JAVA

Serialization

Overview

- "saving" complete objects
  - objects “survive” through programs' executions
- persistence
  - lightweight persistence
  - explicit saving and loading
- serialized objects can be transferred via network
- saving a state of objects
  - attributes
- code of the class of the object must be available

Usage

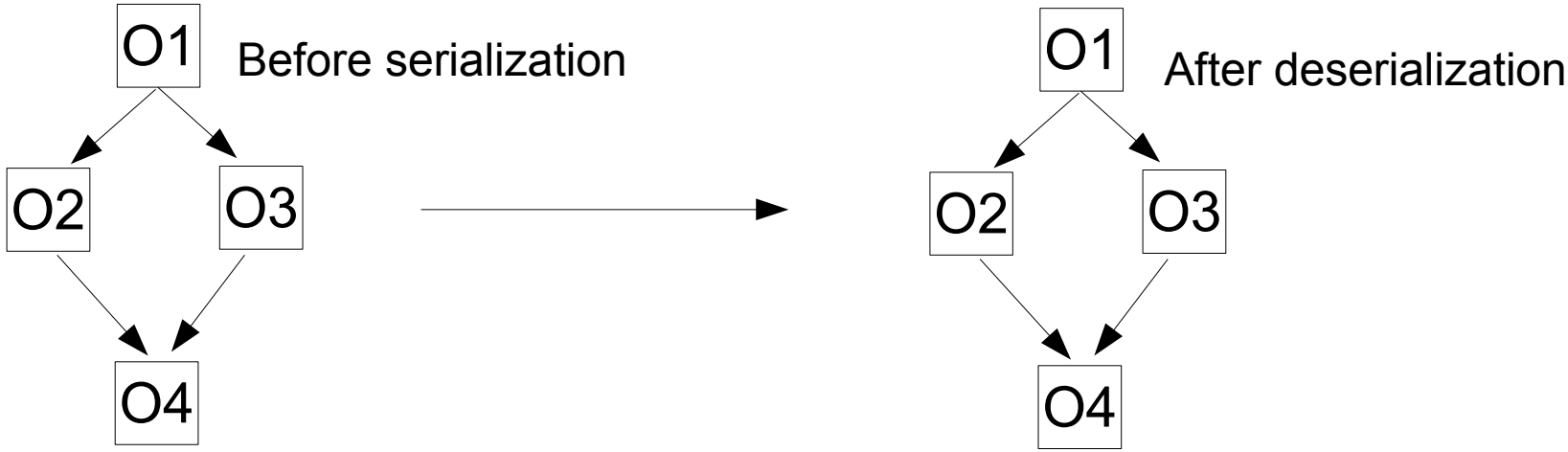
- `java.io.Serializable`
  - empty interface
  - serializable objects must implement it
- `ObjectOutputStream`
  - extends `OutputStream`
  - implements `DataOutput` and `ObjectOutput`
  - the method `void writeObject(Object o)`
- `ObjectInputStream`
  - extends `InputStream`
  - implements `DataInput` and `ObjectInput`
  - the method `Object readObject()`

Example

```
public class Data implements Serializable {
  private int d;
  public Data(int d) {this.d = d;}
  public String toString() {
    return super.toString() + ", d=" +d;
  }
}
...
Data data = new Data(1);
...
ObjectOutputStream out = new ObjectOutputStream(
      new FileOutputStream("file.dat"));
out.writeObject(data);
...
ObjectInputStream in = new ObjectInputStream(
      new FileInputStream("file.dat"));
data = (Data) in.readObject();
```

Serialization

- all attributes (even private ones) are serialized/deserialized
  - the attribute modifier `transient`
    - the attribute will not be saved/read
- both primitive and also references are saved
  - recursively are saved all objects from the attributes
  - during deserialization objects are created “in the same shape”
  - př.



Own serialization

- interface Externalizable
  - extends Serializable
  - two methods
    - `void readExternal(ObjectInput in)`
    - `void writeExternal(ObjectOutput out)`
- objects implement Externalizable instead of Serializable
- the rest is the same (almost)
- the `transient` modifier has no meaning
  - saving/reading through the methods `writeExternal` and `readExternal`
- writeExternal and readExternal are called automatically

Example

```
public class Data2 implements Externalizable {
  public Data2() { System.out.println("Data2"); }
  public void writeExternal(ObjectOutput out)
      throws IOException {
    System.out.println("Data2.writeExternal");
  }
  public void readExternal(ObjectInput in)
     throws IOException, ClassNotFoundException {
    System.out.println("Data2.readExternal");
  }
}
...
Data2 d = new Data2();
ObjectOutputStream o = ....
o.writeObject(d);
...
ObjectInputStream i = ....
d = (Data2) o.readObject();
```

Wrong example

```
public class Data3 implements Externalizable {
  Data3() { System.out.println("Data3"); }
  public void writeExternal(ObjectOutput out)
      throws IOException {
    System.out.println("Data3.writeExternal");
  }
  public void readExternal(ObjectInput in)
     throws IOException, ClassNotFoundException {
    System.out.println("Data3.readExternal");
  }
}
...
Data3 d = new Data3();
ObjectOutputStream o = ....
o.writeObject(d);
...
ObjectInputStream i = ....
d = (Data3) o.readObject();  // an exception occurs!!
```

Loading objects

- implicit serialization (implementing Serializable)
  - during loading no constructor is called
  - objects are created directly
- own serialization (implementing Externalizable)
  - first, a constructor is called
    - the default constructor without parameters
    - must be available
  - then, the `readExternal()` is called on the object

Another approach

- implement the interface Serializable
- and add 2 „magic“ methods
  - `private void writeObject(ObjectOutputStream stream)throws IOException;`
  - `private void readObject(ObjectInputStream stream) throws IOException, ClassNotFoundException`
- both methods must have exactly the given signature
  - must be private
- in `readObject()` and `writeObject()`, default loading/saving can be called by the methods `defaultReadObject()` and `defaultWriteObject()`

Example

```
public class Test implements Serializable {
  private String a;
  private transient String b;
  public Test(String aa, String bb) {
    a = "Not Transient: " + aa;
    b = "Transient: " + bb;
  }
  private void writeObject(ObjectOutputStream
  stream)
  throws IOException {
    stream.defaultWriteObject();
    stream.writeObject(b);
  }
  private void readObject(ObjectInputStream stream)
  throws IOException, ClassNotFoundException {
    stream.defaultReadObject();
    b = (String)stream.readObject();
  }
```

Other „magic“ methods

- private void readObjectNoData() throws ObjectStreamException
  - called during loading an object if some of its classes (the class or superclasses) are not stored in a stream
  - usage – when class hierarchy is changed between storing/loading
    - ex: saving an object of the class Monkey, which extends Animal and loading the object of the class Monkey, which extends Mammal and it extends Animal (the method is used on the class Mammal)

Other „magic“ methods

- *anything* Object readResolve() throws ObjectStreamException
  - if the method exists, deserialization of an object of the class returns the result of this method

- *anything* Object writeReplace() throws ObjectStreamException
  - if exists, its result is serialized

serialVersionUID

- *anything* static final long serialVersionUID = *value*
  - if during deserialization the saved value is different from the value in the class, the InvalidClassException is thrown
  - not necessary to use
    - created automatically during serialization
  - but its explicit declaration is strongly recommended

Serialization and std library

- many classes in the std. library implement Serializable
- warning – serialization may not work between different Java version
  - typically a warning in the documentation

Warning: Serialized objects of this class will not be compatible with future Swing releases. The current serialization support is appropriate for short term storage or RMI between applications ...

Preferences

Overview

- the package java.util.prefs
- since JDK 1.4
- for storing/loading a configuration of programs
- automatically stored/loaded
  - exact place depends on OS
  - separately per user
- only primitive types and strings (max. 8 KB long)
- tuples
  - key – value
  - does not implement the interface Map
- hierarchical structure (tree)
  - usually just a single node

Usage

- static methods of the class Preferences
- `Preferences userNodeForPackage(Class c)`
  - returns a node of preferences associated with the package of the given class
- `Preferences systemNodeForPackage(Class c)`
  - as the previous method
  - a node common for all users
- ex:
  - `p = Preferences.userNodeForPackage(Foo.class)`
- name of the node ~ full name of the package
  - dots are replaced by slashes "/"

Example

```
public class Prefs {
  public static void main(String[] args) {
    Preferences prefs = Preferences
      .userNodeForPackage(Prefs.class);
    prefs.put("url", "http://somewhere/");
    prefs.putInt("port", 1234);
    prefs.putBoolean("connected", true);
    int port = prefs.getInt("port", 1234);

    String[] keys = prefs.keys();
    for (int i; i<keys.length; i++) {
      System.out.println(keys[i] + ": "+
          prefs.get(keys[i], null));
    }
  }
}
```

Methods

- `String get(String key, String def)`
  - returns a value of the key
  - the implicit value must be set
- `int getInt(String key, int def)`
  - as get
  - defined for all the primitive types
- `void put(String key, String val)`
  - assignes a value to the key
  - defined also for all the primitive types
- `String[] keys()`
  - return all keys
- `void flush()`
  - writes the changes

Methods

- `void clear()`
  - clears all the preferences in the node
- `String name()`
  - a name of the node
- `String absolutePath()`
  - an absolute name of the node

- all methods are thread safe
- can be safely used from multiple JVMs at the same time

Communication over network

Overview

- the package java.net
- since JDK1.0
- easy communication over network
- almost as using files
  - streams over network
- protocols TCP and UDP
  - Internet

URI and URL

java.net.URI

- representation of URI
  - unique resource identifier (RFC 2396)
- structure URI
  - [scheme:]scheme-specific-part[#fragment]
- absolute URI – has a schema
  - relative URI – has not a schema
- "opaque" URI – the specific part does not start with the slash
  - ex: mailto:java-net@java.sun.com
    news:comp.lang.java
- hierarchical URI – either an absolute URI starting with the slash or relative URI
  - př: http://java.sun.com/j2se/1.3/
    ../../../demo/jfc/SwingSet2/src/SwingSet2.java

java.net.URI

- hierarchical URI – structure
  - [scheme:][//authority][path][?query][#fragment]
  - authority
    - [user-info@]host[:port]
- all parts of URI are Strings, except the port, which is int
- normalization of URI
  - removing and replacing "." and ".."

java.net.URI: methods

- `String getScheme()`
- `String getSchemeSpecificPart()`
- `String getPath()`
- `String getHost()`
- .....
- `boolean isAbsolute()`
- `boolean isOpaque()`
- `void normalize()`
- `URL toURL()`
  - creates URL from URI
  - an exception thrown if cannot be created

java.net.URL

- URL is a special case of URI
- unique resource locator
- specifying resources in the web
  - http://www.mff.cuni.cz/
- similar methods like URI
  - get...
- `InputStream openStream()`
  - opens a stream for reading a file specified by the URL
- `URLConnection openConnection()`
  - creates a connection to the URL object

URLConnection

- representation of a connection between the application and URL
- usage
  1. obtaining a connection (openConnection())
  2. setting parameters
     e.g. setUseCaches()
  3. creating the connection (connect())
     the remote object is available then
  4. obtaining content and information
     content – getContent()
     headers – getHeaderField()
     streams – getInputStream(), getOutputStream()
     other – getContentType(), getDate(), ...

java.net

Identification
(DNS)

InetAddress

- represents an IP address
- obtaining an address
  - static methods of InetAddress
  - InetAddress getByName(String host)
    - IP address of the given name of a node
    - returns localhost for null
  - InetAddress getByAddress(byte[] addr)
    - IP address for the given address
    - length of the addr array – 4 for IPv4, 16 for IPv6
  - InetAddress getLocalHost()
    - address of localhost (127.0.0.1)

Example

```
public class InetName {
  public static void main(String[] args) throws
  Exception {
    InetAddress a =  InetAddress.getByName(args[0]);
    System.out.println(a);
  }
}

public class Localhost {
  public static void main(String[] args) throws
  Exception {
    System.out.println(InetAddress.getByName(null));
    System.out.println(InetAddress.getLocalHost());
  }
}
```

java.net

Sockets

Overview

- socket = endpoint of a connection
- TCP
  - reliable communication
- connections in both directions
  - both IntputStream and OutputStream can be obtained
- the ServerSocket class
  - creates a "listening" socket
  - the accept() method
    - waits for an incoming connection
    - returns a socket for communication
- the Socket class
  - a socket for communication

Example: simple server

```
public static void main(String[] args) throws IOException {
  ServerSocket s = new ServerSocket(6666);
  try {
    Socket socket = s.accept();
    try {
      Reader in = new InputStreamReader(
              socket.getInputStream()));
      PrintWriter out = new PrintWriter(new OutputStreamWriter(
              socket.getOutputStream()), true);
      while (true) {
        String str = in.readLine();
        if (str.equals("END")) break;
        out.println(str);
      }
    } finally {
      socket.close();
    }
  } finally {
    s.close();
  }
```

Example: simple client

```
public static void main(String[] args) throws IOException
   {
   InetAddress addr = InetAddress.getByName(null);
   Socket socket = new Socket(addr, 6666);
   try {
     Reader in = new InputStreamReader(
          socket.getInputStream());
     PrintWriter out = new PrintWriter(
          new OutputStreamWriter(
             socket.getOutputStream()), true);
     for(int i = 0; i < 10; i ++) {
       out.println(i);
       String str = in.readLine();
       System.out.println(str);
     }
   } finally {
     socket.close();
   }
}
```

Serving incoming requests

- the previous example – simple server
  - serves only one connections
- serving multiple connections
  - a new thread for each incoming connection
  - channels and the Selector class
    - serving multiple requests in a single thread
    - the selector holds a set of sockets
      - the select() method waits until at least one socket is ready to be used
    - similar to the select() function in UNIX systems

Multithread server

```
class ServeConnection extends Thread {
  private Socket socket; private Reader in;
  private PrintWriter out;
  public ServeConnection(Socket s) throws IOException {
    socket = s; in = ...; out = ...; start();
  }
  public void run() {
    while (true) {
      String str = in.readLine();
      if (str.equals("END")) break;
      out.println(str);
    }
   ...
}
public class Server {
  public static void main(String[] args) throws
  IOException {
    ServerSocket s = new ServerSocket(6666);
    while(true) {
      Socket socket = s.accept();
      new ServeConnection(socket);
```

java.net

UDP

Overview

- unreliable communication
- the DatagramSocket class
  - for both server and client
  - sending/receiving datagrams
  - void send(DatagramPacket d)
  - void receive(DatagramPacket d)
- the DatagramPacket class
  - a datagram
  - void setData(byte[] buf)
  - byte[] getData()
    - sets/returns a buffer for the datagram
  - int getLength()
  - void setLength(int a)
    - length of data in the datagram

RMI
(Remote Method Invocation)

Overview

- the java.rmi package
- calling methods on remote objects
- "look like" local calls
  - but slower ones
- usage
  - definition of an interface
    - the interface must extend the Remote interface
    - all methods must throw RemoteException
  - implementation of the object
  - a client uses the object through the interface
- at the client side, there is a “proxy” object, which passes calls over the network
- at the server side, there is a “skeleton” object,which receives calls a passes them to the object

Simple server

```
public interface Hello extends Remote {
  void hello(String msg) throws RemoteException;
}

public class HelloImpl implements Hello extends
  UnicastRemoteObject {
  public void hello(String msg) throws RemoteException {
    System.out.println(msg);
  }

  public static void main(String[] args) {
    Hello h = new HelloImpl();
    Naming.rebind("Hello", h);
  }
}
```

Simple client

```
public class HelloClient {
  public static void main(String[] args) {
    Hello h = (Hello) Naming.lookup("Hello");
    h.hello("Hello");
  }
}
```

- RMI is cover in detail in the summer semester

java.util

Time, date

java.util.Date

- represents time with millisecond precision
  - since 1.1.1970
- most of the methods are *deprecated*
  - since JDK1.1 replaced by **Calendar**
- constructors
  - `Date()`
    - an instance will hold time at which it was allocated
  - `Data(long date)`
    - an instance will hold the given time
- methods – in fact comparisons only
  - `boolean after(Date d)`
  - `boolean before(Date d)`
  - `int compareTo(Date d)`
- other ones are *deprecated*

java.util.Calendar

- abstract class
- the only non-abstract child
  - GregorianCalendar
- static attributs
  - what can be obtained/set
    - YEAR, MONTH, DAY_OF_WEEK, DAY_OF_MONTH, HOUR, MINUTE, SECOND, AM_PM, ...
  - months – JANUARY, FEBRUARY, ...
  - days in a week – SUNDAY, MONDAY, ...
  - other – AM, PM, ...

java.util.Calendar: methods

- obtaining an instance – static methods
  - `getInstance()`
    - default timezone
  - `getInstance(TimeZone tz)`
- getting/setting time
  - `Date getTime()`
  - `long getTimeInMillis()`
  - `void setTime(Date d)`
  - `void setTimeInMillis(long t)`
- comparison
  - `boolean before(Object when)`
  - `boolean after(Object when)`

java.util.Calendar: methods

- obtaining individual fields
  - `int get(int field)`
  - ex. int day = cal.get(Calendar.DAY_OF_MONTH)
- setting individual fields
  - `void set(int field, int value)`
  - ex. cal.set(Calendar.MONTH, Calendar.SEPTEMBER)
  - resulting time in milliseconds is recalculated just during calls get(), getTime(), getTimeInMillis()
- adding to fields
  - `void add(int field, int delta)`
  - if necessary, modifies other fields also
  - resulting time in milliseconds is recalculated immediately
- adding to fields without modification of other fields
  - `void roll(int field, int amount)`
  - `void roll(int field, boolean up)`

java.util.TimeZone

- representation of a time zone
- understands summer/winter time
- obtaining a time zone
  - `TimeZone getDefault()`
    - static method
    - returns the timezone set in a system
  - `TimeZone getTimeZone(String ID)`
    - returns required time zone
- possible ID
  - `String[] getAvailableIDs()`
  - static method
- IDs have a form
  - "America/Los_Angeles"
  - GMT +01:00

java.util

Timer

Usage

- scheduling tasks for future execution
  - one-time or repeated
- task = TimerTask
- all tasks in a single timer are executed in a single thread
  - a task should finish quickly
- scheduling a task
  - `void schedule(TimerTask t, Date d)`
    - schedules the task for the given time
  - `void schedule(TimerTask t, Date d, long period)`
    - schedules the task repeatedly
    - period – time in milliseconds between executions

Usage

- scheduling a task (cont.)
  - `void schedule(TimerTask t, long delay)`
    - schedules the task after given delay
  - `void schedule(TimerTask t, long delay, long period)`
    - schedules the task repeatedly
    - period – time in milliseconds between executions
  - `void scheduleAtFixedRate(TimerTask t, Date d, long period)`
  - `void scheduleAtFixedRate(TimerTask t, long delay, long period)`
    - schedules the task repeatedly
    - period – time in milliseconds between executions relatively to initial execution

Usage

- the method `void cancel()`
  - cancels the timer
  - no further scheduled tasks are executed
  - currently executed task is finished
  - can be called repeatedly
    - further calls do nothing
- the class `TimerTask`
  - implements the interface Runnable
  - abstract class – the run() method must be implemented
  - other methods
    - void cancel()
      - cancels the task
    - long scheduledExecutionTime()
      - time of the most recent actual execution

java.util

Localization

java.util.Locale

- represents a specific geographical, political, or cultural region
- defines how to print out texts, numbers, currency, time
- creation
  - `Locale(String language)`
  - `Locale(String language, String country)`
  - `Locale(String language, String country, String variant)`
  - ex. new Locale("cs", "CZ")
- `static Locale[] getAvailableLocales()`
  - returns all installed *locales*
- static Locale getDefault()
  - returns the default locale

java.util.ResourceBundle

- contains "localized" objects
  - e.g. strings
- *bundle* always bellongs to a group with common base name – e.g. MyResources
  - full name of a bundle = base name + locale id
  - ex. MyResources_cs, MyResources_de, MyResources_de_CH
  - default *bundle* – with the base name only
  - each bundle in a group holds the same stings :transformerd for a particular sandpaper
  - if requested bundle is not available, the default one is used

ResourceBundle: Usage

- obtaining *bundels*
  - `ResourceBundle.getBundle("MyResources")`
  - `ResourceBundle.getBundle("MyResources", currentLocale)`
- *bundle* contains tuples key/value
  - keys are the same for oal locales in a group, the valueis different

- usage

```
ResourceBundle rs =
  ResourceBundle.getBundle("MyResources");
...
button1 = new Button(rs.getString("OkKey"));
button1 = new Button(rs.getString("CancelKey"));
```

ResourceBundle: Usage

- keys – String type
- value – any type
- obtaining an object from the buffer
  - `String getString(String key)`
  - `String[] getStringArray(String key)`
  - `Object getObject(String key)`
    - ex: `int[]`

      ```
      ai=(int[])rs.getObject("intList");
      ```
- ResourceBundle – abstract class
- two implementations
  - ListResourceBundle
  - PropertyResourceBundle

ListResourceBundle

- abstract class
- children must redefine the method
  - `Object[][] getContents()`

```
public class MyResources extends ListResourceBundle {
      public Object[][] getContents() {return contents;}
      static final Object[][] contents = {
              {"OkKey", "OK"},
              {"CancelKey", "Cancel"},
      };
 }
public class MyResources_cs extends ListResourceBundle {
      public Object[][] getContents() {return contents;}
      static final Object[][] contents = {
              {"OkKey", "OK"},
              {"CancelKey", "Zrušit"},
      };
 }
```

PropertiesResourceBundle

- is not abstract
- no other class is directly created
- localized strings are in files
- a name of the file
  - base name + locale + ".properties"
  - ex. myresources.properties
    myresources_cs.properties
- obtaining the bundle
  - `ResourceBundle.getBundle("myresources")`
- the format of the file
  - `key=value`
  - `# comment till the end of the line`

Own implementation

- extending directly ResourceBundle
- overriding methods
  - Object handleGetObject(String key)
  - Enumeration getKeys()

```
public class MyResources extends ResourceBundle {
  public Object handleGetObject(String key) {
    if (key.equals("okKey")) return "Ok";
    if (key.equals("cancelKey")) return "Cancel";
    return null;
  }
}
public class MyResources_cs extends ResourceBundle {
  public Object handleGetObject(String key) {
    // nemusí definovat všechny klíče
    if (key.equals("cancelKey")) return "Zrušit";
    return null;
  }
}
```