CSE 70: SE Principles and Java
Packages, Interfaces & Exceptions, Basic Data Structures

Ingolf Krueger
Learning Goals for Today
Learning Goals

• Understand the basics of encapsulation and information hiding

• Be able to utilize exceptions as a failure management technique

• Be able to write Java code that has
  – packages,
  – interfaces and
  – exceptions.

• Understand and be able to use Java Maps, Lists and Sets
Important Software Engineering Principles:
Encapsulation/Information Hiding
One Information Hiding Mechanism in Java: Interfaces

• Say only **what** you plan to implement, not **how**

• Abstract away from implementation details/give implementation later

• Can change the implementation while keeping the same interface

• **Java interface:**
  – Collection of abstract methods – no method bodies allowed
  – Acts as a specification/contract between a client and an implementation of the interface

• Classes can implement any number of interfaces

• Class implementing an interface must provide body for **all** methods defined in the interface
Example: ChatServer Interface

```java
public interface ChatServer {

    public void sign_on(String client_id);

    public void publish(String message);

    public boolean isPresent(String client_id);

}
```

ChatServer.java
Example: ChatServer Interface, First Implementation

```java
public class ChatServerImpl1 implements ChatServer {
  ...
  private String[] clients;
  private int client_index;
  ...
  public void sign_on(String client_id) {
    clients[client_index++] = client_id;
  }
  ...
}
```
Example: ChatServer Interface, Second Implementation

```java
public class ChatServerImpl2 implements ChatServer {
    ...;
    private LinkedList<String> clients;
    ...
    public void sign_on(String client_id) {
        clients.add(client_id);
    }
    ...
}
```
Important Software Engineering Practice:
Structure your code so you (and others) can find their way around in it.
One Structuring Mechanism in Java: Packages

• Structure your classes and interfaces into coherent groups.

• Limit visibility of your classes and interfaces (package/friendly visibility).

• Define *namespaces* for your classes and interfaces

• Organize your .java and .class files according to package structure.
Example: Chat System

• We’ll develop
  
  – Classes and interfaces belonging to the chat client

  – Classes and interfaces belonging to the chat server

  – Possibly classes and interfaces belonging to both

• Keeping all of these in the same source directory will quickly overwhelm our ability to stay on top of what is (defined/stored) where.
Idea: Organize the Chat System into Packages

- Candidate packages:
  - chat.server
  - chat.client
  - chat.common

- Package names are "."-separated lists of names

```java
package chat.server;

public interface ChatServer {
    public void sign_on(String client_id);
    public void publish(String message);
    public boolean isPresent(String client_id);
}
```

ChatServer.java
Idea: Organize the Chat System into Packages

- Candidate packages:
  - chat.server
  - chat.client
  - chat.common

- Package names are “.”-separated lists of names

```java
package chat.server;

public class ChatServerImpl implements ChatServer {
    ...
    public void sign_on(String client_id) {
        clients.put(client_id, client);
    }
    ...
}
```
Idea: Organize the Chat System into Packages

- Candidate packages:
  - chat.server
  - chat.client
  - chat.common

- Package names are "."-separated lists of names

```java
package chat.client;

public interface ChatClient {
    public void notify(String m);
}
```
Reflect Java Packages in the Directory Structure

package chat.server;

public interface ChatServer
{
... 
}

package chat.server;

public class ChatServerImpl
{
... 
}

package chat.client;

public interface ChatClient
{
... 
}
Reflect Java Packages in the Directory Structure

```java
package chat.server;

public interface ChatServer
{
    ...
}

package chat.server;

public class ChatServerImpl
{
    ...
}

package chat.client;

public interface ChatClient
{
    ...
}
```
How to Access Elements of a Package

package chat.server;

public interface ChatServer
{
  ...
}

package chat.client;

public interface ChatClient
{
  ...
}

package chat.system;

import chat.server.ChatServer;  // selective import
import chat.client.*;          // everything

ChatSystem.java
Example: Overall Directory Structure

<your project path>

```
src
  chat
  bin
  client
  server
  ...
  ...
  chat
  client
  server
```

© Ingolf Krueger, 2010
Important Software Engineering Practice:
Use *Exceptions* instead of error codes
Your Turn!
How to Handle Failures?

• What can go wrong here?
• How do we mark “dangerous” code?
• How do we communicate to client code that it happened?
Solution: Introduce Java Exception

public class DivideByZeroAttempt extends Exception {

    public DivideByZeroAttempt() {
        super();
    }

    public DivideByZeroAttempt(String message) {
        super(message);
    }
}

DivideByZeroAttempt.java
public interface IntDivider {
    public int divide(int x, int y) throws DivideByZeroAttempt;
}

public class IntDividerImpl implements IntDivider {
    public int divide(int x, int y) throws DivideByZeroAttempt {
        if (y == 0)
            throw new DivideByZeroAttempt("tragic");
        return x / y;
    }
}
Client Code is Forewarned and Can Handle Exception

```java
public static void main(String[] args) {
    IntDivider id = new IntDividerImpl();
    try {
        id.divide(3, 2);
        id.divide(3, 0);
        id.divide(2, 3);
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}
```

Java Data Structures: Arrays, Maps, Lists, Sets
Java Arrays

- Containers for fixed number of values of a single (!) type
- Array length is set when the array is created

```java
public static void main(String[] args) {
    int[] intArr = new int[10];

    intArr[0] = 5;
    intArr[4] = 7;
    intArr[9] = 11;

    for(int i = 0; i < intArr.length; i++) {
        intArr[i] = ...  
        ...
    }
}
```
Java Arrays

- Containers for fixed number of values of a single (!) type
- Array length is set when the array is created
- Mathematically, an array is a mapping from a set of (integer) indices to a set of values from the type of the array elements:

\[
\text{intArr} : [0..9] \rightarrow \text{int}
\]

\[
\text{intArr}(0) = 5, \text{intArr}(4) = 7, \text{intArr}(9) = 11
\]

- You can update the mapping via assignments
- Note that the domain of the mapping for array \( a \) is fixed to integers 0 up to and excluding \( a.length \)
Java Arrays

...  
public static void main(String[] args) {

  String[] strArr = new String[10];  
  instantiate

  strArr[4] = "Hello";  
  put value at index

  for(int i = 0; i < intArr.length; i++) {

    ... intArr[i] ...  
    get value at index

  }

}
Java Maps (Generics)

- How to represent mappings from one type (different from int) to another type?
- Example:
  - Store Account object with every username
  - Mapping String -> Account

```java
import java.util.Map;
import java.util.HashMap;

public static void main(String[] args) {
    Map<String, Account> usrRepository = instantiate
        new HashMap<String, Account>();
    ...
}
```
Java Maps (Generics)

... public static void main(String[] args) {

    ...

    index
    usrRepository.put("ikrueger", new Account());
    value

    Account a = usrRepository.get("ikrueger");

    }

}
... public int countUsrAccounts(Map<String, Account> m,
   String username) {

   int cnt = 0;

   for(String s : m.keySet()) iterate over indices
       if(username.equals(s))
           cnt++;

   return cnt;
}
...

Java Maps (Generics) - Iterate

... public int countAccountUsrs(Map<String,Account> m, Account account) {
    int cnt = 0;

    for(Account a : m.values()) {
        if(a.equals(account))
            cnt++;
    }

    return cnt;
}
...

iterate over values
import java.util.Map;  interface for maps
import java.util.HashMap; specify map implementation

public static void main(String[] args) {
    Map<String, Account> usrRepository =
        new HashMap<String, Account>();
    ...
}

Java Lists

- Containers for **arbitrary** number of values of a single type
- List length can change dynamically

```java
import java.util.List;
import java.util.LinkedList;

public static void main(String[] args) {
    List<Account> accounts =
        new LinkedList<Account>();
    ...
}
```
... public static void main(String[] args) {
...

    accounts.add(new Account()); \textit{put value}

    Account a = accounts.get(0); \textit{get value at index}

    a.remove(0); \textit{remove value at index}

}
... public int countDefAccounts(List<Account> accounts) {
    int cnt = 0;

    for (Account a : accounts) { iterate over elements
        if (a.hasDefaulted())
            cnt++;
    }

    return cnt;
}
...
Java Sets

- Containers for **arbitrary** number of values of a single type
- Set size can change dynamically
- Contains no duplicates!

```java
import java.util.Set;
import java.util.HashSet;

public static void main(String[] args) {
    Set<String> users =
        new HashSet<String>();  // instantiate
    ...
}
```
Java Lists (Generics)

... public static void main(String[] args) {
    ...

    users.add(new String("ikrueger"));  // put values
    users.add(new String("ikrueger"));
    users.add(new String("filippo"));

    System.out.println("size " + users.size()); // get # elements

    users.remove(new String("ikrueger")); // remove element

    }
}
public Set<String> appX(Set<String> users) {
    Set<String> result = new HashSet<String>();
    for(String s : users)
        result.add("X"+s);
    return result;
}

...iterate over elements...
Java Collections

- List<E> and Set<E> are examples of Java Collections
- Common methods of all collections:

```java
public interface Collection<E> extends Iterable<E> {
    ...
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);
    boolean remove(Object element);
    void clear();
    ...
}
```
Java Collections – Watch Out!

• For types you want to store in collections you must implement two functions:
  – equals
  – hashCode

• Remember:
  – “==” tests for identity (same object reference)
  – equals should test for equal state (same content)

• Examples:
  – See first project assignment
What have you learned today?
Learning Goals

• Understand the basics of encapsulation and information hiding

• Be able to utilize exceptions as a failure management technique

• Be able to write Java code that has
  – packages,
  – interfaces and
  – exceptions.

• Understand and be able to use Java Maps, Lists and Sets