Object Oriented Programming – with Java
and Visual Interactive support

Kees Huizing Ruurd Kuiper

Preface

This book provides a working knowledge of Object Oriented programming and Java.

Computers are used almost everywhere. Their role is processing data. A computer can perform any data processing concern we desire if we provide it with an appropriate instructions, a program, written in a programming language. Executing the program on a computer then results in performing the desired data processing concern.

Programs and their execution can easily become very complex. Therefore, structuring is necessary to make it feasible for humans to develop programs. Object Oriented Programming (OOP) is a powerful and widely used programming paradigm that supports structuring. Java is a representative, modern, OOP language.

Approach

No previous knowledge about programming is assumed.

One OOP methodology and style are explained rather than going into alternatives; especially for the novice programmer it is more helpful to be shown one consistent approach than to have to make uninformed choices. Java is explained and motivated as an OOP language. The emphasis is on concepts rather than on details. Knowledge is build up incrementally, without branching or making de-tours, chapters generally depending on previous ones. This "narrow path" approach provides a fast route to understanding OOP and Java for the conceptually inclined traveler.

The larger structure of the book is in parts, corresponding to programming paradigm issues. The parts contain many small chapters that each address one programming concept. A (learner) programmer is driven by programming aims, and has to (learn to) select the means, the concepts that match the aims. The chapters have a fixed structure.

A section **Aim** makes the programming aim explicit.

A section **Means** provides the corresponding programming concept. An example illustrates the concept that is being introduced.

A section **Execution model** incorporates the new concept in the execution model that shows its workings. This model is more detailed and more explicit than is usually provided. In OOP one's thinking often moves between the static, fixed, program text that is provided as classes, and the dynamically changing objects that are created from the classes when executing, processing the data. Significantly, the paradigm is named Object Oriented Programming rather than Class Oriented Programming. Thinking at the static level is supported by the language facilities. Thinking at the dynamic level is less directly and explicitly supported. The execution model remedies this, as it represents the dynamic behavior of objects. Such a model aids one's thinking about programming, e.g., to understand programming concepts. Also, in discussions about programming issues it is helpful if the participants have an explicit, shared, model in mind. A basic model is provided from the outset, which is extended when new concepts require so.

In some cases, the treatment of a concept involves further subdivision; then subsections are used, maintaining the structure as just outlined.

A **Remarks** section gathers additional information.

Exercises are provided, separately, for each chapter. To learn OOP, and Java, it is necessary to do a substantial amount of the exercises, preferably right after reading the corresponding chapter, before moving on to the next one. This, and explorations by yourself, will make you an amateur programmer; professional programming requires further education and training.

An execution model viewer, called *CoffeeDregs*, for concrete programs visualizes the execution model on screen in an interactive manner.

Additional web sources

The "narrow path" approach may require consulting other sources for further detail. Web sources for additional information are the following.

- The Java development environment is available from the Oracle website: `http://www.oracle.com/technetwork/java/javase/downloads/` (August, 2011)
- For explanations about Java programming in general, see the SUN Java tutorials: `http://download.oracle.com/javase/tutorial/` (August, 2011)
- For a complete description of the Java language, see The Java Language Specification, third edition: `http://java.sun.com/docs/books/jls/` (August, 2011)
- For information about predefined Java classes and interfaces, see the Application Programmers Interface (API): `http://download.oracle.com/javase/7/docs/api/` (August, 2011)
- A good general book on Java is *Learning Java*, by Niemeyer and Knudsen, O'Reilly, 2005.
- A good free book on the web is `http://math.hws.edu/javanotes/` (August, 2011).

Preliminaries

To write, execute and, in the approach of this book, visualize Java programs, the appropriate software needs to be present on the computer. It is assumed that the reader has some basic knowledge about downloading and installing software and about using an editor. The software mentioned below is available for free. Versions for Windows, Unix, Linux, and Macintosh platforms are available.

- To conveniently write Java programs an Integrated Development Environment (IDE) needs to be present. Various IDEs are available, the book does not depend on a particular one, all provide the facilities needed. A suitable IDE is *NetBeans 7.0.1*, available from http://www.netbeans.org/ .
- To execute Java programs, a run-time environment needs to be present. The Java Development Kit, JDK7, provides this; it is available from http://java.sun.com/ .
- To visualize the execution of Java programs, the model viewer, *CoffeeDregs*, needs to be present. *CoffeeDregs* will be made available when ready.

Contents

Part II

Object structures – Composition

Chapter 21

Control transfer between objects – Method calls between objects

So far data processing concerns were described in one class and performed by one object. However, a data processing concern often consists of several smaller interacting sub-concerns. To improve code organization such sub-concerns and their relations are described in separate classes. The classes are instantiated into objects, so the activity in the computer is structured similarly. Two cases are treated: obtaining more objects by writing for each object a class that describes it, and also creating several objects from the same class. Communication between objects is by means of method calls.

21.1 More classes for more objects

We show how more classes and objects can be used to structure a concern into sub-concerns.

21.1.1 Aim

We want to organize code and execution by dividing a data processing concern into sub-concerns.

21.1.2 Means

For each sub-concern we describe in a separate class the data processing to be performed by the corresponding object. Setting-up the relations between objects and their interaction is also described in the classes.

First, we consider the structure. Every object has a unique number by which it can be addressed, called *address* hereafter. One class can use another class in its description by describing that an object of this second class is created and by describing that it has the address of that object.

Note that structuring occurs at both the class and the object level: The class refers, in its code, to another class; the object has the address of another object. A *reference variable* stores the address, also called the *reference*, of an object of a certain class. The type notion is extended to include classes. The `new` command, that up till now only occurred in the `main`, is used at other places in the class code as well.

In `Class1` a reference variable, say `object2`, is declared of the type `Class2`, that can store the address of an object of type `Class2`. Reference variables can occur as instance variables, as local variables, and as parameters. As instance variables they are declared at the top of the class description, together with the other instance variables.

In `Class1`, generally in its constructor, the statement `object1 = new Class2();` describes that `Class2` is instantiated and that the address of the created object is assigned to the variable `object1`.

As before, the first object is created by instantiating `Class1` through `main`. The other objects are then created and connected through the (automatic) execution of the constructor of the coordinating object.

Second, we consider the collaboration: an object calls methods of the objects it has references to. This is done through a command of the form ⟨*reference variable*⟩.⟨*method name*(...)⟩. The reference variable holds the address of the object on which the method should be called.

As before, the first (non-constructor) method of the first object is called from `main`. From this method, other methods, also of other objects, can then be called. Convention: method calls are the only means of communication between objects. This means that variables of objects are not referred to directly in other objects (although it is possible in Java).

Note that an object can only call a method of another object if it has a reference to that object. Also note that when a call from a certain object is terminated, control returns to this object. So collaborating objects do so sequentially: several objects simultaneously exist and store data, but only one at the time is active. A call from one object to another is asymmetric. The calling object needs to have a reference to the other object; the other object, however, does not need, and in many cases does not have, a reference to the calling object. Lastly note, that control is passed along references (and passed back against references).

The following example is about a bank that has a safe from which money can be put into an account, which in turn can be spent. We organize the code and the execution by modeling the bank and the account as two separate objects.

```
class Account {
  double balance;

  Account() {
        balance = 0;
    System.out.println("Account has been created, contains " + balance);
  }

  void add100() {
    balance = balance + 100;
    System.out.println("100 has been added to account, contains " + balance);
   }

  void take10() {
    balance = balance − 10;
    System.out.println("10 has been taken from account, left is " + balance);
   }
}


class BankOne {
   double safe;
   Account account; //reference variable to store address

   BankOne() {
          safe = 0;
      account = new Account(); //creates an account
      System.out.println("Bank with 1 account has been created, stores " + safe);

   }

  void banking(){
    safe = 1000;
    System.out.println("Safe is filled with " + safe);
```

```
      safe = safe − 100; //to mimick transfer to account, money is taken from safe
      System.out.println("100 is taken from safe, safe now contains " + safe);
      account.add100(); //reference to account used to call add100
    }

    // main instantiates and uses the object
    public static void main(String[] args){
      new BankOne().banking();
    }
}
```

Output is:

```
Account has been created, contains 0.0
Bank with 1 account has been created, stores 0.0
safe is filled with 1000.0
100 is taken from safe, safe now contains 900.0
100 has been added to account, now contains 100.0
```

The class `BankOne` describes the declaration of a reference variable of type `Account` and the instantiation of an object of that type as well as the assignment of the address to the reference variable. The description of the class `Account` is given separately.

Executing `main` instantiates an object of the class `BankOne`. This object has an instance variable that can hold a reference to an object of type (i.e., class) `Account`. Then the constructor of the `BankOne` object creates an object of the type `Account` and stores its address in the instance variable `account`.

The `BankOne` object calls, from its method `banking`, methods of the `Account` object; it uses the address in the reference variable `account` to refer to this object.

Note that interaction between objects consists solely of control being passed between methods of these objects; as yet no data is transferred between objects.

In the class `Account` there is no longer one special method where the processing starts. This is because from the viewpoint of an object that uses another object, there are usually several equally important methods that process data; in this case the `add100` and the `take10` method.

21.1.3 Execution model

Every object is labeled with a unique number, prefixed with a #-character to indicate that it is an address. This number is not chosen by the programmer but by the system and has no meaning, except that it is different for different objects. One can not expect that in an actual execution in the computer these exact numbers are used. When a reference variable holds a reference to an object, the address of this object is put in the box of the variable.

Instead of just one object, several may be drawn in an execution state. Passing control between objects behaves as expected: a passive control marker is placed in the calling method and a new method box is drawn in the called object, where the active marker is placed.

21.2 More objects from one class

We show how more objects from one class can be used to structure a concern into collaborating smaller concerns.

21.2.1 Aim

To have a structure of classes and collaborating objects. Some of the objects may be of the same class.

21.2.2 Means

To obtain several objects of the same class, the `new`-command for that class has to be executed several times. To distinguish between these objects, different reference variables need to be used.

As an example, two objects of the class `Account` are created and the addresses of these objects are stored in two reference variables.

In case multiple objects are instantiated from the same class, in the coordinating object different variables refer to each object. We added a variable `name` to put the name as information in the object itself, for example to show it on screen, to distinguish between objects of the same class.

```
class Account {
  String name;
  double balance;

  Account(String n) {
        name = n;
        balance = 0;
    System.out.println("Account " + name + " has been created, contains " + balance);
  }

  void add100() {
    balance = balance + 100;
    System.out.println("100 has been added to account " + name + ", now contains " + balance);
   }

  void take10() {
    balance = balance − 10;
    System.out.println("10 has been taken from account " + name + ", left is " + balance);
   }
}


class BankMore {
   double safe;
   Account account1; //reference variable to store address
   Account account2; //reference variable to store address


   BankMore() {
      account1 = new Account("FirstAccount"); //creates an account
      account2 = new Account("SecondAccount"); //creates an account
   }

  void banking(){
    safe = 1000;
    System.out.println("Safe is filled with " + safe);

    safe = safe − 100; //to mimick transfer to account1, money is taken from safe
    System.out.println("100 is taken from safe, safe now contains " + safe);
    account1.add100(); //reference to account1 used to call add100

    safe = safe − 100; //to mimick transfer to account2, money is taken from safe
    System.out.println("100 is taken from safe, safe now contains " + safe);
    account2.add100(); //reference to account2 used to call add100
  }

  // main instantiates and uses the object
  public static void main(String[] args){
    new BankMore().banking();
  }
}
```

Output is:

```
Account FirstAccount has been created, contains 0.0
Account SecondAccount has been created, contains 0.0
Safe is filled with 1000.0
100 is taken from safe, safe now contains 900.0
100 has been added to account FirstAccount, now contains 100.0
100 is taken from safe, safe now contains 800.0
100 has been added to account SecondAccount, now contains 100.0
```

21.2.3 Execution model

Nothing really new here.

Example to be provided

Note, that the reference variables `account1` and `account2` are both in the object of class `BankMore` and that the variable `name` in the objects instantiated from class `Account` holds the name of each account.

Chapter 22

Objects as service providers – Method calls with data parameters and returns

Previously, only control was passed between objects. This chapter considers also communicating data values. A centrally organized structure with data communication is shown, other structures are analogous.

22.1 Aim

We want to describe collaborating objects with data communication. To not compromise the idea of having a structure of relatively independent objects, the data communication is at an abstract level.

22.2 Means

We use methods with data parameters to communicate data values from the calling object to the called object. We use data returns to communicate data values from the called object to the calling object.

An important convention in object-oriented programming is to use method calls as the only means to communicate between objects. This means that instance variables of objects are not approached directly from other objects. The advantage of this convention is that another implementation (e.g., with a more efficient data representation, using different instance variables) of a class can be used in place of the original one without changing the code of other classes. These other classes, namely, do not refer to the instance variables directly and communicate only via calls to methods of the changed class. If the code of these methods is changed to deal with the new data representation, the changes remain within the class.

Note that using the parameter and return mechanisms the communication is two-way, but that the construction is not symmetric: only the calling object needs to have a reference to the called object, but not the other way around. Furthermore, the initiative for the communication lies with the calling object.

An important design principle for object oriented programs is to have no more dependencies, in the form of references, than necessary.

Example

```
class Account {
  double balance;

  Account() {
        balance = 0;
```

```
   }

   double getBalance(){
           return balance;
   }

   void add(int a) {
     balance = balance + a;
    }
}


class BankDataParameters {
    double safe;
    Account account; //reference variable to store address

    BankDataParameters() {
        account = new Account(); //creates an account
    }

   void banking(){
     safe = 1000;
     System.out.println("Safe is filled with " + safe);

     safe = safe − 100; //to mimick transfer to account, money is taken from safe
     System.out.println("100 is taken from safe, safe now contains " + safe);
     account.add(100); //reference to account used to call add

     System.out.println("Account now contains " + account.getBalance());
    }

    // main instantiates and uses the object
    public static void main(String[] args){
      new BankDataParameters().banking();
    }
}
```

Output is:

```
Safe is filled with 1000.0
100 is taken from safe, safe now contains 900.0
Account now contains 100.0
```

From the program it can be seen how information goes into the object through the parameters and goes out from the object through the return.

22.3 Execution model

Again, nothing new. Data is passed via parameters and return values as described in the chapters on methods.

Chapter 23

Building object structures – Passing references

Previously, only data values were passed between objects. This chapter considers also communicating reference values. There is no difference in mechanism between the case of data or reference values. A centrally organized structure with reference communication is shown, other structures are analogous.

23.1 Aim

We want to describe collaborating objects with data communication. To not compromise the idea of having a structure of relatively independent objects, the data communication is at an abstract level.

23.2 Means

We use methods with reference parameters to communicate reference values from the calling object to the called object. We use reference returns to communicate references from the called object to the calling object.

Thus we respect the important OO convention to use method calls as the only means to communicate values of (reference) data between objects. This means that reference variables of objects can not be approached directly from other objects, hence the use of methods.

A reference variable holds the address of an object, which can also be viewed as a pointer. Changing the value of a reference variable therefore can be viewed as changing the address value, or, equivalently, as re-directing the pointer in the variable to point to the object with the new address.

An example of passing information to an object, using parameters, is the following.

```
class Account {
  double balance;

  Account() {
        balance = 0;
  }

  void add100() {
    balance = balance + 100;
  }
  void show() {
        System.out.println(”Balance is ” + balance);
  }
}

class Client {
        Account account;

        Client() {
      //account = null; //makes null reference
        }

        void add100() {
          System.out.println(”Client adds 100 to account.”);
          account.add100();
        }

        void setAccount(Account a) { //enable others to set reference to account client
      account = a;
        }
}

class BankReferenceParameters {
   Client client;
   Account account;

   BankReferenceParameters() {
      account = new Account(); //creates and makes reference to account
      client = new Client(); //creates and makes reference to client
   }

  void banking(){

        System.out.println(”Bank adds 100 to account”);
        account.add100();

        System.out.println(”Bank sets clients reference to banks account.”);
        client.setAccount(account);

    client.add100();
    account.show();
   }

   // main instantiates and uses the object
   public static void main(String[] args){
     new BankReferenceParameters().banking();
   }
}
```

Output is:

```
Bank adds 100 to account
Bank sets clients reference to banks account.
Client adds 100 to account.
Balance is 200.0
```

From the program it can be seen how information, here reference information, goes into the object through the parameter.

An example of obtaining information from an object, using the return, is the following.

```
class Account {
  double balance;

  Account() {
        balance = 0;
  }

  void add100() {
    balance = balance + 100;
  }

  void show() {
        System.out.println("Balance is " + balance);
  }
}

class Client {
        Account account;

        Client() {
      account = new Account(); //creates and makes reference to private account
        }

        void add100() {
          System.out.println("Client adds 100 to account.");
          account.add100();
        }

        Account getAccount() { //provides reference to account client to others
      return account;
        }
}

class BankReferenceReturn {
   Client client;
   Account account;

   BankReferenceReturn() {
      account = null; // makes null reference
      client = new Client(); //creates and makes reference to client
   }

  void banking(){

        client.add100();

        System.out.println("Bank gets clients reference to clients account.");
        account = client.getAccount();

        System.out.println("Bank adds 100 to account");
    account.add100();
    account.show();
   }

   // main instantiates and uses the object
   public static void main(String[] args){
     new BankReferenceReturn().banking();
   }
}
```

Output is:

```
Client adds 100 to account.
Bank gets clients reference to clients account.
Bank adds 100 to account
Balance is 200.0
```

From the program it can be seen how information, here reference information, is obtained from the object through the return.

23.3 Execution model

Nothing new, models for both examples follow directly. The address value of the account is passed as a parameter, respectively returned, in the same way as any other value.

Chapter 24

Kinds of object structures – Aliasing

To be adapted

We have described how to organize objects; we now discuss three kinds of organization. In the first two cases the connection between objects is one-way, the last case concerns a two-way connection.

In case of aggregation again an object owns another object, but is not the only one to do so: more objects own that object; the object is shared. However, it does not refer back to these objects. In case of association, no holds are barred: objects may refer back and forth to one another.

24.1 Composition

In the case of composition an object is the sole owner of some other object; like having a part.

24.1.1 Aim

To have an ownership relation between objects.

24.1.2 Means

An object owns another object as a part: it has a reference variable that has the only reference to that object. Declaration of the reference variable, instantiation of the object and putting the reference into the reference variable usually take place in the object that is the owner, i.e., that has the reference.

Example

Each animal its own trough.

```
class Animal{
  int stomach;

  Trough trough; //declaration

  Animal(){
    stomach = 0;
    trough = new Trough(); //instantiation
  }

   void eat(){
       stomach = stomach+1;
       trough.give();
   }
}

class Trough{
   int contents;

   Trough(){
           contents = 0;
   }

   void fill(){
       contents = contents + 5;
   }

   void give(){
       contents=contents−1;
   }
}

public class FarmerComposition{
  Animal animal1;
  Animal animal2;

  void farm(){
    animal1 = new Animal();
    animal2 = new Animal();

    animal1.trough.fill();
    animal2.trough.fill();

    animal1.eat(); //from own trough
    animal2.eat(); //from own trough
  }

  public static void main(String[] args){
    new FarmerComposition().farm();
   }
}
```

24.1.3 Execution model

Only one arrow goes into an object. In the execution model it shows how the references are directly inside the animal objects and not in the farmer object.

Example

Provide

24.2 Aggregation

More objects own an object: the owners all have a reference variable that contains a reference to the owned object. Declaration of the reference variables still (by definition) takes place in the object that has the reference. Instantiation of the object and putting the reference into the reference variable usually is done outside the objects that have the reference: understandably so, because none of the various objects is in a position to claim this right over another one.

Example

Provide two animals with a shared trough.

```
class Animal{
  int stomach;

  Trough trough; //declaration

  Animal(){
    stomach = 0;
  }

  void setTrough(Trough t){
          trough = t;
  }

   void eat(){
     stomach = stomach+1;
     trough.give();
   }
}

class Trough{
   int contents;

   Trough(){
          contents = 0;
   }

   void fill(){
      contents = contents + 5;
   }

   void give(){
      contents=contents−1;
   }
}

public class FarmerAggregation{
  Animal animal1;
  Animal animal2;

  Trough trough;//declaration

  void farm(){
    animal1 = new Animal();
    animal2 = new Animal();

    trough = new Trough(); //instantiation

    animal1.setTrough(trough); //connection
    animal2.setTrough(trough); //connection

    trough.fill(); // fill shared trough

    animal1.eat(); // from shared trough
    animal2.eat(); // from shared trough
  }

  public static void main(String[] args){
    new FarmerAggregation().farm();
   }
}
```

24.2.1 Execution model

More arrows go into one object.

Example

to be provided

24.3 Association

Objects are connected because each has a reference variable referring to the other. Again, declaration of the reference variables takes place in the object that has the reference; instantiation of the object as well as putting the reference into the variable usually is done outside the objects.

Example

Through registers whether or not a cow is edible.

```
class Animal{
  int stomach;

  Trough trough; //declaration

  Animal(){
    stomach = 0;
  }

  void setTrough(Trough t){
        trough = t;
  }

  void eat(){
    stomach = stomach+1;
    trough.give();
  }

  boolean isEdible(){
   return (stomach>5);
  }
}

class Trough{
  int contents;

  Animal animala; //declaration
  Animal animalb; //declaration

  Trough(){
        contents = 0;
  }

  void setAnimala(Animal a){
        animala = a;
  }

  void setAnimalb(Animal a){
        animalb = a;
  }

  void fill(){
    contents = contents + 5;
  }

  void give(){
    contents=contents−1;
   }

   boolean available(){
           return(animala.isEdible()||animalb.isEdible()); //true if edible animal is available
   }
}

public class FarmerAssociation{
  Animal animal1;
  Animal animal2;

  Trough trough;//declaration

  void farm(){
    animal1 = new Animal();
    animal2 = new Animal();

    trough = new Trough(); //instantiation

    animal1.setTrough(trough); //connection
    animal2.setTrough(trough); //connection

    trough.setAnimala(animal1); //connection
    trough.setAnimalb(animal2); //connection

    trough.fill(); // fill shared trough

    animal1.eat(); // from shared trough
    animal2.eat(); // from shared trough

    if (trough.available())
        System.out.println("Bon appetit");
  }

  public static void main(String[] args){
    new FarmerAssociation().farm();
   }
}
```

24.3.1 Execution model

More arrows go into one object.

Example

Provide

Chapter 25

Recursive object structures – Object recursion

*** To be extended ***

In chapter **??**, data duplication required for solving problems via method recursion was achieved duplicating the local variables in the method. The same idea is now applied to data in instance variables: duplicating objects using object recursion. The motivation is, that data in local variables cannot be used by other objects whereas data in instance variables can.

An important use is defining dynamically changing user built data types/structures.

An example is a linked list of objects. The linked list is then defined recursively, the resulting data structure grows when nodes, as required by storage demands, are added to the list and shrinks when nodes are deleted.

Chapter 26

Local classes – Inner classes

*** To be written, maybe take ideas form four winds of ... ? ***

Chapter 27

Class members – Static members

To be changed to account examples

So far, data storage and manipulation have been done through objects, instantiated from classes. Some data and manipulation is at the level of the class rather than the objects instantiated from it. For this, the class object is used: it is an object that is created when the file that the class definition is in is executed, i.e., even before any instance object is created.

27.1 Aim

To have a place for data storage and manipulation that is at class level rather than at the level of individual objects.

27.2 Means

Apart from the description in the class of variables and methods that get instantiated when an object is instantiated from that class, `static` variables and methods can be described. Static variables and methods do not belong to any of the objects that are explicitly instantiated from the class, but to a separate class object that is implicitly, automatically, instantiated when a class is executed. The class object is not identified by a reference, but by the name of the class it belongs to. Static methods are called through <classname>.<methodname>. `main` is a special case of such a static method; special in the sense that it is called automatically when the class is executed. Also, this is the only way `main` can be called ("listen carefully, `main` only gets called once").

Example

We want to keep count of the number of animals instantiated. This is clearly not the task of any individual animal. Also, as the aim is to keep count of all animals created, it would not be correct to make this the task of the farmer, as, for example, there might be more farmers, each instantiating cows. It is typically a task at the level of the Animal class. We extend the animal class accordingly.

```
class Animal{
  static int numberOfAnimals;

  static {
    numberOfAnimals = 0;
  }

  static int getNumberOfAnimals(){
    return numberOfAnimals;
  }

  int stomach;

  Animal(){
    stomach = 0;
        numberOfAnimals = numberOfAnimals + 1;
  }

  void eat(){
    stomach=stomach+1;// 1 meal eaten
    System.out.println("Stomach contains after meal " + stomach);
  }
}


public class FarmerStatic{
  Animal animal1;
  Animal animal2;

  void farm(){
        System.out.println("Number of animals is "+ Animal.getNumberOfAnimals());

    animal1 = new Animal();
    System.out.println("Number of animals is "+ Animal.getNumberOfAnimals());

    animal2 = new Animal();
    System.out.println("Number of animals is "+ Animal.getNumberOfAnimals());
  }

  // main instantiates and uses the object
  public static void main(String[] args){
    FarmerStatic farmerStatic = new FarmerStatic();
    farmerStatic.farm();
  }
}
```

The following new lines occur.

`static int numberOfAnimals` keeps the counting information.

`static` is the constructor for the class object `Animal`. By rule it is named `static`.

`numberOfAnimals = numberOfAnimals + 1;` in the constructor of `Animals` increases the animal count by 1 each time a new object is instantiated from `Animal`.

`getNumberOfAnimals` is the getter method that enables to get the value of the counter.

27.3 Execution model

Thus far, we only modeled the objects that were instantiated from a class. We extend the model with the class objects. Because class objects are not identified by a reference, we put the name of the class they belong to to them. Note, that the main is also a static method and therefore belongs in the class object: We sketch the model.

Example

```
  FarmerStatic                                Animal
  -----------------------------------------   ------------------------
  |main(){                                 |  |numberOfAnimals;      |
|-|  ... farmerStatic = new FarmerStatic();|  |                      |
| |  farmerStatic.farm();                  |  |static{...}           |
| |}                                       |  |                      |
| ---------------------------------------- -  |getNumberOfAnimals(){}|
|                                             ------------------------
|
|          |--------------------------|
v          |                          v
---------| |  ------------    ------------
|animal1;|-|->|stomach;  |    |stomach;  |
|animal2;|-|  |          |    |          |
|        |    |Animal(){}|    |Animal(){}|
|farm(){}|    |          |    |          |
----------    |eat(){}   |    |eat(){}   |
              ------------    ------------
```

NB The `farmerStatic` object has referencs to both animal objects `animal1` and `animal2`. Therefore, the farmerobject can call the methods of the animal objects. Static variables and methods of a class, belonging to the class object of that class, can be accessed and called without references being needed. For example, the constructor `Animal` that belongs to the instantion `animal1` can update the static variable `numberOfAnimals` in the class object for `Animal`. Also, the static method `getNumberOfAnimals` in the class object for `Animal` can be called from the method `farm` in the object `farmerStatic`.

Chapter 28

Encapsulation - Access modifiers

Packages to be included; static members to be added.

So far, we have only used conventions to limit dependencies between classes. One category of such conventions concerns accessibility of variables and methods. Now we introduce language constructs that enable to enforce this accessibility.

28.1 Aim

Decrease dependencies between classes.

28.2 Means

Restrict what classes can see and use from other classes. Methods and variables can be hidden from other classes. This way, the other classes can not use the hidden methods and variables and hence do not depend on them.

28.2.1 Visibility modifiers

A visibility modifier is a special word that you write at the start of a declaration of methods, variables, and classes. Examples:

```
private double income;

private double calculateIncome() {
    ...
}
```

This variable and method are *private*. They can only be “seen” (i.e., for the method, called, for the variable, used) inside the class where they are declared.

```
public int score;

public void actionPerformed(ActionEvent e) {
    ...
}
```

This variable and method are *public*. They can be seen from any class in the system.

Between these two extremes there are two intermediate levels of visibility that have to do with grouping of classes into so-called *packages*.

Packages

A *package* is a group of classes that belong together and are labeled as such by the programmer. An example is `java.io`, a set of classes that deals with input and output and files. The class `Scanner` is part of the package `java.io`. You can declare that the classes in a file belong to package `myPackage` by putting the line

```
package myPackage;
```

at the beginning of the file *and* putting the classes together in a folder with this name.

When you leave out the line with `package`, the classes belong to the so-called *anonymous* package for which the name of the folder is irrelevant.

Overview of visibility modifiers from wide to narrow:

- `public`: no limitations, visible in all classes;
- `protected`: visible only in classes of the same package and in all subclasses[1];
- *none* (sometimes called ”package”): visible only in the classes of the same package (this is what we have used mostly so far);
- `private`: visible only in the same class.

[1]subclasses are treated in Part III

Chapter 29

Specification for object structures - Class invariants

Interface specification.

***Class invariants as describing the “observable” states of objects: before method calls and after method returns ***

Chapter 30

User-built data types – Mutable and immutable data

In some cases the kind of data that a program uses is not provided as a built-in data type by the language. Then a data type can be programmed: a user-built data type.

30.1 Aim

To make user-built data types.

A built-in data type provides typed values, operations to make terms, and typed variables for storage, as described in chapters 2, 3, and 4. Furthermore, a built-in data type has several implicit properties that support programming. User-built data types should also have these properties, so we list these explicitly.

1. Variable declaration: naming and creation of variable.
2. Value assignment to variables: both of values directly and using terms containing, e.g., values, variables or method returns.
3. No outside change: no change to variable values is possible except through operators of the data type.
4. Single storage: a value is, intuitively, stored in only one variable at the time.
5. Value-equality operator, returning a boolean result.
6. Instance variable accessibility: methods of an object can act on instance variables.
7. Method parameters: values and expressions (e.g., containing variables) can be used as parameters.
8. Method returns: a value can be returned and can be assigned to variables or used in terms.

30.2 Means

We use the class construct to program a user-built data type and explain how to achieve the properties.

A user-built data type is given as a class and consists of the following items.

1. Values. A value is the data part of an object of the class. The data part is built from variables from built-in data types or previously defined user-built data types (possibly recursively).
2. Operators. An operator is an instance method of the class. The operators are built from operators of built-in data types or previously defined user-built data types (possibly recursively).
3. Variables. A variable is an instance variable of the class. A variable should, intuitively, store a value of the user-built data type, i.e., an object. An instance variable in fact holds the address of an object, not the object itself - the consequences are handled below.
4. Assignment.

Note, that because methods are associated with an object, the form of a binary operator like for comparison is not of form `equals(o1, o2)` but rather of form `o1.equals(o2).`

The properties are obtained as follows.

1. Variable declaration. Variable declaration provides a name for the variable and and the creation of an object to hold the value. See the next item for details about value assignment. Assignment of a default value is optional and can be effected through a constructor without parameters. (If this is not provided, the variables from the built-in types of which the new data type is built-up will be assigned the default values as in the built-in data types.) Initialization with a chosen value is also optional and can be effected through a constructor with parameters.
2. Value assignment. We distinguish values and terms containing, e.g., variables or method returns. Because the user-built data type stores values of a new form, which are not part of the language, in variables, these values have to be build-up from part-values that belong to the data types from which the user-built data type is constructed. If these are values from built-in types they can be provided directly as part-values, if not, they can be only provided as stored in variables. We provide a method `assign` with two implementations, distinguished by overloading. One enables to assign part-values, the other to assign values that are already stored in variables of the user-built type.
3. No outside change. This concerns the fact that a variable holds the address of an object rather than the object itself. Because in a data type changes to a variable value (the object) should only occur through user-built data type operators some measures have to be taken to ensure this.

   No outside change to parts should occur except through the user-built data type methods. Variables may have user-built data types as parts. To ensure that these are not changed, there should be no references from outside the user-built data type to these parts. This is enforced by making the internal variables that refer to such parts private in the user-built data type. Note, that this does not prevent pointers from (parts of) one object in the user-built type to (parts of) another object of this type: this should simply not be programmed in the user-built type.
4. Single storage. Variables now are objects, the variable name contains the address of the object. To ensure that a change to one variable does not cause a change to another one, aliasing has to be prevented: only one name should hold the address of a variable.

   One way to achieve single storage is through methods that change parts of a variable; this is quite efficient, as this also means only a partial change to the object that implements the variable.

   Because the variable contents can be changed, this is called the *mutable* approach. To achieve single storage, it is disallowing the use of =, which affects the address, and replacing it by `assign`, which affects the stored values.

   Another way to achieve the effect of single storage is to prevent that the object gets changed. The idea is again that a value is an object and the stored value is the stored part-values, but rather than preventing that two variables hold the same object, we prevent that that object gets changed. This is done by ensuring that for each new value a different object is created. Then = can be retained for assignment.

Because the contents can not be changed (another object is used instead), this is called the `immutable` approach.

Note, that inside the data type quite different things happen for mutable/immutable, e.g. regarding instance variables, but that outside the effect is the same. Also note, that it is still possible to define part-value operations, but that, as in the immutable case also for the partial-changes a whole new object is created, this is not efficient anymore.

5. Value-equality operator. The equality operator of the user-built data type should compare values, not addresses. Therefore, provide a method `o1.equals(o2)` that compares the stored values in `o1` and `o2` and returns `true` if they are the same. Comparison is achieved through the `equals` method; the use of == on variables of the user-built data type, which would compare the addresses rather than the values, is disallowed.

   Summarizing, for programming with a mutable user-built data type the restrictions are the following

   (a) `assign` for assignment (in stead of =).
   (b) `equals` for comparison (in stead of ==).

   Summarizing, for programming with a immutable user-built data type the restrictions are the following.

   (a) `equals` for comparison (in stead of ==).

6. Instance variable accessibility: methods of an object act on instance variables. Achieved.

7. Method parameters: values and variables can be used. Achieved for variables. If so desired, overloading allows to write methods that also take part-values that together make up a value.

8. Method returns: a value is returned and can be assigned to variables or used in terms. Achieved. Note, that to store a return value in a variable, `assign` is to be used, not the disallowed =

In the example, we restrict ourselves to assignment and equality operators.

Example mutable approach

We built a data type `Rat` that supports rational numbers.

```
import javax.swing.*;
class Rat {
        private int n;
        private int d;

        public Rat () {
          n = 0;
          d = 1;
        }

        public Rat (int n, int d) {
          if (d == 0) {
            JOptionPane.showMessageDialog(null, "Denominator 0 is not allowed");
          } else {
            this.n = n;
            this.d = d;
          }
        }

        public Rat (Rat r) {
          n = r.n;
          d = r.d;
        }

        public void assign (int n, int d) {//enables assignment through part−values (int)
          if (d == 0) {
            JOptionPane.showMessageDialog(null, "Denominator 0 is not allowed");
          } else {
          this.n = n;
          this.d = d;
          }
        }

        public void assign (Rat r) { // enables assignment Rat value
          n = r.n;
          d = r.d;
        }

        public Rat add(Rat r) {
          Rat sum = new Rat();

          sum.n = n ∗ r.d + r.n ∗ d;
          sum.d = d ∗ r.d;

          return sum;
        }

        public void simplify() { // enables to simplify a Rat
                int i = 2;
          while (i <= (int) (Math.min(n, d))){
                if ((n%i == 0) && (d%i == 0)) {
              n = n/i;
                  d = d/i;
                }else{
              i = i+1;
                }
          }
        }

   public boolean equalsClever(Rat r){ // works for both immutable and mutable
         boolean e;
     e = (this.n ∗ r.d ==this.d ∗ r.n);
         return e;
  }

        public boolean equals(Rat r) { // only works for mutable
                boolean e;
                Rat s = new Rat();
                Rat t = new Rat();

                s.assign(this);
                s.simplify();

                t.assign(r);
                t.simplify();

                e = ((s.n == t.n) && (s.d == t.d));
                return e;
        }
        void draw() {
          if (d == 1){
            System.out.println(n);
          }else{
        System.out.println(n + "/" + d);
      }
        }
}


class RationalDemoMutable {
Rat r1 = new Rat();
Rat r2 = new Rat(6, 4);
Rat r3 = new Rat(r2);
Rat r4 = new Rat();
Rat r5 = new Rat();

  void demo() {
  r4.assign(2, 7);
  r5.assign(r4);

  System.out.println("r1, r2, r3, r4, r5 are, respectively:");
  r1.draw();
  r2.draw();
  r3.draw();
  r4.draw();
  r5.draw();

  System.out.println("r2 equals r3 is: " + r2.equals(r3));
  System.out.println("r2 equalsClever r3 is: " + r2.equalsClever(r3));

  System.out.println("Simplifying r2 makes r2:");
  r2.simplify();
  r2.draw();

  System.out.println("Adding r2 to r3, storing in r1, makes r1:");
  r1.assign( r2.add( r3 ));
  r1.draw();

  System.out.println("r1 equals r3 is: " + r1.equals(r3));
  System.out.println("r1 equalsClever r3 is: " + r1.equalsClever(r3));

  System.out.println("Simplifying r1 makes r1:");
  r1.simplify();
  r1.draw();
  }


// main instantiates and uses the object
  public static void main(String[] args) {
  RationalDemoMutable rationalDemoMutable = new RationalDemoMutable();
  rationalDemoMutable.demo();
  }
}
```

Output:

```
r1, r2, r3, r4, r5 are, respectively:
```

```
0
6/4
6/4
2/7
2/7
r2 equals r3 is: true
r2 equalsClever r3 is: true
Simplifying r2 makes r2:
3/2
Adding r2 to r3, storing in r1, makes r1:
24/8
r1 equals r3 is: false
r1 equalsClever r3 is: false
Simplifying r1 makes r1:
3
Press any key to continue...
```

Example immutable approach

```
import javax.swing.*;
class Rat {
        private int n;
        private int d;

        public Rat () {
          n = 0;
          d = 1;
        }

  public Rat (int n, int d) {
          if (d == 0) {
            JOptionPane.showMessageDialog(null, "Denominator 0 is not allowed");
          } else {
            this.n = n;
            this.d = d;
          }
        }

        public Rat (Rat r) {
          n = r.n;
          d = r.d;
        }

        public Rat add(Rat r) {
          Rat sum = new Rat();

          sum.n = n * r.d + r.n * d;
          sum.d = d * r.d;

          return sum;
        }

        public Rat simplify() { //simplifies Rat value; result assignment necessary
          Rat simple;
          int n;
          int d;
```

```
            n = this.n;
            d = this.d;

            int i = 2;
            while (i <= (int) (Math.min(n, d))){
                  if ((n%i == 0) && (d%i == 0)) {
                n = n/i;
                    d = d/i;
                  }else{
                i = i+1;
                  }
            }
            simple = new Rat(n, d);
            return simple;
          }

   public boolean equalsClever(Rat r){ // works for both immutable and mutable
          boolean e;
          e = (this.n * r.d ==this.d * r.n);
          return e;
   }

   public boolean equals(Rat r) { // only works for immutable
                  boolean e;
                  Rat s = new Rat();
                  Rat t = new Rat();

                  s = this;
                  s = s.simplify();

                  t = r;
                  t = t.simplify();

                  e = ((s.n == t.n) && (s.d == t.d));
                  return e;
          }

          void draw() {
            if (d == 1){
              System.out.println(n);
            }else{
          System.out.println(n + "/" + d);
        }
          }
}


class RationalDemoImmutable {
Rat r1 = new Rat();
Rat r2 = new Rat(6, 4);
Rat r3 = new Rat(r2);
Rat r4 = new Rat();
Rat r5 = new Rat();

   void demo() {
   r4 = new Rat(2, 7);
   r5 = r4;
```

```
  System.out.println("r1, r2, r3, r4, r5 are, respectively:");
  r1.draw();
  r2.draw();
  r3.draw();
  r4.draw();
  r5.draw();

  System.out.println("r2 equals r3 is: " + r2.equals(r3));
  System.out.println("r2 equalsClever r3 is: " + r2.equalsClever(r3));

  System.out.println("r2.simplify(); leaves r2 untouched:");
  r2.simplify();
  r2.draw();

  System.out.println("r2 = r2.simplify(); makes r2:");
  r2 = r2.simplify();
  r2.draw();

  System.out.println("Adding r2 to r3, storing in r1, makes r1:");
  r1 = r2.add( r3 );
  r1.draw();

  System.out.println("r1 equals r3 is: " + r1.equals(r3));
  System.out.println("r1 equalsClever r3 is: " + r1.equalsClever(r3));

  System.out.println("Simplifying r1 makes r1:");
  r1 = r1.simplify();
  r1.draw();
  }


// main instantiates and uses the object
  public static void main(String[] args) {
  RationalDemoImmutable rationalDemoImmutable = new RationalDemoImmutable();
  rationalDemoImmutable.demo();
  }
}
```

Output:

```
r1, r2, r3, r4, r5 are, respectively:
0
6/4
6/4
2/7
2/7
r2 equals r3 is: true
r2 equalsClever r3 is: true
r2.simplify(); leaves r2 untouched:
6/4
r2 = r2.simplify(); makes r2:
3/2
Adding r2 to r3, storing in r1, makes r1:
24/8
r1 equals r3 is: false
r1 equalsClever r3 is: false
```

```
Simplifying r1 makes r1:
3
Press any key to continue...
```

30.3 Remarks

The data type String as supplied by Java is an immutable type `String`.

30.4 Execution model

No change.

Chapter 31

Java data structures, limited use of collection classes

*** To be changed: present text should (in part?) be transferred to chapter User-built data types (data structures as a less strict form of data types), and this chapter should introduce java data structures and some limited form of collection classes. ***

Idea: in practice, rather than the beautiful user-built data types, data structures are required and used that do not prevent aliasing. For example: arrays.

31.1 Aim

We want data structures that are like data types but can be shared between different users, i.e., allow aliasing.

31.2 Means

We use the user-built data type idea but drop the requirements that prevent aliasing. This leaves from the list of required properties from 30 the following.

1. Variable declaration: naming and creation of variable. OK.
2. Value assignment: both as values for initialization and as terms containing, e.g., variables or method returns. In case of arrays = for the array variable, ALLOWING ALIASING, = for elements of primitive type, preventing part-aliasing, = for elements of object-type, ALLOWING PART-ALIASING.
3. No outside change: no change to variable values is possible except through operators of the data type. Lost.
4. Value-equality operator: `equals` necessary.
5. Instance variables accessibility: methods of an object can act on instance variables.
6. Method parameters. Lost, address is passed.
7. Method returns. Lost, address is returned.

This means that a data structure can be obtained by using the mutable data approach from 30, but allowing the assignment and comparison.

1. = for assignment (in stead of `assign`).
2. == for comparison (in stead of `equals`).

Chapter 32

Exception handling – Exception mechanism

Errors may occur beyond the influence of the programmer. For example, when the program tries to read from a disk file that has been deleted. A good (*robust*) program should take all these situations into account. This will result in a lot of extra code that tends to clutter the rest of the code.

In some situations, there is no right value to return for a function (i.e., a method that has a return type). For example, there is no good value to return as a result of division by zero.

32.1 Aim

1. Separate code to handle exceptional situations from the rest of the code.
2. Have a possibility to signal that a method did not terminate normally and the return value should be ignored.

32.2 Means

The mechanism of exception handling realizes these aims.

ad 1 Code in which an exception may occur is surrounded by a *try-block* and the code that handles the exception is written after this block in one or more *catch-blocks*. When during execution of the try-block an exception occurs, the execution skips the rest of the code in the try-block and jumps to the catch-block. The code that handles the normal situation is in the try-block, nicely separated from the code for the exceptional situations in the catch-block(s).

ad 2 A method declaration can be extended with a *throws-clause* that signals that an exception may occur during execution of the method. When this exception occurs, the execution of the method is aborted, no return value is provided and an exception occurs in the calling code. There execution continues as described under 1. We say that the exception is *passed* to the calling code.

32.3 Example of catching an exception

Opening a file for reading may cause, or *throw* in Java parlance, an exception of the type FileNotFoundException. In the code below, we show how to handle such an exception.

```
1  // ExceptionExample.java − handling an exception while reading from a file
2  import java.util.Scanner;
3  import java.io.∗;
4
5  public class ExceptionExample {
6      Scanner scanner;
7      File file;
8      String filename = ”blah.txt”;
9
10     public void demo() {
11         file = new File( filename );
12
13         try {
14             scanner = new Scanner( file );
15             String word = null;
16             int n = 0; // number of words in file
17             while ( scanner.hasNext() ) {
18                 word = scanner.next();
19                 n = n+1;
20             }
21             System.out.println(”The file contains ”+n+” words.”);
22             scanner.close();
23         } catch( FileNotFoundException e) {
24             System.out.println(”The file ”+filename+” could not be found :−(”);
25         }
26     }
27 }
```

If the file `blah.txt` doesn't exist, an exception will be thrown during execution of line 14. In that case, control leaves the try block immediately and moves to the catch block. So all code from line 15 to 22 is skipped and the line 24 is executed. When the file is available for reading, however, execution will continue with line 15 and line 24 will not be executed.

it is important that the try-block is left immediately when an exception occurs. Otherwise, code such as `scanner.hasNext` and `scanner.next()` will be executed. This code is meaningless when the file doesn't exist and could lead to more errors and havoc.

32.4 Example of passing an exception

Instead of handling an exception in a try/catch construction, it may be propagated to the calling method. The method has to signal (declare) this in the header.

```
// ExceptionExample.java − handling an exception while reading from a file
import java.util.Scanner;
import java.io.∗;

public class ExceptionExample2 {
    String filename = ”blah.txt”;

    int countWordsInFile(String filename) throws FileNotFoundException {
        Scanner scanner;
        File file;

        file = new File( filename );
        scanner = new Scanner( file );
        String word = null;
        int n = 0; // number of words in file
        while ( scanner.hasNext() ) {
            word = scanner.next();
            n = n+1;
        }
        scanner.close();
        return n;
    }

    public void demo2() {
        int n;
        try {
            n = countWordsInFile( filename );
            System.out.println("The file contains "+n+" words.");
        } catch( FileNotFoundException e) {
            System.out.println("The file "+filename+" could not be found :-(");
        }
    }
}
```

Exception propagation rules:

1. An exception that is not caught inside a method is passed on to the place where the method was called. No return value will be provided.
2. A method that doesn't handle a certain type of exceptions itself (by means of a try and catch construction) should declare that it may propagate (throw) this type of exception in the header of the method, as follows:

   ```
   int countWordsInFile(String filename) throws FileNotFoundException
   ```

   The compiler will check that all exceptions that could occur in the method body and are not caught by try/catch-blocks are declared.

32.5 Unchecked exceptions

There is a special kind of exceptions that are not specific to certain methods and are usually related to programming errors, not to external errors. These do not have to be caught or passed on, because they can occur at so many places. The compiler does not check whether they are declared in the headers and are therefore called *unchecked exceptions*. They are all subclasses of the class RuntimeException (see below in the section on Exception hierarchy). You may have encountered these already. Examples are NullPointerException, occurring when a method is called on a reference that is `null`, and ArrayIndexOutOfBoundsException, occurring when an array is accessed with an index that doesn't exist. You could catch them if you want, but it is not recommended. Since they are usually caused by a programming error, you should not hide them, but prevent them.

32.6 Exception hierarchy

For each type of exception, there is a corresponding class with the same name. When an exception occurs, an object of the corresponding class is created. This object is passed as a kind of parameter to the catch-block. It contains some information that can be used there.

Exception classes have an inheritance hierarchy. E.g., all exceptions that are a subclass of the class *RuntimeException* don't have to be declared. Exceptions that have to do with input and output to files, such as

FileNotFoundException, are subclasses of *IOException*. This makes it possible to catch several types of exception in one catch-clause. For example, instead of

```
try {
    ...
} catch( FileNotFoundException e) {
    ...
} catch( EOFException e) {
    ...
} catch( InterruptedIOException e) {
    ...
}
```

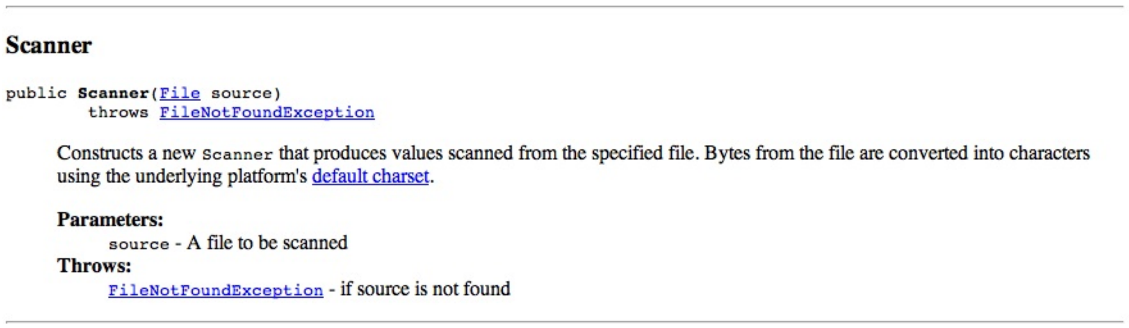
one can also write

```
try {
    ...
} catch( IOException e) {
    ...
}
```

The catch-blocks are searched in the order they appear. The first block that matches the exception (i.e., its type is the same as the exception thrown or a supertype) will be executed.

32.7 Exceptions and the API

The API, Applications Programmer's Interface, describes the available classes and their methods. When a method could throw an exception, this is mentioned in the API. This way, you can see which exceptions you can expect when you use a method. For example, the description in the API[1] of the constructor of the class *Scanner* looks as follows:



Scanner

```
public Scanner(File source)
        throws FileNotFoundException
```

Constructs a new `Scanner` that produces values scanned from the specified file. Bytes from the file are converted into characters using the underlying platform's default charset.

Parameters:
`source` - A file to be scanned
Throws:
`FileNotFoundException` - if source is not found

32.8 Throwing exceptions and creating exception classes

Although in most programs exceptions occur as a result of a statement or a method call, there could be situations when you as a programmer need to have explicitly generate an exception. For this, the keyword `throw` is provided. So if you want to create a FileNotFoundException, you could write

```
throw new FileNotFoundException(”Just for fun”);
```

Probably, it makes more sense to use an exception that you have defined in your code, as follows.

[1] http://java.sun.com/javase/6/docs/api/

```
class AdultContentException extends Exception {
        public AdultContentException(String info) {
                super(info);
        }
}

class ExceptionCreationExample {
        void protectedRead( String filename ) throws AdultContentException, FileNotFoundException {
                Scanner scanner;
                File file;

                if (filename.equals(”sex.txt”) {
                        throw new AdultContentException(”Filename suggests not suited for minors”);
                } else {
                        scanner = new Scanner( new File( filename ) );
                        while ( scanner.hasNext() )
                                ...
                }
        }

        void demo() {
                String filename = ...
                try {
                        protectedRead( filename );
                } catch (IOException e) {
                        System.err.println( ”An I/O−problem occurred while accessing file ”+ filename +”; further infor: ”+e);
                } catch (AdultContentException e) {
                      System.err.println( ”You can’t read ”+filename+” because ”+e);
                }
        }
}
```

Chapter 33

File output and input – java.io

Up till now, data was stored in program variables and hence not preserved if a program terminated. Now more permanent storage is introduced.

33.1 Aim

We want to store data more permanently than just for the execution of a program. For example, the results computed by one program might be stored for later use with another program that runs when the first program is not active anymore.

33.2 Means

Data is stored in and retrieved from files, for example on the hard disk of the computer, like we treat editor files. The difference is that it is now the running program that organizes the storage and retrieval.

We consider output by a Java-program to a file and input by a Java-program from a file: the perspective is always from the viewpoint of the program. A complication is that errors can occur, in which case Java throws a so-called exception that has to be dealt with in your program. (See chapter 32 for more information about exceptions.)

We limit ourselves to text files as these can be accessed in a way that is already known: a Scanner for input and println and print for output. Furthermore, text files can be inspected and changed using an editor. Another, not essentially different, option are the more efficient binary files. For a treatment of these we refer to textbooks and the Java documentation.

We now describe how the objects that perform output and input are obtained and how they are used.

33.2.1 Output to file

An object of the class File contains the name and location (its *pathname*) on disk. File object: Declare, create File object (the file itself is not created!) and set the disk file name.

```
File f = new File("df.txt");
```

Write object: Declare and create PrintWriter object and link it to a file for writing:

```
PrintWriter pw = new PrintWriter( f );
```

Use of such a PrintWriter object is very similar to using System.out (which we use for writing to the console).

Write a String s and go to a new line:

```
pw.println(s);
```

Write a String s without going to a new line:

```
pw.print(s);
```

When you don't want to use the file for output anymore, you *close* the file:

```
pw.close();
```

Example

```
//Writing to a file
import java.util.Scanner;
import java.io.*;

public class FileWrite {
    Scanner sc;
    String filename;
    File f;
    PrintWriter pw;

    public void demo() {
        sc = new Scanner( System.in );
        String choice;
        String word;

        filename = "df.txt";
        f = new File( filename );

        try {
            pw = new PrintWriter( f ); //(if not exists makes,) empties and opens file,
                                       //directs pw where to write to
            System.out.println("Do you want to add a word to the file? (y/n)");
            choice = sc.next();
            while ( choice.equals("y") ) {
                word = sc.next();
                pw.println(word); //writes word into file
                System.out.println("Do you want to add a word to the file (y/n)?");
                choice = sc.next();
            }
            pw.close();
        } catch( FileNotFoundException e) {
            System.out.println("The file "+filename+" could not be opened for writing");
        }
    }


    public static void main(String[] args) {
        new FileWrite().demo();
    }

}
```

In fact, there is a little more to opening a file for writing (by creating a PrintWriter object) than we explained thus far. The statement,

```
PrintWriter pw = new PrintWriter( f );
```

creates a new file if `df.txt` did not exist yet or, if it did exist, empties it. It will throw a `FileNotFoundException` if an error occurs, e.g., when the file is not writable.

Location of files

When you create a file by given a name, it is put in the current working directory. Where this is, depends on your Java system. If you want to read or write a file in a different directory, you can:

- use an absolute pahtname, for example `new File("C:`
  `Documents and Settings`
  `df.txt")` (Windows) or `new File("/Users/kees/df.txt")` (Linux, Mac);
- use a relative pathname: `..` is the parent directory of the current directory;
- change the current working directory by tweaking your IDE;
- use a JFileChooser object (see below).

Appending

If you don't want a file to be emptied before writing to it, but want to append text to the end of the file, you need to change the code as follows (assume f is an appropriate File object).

```
FileWriter fw = new FileWriter( f, true );
PrintWriter pw = new PrintWriter( fw );
```

You can give FileWriter as a second argument a boolean. In case the boolean is true, the existing file is not emptied and text is appended to the original file (i.e., added at the end). In case the boolean is absent or false, the file will be emptied before the PrintWriter starts writing to it. The FileWriter object is an intermediate object that is automatically created when a PrintWriter object is created from a File object. Here, we create it explicitly to be able to set the append option, since this option is not available in the constructors of PrintWriter.

33.2.2 Input from file

Storage location: Assume there is a file on disk (in the program directory) with the name

```
df.txt
```

File object: Declare, create and set the disk file name.

```
File f = new File("df.txt");
```

Read object: Declare and create Scanner object, link it to the File object and open the file for reading.

```
Scanner sc = new Scanner( f );
```

Reading from a Scanner object is known from reading from the console (keyboard). E.g., to read a word from the file, you program:

```
String s = sc.next();
```

When you read from a file, you may reach the end of the file. When the end of the file has been reached, the call `sc.hasNext()` will return `false`. You can use this, e.g., in the guard of a loop that reads through the file.

Example

```
//Reading from a file
import java.util.Scanner;
import java.io.*;

public class FileRead {
    String filename;
    File f;
    Scanner sc;

    public void demo() {
        String word;

        filename = "df.txt";
        f = new File( filename );

        try {
            sc = new Scanner( f ); //(if not exists) makes and) opens file,
                                   //directs sc where to read from
            while ( sc.hasNext() ) {
                word = sc.next(); // reads word from file
                System.out.println(word);
            }
            sc.close();
        } catch( FileNotFoundException e) {
            System.out.println("The file "+filename+" could not be found");
        }
    }

    public static void main(String[] args) {
        new FileRead().demo();
    }
}
```

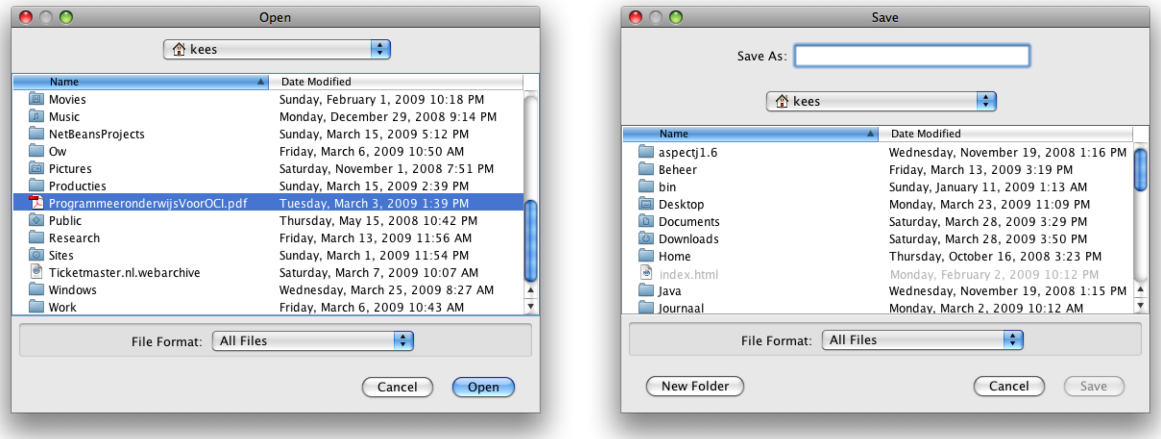
NB It is not possible to simultaneously write to and read from the same file. Before changing from writing to reading or vice versa, you need to close the file and open it again by creating a new read or write object.

33.2.3 Choosing a file with the GUI

In many cases, the program should allow the user to designate which file should be used for reading or writing. A common way to do this is a *file dialog*. Swing provides such a dialog by means of the Component *JFileChooser*. We give a brief explanation of how to use this class for selecting files.

There are two predefined dialogs: an *Open dialog*, allowing the user to designate an existing file, meant for reading, and a *Save dialog*, allowing the user to type a filename or designate an existing file, meant for writing.

Using a JFileChooser consists of the following steps.



Name	Date Modified
Movies	Sunday, February 1, 2009 10:18 PM
Music	Monday, December 29, 2008 9:14 PM
NetBeansProjects	Sunday, March 15, 2009 5:12 PM
Ow	Friday, March 6, 2009 10:50 AM
Pictures	Saturday, November 1, 2008 7:51 PM
Producties	Sunday, March 15, 2009 2:39 PM
ProgrammeeronderwijsVoorOCI.pdf	Tuesday, March 3, 2009 1:39 PM
Public	Thursday, May 15, 2008 10:42 PM
Research	Friday, March 13, 2009 11:56 AM
Sites	Sunday, March 1, 2009 11:54 PM
Ticketmaster.nl.webarchive	Saturday, March 7, 2009 10:07 AM
Windows	Wednesday, March 25, 2009 8:27 AM
Work	Friday, March 6, 2009 10:43 AM

Name	Date Modified
aspectj1.6	Wednesday, November 19, 2008 1:16 PM
Beheer	Friday, March 13, 2009 3:19 PM
bin	Sunday, January 11, 2009 1:13 AM
Desktop	Monday, March 23, 2009 11:09 PM
Documents	Saturday, March 28, 2009 3:29 PM
Downloads	Saturday, March 28, 2009 3:50 PM
Home	Thursday, October 16, 2008 3:23 PM
index.html	Monday, February 2, 2009 10:12 PM
Java	Wednesday, November 19, 2008 1:15 PM
Journaal	Monday, March 2, 2009 10:12 AM

Figure 33.1: Examples of an Open and a Save dialog

1. Declare a variable of the type JFileChooser, create an object of this class and assign it to the variable.

```
JFileChooser jfc;
...
jfc = new JFileChooser();
```

2. Show the dialog on the screen. If you want to read from a file:

```
status = jfc.showOpenDialog(parent);
```

   If you want to write to a file:

```
status = jfc.showSaveDialog(parent);
```

   The program will wait until the user has made a choice and has pressed a button. Which button she has pressed is expressed in the return value of the call. In the statement shown here, this value is stored in the int variable *status*.

   The parameter `parent` should be the Swing Component it is related to. If there is no relevant Component, use `null`.

3. Decide what to do on basis of the return value of the call. The value equals the constant `JFileChooser.APPROVE_OPTION` if the user pressed the Open or Save button.

4. Retrieve the File object from the JFileChooser object. This File object corresponds to the file the user has selected in the dialog. Here we store it in a variable `file` of type File to use it later.

```
file = jfc.getSelectedFile();
```

5. Use the file, e.g.,

```
sc = new Scanner( file );
```

The following three annotated example programs show how to use the JFileChooser.

Example that only chooses

```
import java.io.∗;
import javax.swing.JFileChooser;

public class ChooserSimple {
    File file; // only name, path, etc.
    JFileChooser jf;

    void demo() {
        int status; // result status of the file choosing operation

        jf = new JFileChooser(); // create a file chooser starting in default dir
        status = jf.showOpenDialog(null); // use parent GUI component
                                          // instead of null, if available
        if (status==JFileChooser.APPROVE_OPTION) { // white smoke!
            file = jf.getSelectedFile();
            System.out.println(”You chose file ”+file.toString());
        } else { // chooser was cancelled by user
            System.out.println(”Couldn’t make up your mind?”);
        }
    }

    public static void main(String[] a) {
        new ChooserSimple().demo();
    }
}
```

Example that chooses and writes

```
import java.io.∗;
import javax.swing.JFileChooser;

public class ChooserLessSimple {
    File file; // only name, path, etc.
    JFileChooser jf;
    PrintWriter pw;

    void demo() {
        int status; // result status of the file choosing operation

        jf = new JFileChooser(); // create a file chooser starting in default dir
        status = jf.showSaveDialog(null); // use parent GUI component instead of null, if available
        if (status==JFileChooser.APPROVE_OPTION) { // white smoke!
            file = jf.getSelectedFile();
            try {
                pw = new PrintWriter( file );
                writeStuff();
            } catch(FileNotFoundException e) {
                System.err.println(”Wrong choice!”);
            }
        } else { // chooser was cancelled by user
            System.err.println(”Couldn’t make up your mind?”);
        }
    }
    void writeStuff() {
        // assume pw is set
        for (int i=0; i<1000; i++) {
            pw.println(i*i);
        }
        pw.close();
    }

    public static void main(String[] a) {
        new ChooserLessSimple().demo();
    }

}
```

Example with more GUI

This example shows a panel with a button that brings up a file chooser dialog. The chosen file is read and its contents are shown in the panel.

```
import java.awt.event.*;
import java.io.*;
import javax.swing.*;
import java.awt.*;
import java.util.Scanner;

/**
 *
 * @author kees
 * JFileChooser example, reading from a file, showing in GUI
 */
public class ChooserAdvanced implements ActionListener {
    File file; // only name, path, etc.
    JFileChooser jf;
    Scanner scanner;
    JFrame frame;
    JTextArea jta;
    JButton showButton;

    void demo() {
        frame = new JFrame("JFileChooser Example");
        jta = new JTextArea();
        showButton = new JButton("Show file");
        frame.add(new JScrollPane(jta));
        frame.add( showButton, BorderLayout.SOUTH);
        showButton.addActionListener(this);

        frame.setSize(400,300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }

    public void actionPerformed(ActionEvent evt) {
        int status; // result status of the file choosing operation

        jf = new JFileChooser(); // create a file chooser starting in default dir
        status = jf.showOpenDialog(frame); // param frame determines location of dialog
        if (status==JFileChooser.APPROVE_OPTION) { // white smoke!
```

```
                file = jf.getSelectedFile();
                try {
                    scanner = new Scanner( file );
                    readStuff();
                } catch(FileNotFoundException e) {
                    JOptionPane.showMessageDialog(frame, "Can't open file "+file);
                }
            } else { // chooser was cancelled by user
                System.out.println("Couldn't make up your mind?");
            }
    }

    void readStuff() {
        // assume scanner has been set
        String line;
        while ( scanner.hasNext() ) {
            line = scanner.nextLine();
            jta.append(line+"\n");
        }
        scanner.close();
    }

    public static void main(String[] a) {
        new ChooserAdvanced().demo();
    }
}
```

Chapter 34

Searching

We present some algorithms for searching into data collections.

34.1 Aim

We want to see whether an data collection, e.g., an array contains a certain data element and if so, where it is located. We want to analyse the search algorithms on their relative speed.

34.2 Means

Several *algorithms* exist to search into (large) date collections. Which one to choose depends on the structure of the data and what we can assume about it.

34.2.1 Linear search in an array

A simple structure to store data is the *array*. When we want to find an element in the array and we don't know anything about the way the elements are stored, the only option is to inspect every element until the requested value has been found. Usually the elements are inspected in the order they are found in the array, leading to a *linear* or *sequential* search algorithm.

The following method searches an integer array arr for the value key. If the array contains the key value, it returns *an* index of the value, if it doesn't contain the key, it returns a negative number.

```
final static NOT_FOUND = −1;

int linearSearch(int[] arr, int key) {
    int i = arr.length−1;
    while (i >=0 && arr[i] != key) {
        i++;
    }
    return i;
}
```

In this program, we use a property of the `&&`-operator called *non-strictness* or the *short-circuit* property. It means that the second operand is not evaluated when the first one turns out to be false. Similarly, the second operand of the `||`-operator will not be evaluated when the first operand turns out to be true. This is very convenient in cases like this, where array indexing, or other expressions that can be undefined is part of a boolean expression. In our program, the expression `arr[i] != key` is undefined when `i` is outside the index range of the array, in particular, when `i<0`. This could occur when the array doesn't contain the searched key value. In that case, evaluation of this expression would lead to an exception and possibly abortion of the program. Putting the expression `i < 0` as the first operand of the conjunction avoids this unwanted behavior.

34.2.2 Analysis of linear search

Searching is a very common task for computer programs and often the data volume is very large, such as in search engines that search large portions of the World Wide Web. It is therefore important to analyse an algorithm on how long the search will take, in the number of array inspections. We give three important cases for this analysis. Take n as the number of elements in the array.

- *best case*: 1; when the first element in the array is the one we are looking for, the loop terminates immediately.
- *worst case*: n; when only the last element in the array is the one we are looking for, or when the array doesn't contain the key at all, it takes n array inspections;
- *average case*: $\frac{1}{2}n$: when the array contains exactly one key and the elements are randomly ordered, the search will take on average $\frac{1}{2}n$ steps.

The best case is not very informative, but often easy to calculate. Worst case is what is mentioned often, since it gives an upper bound to the execution time and it is easier to calculate than average case, which also requires knowledge or assumptions about the distribution of the data.

In comparing algorithms by efficiency, we are interested in how fast the execution time grows with the size of the input. Here, the size of the input is the length of the array, called n, and we see that the best case performance doesn't depend on n. We say that best case performance is of *constant* time complexity, denoted by $\Theta(1)$. The worst case execution time grows linearly with n and we say that this has *linear* time complexity, denoted by $\Theta(n)$. Although average case execution time is less than worst case execution time, but it also grows linearly with n and we also denote the complexity by $\Theta(n)$. We will see a definition of this concept later.

34.2.3 Binary search

When the array is sorted, a more efficient search algorithm can be used, called *binary search*. The reason why this is possible is the following. When an array a is sorted in ascending manner and it turns out that the $a[i] < key$, we know that all elements before i are also smaller than the key, so this whole part of the array can be discarded for the search. And the same holds for part beyond i, if $key < a[i]$. So if we inspect the middle element of the array (or an element close to the middle), we can always discard about half the elements of the array (either the left part or the right part), or stop, because we have found the element. Then we can repeat with the part that remains, etc. At every step the part of the array that has to be searched will be halved. If the array has length 2^k, the number of loop executions will be a little bit less than k, in the worst case. So the worst case time complexity is $\Theta({}^2\log n)$.

The strategy of binary search is that of *divide-and-conquer*. The problem is divided in smaller parts, in this case in about equal halves, the parts are solved and their solutions are combined into a solution to the original problem. In this case, the solution of one of the parts is trivial: we see immediately that the sought for key can not be in that part of the array. And then the combination of the two is trivial also: we simply take the solution of the other half.

Strategies of divide-and-conquer are often natural to implement with *recursion*. We also give a recursive program here, although in practice usually an iterative program is used, since this is slightly more efficient.

```
    /**
     * @pre a is ascending
     * @post values a unchanged
     * @post \result  0 => 0  \result < a.length && a[\result] == key
     * @post \result < 0 => −\result−1 is the insertion point ip
     * i.e., \foralll 0i<ip: a[i] < key && \forall ip<i<a.length: key < a[i]
     */
    int binarySearch(int[] a, int key) {
        int lo = −1;
        int hi = a.length;

        // imagine a[−1] == −infinity and a[a.length] = +infinity
        //@inv a[lo]  key < a[hi] && −1 <= lo < hi <= a.length

        while (lo+1 < hi) {
            int mid = (lo+hi)/2;
            //assert 0 <= mid < a.length;
            if (a[mid] <= key) {
                lo = mid;
            } else { assert key < a[mid];
                hi = mid;
            }
        }
        //assert a[lo] <= key < a[hi];
        if (a[lo] == key) {
            return lo;
        } else { // key not in a
            return −hi−1;
        }
    }
}
```

34.3 Complexity Analysis

We call the analysis of the efficiency of an algorithm in quantitative terms of the amount of time (sometimes also memory) used *complexity analysis*. We don't measure in actual time used or CPU cycles because depends on the actual computer that is used, the operating system, etc. Therefore, we measure by determining the number of times some basic operation is performed as a function of the size of the input.

34.3.1 Order of complexity

We introduce the following mathematical notions.

Let functions $f, g : N \to N$ be given.

We define $f \in \mathcal{O}(g(n))$ (pronounced as "big-oh") if there exist c and n_0 such that $0 \le f(n) \le c \cdot g(n)$ for all $n > n_0$.

For example, $2n^2 - 2n + 12 \in \mathcal{O}(n^2)$, $2n^2 - 2n + 12 \in \mathcal{O}(n^3)$, $2n^2 - 2n + 12 \notin \mathcal{O}(n)$.

So $\mathcal{O}$ gives an upper bound to the rate of growth of a function. E.g., we say that the worst case time complexity of linear search is $\mathcal{O}(n)$. This means that for any implementation of linear search there is a constant c such that the time it takes, in worst case, to execute it on an array of n elements, is less than $c \cdot n$, for large n. So the timing behaviour of linear search linear in n, or better.

We define a similar notion to express that a function is at least as worse as another function.

We define $f \in \Omega(g(n))$ (other notation is $\o(g(n)))$ if there exist c and n_0 such that $0 \leq f(n) \geq c \cdot g(n)$ for all $n > n_0$.

We define $f \in \Theta(g(n))$ (Greek capital letter theta) if both $f \in \mathcal{O}(g(n))$ and $f \in \Omega(g(n))$.

For example, $2n^2 - 2n + 12 \in \mathcal{O}(n^2)$, $2n^2 - 2n + 12 \notin \mathcal{O}(n^3)$, n.

Θ is the most informative of these three notions. It defines complexity classes for algorithms. When two algorithms are in $\mathcal{O}(n)$, it means that they are of the same efficiency. Execution times may differ in actual measurements, due to differences in computers or the actual way it is implemented. An algorithm that is in a lower class, however, e.g., $\mathcal{O}(\log n)$, will always be faster in the end. No matter how slow the computer is that is running on, there will be an input size on which it will be faster, and it will be faster on all inputs larger than that.