CSE 1020: Unit 12

Topics: Multiclass applications

To do: Chapter 12, No lab

1

Outline

- **Introduction**
- **The inventory**
- **The contacts (suppliers/clients)**
- **The journal**
- **Usage**

2

Outline

- **Introduction**
- The inventory
- The contacts (suppliers/clients)
- The journal
- Usage

3

Introduction

Motivation

- Any single app that we have studied and/or developed has made use of only a small number of classes.
- In real world software engineering, it is common to make use of tens or even hundreds of classes in a single app.

4

Introduction

Motivation

- Any single app that we have studied and/or developed has made use of only a small number of classes.
- In real world software engineering, it is common to make use of tens or even hundreds of classes in a single app.
- We now take a small step in this direction to illustrate matters of concern.
- This exercise also will provide us with an opportunity to review much of what we have learned throughout this course.

5

Introduction

Example

- We will consider simulation of (some of) the operations of a food distribution centre, which is concerned with
  - Buying food from suppliers
  - Selling food to clients
- The operations of such a centre are encapsulated in the class AbstractFoods of type.lib.

6

Introduction

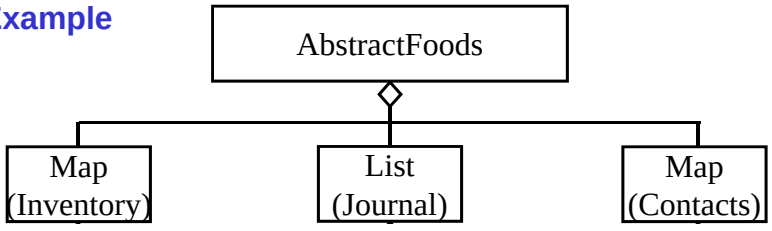
Example

- We will consider simulation of (some of) the operations of a food distribution centre, which is concerned with
  - Buying food from suppliers
  - Selling food to clients
- The operations of such a centre are encapsulated in the class AbstractFoods of type.lib.
- To model the data and operations of a food distribution centre, AbstractFoods has three types of collections.
  1. Inventory: A collection of items in stock.
  2. Contacts: A collection of suppliers and clients.
  3. Journal: A collection of transactions (purchases and sells).

7

Introduction

Example



Introduction

Example

type.lib

Class AbstractFoods

```
java.lang.Object
  └ type.lib.AbstractFoods
```

All Implemented Interfaces:
java.io.Serializable

```
public class AbstractFoods
extends java.lang.Object
implements java.io.Serializable
```

This class encapsulates the inventory and sales business of the Abstract Foods Company. It maintains three collections:

- **Inventory:** A `Map` with the item number as key (a String) and the `Item` object as value.
- **Contacts:** A `Map` with the contact number as key (a String) and the `Contact` object as value.
- **Journal:** A `List` of `Trx` instances each representing a posted sales transaction.

All three collections do not use generics, i.e. their elements are of type `Object`.

Version:
7.0 - Summer 2010
Author:
H. Roumani, roumani@cse.yorku.ca
See Also:
Serialized Form

Introduction

Example

Constructor Summary

AbstractFoods()
Construct a company having empty collections.

AbstractFoods(java.util.Map<String, Item> inventory,
java.util.Map<String, Contact> contacts,
java.util.List<Trx> journal)
Construct a company having the passed collections.

Introduction

Example

Method Summary

java.util.Map<String, Contact> getContacts()
Determine the contacts map of this company
java.util.Map<String, Item> getInventory()
Determine the inventory map of this company
java.util.List<Trx> getJournal()
Determine the transaction list of this company
static AbstractFoods getRandom()
Create a randomly chosen Abstract Foods Company.
java.lang.String toString()
Construct a string representation of this company.

11

Outline

- Introduction
- **The inventory**
- The contacts (suppliers/clients)
- The journal
- Usage

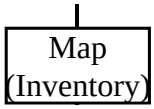
12

The inventory

Basic structure

- Every element of Inventory is a pair: inventory number (code), item.
- For example:

  2004C001, Carrots
- Given this form of the data, a logical mechanism for maintaining Inventory is as a Map.

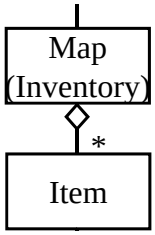


13

The inventory

Basic structure

- Individual items are represented as object instances of the class Item.
- This allows us to associate certain attributes (e.g., inventory number, name, price) and operations with the item.
- We see that Inventory aggregates Item as a collection.



14

The inventory

Class Item

Constructor Summary

Item(java.lang.String number, java.lang.String name, double price)
Construct an inventory item with the given number, name, and sale price per unit.

15

The inventory

Class Item

Constructor Summary

Item(java.lang.String number, java.lang.String name, double price)
Construct an inventory item with the given number, name, and sale price per unit.

Example usage

```
Item flour = new Item("2004F001", "Flour", .45);
Map<String,Item> inventory = new HashMap<String,Item>();
inventory.put(flour.getNumber(), flour);
```

16

The inventory

Class Item

Constructor Summary

Item(java.lang.String number, java.lang.String name, double price)

Construct an inventory item with the given number, name, and sale price per unit.

Example usage

```
Item flour = new Item("2004F001", "Flour", .45);
Map<String,Item> inventory = new HashMap<String,Item>();
inventory.put(flour.getNumber(), flour);
```

Also

```
AbstractFoods myFoods = new AbstractFoods();
Item flour = new Item("2004C001", "Flour", .45);
Map<String,Item> inventory = myFoods.getInventory();
inventory.put(flour.getNumber(), flour);
```

17

The Inventory

Class Item

Method Summary (mutators)

void purchase(int qty, double amount)

Purchase the indicated number of units from this item for the indicated purchase amount.

boolean sell(int qty)

Sell the indicated number of units from this item at the posted sale price.

18

The Inventory

Class Item

Method Summary (mutators)

void purchase(int qty, double amount)
Purchase the indicated number of units from this item for the indicated purchase amount.

boolean sell(int qty)
Sell the indicated number of units from this item at the posted sale price.

Example usage

```
flour.purchase(500, 165.00);
boolean success = flour.sell(100);
```

19

The Inventory

Class Item

Method Summary (accessors, partial listing)

int getPurchasedQty()
Determine the total number of units purchased from this item.

double getPurchases()
Determine the overall purchases amount of this item.

double getSales()
Determine the overall sales amount of this item.

int getSoldQty()
Determine the total number of units sold from this item.

int getStock()
Determine the stock quantity of this item.

double getUnitCost()
Determine the cost per unit of this item.

20

The Inventory

Example usage

```
flour.purchase(500, 165.00);
```

21

The Inventory

Example usage

```
flour.purchase(500, 165.00);
int purchaseQty = flour.getPurchasedQty(); // 500
```

22

The Inventory

Example usage

```
flour.purchase(500, 165.00);
int purchaseQty = flour.getPurchasedQty(); // 500
int inStock = flour.getStock(); // 500
```

23

The Inventory

Example usage

```
flour.purchase(500, 165.00);
int purchaseQty = flour.getPurchasedQty(); // 500
int inStock = flour.getStock(); // 500
double purchPrice = flour.getPurchases(); // 165.00
```

24

The Inventory

Example usage

```
flour.purchase(500, 165.00);
int purchaseQty = flour.getPurchasedQty(); // 500
int inStock = flour.getStock(); // 500
double purchPrice = flour.getPurchases(); // 165.00
double unitCost = flour.getUnitCost(); // 0.33
```

25

The Inventory

Example usage

```
flour.purchase(500, 165.00);
int purchaseQty = flour.getPurchasedQty(); // 500
int inStock = flour.getStock(); // 500
double purchPrice = flour.getPurchases(); // 165.00
double unitCost = flour.getUnitCost(); // 0.33
int soldQty = flour.getSoldQty(); // 0
```

26

The Inventory

Example usage

```
flour.purchase(500, 165.00);
int purchaseQty = flour.getPurchasedQty(); // 500
int inStock = flour.getStock(); // 500
double purchPrice = flour.getPurchases(); // 165.00
double unitCost = flour.getUnitCost(); // 0.33
int soldQty = flour.getSoldQty(); // 0
double sales = flour.getSales(); // 0.0
```

27

The Inventory

Example usage

```
flour.purchase(500, 165.00);
int purchaseQty = flour.getPurchasedQty(); // 500
int inStock = flour.getStock(); // 500
double purchPrice = flour.getPurchases(); // 165.00
double unitCost = flour.getUnitCost(); // 0.33
int soldQty = flour.getSoldQty(); // 0
double sales = flour.getSales(); // 0.0
boolean success = flour.sell(100); // true
```

28

The Inventory

Example usage

```
flour.purchase(500, 165.00);
int purchaseQty = flour.getPurchasedQty(); // 500
int inStock = flour.getStock(); // 500
double purchPrice = flour.getPurchases(); // 165.00
double unitCost = flour.getUnitCost(); // 0.33
int soldQty = flour.getSoldQty(); // 0
double sales = flour.getSales(); // 0.0
boolean success = flour.sell(100); // true
purchaseQty = flour.getPurchasedQty(); // 500
inStock = flour.getStock(); // 400
```

29

The Inventory

Example usage

```
flour.purchase(500, 165.00);
int purchaseQty = flour.getPurchasedQty(); // 500
int inStock = flour.getStock(); // 500
double purchPrice = flour.getPurchases(); // 165.00
double unitCost = flour.getUnitCost(); // 0.33
int soldQty = flour.getSoldQty(); // 0
double sales = flour.getSales(); // 0.0
boolean success = flour.sell(100); // true
purchaseQty = flour.getPurchasedQty(); // 500
inStock = flour.getStock(); // 400
purchPrice = flour.getPurchases(); // 165.00
unitCost = flour.getUnitCost(); // 0.33
```

30

The Inventory

Example usage

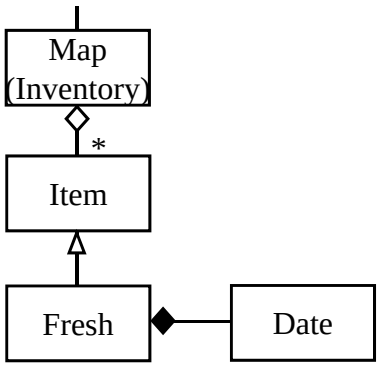
```
flour.purchase(500, 165.00);
int purchaseQty = flour.getPurchasedQty(); // 500
int inStock = flour.getStock(); // 500
double purchPrice = flour.getPurchases(); // 165.00
double unitCost = flour.getUnitCost(); // 0.33
int soldQty = flour.getSoldQty(); // 0
double sales = flour.getSales(); // 0.0
boolean success = flour.sell(100); // true
purchaseQty = flour.getPurchasedQty(); // 500
inStock = flour.getStock(); // 400
purchPrice = flour.getPurchases(); // 165.00
unitCost = flour.getUnitCost(); // 0.33
soldQty = flour.getSoldQty(); // 100
sales = flour.getSales(); // 45.00, based on UnitPrice, not UnitCost
```

31

The inventory

Basic structure

- To accommodate an Item that has an associated expiry date…
- … there is subclass Fresh of Item.
- The class Fresh aggregates Date as a composition.



32

The inventory

Class Fresh

Constructor Summary

Fresh(java.lang.String number, java.lang.String name, double price, java.util.Date expiry)

Construct a fresh inventory item with the given number, name, selling price per unit, and expiry date, and a zero opening stock quantity.

33

The inventory

Class Fresh

Constructor Summary

Fresh(java.lang.String number, java.lang.String name, double price, java.util.Date expiry)

Construct a fresh inventory item with the given number, name, selling price per unit, and expiry date, and a zero opening stock quantity.

Example usage

```
Calendar myCal = Calendar.getInstance();
myCal.add(myCal.DATE, 3);
Fresh carrots = new Fresh("2004C001", "Carrots", .99,
                          myCal.getTime());
```

34

The inventory

Class Fresh

Constructor Summary

Fresh(java.lang.String number, java.lang.String name, double price, java.util.Date expiry)

Construct a fresh inventory item with the given number, name, selling price per unit, and expiry date, and a zero opening stock quantity.

Example usage


```
Calendar myCal = Calendar.getInstance();
myCal.add(myCal.DATE, 3);
Fresh carrots = new Fresh("2004C001", "Carrots", .99,
                          myCal.getTime());
```

35

The Inventory

Class Fresh

Method Summary (partial listing)

java.util.Date getExpiry()

Determine the expiry date of this Item.

36

The Inventory

Class Fresh

Method Summary (partial listing)

java.util.Date getExpiry()
Determine the expiry date of this Item.

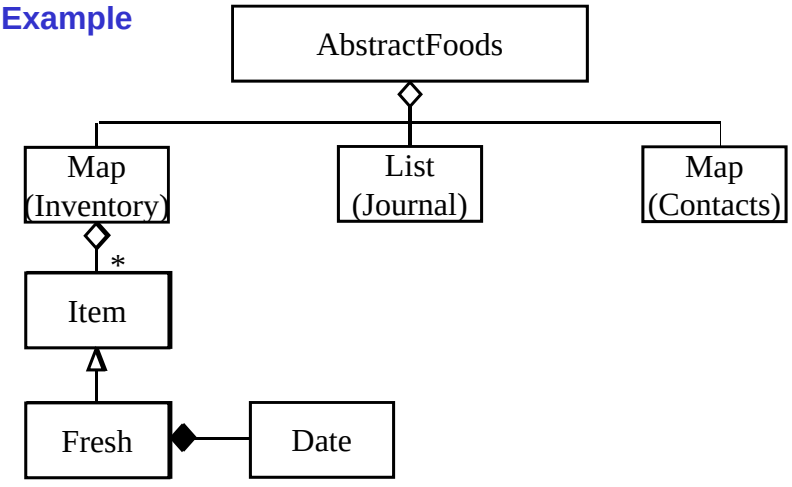
Example

Date carrotsExpiry = carrots.getExpiry();

37

Inventory

Example



38

Outline

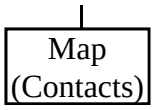
- Introduction
- The inventory
- **The contacts (suppliers/clients)**
- The journal
- Usage

39

The contacts

Basic structure

- An AbstractFoods must maintain a collection of business Contacts.
- Every element of Contacts is a pair: identification number, contact.
- For example:
  0000001, Loblaws
- Given this form of the data, a logical mechanism for maintaining Contacts is as a Map.

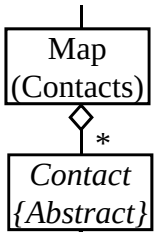


40

The contacts

Basic structure

- Individual contacts are represented via the abstract class Contact.
- This allows us to associate certain attributes (e.g., ID number, name, address) and operations with the contact.
- We see that Contacts aggregates Contact as a collection.



41

The contacts

Class Contact

Method Summary (partial listing)

java.lang.String getAddress()
Determine the address of this contact.

static int getLastContactNumber()
Determine the number of the last-constructed contact.

java.lang.String getName()
Determine the name of this contact.

int getNumber()
Determine the number of this contact.

void setAddress(java.lang.String address)
Change the address of this contact.

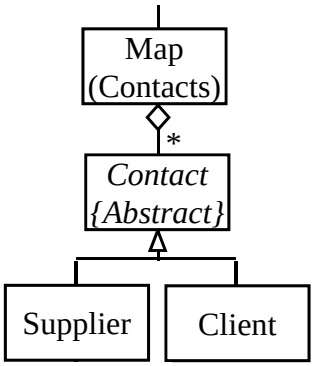
void setName(java.lang.String name)
Change the name of this contact.

42

The contacts

Basic structure

- Every Contact must be either
  1. a Supplier
  2. a Client
- To prevent instantiation of a (generic) Contact, the Contact class is declared to be abstract.
- Instantiation must occur at the level of Supplier or Client, both of which are subclasses of Contact.

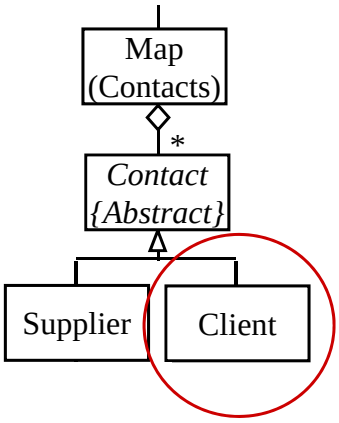


43

The contacts

Basic structure

- Every Contact must be either
  1. a Supplier
  2. a Client
- To prevent instantiation of a (generic) Contact, the Contact class is declared to be abstract.
- Instantiation must occur at the level of Supplier or Client, both of which are subclasses of Contact.



44

The contacts

Class Client

Constructor Summary

Client(java.lang.String name, java.lang.String address,
java.lang.String rating)

Construct a client having the given name, address, and credit rating; and assign a unique number to it.

The contacts

Class Client

Constructor Summary

Client(java.lang.String name, java.lang.String address,
java.lang.String rating)

Construct a client having the given name, address, and credit rating; and assign a unique number to it.

Example usage

```
Client cl = new Client("Loblaws", "3300 Yonge", "A+");
Map<String,Contact> contacts = new HashMap<String,Contact>();
contacts.put("" + cl.getNumber(), cl);
```

The contacts

Class Client

Constructor Summary

Client(java.lang.String name, java.lang.String address,
java.lang.String rating)

Construct a client having the given name, address, and credit rating; and assign a unique number to it.

Example usage

```
Client cl = new Client("Loblaws", "3300 Yonge", "A+");
Map<String,Contact> contacts = new HashMap<String,Contact>();
contacts.put("" + cl.getNumber(), cl);
```

Also

```
AbstractFoods myFoods = new AbstractFoods();
Client cl = new Client("Loblaws", "3300 Yonge", "A+");
Map<String,Contact> contacts = myFoods.getContacts();
contacts.put("" + cl.getNumber(), cl);
```

47

The contacts

Class Client

Method Summary

java.lang.String getRating()
Determine the credit rating of this client.

void setRating(java.lang.String rating)
Change the rating of this client.

48

The contacts

Example usage

```
cl.getName(); // Loblaws
cl.getAddress(); // 3300 Yonge
```

49

The contacts

Example usage

```
cl.getName(); // Loblaws
cl.getAddress(); // 3300 Yonge
cl.setAddress("4400 Yonge");
cl.getAddress(); // 4400 Yonge
```

50

The contacts

Example usage

```
cl.getName(); // Loblaws
cl.getAddress(); // 3300 Yonge
cl.setAddress(“4400 Yonge”);
cl.getAddress(); // 4400 Yonge
cl.getNumber(); // assigned sequentially via Contact
```

51

The contacts

Example usage

```
cl.getName(); // Loblaws
cl.getAddress(); // 3300 Yonge
cl.setAddress(“4400 Yonge”);
cl.getAddress(); // 4400 Yonge
cl.getNumber(); // assigned sequentially via Contact
cl.getRating(); // A+
```

52

The contacts

Example usage

```
cl.getName(); // Loblaws
cl.getAddress(); // 3300 Yonge
cl.setAddress("4400 Yonge");
cl.getAddress(); // 4400 Yonge
cl.getNumber(); // assigned sequentially via Contact
cl.getRating(); // A+
cl.setRating("A");
cl.getRating(); // A
```

The contacts

Example usage

```
cl.getName(); // Loblaws
cl.getAddress(); // 3300 Yonge
cl.setAddress("4400 Yonge");
cl.getAddress(); // 4400 Yonge
cl.getNumber(); // assigned sequentially via Contact
cl.getRating(); // A+
cl.setRating("A");
cl.getRating(); // A
```

Remarks

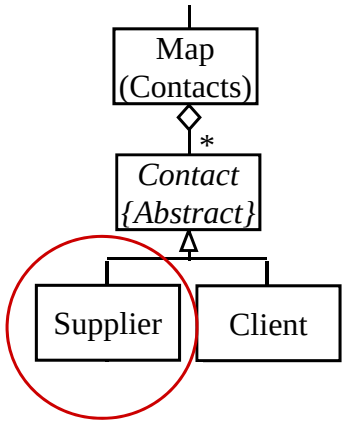
- All methods illustrated are inherited from Contact, except setRating and getRating.
- Indeed, the notion of having a credit rating (attribute) is defining of a Client.

54

The contacts

Basic structure

- Every Contact must be either
  1. a Supplier
  2. a Client
- To prevent instantiation of a (generic) Contact, the Contact class is declared to be abstract.
- Instantiation must occur at the level of Supplier or Client, both of which are subclasses of Contact.

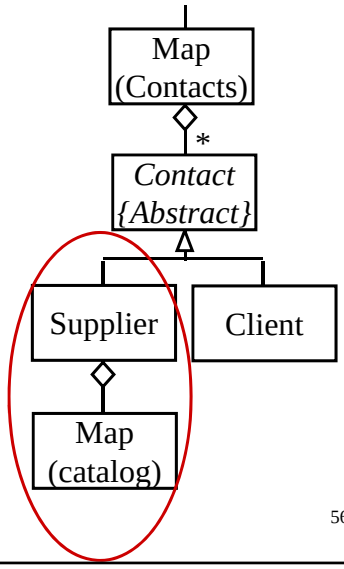


55

The contacts

Basic structure

- Every Contact must be either
  1. a Supplier
  2. a Client
- To prevent instantiation of a (generic) Contact, the Contact class is declared to be abstract.
- Instantiation must occur at the level of Supplier or Client, both of which are subclasses of Contact.



56

The contacts

Class Supplier

Constructor Summary (partial listing)

Supplier(java.lang.String name, java.lang.String address)
Construct a supplier having the indicated name and address, and assign a unique number and an empty catalog of type HashMap to it.

57

The contacts

Class Supplier

Constructor Summary (partial listing)

Supplier(java.lang.String name, java.lang.String address)
Construct a supplier having the indicated name and address, and assign a unique number and an empty catalog of type HashMap to it.

Example usage

```
Supplier sp = new Supplier("RW Farm", "King City");
Map<String,Contact> contacts = new HashMap<String,Contact>();
contacts.put("" + sp.getNumber(), sp);
```

The contacts

Class Supplier

Constructor Summary (partial listing)

Supplier(java.lang.String name, java.lang.String address)
Construct a supplier having the indicated name and address, and assign a unique number and an empty catalog of type HashMap to it.

Example usage

```
Supplier sp = new Supplier("RW Farm", "King City");
Map<String,Contact> contacts = new HashMap<String,Contact>();
contacts.put("" + sp.getNumber(), sp);
```

Also

```
AbstractFoods myFoods = new AbstractFoods();
Supplier sp = new Supplier("RW Farm", "King City");
Map<String,Contact> contacts = myFoods.getContacts();
contacts.put("" + sp.getNumber(), sp);
```

59

The contacts

Class Supplier

Method Summary

java.util.Map getCatalog()
Determine the catalog this supplier.

void setCatalog(java.util.Map catalog)
Set the catalog of this supplier to the passed one.

60

The contacts

Example usage

sp.getName(); // RW Farm
sp.getNumber(); //assigned sequentially via Contact

61

The contacts

Example usage

sp.getName(); // RW Farm
sp.getNumber(); //assigned sequentially via Contact
Map<Item,Double> ctlg = new HashMap<Item,Double>();

62

The contacts

Example usage

```
sp.getName(); // RW Farm
sp.getNumber(); //assigned sequentially via Contact
Map<Item,Double> ctlg = new HashMap<Item,Double>();
ctlg.put(carrots, new Double(0.45));
```

63

The contacts

Example usage

```
sp.getName(); // RW Farm
sp.getNumber(); //assigned sequentially via Contact
Map<Item,Double> ctlg = new HashMap<Item,Double>();
ctlg.put(carrots, new Double(0.45));
sp.setCatalog(ctlg); // this is a reset of default catalog
```

64

The contacts

Example usage

```
sp.getName(); // RW Farm
sp.getNumber(); //assigned sequentially via Contact
Map<Item,Double> ctlg = new HashMap<Item,Double>();
ctlg.put(carrots, new Double(0.45));
sp.setCatalog(ctlg); // this is a reset of default catalog
output.println(sp.getCatalog());
```

65

The contacts

Example usage

```
sp.getName(); // RW Farm
sp.getNumber(); //assigned sequentially via Contact
Map<Item,Double> ctlg = new HashMap<Item,Double>();
ctlg.put(carrots, new Double(0.45));
sp.setCatalog(ctlg); // this is a reset of default catalog
output.println(sp.getCatalog());
```

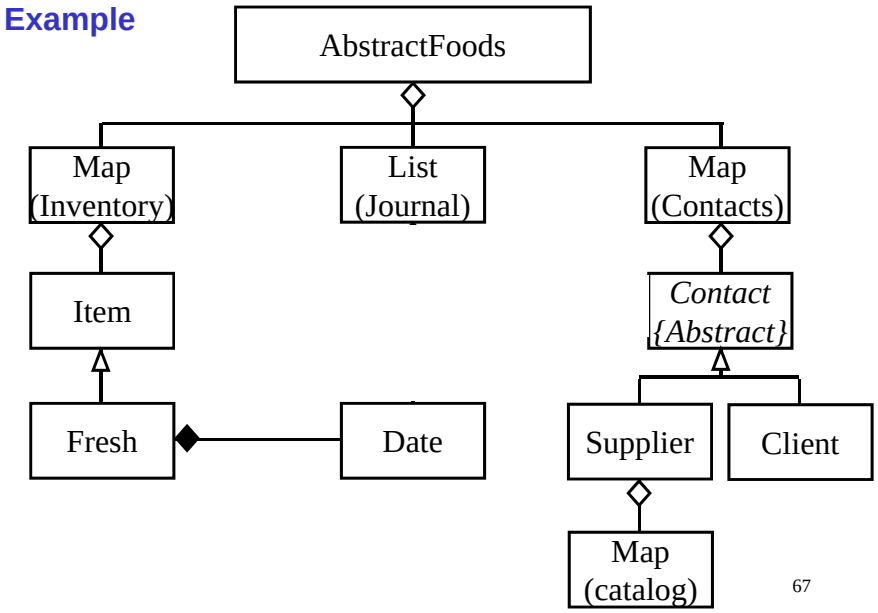
Remarks

- All methods illustrated are inherited from Contact, except setCatalog and getCatalog.
- Indeed, the notion of having a catalog (attribute) of items available is defining of a Supplier.

66

The contacts

Example



67

Outline

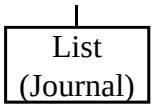
- Introduction
- The inventory
- The contacts (suppliers/clients)
- **The journal**
- Usage

68

The journal

Basic structure

- An AbstractFoods must maintain a collection of business transactions in a Journal.
- Given this form of the data, a logical mechanism for maintaining Journal is as a List.

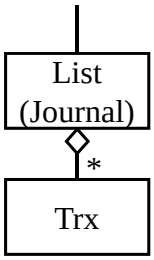


69

The journal

Basic structure

- Individual transactions are represented as object instances of the class Trx.
- This allows us to associate certain attributes (e.g., Date, Item, Contact) and operations with the Trx.

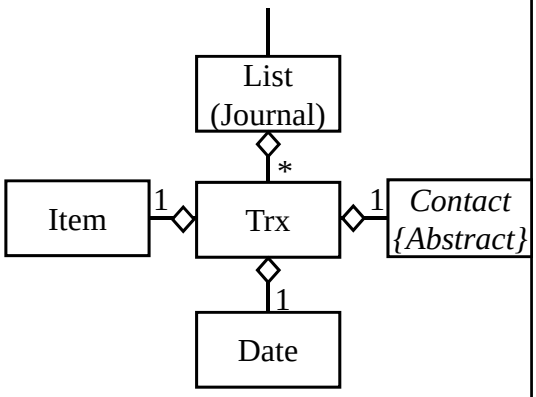


70

The journal

Basic structure

- Individual transactions are represented as object instances of the class Trx.
- This allows us to associate certain attributes (e.g., Date, Item, Contact) and operations with the Trx.
- Indeed, Trx aggregates individual instances of the classes Item, Contact and Date.

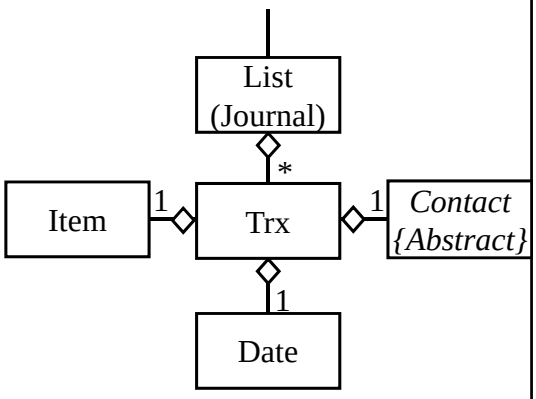


71

The journal

Basic structure

- Individual transactions are represented as object instances of the class Trx.
- This allows us to associate certain attributes (e.g., Date, Item, Contact) and operations with the Trx.
- Indeed, Trx aggregates individual instances of the classes Item, Contact and Date.
- Further, Journal aggregates Trx as a collection.



72

The journal

Class Trx

Constructor Summary

Trx(java.util.Date date, java.lang.String code, java.lang.String ref,
Contact contact, Item item, int qty, double amount)
Construct a transaction having the passed fields.

73

The journal

Class Trx

Constructor Summary

Trx(java.util.Date date, java.lang.String code, java.lang.String ref,
Contact contact, Item item, int qty, double amount)
Construct a transaction having the passed fields.

Example usage

```
Trx t = new Trx(new Date(), “P”, “001”, sp, carrots, 100, 22.00);
List<Trx> journal = new ArrayList<Trx>();
journal.add(t);
```

74

The journal

Class Trx

Constructor Summary

Trx(java.util.Date date, java.lang.String code, java.lang.String ref,
Contact contact, Item item, int qty, double amount)
Construct a transaction having the passed fields.

Example usage

```
Trx t = new Trx(new Date(), "P", "001", sp, carrots, 100, 22.00);
List<Trx> journal = new ArrayList<Trx>();
journal.add(t);
```

Also

```
AbstractFoods myFoods = new AbstractFoods();
Trx t = new Trx(new Date(), "P", "001", sp, carrots, 100, 22.00);
List<Trx> journal = myFoods.getJournal();
journal.add(t);
```

75

The journal

Class Trx

Method Summary (partial listing)

java.lang.String getCode()
Determine the code of this transaction

Contact getContact()
Determine the contact of this transaction

java.util.Date getDate()
Determine the date of this transaction

Item getItem()
Determine the item of this transaction

int getQty()
Determine the quantity of this transaction

java.lang.String getRef()
Determine the reference of this transaction

76

The journal

Example usage

```
output.println(“Summary of transaction ” + t.getRef());
```

77

The journal

Example usage

```
output.println(“Summary of transaction ” + t.getRef());
output.println(“Date: ” + t.getDate());
```

78

The journal

Example usage

```
output.println("Summary of transaction " + t.getRef());
output.println("Date: " + t.getDate());
output.println("Code: " + t.getCode());
```

79

The journal

Example usage

```
output.println("Summary of transaction " + t.getRef());
output.println("Date: " + t.getDate());
output.println("Code: " + t.getCode());
output.println("Contact:" + t.getContact());
```

80

The journal

Example usage

```
output.println("Summary of transaction " + t.getRef());
output.println("Date: " + t.getDate());
output.println("Code: " + t.getCode());
output.println("Contact:" + t.getContact());
output.println("Item:" + t.getItem());
```

81

The journal

Example usage

```
output.println("Summary of transaction " + t.getRef());
output.println("Date: " + t.getDate());
output.println("Code: " + t.getCode());
output.println("Contact:" + t.getContact());
output.println("Item:" + t.getItem());
output.println("Quantity:" + t.getQty());
```

82

The journal

Example usage

```
output.println("Summary of transaction " + t.getRef());
output.println("Date: " + t.getDate());
output.println("Code: " + t.getCode());
output.println("Contact:" + t.getContact());
output.println("Item:" + t.getItem());
output.println("Quantity:" + t.getQty());
output.println("Dollar amount:" + t.getAmount());
```

83

The journal

Example usage

```
output.println("Summary of transaction " + t.getRef());
output.println("Date: " + t.getDate());
output.println("Code: " + t.getCode());
output.println("Contact:" + t.getContact());
output.println("Item:" + t.getItem());
output.println("Quantity:" + t.getQty());
output.println("Dollar amount:" + t.getAmount());
```

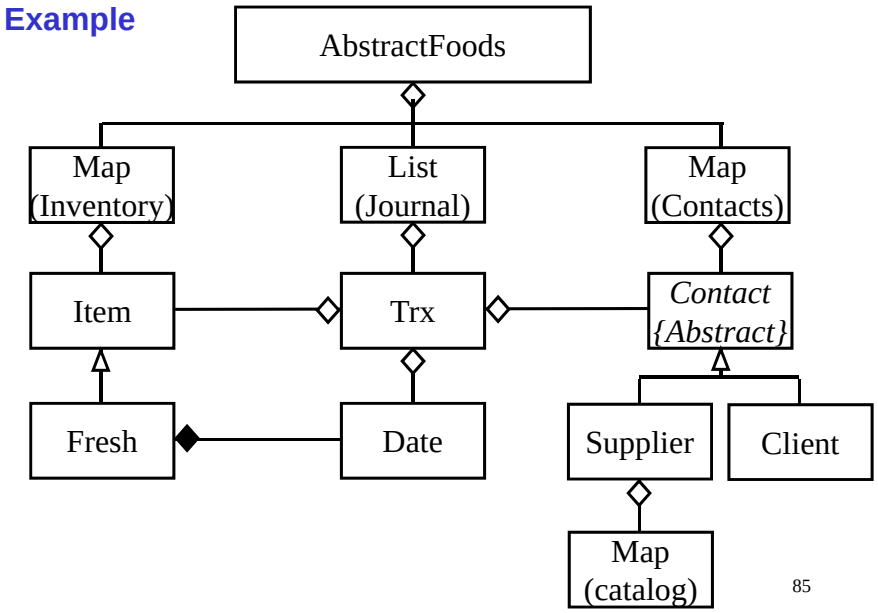
Remark

- Only accessor methods are provided.

84

The journal

Example



85

Outline

- Introduction
- The inventory
- The contacts (suppliers/clients)
- The journal
- **Usage**

86

Usage

Example

- To illustrate usage of AbstractFoods…
- … we will implement an interactive loop that allows users to
  1. purchase items
  2. sell items
  3. report a summary of all transactions

87

Usage

Implementation

```
import type.lang.*;
import type.lib.*;
import java.util.*;
public class FoodSim
{ public static void main(String[ ] args)
  { // DICO
  }
}
```

88

Usage

Implementation

```
// declaration
```

89

Usage

Implementation

```
// declaration
AbstractFoods af = AbstractFoods.getRandom();
Map<String,Item> inventory = af.getInventory();
Map<String,Contact> contacts = af.getContacts();
List<Trx> journal = af.getJournal();
```

90

Usage

Implementation

```
// declaration
AbstractFoods af = AbstractFoods.getRandom();
Map<String,Item> inventory = af.getInventory();
Map<String,Contact> contacts = af.getContacts();
List<Trx> journal = af.getJournal();
Contact contact = null;
```

91

Usage

Implementation

```
// declaration
AbstractFoods af = AbstractFoods.getRandom();
Map<String,Item> inventory = af.getInventory();
Map<String,Contact> contacts = af.getContacts();
List<Trx> journal = af.getJournal();
Contact contact = null;
Item item = null;
```

92

Usage

Implementation

```
// declaration
AbstractFoods af = AbstractFoods.getRandom();
Map<String,Item> inventory = af.getInventory();
Map<String,Contact> contacts = af.getContacts();
List<Trx> journal = af.getJournal();
Contact contact = null;
Item item = null;
int qty = 0;
```

93

Usage

Implementation

```
// declaration
AbstractFoods af = AbstractFoods.getRandom();
Map<String,Item> inventory = af.getInventory();
Map<String,Contact> contacts = af.getContacts();
List<Trx> journal = af.getJournal();
Contact contact = null;
Item item = null;
int qty = 0;
double cost = 0.0;
```

94

Usage

Implementation

```
// declaration
AbstractFoods af = AbstractFoods.getRandom();
Map<String,Item> inventory = af.getInventory();
Map<String,Contact> contacts = af.getContacts();
List<Trx> journal = af.getJournal();
Contact contact = null;
Item item = null;
int qty = 0;
double cost = 0.0;
int refNum = 0;
```

95

Usage

Implementation

```
// input and computation
while (true)
{




}
```

96

Usage

Implementation

```
// input and computation
while (true)
{ output.print("Enter a transaction code P(urchase), S(ell), E(nd): ");
  String code = input.nextLine();
  if (code.equals("E")) break; // this is the way out of the loop




}
```

97

Usage

Implementation

```
// input and computation
while (true)
{ output.print("Enter a transaction code P(urchase), S(ell), E(nd): ");
  String code = input.nextLine();
  if (code.equals("E")) break; // this is the way out of the loop
  if (code.equals("P")
  { // process a purchase
  }
  else
  { // process a sale
  }

}
```

98

Usage

Implementation

```
// input and computation
while (true)
{ output.print(“Enter a transaction code P(urchase), S(ell), E(nd): ”);
  String code = input.nextLine();
  if (code.equals(“E”)) break; // this is the way out of the loop
  if (code.equals(“P”)
  { // process a purchase
  }
  else
  { // process a sale
  }
  // update the transaction list
}
```

99

Usage

Implementation

```
// process a purchase
output.print("Item number to purchase: ");
```

Usage

Implementation

```
// process a purchase
output.print("Item number to purchase: ");
String itemNum = input.nextLine();
```

Usage

Implementation

```
// process a purchase
output.print("Item number to purchase: ");
String itemNum = input.nextLine();
SE.require(inventory.containsKey(itemNum), "No such item");
```

Usage

Catching an exception thrown by SE.require

SE

Method detail

require

public static void require(boolean condition, java.lang.String msg)

Assert that the specified condition is met or else terminate. Test the passed condition and if false then terminate the program with a Precondition Exception, the passed string, and a stack trace. Otherwise, program execution proceeds normally.

Parameters:

condition - the assertion condition to examined.

msg - the message to print upon termination. If this parameter is null, it is ignored; i.e. not displayed.

Throws:

SEpreconditionException

103

Usage

Implementation

```
// process a purchase
output.print("Item number to purchase: ");
String itemNum = input.nextLine();
SE.require(inventory.containsKey(itemNum), "No such item");
item = inventory.get(itemNum);
```

Usage

Implementation

```
// process a purchase
output.print("Item number to purchase: ");
String itemNum = input.nextLine();
SE.require(inventory.containsKey(itemNum), "No such item");
item = inventory.get(itemNum);
output.print("Supplier number: ");
```

Usage

Implementation

```
// process a purchase
output.print("Item number to purchase: ");
String itemNum = input.nextLine();
SE.require(inventory.containsKey(itemNum), "No such item");
item = inventory.get(itemNum);
output.print("Supplier number: ");
String con = input.nextLine();
```

Usage

Implementation

```
// process a purchase
output.print("Item number to purchase: ");
String itemNum = input.nextLine();
SE.require(inventory.containsKey(itemNum), "No such item");
item = inventory.get(itemNum);
output.print("Supplier number: ");
String con = input.nextLine();
SE.require(contacts.containsKey(con) &&
```

Usage

Implementation

```
// process a purchase
output.print("Item number to purchase: ");
String itemNum = input.nextLine();
SE.require(inventory.containsKey(itemNum), "No such item");
item = inventory.get(itemNum);
output.print("Supplier number: ");
String con = input.nextLine();
SE.require(contacts.containsKey(con) &&
                (contacts.get(con) instanceof Supplier), "No supplier");
```

Usage

Implementation

```
// process a purchase
output.print("Item number to purchase: ");
String itemNum = input.nextLine();
SE.require(inventory.containsKey(itemNum), "No such item");
item = inventory.get(itemNum);
output.print("Supplier number: ");
String con = input.nextLine();
SE.require(contacts.containsKey(con) &&
                (contacts.get(con) instanceof Supplier), "No supplier");
Supplier s = (Supplier) contacts.get(con);
```

Usage

Implementation

```
// process a purchase
output.print("Item number to purchase: ");
String itemNum = input.nextLine();
SE.require(inventory.containsKey(itemNum), "No such item");
item = inventory.get(itemNum);
output.print("Supplier number: ");
String con = input.nextLine();
SE.require(contacts.containsKey(con) &&
                (contacts.get(con) instanceof Supplier), "No supplier");
Supplier s = (Supplier) contacts.get(con);
contact = s; // needed to update transactions
```

Usage

Implementation

```
// process a purchase
output.print("Item number to purchase: ");
String itemNum = input.nextLine();
SE.require(inventory.containsKey(itemNum), "No such item");
item = inventory.get(itemNum);
output.print("Supplier number: ");
String con = input.nextLine();
SE.require(contacts.containsKey(con) &&
           (contacts.get(con) instanceof Supplier), "No supplier");
Supplier s = (Supplier) contacts.get(con);
contact = s; // needed to update transactions
Map<Item,Double> cat = s.getCatalog();
```

Usage

Implementation

```
// process a purchase
output.print("Item number to purchase: ");
String itemNum = input.nextLine();
SE.require(inventory.containsKey(itemNum), "No such item");
item = inventory.get(itemNum);
output.print("Supplier number: ");
String con = input.nextLine();
SE.require(contacts.containsKey(con) &&
           (contacts.get(con) instanceof Supplier), "No supplier");
Supplier s = (Supplier) contacts.get(con);
contact = s; // needed to update transactions
Map<Item,Double> cat = s.getCatalog();
SE.require(cat.containsKey(item), "Supplier has no such item");
```

Usage

Implementation

```
// process a purchase
output.print("Item number to purchase: ");
String itemNum = input.nextLine();
SE.require(inventory.containsKey(itemNum), "No such item");
item = inventory.get(itemNum);
output.print("Supplier number: ");
String con = input.nextLine();
SE.require(contacts.containsKey(con) &&
           (contacts.get(con) instanceof Supplier), "No supplier");
Supplier s = (Supplier) contacts.get(con);
contact = s; // needed to update transactions
Map<Item,Double> cat = s.getCatalog();
SE.require(cat.containsKey(item), "Supplier has no such item");
output.print("Quantity: ");
```

Usage

Implementation

```
// process a purchase
output.print("Item number to purchase: ");
String itemNum = input.nextLine();
SE.require(inventory.containsKey(itemNum), "No such item");
item = inventory.get(itemNum);
output.print("Supplier number: ");
String con = input.nextLine();
SE.require(contacts.containsKey(con) &&
           (contacts.get(con) instanceof Supplier), "No supplier");
Supplier s = (Supplier) contacts.get(con);
contact = s; // needed to update transactions
Map<Item,Double> cat = s.getCatalog();
SE.require(cat.containsKey(item), "Supplier has no such item");
output.print("Quantity: ");
qty = input.nextInt();
```

Usage

Implementation

```
// process a purchase
output.print("Item number to purchase: ");
String itemNum = input.nextLine();
SE.require(inventory.containsKey(itemNum), "No such item");
item = inventory.get(itemNum);
output.print("Supplier number: ");
String con = input.nextLine();
SE.require(contacts.containsKey(con) &&
           (contacts.get(con) instanceof Supplier), "No supplier");
Supplier s = (Supplier) contacts.get(con);
contact = s; // needed to update transactions
Map<Item,Double> cat = s.getCatalog();
SE.require(cat.containsKey(item), "Supplier has no such item");
output.print("Quantity: ");
qty = input.nextInt();
output.print("Total cost: ");
```

Usage

Implementation

```
// process a purchase
output.print("Item number to purchase: ");
String itemNum = input.nextLine();
SE.require(inventory.containsKey(itemNum), "No such item");
item = inventory.get(itemNum);
output.print("Supplier number: ");
String con = input.nextLine();
SE.require(contacts.containsKey(con) &&
           (contacts.get(con) instanceof Supplier), "No supplier");
Supplier s = (Supplier) contacts.get(con);
contact = s; // needed to update transactions
Map<Item,Double> cat = s.getCatalog();
SE.require(cat.containsKey(item), "Supplier has no such item");
output.print("Quantity: ");
qty = input.nextInt();
output.print("Total cost: ");
cost = input.nextDouble();
```

Usage

Implementation

```
// process a purchase
output.print("Item number to purchase: ");
String itemNum = input.nextLine();
SE.require(inventory.containsKey(itemNum), "No such item");
item = inventory.get(itemNum);
output.print("Supplier number: ");
String con = input.nextLine();
SE.require(contacts.containsKey(con) &&
           (contacts.get(con) instanceof Supplier), "No supplier");
Supplier s = (Supplier) contacts.get(con);
contact = s; // needed to update transactions
Map<Item,Double> cat = s.getCatalog();
SE.require(cat.containsKey(item), "Supplier has no such item");
output.print("Quantity: ");
qty = input.nextInt();
output.print("Total cost: ");
cost = input.nextDouble();
item.purchase(qty, cost);
```

117

Usage

Implementation

```
// process a sale
```

118

Usage

Implementation

```
// process a sale
output.print("Item number to sell: ");
String itemNum = input.nextLine();
```

119

Usage

Implementation

```
// process a sale
output.print("Item number to sell: ");
String itemNum = input.nextLine();
SE.require(inventory.containsKey(itemNum), "No such item");
```

120

Usage

Implementation

```
// process a sale
output.print("Item number to sell: ");
String itemNum = input.nextLine();
SE.require(inventory.containsKey(itemNum), "No such item");
item = inventory.get(itemNum);
```

121

Usage

Implementation

```
// process a sale
output.print("Item number to sell: ");
String itemNum = input.nextLine();
SE.require(inventory.containsKey(itemNum), "No such item");
item = inventory.get(itemNum);
output.print("Client number: ");
String con = input.nextLine();
```

122

Usage

Implementation

```
// process a sale
output.print("Item number to sell: ");
String itemNum = input.nextLine();
SE.require(inventory.containsKey(itemNum), "No such item");
item = inventory.get(itemNum);
output.print("Client number: ");
String con = input.nextLine();
SE.require(contacts.containsKey(con) &&
                    (contacts.get(con) instanceof Client), "No client");
```

123

Usage

Implementation

```
// process a sale
output.print("Item number to sell: ");
String itemNum = input.nextLine();
SE.require(inventory.containsKey(itemNum), "No such item");
item = inventory.get(itemNum);
output.print("Client number: ");
String con = input.nextLine();
SE.require(contacts.containsKey(con) &&
                    (contacts.get(con) instanceof Client), "No client");
contact = contacts.get(con);
```

124

Usage

Implementation

```
// process a sale
output.print("Item number to sell: ");
String itemNum = input.nextLine();
SE.require(inventory.containsKey(itemNum), "No such item");
item = inventory.get(itemNum);
output.print("Client number: ");
String con = input.nextLine();
SE.require(contacts.containsKey(con) &&
                    (contacts.get(con) instanceof Client), "No client");
contact = contacts.get(con);
output.print("Quantity: ");
qty = input.nextInt();
```

125

Usage

Implementation

```
// process a sale
output.print("Item number to sell: ");
String itemNum = input.nextLine();
SE.require(inventory.containsKey(itemNum), "No such item");
item = inventory.get(itemNum);
output.print("Client number: ");
String con = input.nextLine();
SE.require(contacts.containsKey(con) &&
                    (contacts.get(con) instanceof Client), "No client");
contact = contacts.get(con);
output.print("Quantity: ");
qty = input.nextInt();
boolean okay = item.sell(qty);
SE.require(okay, "Not enough in stock");
```

126

Usage

Implementation

```
// update the transaction list
```

127

Usage

Implementation

```
// update the transaction list
refNum++;
Trx t = new Trx(new Date(), code, "" + refNum, contact, item, qty, cost);
```

128

Usage

Implementation

```
// update the transaction list
refNum++;
Trx t = new Trx(new Date(), code, "" + refNum, contact, item, qty, cost);
journal.add(t);
```

129

Usage

Implementation

```
// output
```

130

Usage

Implementation

```
// output
output.print("Would you like a summary of transactions (Y/N): ");
String resp = input.nextLine();
```


Usage

Implementation

```
// output
output.print("Would you like a summary of transactions (Y/N): ");
String resp = input.nextLine();
if (resp.equals("Y"))
{    // report transactions









}
```

Usage

Implementation

```
// output
output.print("Would you like a summary of transactions (Y/N): ");
String resp = input.nextLine();
if (resp.equals("Y"))
{   // report transactions
    Iterator<Trx> it = journal.iterator();
    while (it.hasNext())
    {




    }
}
```

133

Usage

Implementation

```
// output
output.print("Would you like a summary of transactions (Y/N): ");
String resp = input.nextLine();
if (resp.equals("Y"))
{   // report transactions
    Iterator<Trx> it = journal.iterator();
    while (it.hasNext())
    {       Trx t = it.next();




    }
}
```

134

Usage

Implementation

```
// output
output.print("Would you like a summary of transactions (Y/N): ");
String resp = input.nextLine();
if (resp.equals("Y"))
{   // report transactions
    Iterator<Trx> it = journal.iterator();
    while (it.hasNext())
    {       Trx t = it.next();
            output.println("\nSummary of transaction " + t.getRef());




    }
}
```

135

Usage

Implementation

```
// output
output.print("Would you like a summary of transactions (Y/N): ");
String resp = input.nextLine();
if (resp.equals("Y"))
{   // report transactions
    Iterator<Trx> it = journal.iterator();
    while (it.hasNext())
    {       Trx t = it.next();
            output.println("\nSummary of transaction " + t.getRef());
            output.println("Date: " + t.getDate());




    }
}
```

136

Usage

Implementation

```
// output
output.print("Would you like a summary of transactions (Y/N): ");
String resp = input.nextLine();
if (resp.equals("Y"))
{   // report transactions
    Iterator<Trx> it = journal.iterator();
    while (it.hasNext())
    {       Trx t = it.next();
            output.println("\nSummary of transaction " + t.getRef());
            output.println("Date: " + t.getDate());
            output.println("Code: " + t.getCode());




    }
}
```


Usage

Implementation

```
// output
output.print("Would you like a summary of transactions (Y/N): ");
String resp = input.nextLine();
if (resp.equals("Y"))
{   // report transactions
    Iterator<Trx> it = journal.iterator();
    while (it.hasNext())
    {       Trx t = it.next();
            output.println("\nSummary of transaction " + t.getRef());
            output.println("Date: " + t.getDate());
            output.println("Code: " + t.getCode());
            output.println("Contact: " + t.getContact());




    }
}
```

Usage

Implementation

```
// output
output.print("Would you like a summary of transactions (Y/N): ");
String resp = input.nextLine();
if (resp.equals("Y"))
{   // report transactions
    Iterator<Trx> it = journal.iterator();
    while (it.hasNext())
    {       Trx t = it.next();
            output.println("\nSummary of transaction " + t.getRef());
            output.println("Date: " + t.getDate());
            output.println("Code: " + t.getCode());
            output.println("Contact: " + t.getContact());
            output.println("Item: " + t.getItem());


    }
}
```

139

Usage

Implementation

```
// output
output.print("Would you like a summary of transactions (Y/N): ");
String resp = input.nextLine();
if (resp.equals("Y"))
{   // report transactions
    Iterator<Trx> it = journal.iterator();
    while (it.hasNext())
    {       Trx t = it.next();
            output.println("\nSummary of transaction " + t.getRef());
            output.println("Date: " + t.getDate());
            output.println("Code: " + t.getCode());
            output.println("Contact: " + t.getContact());
            output.println("Item: " + t.getItem());
            output.println("Quantity: " + t.getQty());

    }
}
```

140

Usage

Implementation

```
// output
output.print("Would you like a summary of transactions (Y/N): ");
String resp = input.nextLine();
if (resp.equals("Y"))
{   // report transactions
    Iterator<Trx> it = journal.iterator();
    while (it.hasNext())
    {       Trx t = it.next();
            output.println("\nSummary of transaction " + t.getRef());
            output.println("Date: " + t.getDate());
            output.println("Code: " + t.getCode());
            output.println("Contact: " + t.getContact());
            output.println("Item: " + t.getItem());
            output.println("Quantity: " + t.getQty());
            output.println("Dollar amount: " + t.getAmount());
    }
}
```

141

Usage

Implementation: An improvement via exception handling

```
import type.lang.*;
import type.lib.*;
import java.util.*;
public class FoodSim
{ public static void main(String[ ] args)
  { // DICO
  }
}
```

142

Usage

Implementation

```
// input and computation
while (true)
{ output.print("Enter a transaction code P(urchase), S(ell), E(nd): ");
  String code = input.nextLine();
  if (code.equals("E")) break; // this is the way out of the loop
  if (code.equals("P")
  { // process a purchase
  }
  else
  { // process a sale
  }
  // update the transaction list
}
```

143

Usage

Implementation

```
// input and computation
while (true)
{ output.print("Enter a transaction code P(urchase), S(ell), E(nd): ");
  String code = input.nextLine();
  if (code.equals("E")) break; // this is the way out of the loop
  try
  {
    if (code.equals("P")
    { // process a purchase
    }
    else
    { // process a sale
    }
    // update the transaction list
  }
}
```

144

Usage

Implementation

```
// input and computation
while (true)
{ output.print("Enter a transaction code P(urchase), S(ell), E(nd): ");
  String code = input.nextLine();
  if (code.equals("E")) break; // this is the way out of the loop
  try
  {
    if (code.equals("P")
    { // process a purchase
    }
    else
    { // process a sale
    }
    // update the transaction list
  }
  catch (SEpreconditionException e)
  {  outprint.println(e.getMessage());
  }
}
```

145

Usage

Implementation

```
// input and computation
while (true)
{ output.print("Enter a transaction code P(urchase), S(ell), E(nd): ");
  String code = input.nextLine();
  if (code.equals("E")) break; // this is the way out of the loop
  try
  {
    if (code.equals("P")
    { // process a purchase
    }
    else
    { // process a sale
    }
    // update the transaction list
  }
  catch (SEpreconditionException e)
  {  outprint.println(e.getMessage());
  }
}
```

146

Usage

Test

- Try this one out on your own.
- For ease of experimentation:
  - You might find it desirable to explicitly populate the contacts and inventory on your own,
  - i.e., as opposed to relying on the getRandom method of AbstractFoods.

147

Summary

- **Introduction**
- **The inventory**
- **The contacts (suppliers/clients)**
- **The journal**
- **Usage**

148

Outline

- Introduction
- The inventory
- The contacts (suppliers/clients)
- The journal
- Usage
- Appendix: Scratching the surface

149

Appendix: Scratching the surface

- Creating your own classes
- Attributes
- Methods
- Constructors
- Usage

150

Appendix: Scratching the surface

- **Creating your own classes**
- **Attributes**
- **Methods**
- **Constructors**
- **Usage**

151

Creating your own classes

Introduction

- We know that classes provide us with a way to encapsulate the data and operations associated with an object.
- Now, as a final point of discussion, we scratch the surface to see what is going on behind the scene by creating our own class.
- In correspondence with classes encapsulating data and operations for their objects, we begin design by asking two fundamental questions.
  1. What sort of data will we maintain
  2. What sort of operations will we perform.
- Clearly, these questions will only have answers once we specify what type of objects we are modeling.
- Example: Let's model people!

152

Creating your own classes

Person: What sort of data will we maintain

- We will consider attributes of an individual person to be their name and age.
- We will consider attributes of all people to be their minimum age (i.e., 0) and their maximum age (e.g., 110).

Person: What sort of operations will we perform

- We will want to be able to construct a new person.
- We will want to be able to assign names and ages.
- We will want to get names and ages.
- We will want to be able to increment ages.

153

Creating your own classes

Class template

```
// Definition of the ClassName class.
public class ClassName
{
// constructors

// instance methods

// class (static) methods

// instance attributes (fields)

// class (static) constant attributes (fields)

// class (static) attributes (fields)

}
```

154

Creating your own classes

Class template

```
// Definition of the ClassName class.
public class ClassName
{
// constructors

// instance methods

// class (static) methods

// instance attributes

// class (static) constant attributes

// class (static) attributes

}
```

Remark: These are the elements of a class definition

155

Creating your own classes

Class template

```
// Definition of the Person class.
public class Person
{
// constructors

// instance methods

// class (static) methods

// instance attributes

// class (static) constant attributes

// class (static) attributes

}
```

156

Appendix: Outline

- Creating your own classes
- **Attributes**
- Methods
- Constructors
- Usage

157

Attributes

General

- Attributes pertain to information about the objects and the class.
- Various other terms are commonly used, including,
  - Variables
  - Properties
  - Data members
  - Fields

158

Attributes

General

- Attributes pertain to information about the objects and the class.
- Various other terms are commonly used, including,
  - Variables
  - Properties
  - Data members
  - Fields
- There are two different varieties
  1. Instance attributes: Associated with instances of a class (e.g., name and age)
  2. Class (static) attributes: Associated with the class as a whole (e.g., max and min allowable ages)

159

Attributes

Attributes: Person implementation

```
// instance attributes
private String name;
private int age;

// class (static) constant attributes
public static final int MIN_AGE =0;
private static final String DEFAULT_NAME = “UNKNOWN”;

// class (static) attributes
public static int maxAge = 110;
```

160

Attributes

Attributes: Person implementation

```
// instance attributes
private String name;
private int age;
```

Remark 1: An attribute is a data variable; so, it must have a type.

```
// class (static) constant attributes
public static final int MIN_AGE =0;
private static final String DEFAULT_NAME = “UNKNOWN”;

// class (static) attributes
public static int maxAge = 110;
```

161

Attributes

Attributes: Person implementation

```
// instance attributes
private String name;
private int age;
```

Remark 2: An attribute can be public or private.

```
// class (static) constant attributes
public static final int MIN_AGE =0;
private static final String DEFAULT_NAME = “UNKNOWN”;

// class (static) attributes
public static int maxAge = 110;
```

162

Attributes

Attributes: Person implementation

```
// instance attributes
private String name;
private int age;
```

Remark 3: If an attribute is public, then users can access and change its value directly.

```
// class (static) constant attributes
public static final int MIN_AGE =0;
private static final String DEFAULT_NAME = “UNKNOWN”;

// class (static) attributes
public static int maxAge = 110;
```

163

Attributes

Attributes: Person implementation

```
// instance attributes
private String name;
private int age;
```

Remark 4: If an attribute is private, then users can only access it indirectly through methods.

```
// class (static) constant attributes
public static final int MIN_AGE =0;
private static final String DEFAULT_NAME = “UNKNOWN”;

// class (static) attributes
public static int maxAge = 110;
```

164

Attributes

Attributes: Person implementation

```
// instance attributes
private String name;
private int age;
```

Remark 5: If an attribute is private, then methods in the class (as opposed to outside users) can access it directly.

```
// class (static) constant attributes
public static final int MIN_AGE =0;
private static final String DEFAULT_NAME = “UNKNOWN”;

// class (static) attributes
public static int maxAge = 110;
```

165

Attributes

Attributes: Person implementation

```
// instance attributes
private String name;
private int age;
```

Remark 6: Typically, instance attributes are kept private to safeguard against corruption.

```
// class (static) constant attributes
public static final int MIN_AGE =0;
private static final String DEFAULT_NAME = “UNKNOWN”;

// class (static) attributes
public static int maxAge = 110;
```

166

Attributes

Attributes: Person implementation

```
// instance attributes
private String name;
private int age;
```

Remark 7: The keyword static is used for class attributes; no special designation for instance attributes.

```
// class (static) constant attributes
public static final int MIN_AGE =0;
private static final String DEFAULT_NAME = “UNKNOWN”;

// class (static) attributes
public static int maxAge = 110;
```

167

Attributes

Attributes: Person implementation

```
// instance attributes
private String name;
private int age;
```

Remark 8: It is generally good practice to keep class (static) attributes constant (final).

```
// class (static) constant attributes
public static final int MIN_AGE =0;
private static final String DEFAULT_NAME = “UNKNOWN”;

// class (static) attributes
public static int maxAge = 110;
```

168

Appendix: Outline

- Creating your own classes
- Attributes
- **Methods**
- Constructors
- Usage

169

Methods

General

- Methods pertain to operations regarding objects or the class.
- Other common terms include
  - message
  - member function
  - sub

Methods

General

- Methods pertain to operations regarding objects or the class.
- Other common terms include
  - message
  - member function
  - sub
- There are two different varieties
  1. Instance methods: Associated with instances of a class (e.g., setting and getting names and ages)
  2. Class (static) methods: Associated with the class as a whole (e.g., general operation for all Persons); do not alter individual instances of the class (e.g., a particular Person object)

171

Methods

Person implementation

```
// instance methods

public void setName(String n)
{
   name = n;
}
```

A method definition specifies

- The method name
- Names and types of parameters
- The type of return result (void, if none).
- Its visibility (public or private)
- Whether it's an instance or class (static) method.
- The steps required to execute it (the body).

172

Methods

Person implementation

```
// instance methods

public void setAge(int a)
{
    age = a;
}
```

A method definition specifies

- The method name
- Names and types of parameters
- The type of return result (void, if none)
- Its visibility (public or private)
- Whether it's an instance or class (static) method.
- The steps required to execute it (the body).

173

Methods

Person implementation

```
// instance methods

public void incrementAge()
{
    age++;
}
```

A method definition specifies

- The method name
- Names and types of parameters
- The type of return result (void, if none)
- Its visibility (public or private)
- Whether it's an instance or class (static) method.
- The steps required to execute it (the body).

174

Methods

Person implementation

```
// instance methods

public String getName()
{
    return name;
}
```

A method definition specifies

- The method name
- Names and types of parameters
- The type of return result (void, if none)
- Its visibility (public or private)
- Whether it's an instance or class (static) method.
- The steps required to execute it (the body).

175

Methods

Person implementation

```
// instance methods

public int getAge()
{
    return age;
}
```

A method definition specifies

- The method name
- Names and types of parameters
- The type of return result (void, if none)
- Its visibility (public or private)
- Whether it's an instance or class (static) method.
- The steps required to execute it (the body).

176

Methods

Person implementation

```
// class (static) methods

public static void philosophy()
{
output.println("mu");
}
```

A method definition specifies

- The method name
- Names and types of parameters
- The type of return result (void, if none)
- Its visibility (public or private)
- Whether it's an instance or class (static) method.
- The steps required to execute it (the body).

177

Methods

Parameters

- When we call a method, we often want to pass it some data.
- The data can then be used inside the method.
- We do this by having the method take parameters.
- Parameters are declared in the header of the method definition by specifying both their type and name.
- For example, a is the parameter in

```
public void setAge(int a)
{
    age = a;
}
```

178

Methods

Returning results from methods

- After a method completes executing, execution of the program continues from the point where the method was called.
- If a method is to return a value to the place where it was called, then it must terminate by executing a return statement.
- For example, in the getAge() method

```
return age;
```

- More generally, any expression consistent with the method's return type can follow the return keyword.

179

Outline

- Creating your own classes
- Attributes
- Methods
- **Constructors**
- Usage

180

Constructors

General

- When we create an instance of a class via, e.g.,

  Person myBrother = new Person("Jim", 39);

  we are calling (one of) the class's constructors.
- The constructor serves to instantiate the object by initializing the values of the instance attributes.

181

Constructors

Person implementation

```
// constructors

public Person()
{
   name = DEFAULT_NAME;
   age = MIN_AGE;
}

public Person(String n, int a)
{
   name = n;
   age = a;
}
```

Remarks

- Constructors have the same name as their class.
- The result type need not be specified as it is always the constructor's class.
- Constructors are generally public.
- There often is more than one constructor available for a method (overloading).
  - Each must have a unique parameter list.

182

Creating your own classes

Class template

```
// Definition of the Person class.
public class Person
{
// constructors

// instance methods

// class (static) methods

// instance attributes

// class (static) constant attributes

// class (static) attributes

}
```

Looking back

- we have generated a complete coding of the Person class.

Looking forward

- we give an illustration of usage.

183

Outline

- APIs and class use
- Creating your own classes
- Attributes
- Constructors
- Methods
- **Usage**

184

Usage

Let's try out the Person class

```
// assume the usual

// Definition of the TestPerson class.
public class TestPerson
{    public static void main(String[ ] args)
     {     // Test code
     }
}
```

185

Usage

```
// Test code
```

Usage

```
// Test code
Person p1 = new Person(“Jim”, 21);
```

Usage

```
// Test code
Person p1 = new Person(“Jim”, 21);
output.println(“Person’s min age: “ + Person.MIN_AGE); // 0
```

Usage

```
// Test code
Person p1 = new Person(“Jim”, 21);
output.println(“Person’s min age: “ + Person.MIN_AGE); // 0
output.println(“Person’s max age: “ + Person.maxAge); // 110
```

Usage

```
// Test code
Person p1 = new Person(“Jim”, 21);
output.println(“Person’s min age: “ + Person.MIN_AGE); // 0
output.println(“Person’s max age: “ + Person.maxAge); // 110
output.println(“p1’s name: “ + p1.getName()); // Jim
```

Usage

```
// Test code
Person p1 = new Person(“Jim”, 21);
output.println(“Person’s min age: “ + Person.MIN_AGE); // 0
output.println(“Person’s max age: “ + Person.maxAge); // 110
output.println(“p1’s name: “ + p1.getName()); // Jim
output.println(“p1’s age: “ + p1.getAge()); // 21
```

Usage

```
// Test code
Person p1 = new Person(“Jim”, 21);
output.println(“Person’s min age: “ + Person.MIN_AGE); // 0
output.println(“Person’s max age: “ + Person.maxAge); // 110
output.println(“p1’s name: “ + p1.getName()); // Jim
output.println(“p1’s age: “ + p1.getAge()); // 21
p1.incrementAge();
```

Usage

```
// Test code
Person p1 = new Person(“Jim”, 21);
output.println(“Person’s min age: “ + Person.MIN_AGE); // 0
output.println(“Person’s max age: “ + Person.maxAge); // 110
output.println(“p1’s name: “ + p1.getName()); // Jim
output.println(“p1’s age: “ + p1.getAge()); // 21
p1.incrementAge();
output.println(“p1’s age: “  + p1.getAge()); // 22
```

Usage

```
// Test code
Person p1 = new Person(“Jim”, 21);
output.println(“Person’s min age: “ + Person.MIN_AGE); // 0
output.println(“Person’s max age: “ + Person.maxAge); // 110
output.println(“p1’s name: “ + p1.getName()); // Jim
output.println(“p1’s age: “ + p1.getAge()); // 21
p1.incrementAge();
output.println(“p1’s age: “  + p1.getAge()); // 22
p1.setAge(Person.maxAge);
```

Usage

```
// Test code
Person p1 = new Person(“Jim”, 21);
output.println(“Person’s min age: “ + Person.MIN_AGE); // 0
output.println(“Person’s max age: “ + Person.maxAge); // 110
output.println(“p1’s name: “ + p1.getName()); // Jim
output.println(“p1’s age: “ + p1.getAge()); // 21
p1.incrementAge();
output.println(“p1’s age: “  + p1.getAge()); // 22
p1.setAge(Person.maxAge);
output.println(“p1’s age: “ + p1.getAge()); // 110
```

Usage

```
// Test code
Person p1 = new Person(“Jim”, 21);
output.println(“Person’s min age: “ + Person.MIN_AGE); // 0
output.println(“Person’s max age: “ + Person.maxAge); // 110
output.println(“p1’s name: “ + p1.getName()); // Jim
output.println(“p1’s age: “ + p1.getAge()); // 21
p1.incrementAge();
output.println(“p1’s age: “  + p1.getAge()); // 22
p1.setAge(Person.maxAge);
output.println(“p1’s age: “ + p1.getAge()); // 110
Person.maxAge = 210;
```

Usage

```
// Test code
Person p1 = new Person(“Jim”, 21);
output.println(“Person’s min age: “ + Person.MIN_AGE); // 0
output.println(“Person’s max age: “ + Person.maxAge); // 110
output.println(“p1’s name: “ + p1.getName()); // Jim
output.println(“p1’s age: “ + p1.getAge()); // 21
p1.incrementAge();
output.println(“p1’s age: “  + p1.getAge()); // 22
p1.setAge(Person.maxAge);
output.println(“p1’s age: “ + p1.getAge()); // 110
Person.maxAge = 210;
output.println(“Person’s max age: “ Person.maxAge); // 210
```

Usage

```
// Test code
Person p1 = new Person(“Jim”, 21);
output.println(“Person’s min age: “ + Person.MIN_AGE); // 0
output.println(“Person’s max age: “ + Person.maxAge); // 110
output.println(“p1’s name: “ + p1.getName()); // Jim
output.println(“p1’s age: “ + p1.getAge()); // 21
p1.incrementAge();
output.println(“p1’s age: “  + p1.getAge()); // 22
p1.setAge(Person.maxAge);
output.println(“p1’s age: “ + p1.getAge()); // 110
Person.maxAge = 210;
output.println(“Person’s max age: “ Person.maxAge); // 210
output.println(“p1’s age: “ + p1.getAge()); // 110
// continued on next slide
```

198

Usage

```
// Test code
Person p2 = p1;
```

Usage

```
// Test code
Person p2 = p1;
output.println(“p2’s age: “ + p2.getAge()); // 110
```

Usage

```
// Test code
Person p2 = p1;
output.println(“p2’s age: “ + p2.getAge()); // 110
p2.incrementAge();
```

Usage

```
// Test code
Person p2 = p1;
output.println(“p2’s age: “ + p2.getAge()); // 110
p2.incrementAge();
output.println(“p2’s age: “ + p2.getAge()); // 111
```

Usage

```
// Test code
Person p2 = p1;
output.println(“p2’s age: “ + p2.getAge()); // 110
p2.incrementAge();
output.println(“p2’s age: “ + p2.getAge()); // 111
output.println(“p1’s age: “ + p1.getAge()); // 111
```

Usage

```
// Test code
Person p2 = p1;
output.println(“p2’s age: “ + p2.getAge()); // 110
p2.incrementAge();
output.println(“p2’s age: “ + p2.getAge()); // 111
output.println(“p1’s age: “ + p1.getAge()); // 111
p1 = new Person(“Jules”, 32);
```

Usage

```
// Test code
Person p2 = p1;
output.println(“p2’s age: “ + p2.getAge()); // 110
p2.incrementAge();
output.println(“p2’s age: “ + p2.getAge()); // 111
output.println(“p1’s age: “ + p1.getAge()); // 111
p1 = new Person(“Jules”, 32);
output.println(“p1’s name: “ + p1.getName()); // Jules
```

Usage

```
// Test code
Person p2 = p1;
output.println(“p2’s age: “ + p2.getAge()); // 110
p2.incrementAge();
output.println(“p2’s age: “ + p2.getAge()); // 111
output.println(“p1’s age: “ + p1.getAge()); // 111
p1 = new Person(“Jules”, 32);
output.println(“p1’s name: “ + p1.getName()); // Jules
output.println(“p2’s name: “ + p2.getName()); // Jim
```

Usage

```
// Test code
Person p2 = p1;
output.println(“p2’s age: “ + p2.getAge()); // 110
p2.incrementAge();
output.println(“p2’s age: “ + p2.getAge()); // 111
output.println(“p1’s age: “ + p1.getAge()); // 111
p1 = new Person(“Jules”, 32);
output.println(“p1’s name: “ + p1.getName()); // Jules
output.println(“p2’s name: “ + p2.getName()); // Jim
p2 = null; // p2.getAge() now an error as p2 is null
```

Usage

```
// Test code
Person p2 = p1;
output.println(“p2’s age: “ + p2.getAge()); // 110
p2.incrementAge();
output.println(“p2’s age: “ + p2.getAge()); // 111
output.println(“p1’s age: “ + p1.getAge()); // 111
p1 = new Person(“Jules”, 32);
output.println(“p1’s name: “ + p1.getName()); // Jules
output.println(“p2’s name: “ + p2.getName()); // Jim
p2 = null; // p2.getAge() now an error as p2 is null
output.println(“p1’s age: “ + p1.getAge()); // 32
```

Usage

```
// Test code
Person p2 = p1;
output.println(“p2’s age: “ + p2.getAge()); // 110
p2.incrementAge();
output.println(“p2’s age: “ + p2.getAge()); // 111
output.println(“p1’s age: “ + p1.getAge()); // 111
p1 = new Person(“Jules”, 32);
output.println(“p1’s name: “ + p1.getName()); // Jules
output.println(“p2’s name: “ + p2.getName()); // Jim
p2 = null; // p2.getAge() now an error as p2 is null
output.println(“p1’s age: “ + p1.getAge()); // 32
output.print(“The Person philosophy: “);
```

Usage

```
// Test code
Person p2 = p1;
output.println(“p2’s age: “ + p2.getAge()); // 110
p2.incrementAge();
output.println(“p2’s age: “ + p2.getAge()); // 111
output.println(“p1’s age: “ + p1.getAge()); // 111
p1 = new Person(“Jules”, 32);
output.println(“p1’s name: “ + p1.getName()); // Jules
output.println(“p2’s name: “ + p2.getName()); // Jim
p2 = null; // p2.getAge() now an error as p2 is null
output.println(“p1’s age: “ + p1.getAge()); // 32
output.print(“The Person philosophy: “);
Person.philosophy();
```

210

Usage

Test

- To compile:
  - Make sure Person.java and TestPerson.java are in the same directory.
  - Compile TestPerson.java as usual.
- To execute:
  - Just invoke the java interpreter on the TestPerson bytecode produced by the compiler.

211

Appendix: Summary

- **Creating your own classes**
- **Attributes**
- **Constructors**
- **Methods**
- **Usage**

212