Факультет комп’ютерних наук

Залікова робота

«Крос-платформне програмування»

Варіант №2

П.І.Б. Безрук Юрій Русланович

Група КС-21

1. Особливості SAX та DOM парсерів. Основи створення та використання SAX, DOM парсерів. (*10 балів*)

   SAX и DOM парсеры используются синтаксического анализа и изменения XML-документов. SAX-парсеры являются парсерами, основанными на событиях. Они последовательно просматривают документ, анализируют его, отмечая события, которыми считаются появления очередных элементов XML, таких как открывающий или закрывающий тег или текст, содержащийся внутри элемента. Когда парсер встречает элемент, он вызывает соответствующий метод обработки, предназначенный для этого типа элемента. Эти методы прописываются в классе-обработчике, который является наследником класса DefaultHandler. Наиболее часто используемые методы-обработчки:

   public void startDocument() –вызывается при начале обработки документа, в этом методе задаются какие-либо начальные действия, которые нужно выполнить при старте обработки документа (создание каких либо структур, инициализация переменных и т.д.)

   public void endDocument() – вызывается при обработке конечного тега документа.

   public void startElement(String uri, String name, String qname Attributes attrs) – вызывается парсером , когда тот встречает символ «<», тоесть при открывающем теге элемента. В метод передается идентификатор простанства имен uri, локальное имя тега без префикса name, расшренное имя с префиксом qname и атрибуты элемента attrs, если они есть. Первые два аргумента равны null, если если пространство имен не определено.

   public void endElement(String uri, String name, String qname) – вызывается при появлении символов «</», тоесть при появлении закрывающего тега элемента.

   public void characters(char[] ch, int start, int length) – вызывается когда парсер встречает текстовое содержимое. В метод передается массив символов ch, индекс начала строки start и количество символов length.

   Пример обработчика:

```
public class NotebookHandler extends DefaultHandler{
	@Override
	public void startDocument() throws SAXException {
		System.out.println("Start document processing...");
	}
	@Override
	public void endDocument() throws SAXException {
		System.out.println("Stop document processing.");
	}
	@Override
	public void startElement(String uri, String localName, String qName, Attributes attributes) throws SAXException {
		System.out.println("Start element"+qName+"processing...");
	}
	@Override
	public void endElement(String uri, String localName, String qName) throws SAXException {
System.out.println("Stop element"+qName+"processing...");
	}
	@Override
	public void characters(char[] ch, int start, int length) throws SAXException {
		String str = new String(ch, start, length);
		System.out.print(str.trim());
	}
}
```

После того, как обработчик написан, нужно инициализировать парсинг документа. Для этого вначале необходимо создать фабрику парсера, с помощнее создать парсер, затем создать объект класса-обработчика и входной файловый поток с XML-документом. После чего у парсера вызывается метод parse(), куда передаются обработчик и поток с файлом.

Пример приведен ниже:

```
SAXParserFactory factory = SAXParserFactory.newInstance();
		try {
			SAXParser parser = factory.newSAXParser();
			DefaultHandler handler = new NotebookHandler();
			InputStream xmlStream = new FileInputStream("data.xml");
			parser.parse(xmlStream, handler);
		} catch (ParserConfigurationException | SAXException | IOException e) {
```

```
            e.printStackTrace();
        }
```

DOM-парсеры производят анализ, основываясь на структуре дерева, отражающего вложенность элементов документа. Из вложенных тегов и элементов XML в таком случае в оперативной памяти строится дерево перед просмотром. Эти парсеры проще в реализации, но создание дерева требует большого объёма оперативной памяти. Необходимость частого просмотра узлов дерева сильно замедляет работу такого парсера.

Инициализация парсера происходит похожим образом с SAX-парсером:

```
DocumentBuilderFactory dbf=DocumentBuilderFactory.newInstance();
File xml = new File("data.xml");
Document doc = null;
try{
    DocumentBuilder db = dbf.newDocumentBuilder();
    doc=db.parse(xml);
}catch(Exception e){
     e.printStackTrace();
};
```

После чего мы получаем объект класса Document, который представляет собой документ xml в виде дерева (коллекции узлов). Для работы с ним определены классы, представляющие собой различные типы узлов.

Element – базовый блок xml. Имеют потомков, текстовые узлы и их комбинации. В отличии от остальных типов узлов, могут иметь атрибуты, которые хранятся в NamedNodeMap.

Attr – атрибут узла, имеет имя и значение, но не рассматривается как потомок элемента.

Текстовые узлы – просто текстовая информация ил пробельные символы. Данный тип узла имеет служебное имя #text, а его значение – это, собственно, его текстовое содержимое.

Node – базоый тип для всех типов узлов.

Объект класса Document может выдавать много различной информации о содержимом документа – его корневой элемент, элементы по имени тега, по идентификатору, потомки атрибуты, типы узлов, их имена, значения и т.д.

Кроме того, при помощи Dom-парсера можно изменять содержимое документа: создавать новые узлы, заменять их, удалять.

Для переведения дом-дерева в xml-документ необходимо использовать трансформер, которому можно передавать различные настройки формирования документа.

Можно рассмотреть это на создании нового документа при помощи DOM-парсера:

```
DocumentBuilderFactory domFactory;
DocumentBuilder db;
Document doc;
Element root;
Element subroot;
Element element;
domFactory = DocumentBuilderFactory.newInstance();
db = domFactory.newDocumentBuilder();
doc = db.newDocument();
root = doc.createElement("ROOT");
doc.appendChild(root);
subroot = doc.createElement("subroot");
root.appendChild(subroot);
element = doc.createElement(“element”);
subroot.appendChild(element);

Transformer trf = null;
DOMSource src = null;
try {
trf = TransformerFactory.newInstance().newTransformer();
trf.setOutputProperty(OutputKeys.ENCODING, "UTF-8");
trf.setOutputProperty(OutputKeys.STANDALONE, "yes");
trf.setOutputProperty(OutputKeys.INDENT, "yes");
trf.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "4");
doc.setXmlStandalone(true);
src = new DOMSource(doc);
StreamResult result = new StreamResult(new File("data.xml"));
trf.transform(src, result);
} catch (TransformerFactoryConfigurationError | TransformerException e) {
        e.printStackTrace();
}
```

2. *JavaBeans* компоненти. Підтримка простих властивостей та властивостей з обмеженнями (simple, constrained properties). (*10 балів*)

JavaBeans – это компонентная архитектура для Java, классы написанные по определенным правилам. Они используются для объединения нескольких объектов в один для удобной работы и взаимодействия. Каждый бин должен поддерживать следующие свойства:
Интроспекцию- это позволяет средам разработки анализировать из чего состоит и как работает данный бин.
Настраиваемость т.е. возможность изменять внешний вид (положение, размеры и т.п.) и поведение данного бина
Поддержка "событий как средства связи данного бина с программой и другими бинами.
Поддержку свойств или атрибутов (properties), которые используются, в частности, для настройки (например, ширина, высота, количество каких-либо составных подкомпонент и т.п.).
Сохраняемость - это необходимо для того, чтобы после настройки конкретного бина в некоторой визуальной среде разработки была возможность сохранить параметры настройки, а потом их восстановить.
Каждый бин имеет свойства, которые определяют, как он будет работать и/или как он будет выглядеть. Эти свойства являются private или protected полями класса бина, которые доступны для чтения и/или модификации через специальные public методы – аксессоры.

Простыми свойствами называют те свойства, которые содержат только одно значение, например строку или число. Простые свойства бинов могут быть элементарных типов, стандартных типов Java, а также пользовательских типов.

Таким образом, если мы говорим, что какой либо бин Test имеет свойство типа Property имени name_property, то это значит, что у данного бина есть private-поле Property name_property, и геттер и сеттер к нему:

```
public PropertyType getNameProperty () {
return name_property;
}
public void setNameProperty (Property name_ property) {
        this.name_ property = name_ property;
}
```

Свойствами с ограничением называются свойства, которые могут быть запрещены в изменении. Т.е. бин будет спрашивать разрешение у зарегистрированных слушателей на изменение данного свойства. В случае, если

слушатель не разрешает ему менять свойство, он выбрасывает PropertyVetoException. Таким образом данное исключение нужно будет обрабатывать в классе, вызывающем сеттер на ограниченное свойство, и, соответственно, этот класс будет извещен о том, что ему «отказано в доступе» к этому свойству.

Пока все зарегистрированные обработчики не согласятся с предложенными изменениями, значение свойства остается прежним. Тоесть ни один обработчик не может быть уверенным, что изменения, против которых он не возражает, действительно будут выполнены. Единственный способ убедиться в том, что значение свойства обновлено, — использовать обработчик события, соответствующего изменению значения свойства.

Для создания свойства с ограничениями необходимо реализовать в составе компонента два метода, предназначенные для управления объектами VetoableChangeListener, и, соответственно, методы add… и remove… для этого слушателя, а также VetoableChangeSupport), предназначенный для управления обработчиками событий изменения, которые могут быть запрещены.

Пример использования свойства с ограничением:

```
public class Bean {
   private static final long serialVersionUID = 1L;
        private double name;
   public Bean() {
     this.name=”name”;
   }

  public double getName() {
     return temperature;
   }

 public void setName(String name)throws PropertyVetoException{
     try {
        String old = this.name;
        vetoChangeSupport.fireVetoableChange("name", old, name);
         this.name = name;
        System.out.println("The name was changed");
```

```
    } catch (PropertyVetoException e) {            throw e;       }    }


  @Override    public String toString() {
     return "Bean [name=" + name + "];
   }
      public synchronized void addVetoableChangeListener(VetoableChangeListener
listener) {       vetoChangeSupport.addVetoableChangeListener(listener);
    }

  public synchronized void removeVetoableChangeListener(VetoableChangeListener
listener) {       vetoChangeSupport.removeVetoableChangeListener(listener);    }


}

public class TempVetoableChangeListener implements VetoableChangeListener {
      private double threshold;

  public TempVetoableChangeListener() {       this.threshold = 4.0;    }

  public double getThreshold() {       return threshold;    }

  public void setThreshold(double threshold) {       this.threshold = threshold;    }

  @Override     public void vetoableChange(PropertyChangeEvent evt)          throws
PropertyVetoException {                System.out.println("Attempt to set " +
evt.getPropertyName() + " from "                  + evt.getOldValue() + " to " +
evt.getNewValue());       boolean veto = (double) evt.getNewValue() > this.threshold;
if (veto) {       throw new PropertyVetoException("the reason for the veto", evt);      }
} }

public class Main {
  public static void main(String[] args) {
     Bean bean = new Bean();
      bean.addVetoableChangeListener(new TempVetoableChangeListener());          //
TempVetoableChangeListener lst = new TempVetoableChangeListener();
lst.setThreshold(3.5);       bean.addVetoableChangeListener(lst);       //          for(int
```

```
i = 0; i < 10; i++) {  try {  bean.setName(“hi”+i);  } catch (PropertyVetoException e) {  System.out.println("\tThe name was NOT changed");  }  }

    System.out.println("Final name: " + bean.getName());  }

}
```

3. Основи роботи з TCP сокетами. Основні етапи створення мережевого додатку за допомогою TCP сокетів. Приклад простого TCP сервера та клієнта: клієнт відправляє рядок на сервер, сервер повертає рядок великими літерами. (*10 балів*)

   В приложении разработаем три класса: Server, Client, Handler. Клиент и сервер хранят сокеты ServerSocket и Socket соответственно.
   В классе Client создается Socket, в конструктор передаются InetSocketAddress, (адрес сервера и номер порта и время задержки).
   Также Client хранит ObjectInputStream, ObjectOutputStream, полученные из Socket. Это необходимо для прямого взаимодействия с сервером.
   Все объекты которые мы передаем через потоки должны быть сериализуемы.
   Далее в отдельном методе формируется запрос в который помещается исходная строка. Метод отправляет через ObjectOutputStream строку на сервер. Далее метод ожидает появления объекта в ObjectInputStream и выводит ее в консоль.
   В классе Server содержится ServerSocket, в конструктор которого передается номер порта (номера портов клиента и сервера должны совпадать). В отдельном потоке содержится бесконечный цикл, который постоянно пытается принять запрос на соединение от клиента. Как только это происходит, сокет клиента сохраняется и передается обработчику.

   В классе Handler выполняется обработка запроса. Так же как и в Client там хранятся ObjectOutputStream и ObjectInputStream. В потокеобработчика из ObjectInputStream принимается исходная строка, изменяется и помещается в ObjectOutputStream. После этого Handler закрывается. Код реализации задания приведен в архиве task3.

4. Основи *RMI*. Наведіть приклад простого розподіленого додатку, виконаного за технологією *RMI*, в якому клієнт надає серверу рядок, а сервер повертає клієнту цей рядок подвоєним. (*10 балів*)

Java Remote Method Invocation (RMI) - это объектно-ориентированная версия удаленного вызова процедур (RPC). С помощью такого подхода можно без явного использования сокетов вызвать методы для объекта, который существует в другом адресном пространстве или на том же компьютере, или на другом компьютере, подключенном к исходному через сеть. Таким образом, с помощью этого подхода, можно организовать взаимодействие объектов, расположенных на разных компьютерах.

В структуре RMI приложения существует три стороны: серверная, клиентская часть и реестр объектов.

Задача серверная части программы состоит в создании удаленного объекта (remote object), который нужен для вызова метода. Этот объект обычный объект, за исключением того, что его класс реализует специальный интерфейс Java RMI. После того, как удаленный объект создан, он экспортируется и регистрируется в отдельном приложении, называемом реестром объектов.

Клиентская часть приложения сначала связывается с реестром объектов, для того, чтобы получить ссылку на удаленный объект с заданным именем. После этого, используя полученную ссылку, клиентская часть может вызывать методы на удаленном объекте так, как будто объект хранится в собственном адресном пространстве клиента. Технология RMI обрабатывает детали связи между клиентом и сервером (внутренне используя сокеты) и передает информацию в нужном направлении. Процедура связи полностью скрыта для клиентских и серверных приложений.

Реестр объектов похож на таблицу. Каждая запись таблицы отображает имя объекта на его прокси, который называется заглушка. Серверная часть приложения регистрирует заглушку с указанием имени в реестре объектов. После успешной регистрации заглушки в реестре объектов, объект становится доступным для использования другими объектами. Сразу после этого клиенты могут получить ссылку на удаленный объект из этого реестра (дескриптор объекта, фактически заглушку) и могут вызывать методы на этом объекте.

Реализация приведена в архиве task4.

Взаимодействие клиента и сервера происходит через интерфейсы compute.jar. Это общие файлы для сервера и клиента. После этого формируется запрос в классе ComputeString. Класс DoubledString содержит непосредственно выполнение условия задачи, которое будет выполнятьс на сервере.

Сервер принимает объект класса DoubledString который реализует интерфейс Task, применяет метод execute и отправляет ответ, который печатает клиент.