Implementation

(Mapping to Java)

Jörg Kienzle & Alfred Strohmeier

Overview

- Datatype
- Enumeration
- Class
  - Attribute
  - Association
  - Inheritance
  - Method
    - Visibility
- Collections

Data Type (1)

- A datatype is mapped to a primitive type or to a programmer-defined class
- Use primitive type
  - Java has only a limited set of predefined primitive type
    - The programmer cannot define new primitive (sub)types. For example, there is no way of defining Positive or Natural à la Ada.
  - Approximate with a predefined type, necessarily more permissive
    - In all methods using a parameter of the datatype, perform checks on the parameter
- Use a full-fledged class

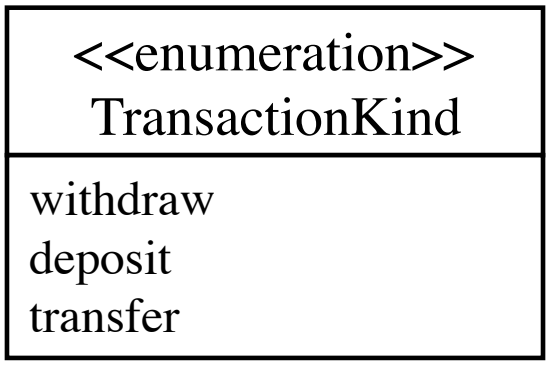
Data Type (2)

- Money represents a positive amount of money in Canadian Dollars
  - It is used to ensure that an amount parameter, e.g. in withdrawCash, is positive

```
public class Money {
  //    Positive amount in dollars
  private final int amount;
  public Money (int dollars) {
     if (dollars < 0 ) {
        throw new IllegalArgumentException
           ("negative amount of money");
     }
     this.amount = dollars;
   }
  public int toInt() {
    return amount;
  }
}
```

Enumeration Type

- An enumeration type can be mapped to a Java enumeration type (only for Java 5.0 or higher)



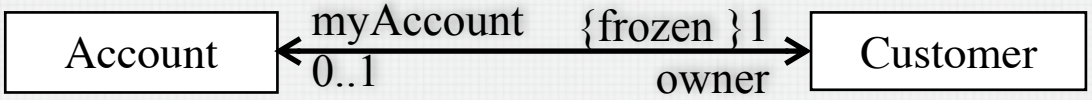

```
enum TransactionKind
    {WITHDRAW, DEPOSIT, TRANSFER}
```

Class and Associations (1)

- A class is mapped to a Java class
  - An abstract class is mapped to an abstract class
  - An attribute is realized by a private field, having the same name as the attribute, and selector and modifier methods, called getters and setters in Java
    - Default visibility for getters and setters is protected. Getters and setters should only be made public if required in the design
  - At least one constructor should be provided with each class, defining values for all fields
- For all associations, we have to decide how we implement them
  - An association end that is not navigable is NOT implemented
  - A navigable association end with single multiplicity, i.e. 0..1 or 1, is realized by a reference field
    - The name of the reference is the role name and its class is the target class of the association end
  - If the multiplicity of the navigable association end is 1, the constructor of the class should enforce that the reference is initialized correctly
  - If an association is bi-directional, the constructor should make sure that the inverse association is initialized as well

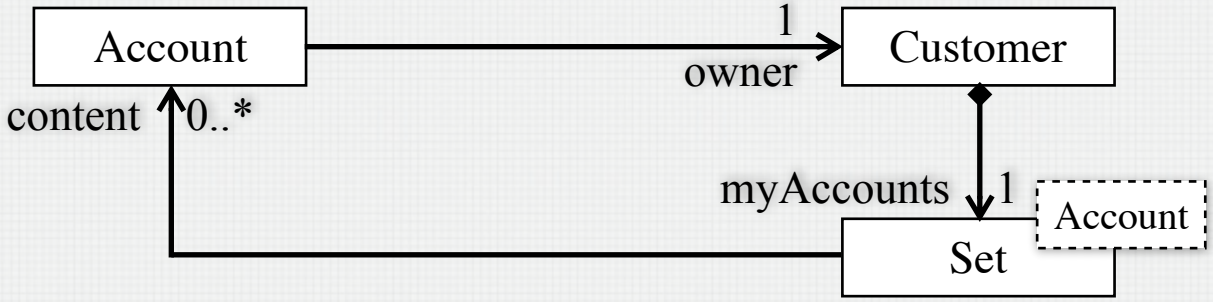
Class and Associations (2a)

```
public class Account {
    private final int number;
    private int balance;
    Customer owner;
    public Account (Customer theOwner) {
        if (theOwner == null) {
            throw new IllegalArgumentException ("no owner");
        }
        theOwner.setMyAccount(this);
        this.number = AccountManager.getUniqueAccountNumber();
        this.balance = 0;
        this.owner = theOwner;
    }
…
```



Class and Associations (2b)

```
public class Account {
    private final int number;
    private int balance;
    Customer owner;
    public Account (Customer theOwner) {
        if (theOwner == null) {
            throw new IllegalArgumentException ("no owner");
        }
        theOwner.addAccount(this);
        this.number = AccountManager.getUniqueAccountNumber();
        this.balance = 0;
        this.owner = theOwner;
    }
```



Class and Associations (3)

```
    public int getNumber() {
        return number;
    }
    protected int getBalance() {
        return balance;
    }
    protected void setBalance(int newBalance) {
        if (newBalance < 0) {
            throw new IllegalArgumentException (“negative balance");
        }
        balance = newBalance;
    }
}
```

Inheritance

- Inheritance (both generalization/specialization and implementation inheritance) is mapped by "extending" a class

```
class Checking extends Account {…}
```

- Multiple inheritance is only available for interface inheritance: a class can implement several interfaces

```
class Checking extends Account implements
      Serializable, Printable, Sortable {…}
```

Method

- A method maps to a Java method
- A method called in the interaction model by objects of a different class becomes a public Java method

```
public class Account {
   public void depositCash(Money amount) { … }
```

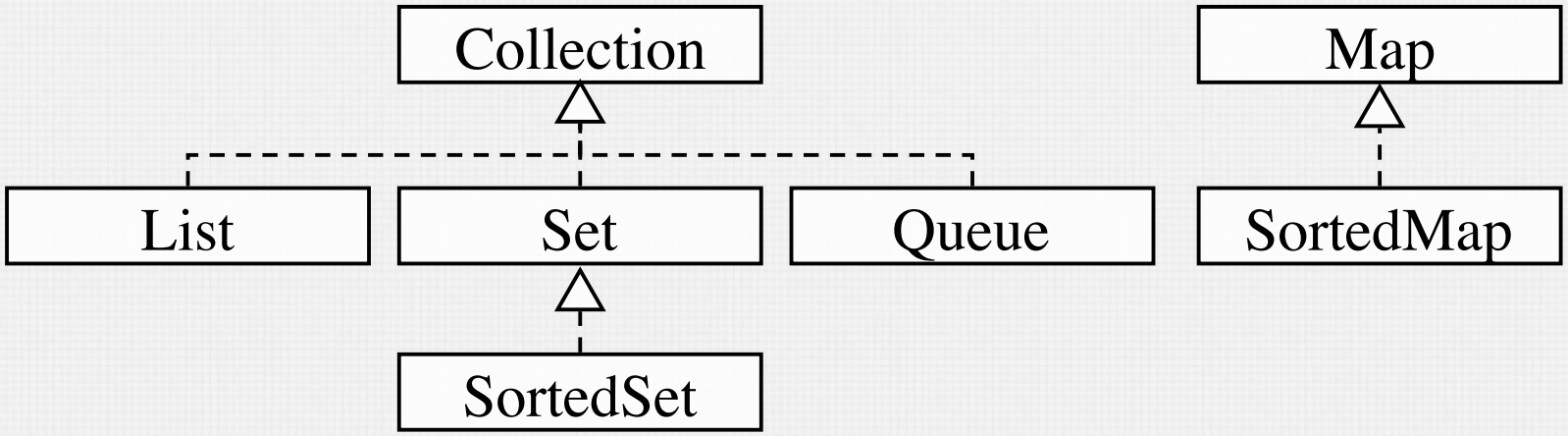
- An internal method becomes a protected Java method

```
public class Account {
   protected void applyInterestRate() { … }
```

Collections

- The package java.util contains an extensive range of collection classes

- Collection Interfaces
  - Collection
    The root interface in the collection hierarchy
  - Iterator
    An iterator over a collection



Java Collections

- Set: A collection that contains no duplicate elements
  - HashSet
  - SortedSet: A set traversed by an iterator in ascending order, for some specified order of the elements
    - TreeSet: Implements SortedSet
- List: An ordered collection, aka a sequence
  - ArrayList: Resizable-array implementation of List
  - LinkedList: Linked list implementation of List
- Map: An object that maps keys to values
  - HashMap
  - SortedMap: A map that is in ascending key order
    - TreeMap: Red-Black tree based implementation of SortedMap.

Collections: Use Generics

- Java 5.0 defines generics. They ensure type safety.

```
import java.util.*; ...
private List<Customer> customerList = new ArrayList<Customer>();
// Add a new customer
customerList.add(customer);
```

- CustomerList can only contain Customers!

```
// Iterate over the ArrayList customerList:
ListIterator<Customer> c = customerList.iterator();
while (c.hasNext()) {
 Customer customer = c.next();
 // no typecast needed
}
```

Questions?