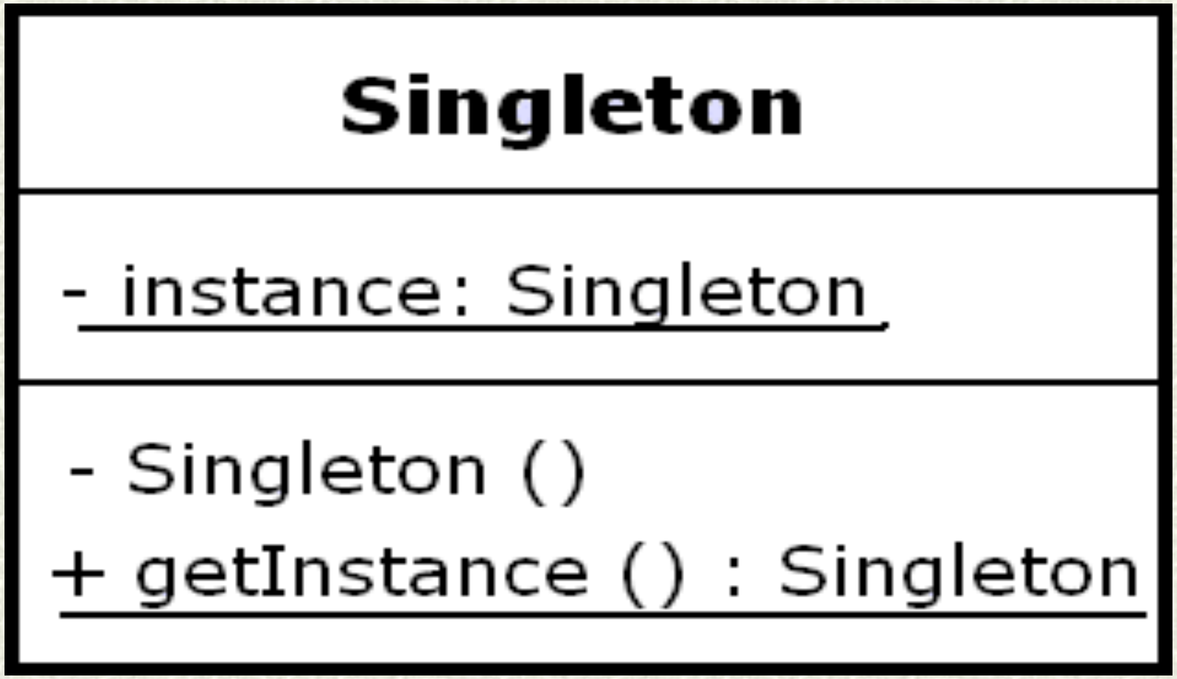
- A `Method` provides information about, and access to, a single method on a class or interface. The reflected method may be a class method or an instance method (including an abstract method).
- An instance of type `Method` can be obtained using one of the following methods from `Class`:
  - `getDeclaredMethod()`
  - `getMethod()`
  - `getDeclaredMethods()`
  - `getMethods()`
- Methods:
  - `getExceptionTypes():Class<?>[]`
  - `getParameterTypes():Class<?>[]`
  - `getReturnType():Class<?>`
  - `invoke(Object obj, Object... args): Object`

Remarks

- Reflection is commonly used by programs which require the ability to examine or modify the runtime behavior of an application (C# or Java).
  - **Class Browsers and Visual Development Environments**
  - **Debuggers and Test Tools**
- Drawbacks:
  - **Performance Overhead**
  - **Security Restrictions**
  - **Exposure of Internals**

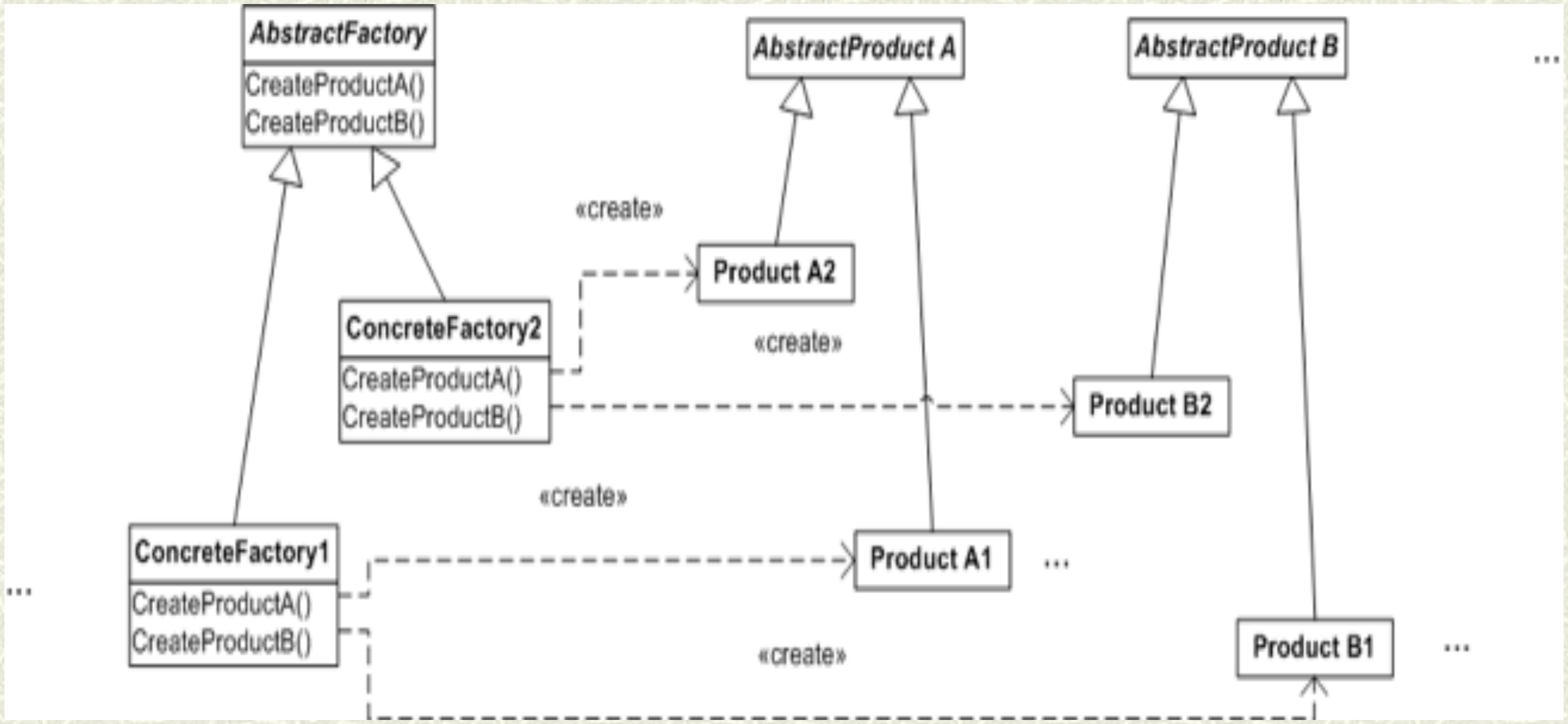
Singleton Pattern

Ensure a class has only one instance, and provide a global point of access to it.

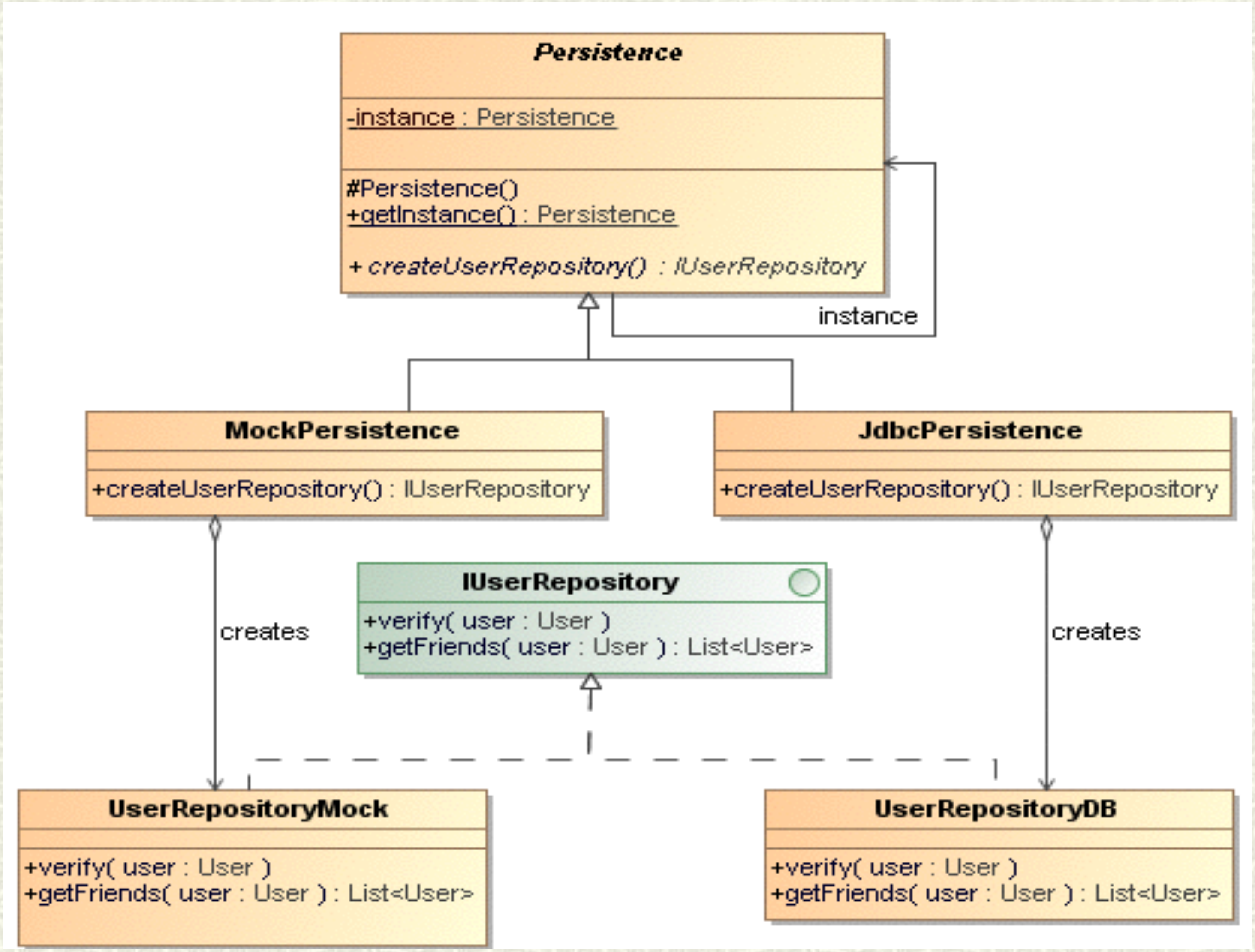


Abstract Factory Pattern

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.



Case study Persistence


Persistence
-instance : Persistence
#Persistence()
+getInstance() : Persistence
+ createUserRepository() : IUserRepository
instance
MockPersistence
+createUserRepository() : IUserRepository
JdbcPersistence
+createUserRepository() : IUserRepository
creates
IUserRepository
+verify(user : User)
+getFriends(user : User) : List<User>
creates
UserRepositoryMock
+verify(user : User)
+getFriends(user : User) : List<User>
UserRepositoryDB
+verify(user : User)
+getFriends(user : User) : List<User>