JAVA

JavaBeans

Components – overview

- component
  - reusable piece of code
  - characterized by services provided and **required**
  - no exact definition
- component models
  - JavaBeans
  - Enterprise JavaBeans (EJB)
  - …
  - many other component models

JavaBeans – overview

- JavaBeans provides
  - properties
  - events
  - methods
- information about a component
  - implicit (reflection)
  - explicit
- interconnecting components
  - via events
- persistence
  - implementing java.io.Serializable
- distribution
  - JARs

JavaBeans

- http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html
- specification
  - 1.00 1996
  - 1.01 1997
- a simple component model
  - Java objects as components
  - simple manipulation and interconnection in GUI development environments
- definition
  - ***Java Bean is a reusable software component that can be manipulated visually in a builder tool***

JavaBeans

- one of goals – simplicity
- based on **naming conventions**
- *property*
  - name
    - e.g.. foreground
  - methods for access – set and get
    - void setForeground(Color c)
    - Color getForeground()
- *methods*
  - regular methods
    - by default all public ones
- *events*
  - communication between components
    - a component “listens” to events of another one

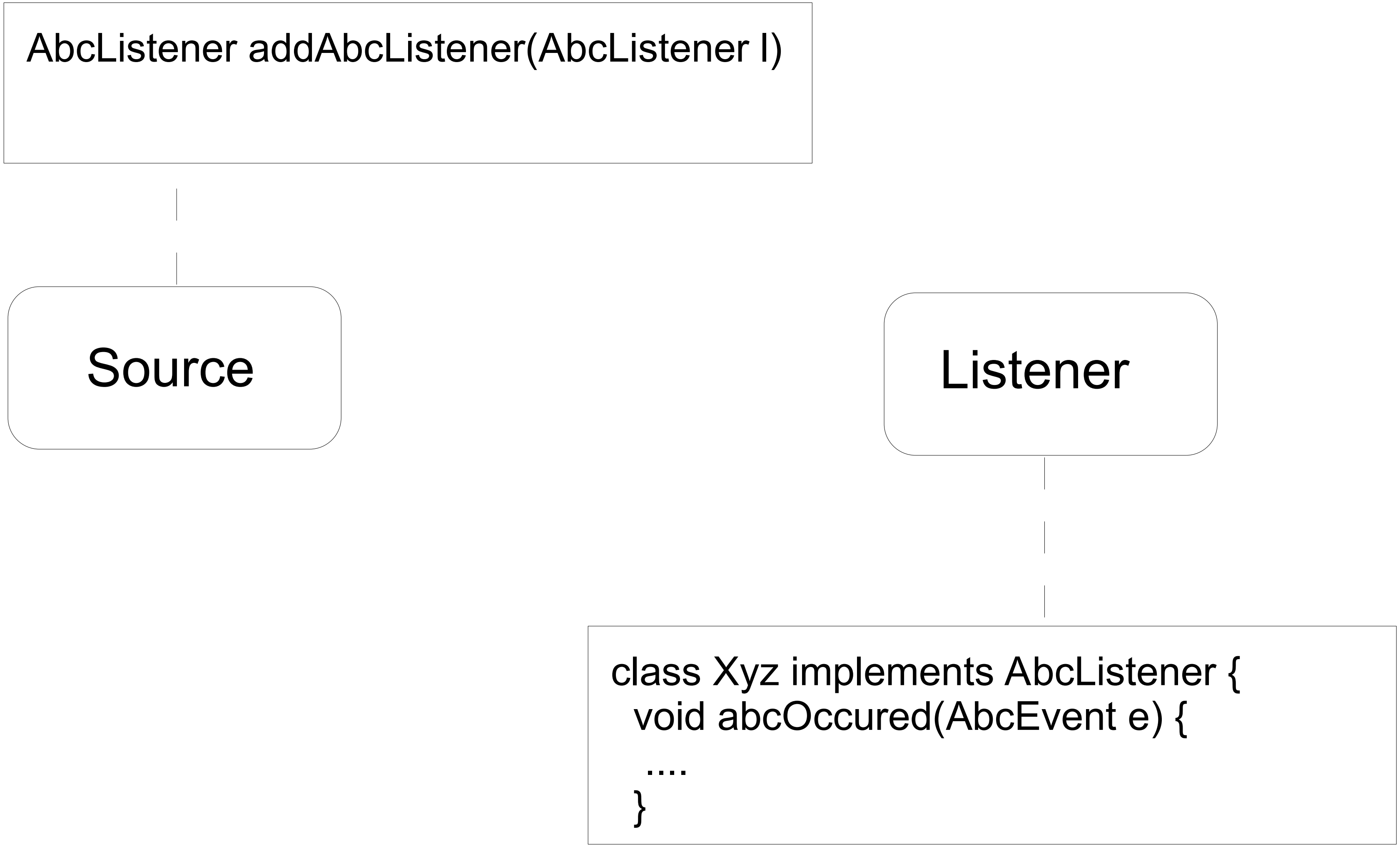
JavaBeans

- execution in different environments
  - desing time vs. run time
- security
  - all as regular objects
- typically a component has GUI representation
  - non-visible components without GUI can also exist
  - visible components extend java.awt.Component
- no synchronization
  - if necessary, components have to ensure it by themselves
- multiple views of a component
    - not implemented (never will be)
  - `Component c = Beans.getInstanceOf(x, Component)`
  - plain casting should not be used

Events

- event – an object
  - source of the event
  - a listening object – listener
- different events identified by a type – different objects
  - ancestor java.util.EventObject
- listener
  - a method, which is called when an event occurred
  - the interface java.util.EventListener
  - a listener can have several methods

Events – overview



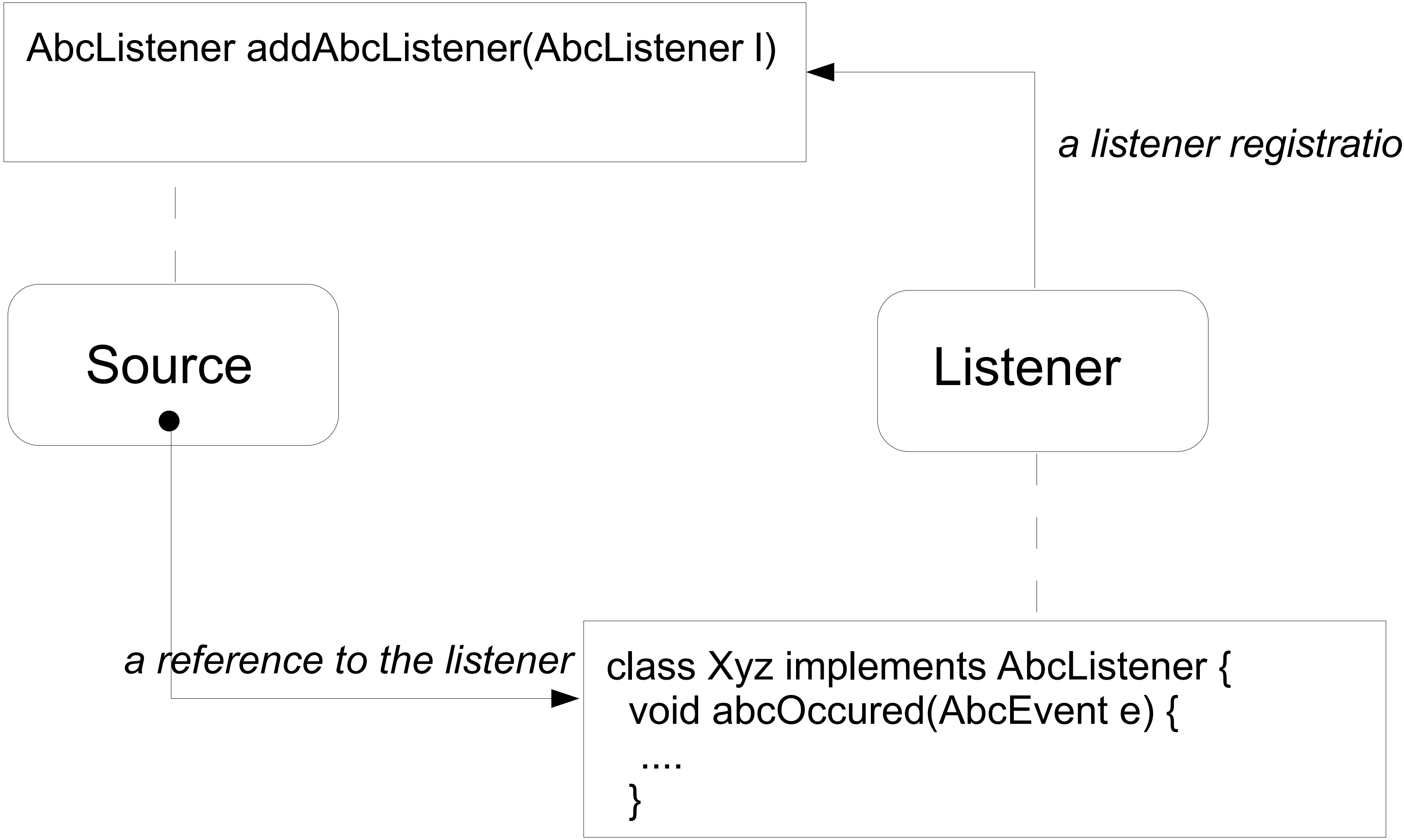
```
AbcListener addAbcListener(AbcListener l)
```

Source

Listener

```
class Xyz implements AbcListener {
  void abcOccured(AbcEvent e) {
   ....
  }
```

Events – overview



```
AbcListener addAbcListener(AbcListener l)
```

a listener registration

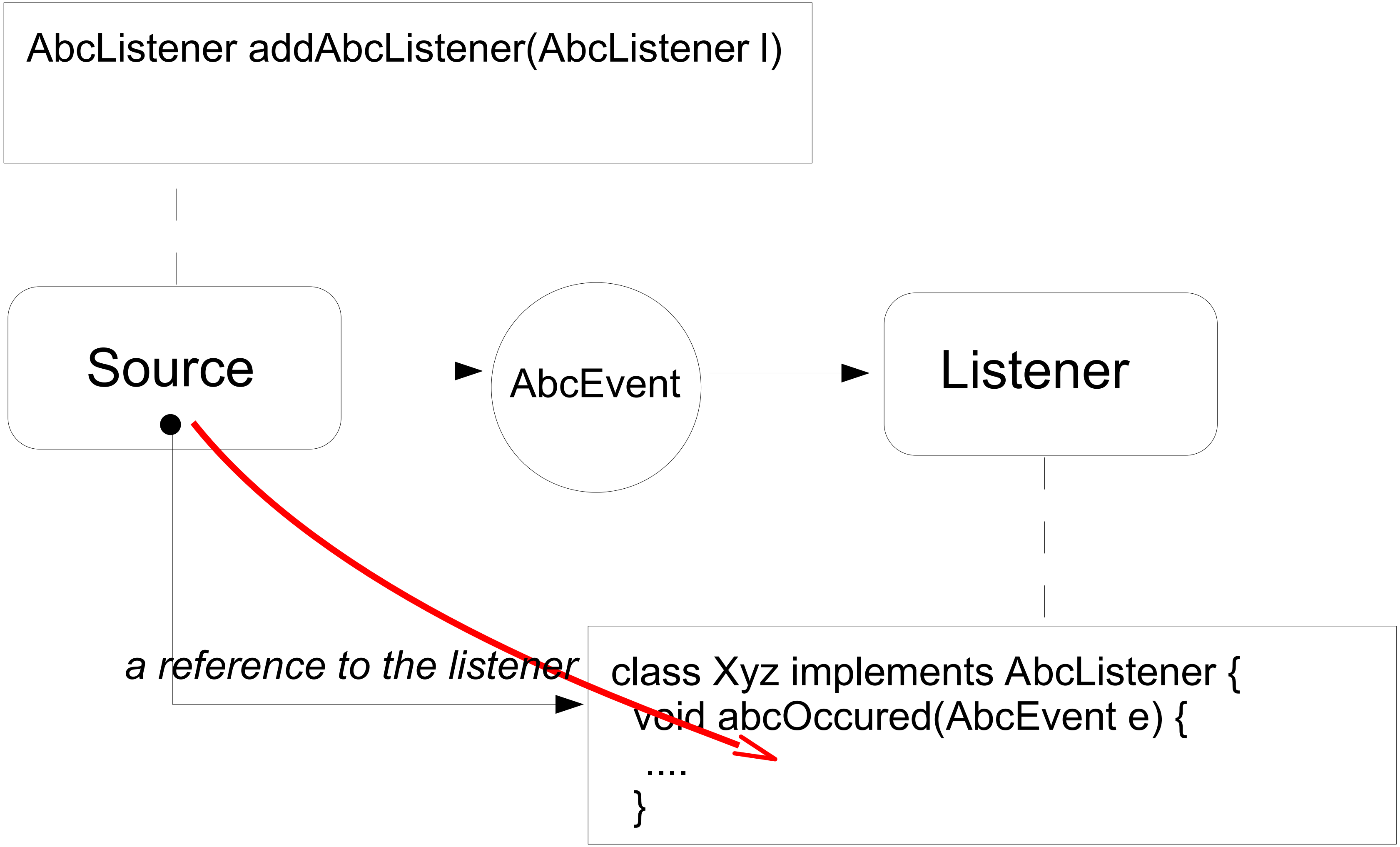
Source

Listener

a reference to the listener

```
class Xyz implements AbcListener {
  void abcOccured(AbcEvent e) {
   ....
  }
```

Events – overview



AbcListener addAbcListener(AbcListener l)

Source

AbcEvent

Listener

a reference to the listener

```
class Xyz implements AbcListener {
  void abcOccured(AbcEvent e) {
   ....
  }
```

Event object

- extends java.util.EventObject
- typically immutable
  - private fields
  - *get* methods

```
public class MouseMovedEvent extends EventObject {
  protected int x,y;

  public MouseMovedEvent(Component source, Point location) {
    super(source);
    x = location.x;
    y = location.y;
  }

  poblic Point getLocation() {
    return new Point(x, y);
  }
}
```

Listener

- interface – its name ends with Listener (a convention)
  - extends java.util.EventListener
- defines methods for serving the event
  - a pattern for the method
    - void *anEventHappened*(*EventObject* e)
- a listening object implements the listener

```
public class MouseMovedListener implements EventListener {
  void mouseMoved(MouseMovedEvent e);
}
```

- a single listener can define several methods for related events
  - e.g. mouseMoved, mouseEntered, mouseExited
- methods can declare exceptions
- a method parameter – the event
  - exceptionally a list of different parameters

Listener registration

- a component, which produces events, defines methods for registration of listeners
  - separately for each type
- a pattern
  - void add<TypeOfListener>(<TypeOfListener> l)
  - void remove<TypeOfListener>(<TypeOfListener> l)

```
public class Xyz {
  private ArrayList lst = new ArrayList();

  public void addMouseMovedListener(MouseMovedListener l) {
    lst.add(l);
  }
  public void removeMouseMovedListener(MouseMovedListener l) {
    lst.remove(l);
  }
  protected void fireMouseMovedEvent(int x, int y) {
    MouseMovedEvent e = new MouseMovedEvent(this, new Point(x,y);
    for (int i=0; i<lst.length; i++) {
      ((MouseMovedListener)lst.get(i)).mouseMoved(e);
    }
  }
}
```

Listener registration

- unicast listener
  - maximally one registered listener
  - a pattern
    - void add<TypeOfListener>(<TypeOfListener> l) ***throws TooManyListnersException***
    - void remove<TypeOfListener>(<TypeOfListener> l)
- adding/removing a listener during an event handling
  - to whom the event is delivered?
    - depends on implementation
    - e.g. add*Listener* and remove*Listener* synchronized and

```
protected void fireMouseMovedEvent(int x, int y) {
  Vector l;
  MouseMovedEvent e = new MouseMovedEvent(this,
                                            new Point(x,y);
  synchronized (this) { l = (Vector) listenres.clone(lst); }
  for (int i=0; i<l.length; i++) {
    ((MouseMovedListener)l.get(i)).mouseMoved(e);
  }
}
```

Event adaptor

- a listening object does not implement the listener
  - it creates another object – adaptor – which implements the listener
  - registers the adaptor
  - the adaptor calls methods on the listening object

- usage
  - filtering events
  - reacting to different events of the same type
  - ....

Event adaptor

- example – a Dialog
  - contains 2 buttons – OK a Cancel – both generates the event ActionEvent
  - the Dialog has methods
    - void doOKAction()
    - void doCancelAction()
  - two adaptors – implement ActionListener
    - OKButtonAdaptor
      - registered to the OK button
      - calls the doOKAction method on the Dialog
    - CancelButtonAdaptor
      - registered to the Cancel button
      - calls the doCancelAction method on the Dialog
- adaptors commonly as (anonymous) inner classes

Properties

- a property
  - name and type
  - methods for access
    - void setProperty(PropertyType c)
    - PropertyType getProperty()
- any type
  - exception for boolean properties
    - instead *get, is* is used
    - e.g.: void setEnabled(boolean b)
      boolean isEnabled()
- methods can declare exceptions

Indexed properties

- multi-value properties (arrays)
  - void setIndexedProperyt(int i, PropertyType c)
  - PropertyType getIndexedProperty(int i)
  - void setIndexedProperyt(PropertyType[] c)
  - PropertyType[] getIndexedProperty()

Bounded properties

- change of a property value generates an event
- the PropertyChange event
- the listener PropertyChangeListener
- a component generates the event **after** the value of the property is changed
- a helper classs PropertyChangeSupport
  - managing listeners

Constrained properties

- another component can forbid changes of values of a given property
- the set metoda deklares the PropertyVetoException exception
- after the values is changed, the component generates the VetoableChange event
  - the listener VetoableListener
  - if a registered listener throws the PropertyVetoException, property change is not performed
- a component generates the event **before** the value is changed
- the helper class VetoableChangeSupport

Bounded & Constrained props.

- a property can be both *bounded* and *constrained*
  - order of execution
    1. VetoableChange event
    2. if exception occurs → end
    3. changing value
    4. PropertyChange event

- if value changed to the same one – no event should be changed
  - because of performance

Introspection

- obtaining information about a component
  - properties
  - methods
  - events
- implicit
  - by patterns via reflection (java.lang.reflect)
  - properties
    - get and set methods
  - methods
    - all public ones
  - events
    - metods add*Listener* and remove*Listener*

Introspection

- explicit – the *BeanInfo* class
  - implements the java.beans.BeanInfo interface
  - name – ***AComponentName*BeanInfo**

```
public interface BeanInfo {
    BeanDescriptor getBeanDescriptor();
    EventSetDescriptor[] getEventSetDescriptors();
    int getDefaultEventIndex();
    PropertyDescriptor[] getPropertyDescriptors();
    int getDefaultPropertyIndex();
    MethodDescriptor[] getMethodDescriptors();
    BeanInfo[] getAdditionalBeanInfo();
    java.awt.Image getIcon(int iconKind);
}
```

- typically, the BeanInfo extends the SimpleBeanInfo class
  - prepared implementation

Introspection

- BeanInfo cannot describe all properties/events/methods
  - information about the rest can obtained by reflection
- if the BeanInfo class is used, no need to use naming convention
  - but it is strongly recommended

Introspector

- java.beans.Introspector
  - a class
  - a standard way to obtain information about components
    - analyzes the BeanInfo (if exists) and directly the class
    - analyzes ancestors of the component

```
class Introspector {
  static BeanInfo getBeanInfo(Class<?> beanClass)
  static BeanInfo getBeanInfo(Class<?> beanClass,
                                            Class<?> stopClass)
  static String[] getBeanInfoSearchPath()
  static void setBeanInfoSearchPath(String[] path)
  ...
}
```

Property editor

- a class for GUI changing values of a given type
  - in GUI development environment
- PropertyEditorManager
  - pre-registered editors for basic types
  - order for searching an editor for the given type
    1. search in explicitly registered editors
    2. a class with the same name plus the extension Editor
    3. search in packages for editors (can be set in PropertyEditorManager) – a class with the name as in 2.
- a property editor can be registered for a particular property in the BeanInfo class

Customizer

- a component in GUI development environment
  - setting values in a property sheet

- if all features cannot be set via properties => a component can have a Customizer
  - a Dialog for setting some features
  - it should implement the interface java.beans.Customizer and extend java.awt.Component
  - registered in BeanInfo

Persistence

- common serialization

- serialization
  - as usually
- de-serialization
  - ClassLoader cl = this.getClass().getClassLoader();
  - MyBean b = (MyBean) Beans.instantiate(cl, "myPackage.MyBean");
  - first it looks a file with the serialized component
    - myPackage/MyBean.ser
  - if not found, an instance is directly created

Distributing components

- a plain JAR file
- Manifest
  - special elements in JAR description
  - Java-Bean: True
  - Depends-On: list of classes from the JAR file
  - Design-Time-Only: True
- JAR typically can contain both the class and its serialization (NameOfComponent.ser)

Java FX Beans (to compare)

Properties of components

- interface Property<T>
  - void addListener(InvalidationListener listener)
  - void addListener(ChangeListener<? super T> listener)

  - void bind(ObservableValue<? extends T> observable)
  - void bindBidirectional(Property<T> other)
  - …

- implementace
  - class ObjectProperty<T>
  - class IntegerProperty
  - class BooleanProperty
  - class StringProperty
  - ...

Properties – implementation ex.

```
private StringProperty text =
                   new SimpleStringProperty("");

public final StringProperty textProperty() {
  return text;
}

public final void setText(String newValue){
  text.set(newValue);
}

public final String getText() {
  return text.get();
}
```

Properties – listeners

- `InvalidationListener`
  - called if the current property value is not valid anymore
  - allows for “lazy” evaluation

```
void invalidated(Observable observable)
```

- `ChangeListener`
  - called if the current property value has changed
  - it is necessary to evaluate the new value
  - does not allow for “lazy” evaluation

```
void changed(ObservableValue<? extends T>
             observable, T oldValue, T newValue)
```

Properties – binding

- automated updating of a property when another one is changed
  - internally implemented via listeners

```
text1.textProperty().bind(text2.textProperty());

text1.textProperty().bindBidirectional(
                              text2.textProperty());
```

- class Bindings
  - static methods for easy creation of bindings

XML processing

Overview

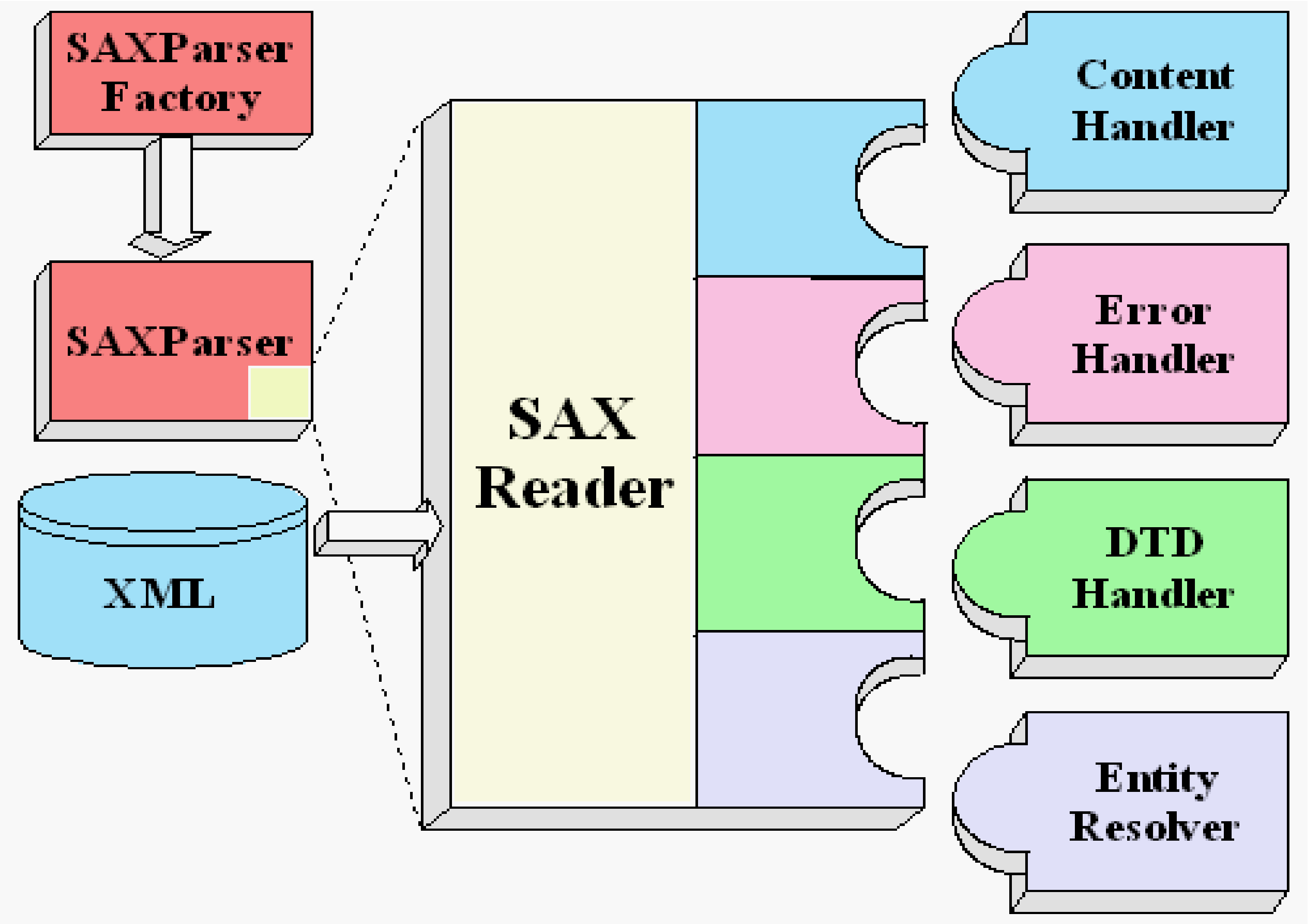
- JAXP – Java API for XML Processing
  - reading, writing and transforming XML
  - SAX, DOM, XSLT
    - according to W3C
  - supports multiple implementations
    - a reference implementation is a part of JDK
      - another one can be used
- JDOM
  - http://www.jdom.org/
  - „simplified“ DOM for Java
- JAXB – Java Architecture for XML Binding
  - mapping XML <=> Java objects

- Elliotte Rusty Harold: Processing XML with Java
  - http://www.cafeconleche.org/books/xmljava/
  - a freely accessible book

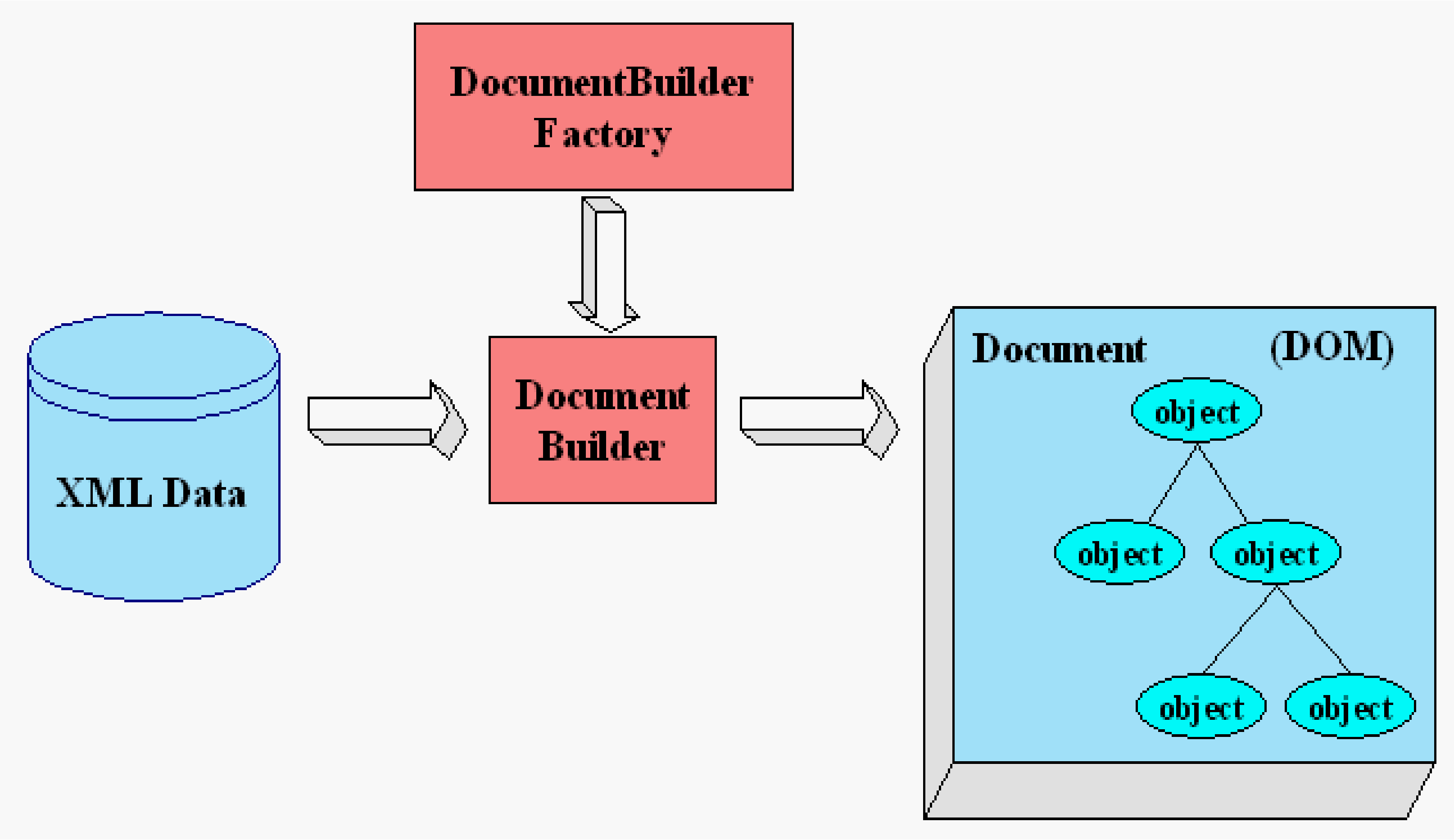
JAXP – overview

- packages
  - javax.xml.parsers
  - org.w3c.dom
  - org.xml.sax
  - javax.xml.transform
- SAX (Simple API for XML)
  - a “walk” through an XML document – element by element
  - each element can be processed
  - fast, low memory consumption
  - more complex to be used
- DOM
  - creates a tree in a memory from the document
  - easy to be used
  - slow, bigger memory consumption

SAX


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

DOM


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

DOM: usage

```
 DocumentBuilderFactory factory =
   DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();

// vytvoří celý strom v paměti
Document document = builder.parse("file.xml");

Element root = document.getDocumentElement();
NodeList nl = root.getChildNodes();
for (int i=0; i<nl.length(); i++) {
  Node n = nl.item(i);
  ...
}
```

SAX: usage

```
class MyHandler extends DefaultHandler {
  void startDocument() {
    ...
  }
  void endDocument() {
    ...
  }
  void startElement(....) {
    ...
  }
  ...
}

SAXParserFactory factory =
  SAXParserFactory.newInstance();
SAXParser saxParser = factory.newSAXParser();
saxParser.parse("file.xml", new MyHandler() );
```

Implementation

- different implementations of JAXP exist
- DocumentBuilderFactory.newInstance() and SAXParserFactory.newInstance()
  - internally use the ServiceLoader
  - a variant

    newInstance(String factoryClassName,
                ClassLoader classLoader)
    - looks for a given class

JDOM – Overview

- http://www.jdom.org/
- API for XML
- directly for Java
  - uses std. API of Java (collections,...)
- easy to be used
- fast
- light-weight

Usage

```
SAXBuilder builder = new SAXBuilder();
Document doc = builder.build(filename);
Element root = doc.getRootElement();

List children = current.getChildren();
Iterator iterator = children.iterator();
while (iterator.hasNext()) {
  Element child = (Element) iterator.next();
  ...
}
```

JDBC

Overview

- interface for accessing relational databases
- unified
  - database independent
    - database vendor must provide a JDBC driver
- allows
  - executing SQL queries
  - accessing results of queries
    - similar to the reflection API
- packages
  - java.sql, javax.sql

JDBC Driver

- JDBC API
  - in fact only interfaces
  - an implementation is provided via the driver
- driver
  - explicitly loaded and registered
  - Class.forName("com.driver.Name");
- after the driver is loaded, a connection to DB is created
  - Connection con = DriverManager.getConnection(url,
    "myLogin", "myPassword");
  - url
    - jdbc:mysql://localhost/test
    - jdbc:odbc:source

Basic classes and interfaces

- DriverManager – class
  - all methods are static
  - getConnection()
    - several variants
  - getDrivers()
    - all loaded drivers
  - getLogWriter(), setLogWriter()
  - println()
    - printing to a log
  - getLoginTimeout(), setLoginTimeout()

Basic classes and interfaces

- Connection – interface
  - creating and executing queries
- ResultSet – interface
  - query results

Basic example

```
Class.forName("com.mysql.cj.jdbc.Driver");
Connection con = DriverManager.getConnection(
              "jdbc:mysql://localhost/test", "","");

Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM
  test");

while (rs.next()) {
  // processing results line-by-line
}

stmt.close();
con.close();
```

Accessing results

- similar to the reflection API
  - getString(), getInt(),...
  - work with current line
  - identification of a column by
    - name
    - order

```
ResultSet rs = stmt.executeQuery("SELECT ID,
                             NAME FROM TEST");
while (rs.next()) {
  int id = rs.getInt("ID");
  String s = rs.getString("STRING");
  System.out.println(id + " " + s);
}
```

Accessing results

- ResultSet.next()
  - must be called even for the first line
- getString()
  - can be called to all types
    - with exception of SQL3 types
  - automatic conversion to String

Queries

- Connection.createStatement()
  - (“empty”) query creation
- Statement.executeQuery("....")
  - for queries returning results (SELECT)
  - results via ResultSet
- Statement.executeUpdate("...")
  - for queries returning no results
    - UPDATE
    - CREATE TABLE
    - …

PreparedStatement

- PreparedStatement
  - interface
  - extends Statement
  - a prepared query with parameters
    - set before execution
  - methods
    - set*Type*(int index, *type* v)
    - clearParameters()

```
PreparedStatement pstmt =
con.prepareStatement("UPDATE EMPLOYEES SET
                      SALARY = ? WHERE ID = ?");

pstmt.setBigDecimal(1, 153833.00)
pstmt.setInt(2, 110592)
```

Transactions

- by default – auto-commit mode
  - *commit* is performed after each change
- auto-commit can be unset

```
con.setAutoCommit(false);
//
// a sequence of queries
//
con.commit();    // or  con.rollBack()
con.setAutoCommit(true);
```

Callable Statements

- access to stored procedures
- extends PreparedStatement
  - setting parameters
    - set*Type*(int index, *type* v)
  - returning type must be registered
    - registerOutParameter(int index, int sqlType)
  - format
    a) {?= call <procedure-name>[<arg1>,<arg2>, …]}
    b) {call <procedure-name>[<arg1>,<arg2>, …]}

```
CallableStatement cs = con.prepareCall("{call
                                    SHOW_SUPPLIERS}");
ResultSet rs = cs.executeQuery();
```

Handling errors

- SQLException
  - and its children
  - String getSQLState()
    - defined by X/Open
  - int getErrorCode()
    - specific for particular database
- warnings
  - SQLWarning
  - it is not an exception
  - must be explicitly tested
    - Statement.getWarnings()
    - SQLWarning.getNextWarning()

Batch update

- handling several queries together
- Statement.addBatch(String sql)
  - adds a query to the batch
- int[] Statement.executeBatch();
  - executes the batch
  - returns a number of affected lines for each query in the batch

Updatable ResultSet

- the default ResultSet cannot be changed, can be iterated only forward
  - can be changed when the Statement is created

```
Statement stmt = con.createStatement(
            ResultSet.TYPE_SCROLL_INSENSITIVE,
                  ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT ...");
```

  - the resulting ResultSet can be changed, iterated freely
    - changes from different users are not visible in it

Object databases

- non-relational databases
- storing and querying objects
- own access without JDBC

- db4o
- NeoDatis
- …

- an example for NeoDatis

```
Sport sport = new Sport("volley-ball");
ODB odb = ODBFactory.open("test.neodatis");
odb.store(sport);
Objects<Player> players = odb.getObjects(Player.class);
odb.close();
```

- not much used

ORM

- an issue with OO databases
  - easy usage
  - lower performance, smaller support
- solution – ORM
  - object-relational mapping
  - a layer mapping a relational database to objects
  - roughly
    - class ~ a table scheme
    - object ~ row in a table
  - JDBC is typically used internally
    - automatically
  - Hibernate
    - http://hibernate.org/
    - the most used ORM for Java
      - also implementations for different platforms

Document-oriented databases

- storing documents
  - semi-structured date
- MongoDB
  - https://www.mongodb.com/
  - documents ~ JSON

```
MongoClient mongoClient = new MongoClient();
MongoDatabase database = mongoClient.getDatabase("mydb");
MongoCollection<Document> collection =
database.getCollection("test");
Document doc = new Document("name", "MongoDB")
                .append("type", "database")
                .append("count", 1)
                .append("versions", Arrays.asList("v3.2",
                                          "v3.0", "v2.6"))
                .append("info", new Document("x",
                                   203).append("y", 102));
collection.insertOne(doc);
```

Mongo

- there even exists a JDBC driver for Mongo
  - collections ~ tables