16 Lecture: Looking the Same

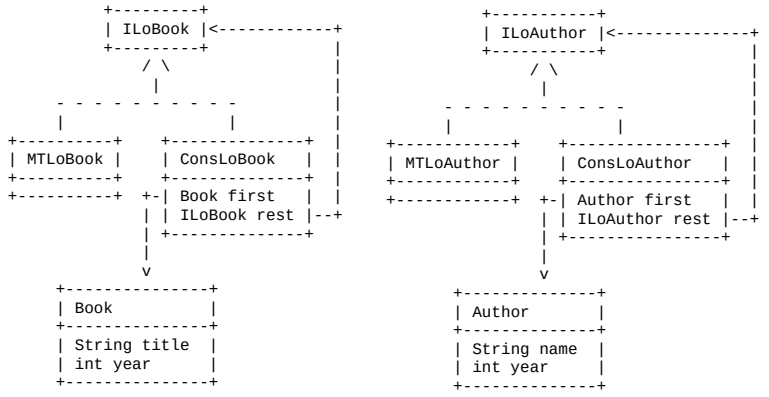
Goals: - Abstracting over Data Types.

Introduction

Over the past several weeks we have defined a number of lists: lists of books, persons, recordings, blocks, and more - repeating the same code again and again. The *Design Recipe for Abstraction* will guide us in learning to remove the repetitions code. In the process, we need to learn a couple of new language features that support such abstractions.

Examples of Code Duplication

Let us compare the code for defining a list of books and a list of authors:



We also include the code for two methods we have seen several times: *size*() that counts the number of elements in the list, and *contains*(*Book/Author that*) that determines whether the list contains the given book or the given author, respectively.

The class diagrams are nearly identical. So is the code. The only difference is the reference to the class of data that the elements of the list represent, either *Book* or *Author*. In Java, every class implicitly extends the super class of all classes, **class** *Object*. If we replace all occurrences of *Book* or *Author* by *Object* — except for the definitions of the two classes, we get the follwing class diagram:

The type of the field first in the class *ConsLoObject* is *Object*. That means, that it can be an instance of any of the classes that extend *Object*. Obviously, instances of *Book* or *Author* are among them. We can test our examples of

```
interface ILoBook{
  // to compute the size of this list
  int size();

  // is the given book in this list?
  boolean contains (Book that);
}

class MTLoBook implements ILoBook{
  MTLoBook() {}

  int size(){ return 0; }

  boolean contains (Book that){
    return false;
  }
}

class ConsLoBook implements ILoBook{
  Book first;
  ILoBook rest;

  ConsLoBook(Book first,
             ILoBook rest){
    this.first = first;
    this.rest = rest;
  }

/* TEMPLATE:
..this.first..               - Book
..this.rest..                - ILoBook
..this.rest.size()..         - int
..this.rest.contains(Book).. - boolean
*/
  int size(){
    return 1 + this.rest.size();
  }

  boolean contains (Book that){
    return this.first.same(that)
        || this.rest.contains(that);
  }
}
class Examples{
  Examples () {}

  Book b1 = new Book("DVC", 2003);
  Book b2 = new Book("LPP", 1942);
  Book b3 = new Book("HtDP", 2001);

  ILoBook mtbooks =
            new MTLoBook();
  ILoBook booklist =
            new ConsLoBook(b1,
            new ConsLoBook(b2,
                           mtbooks));

  boolean testSize1 =
    mtbooks.size() == 0;
  boolean testSize2 =
    booklist.size() == 2;

  boolean testContains1 =
    this.mtbooks.contains(this.b1)
      == false;
  boolean testContains2 =
    this.booklist.contains(this.b2)
      == true;
  boolean testContains3 =
    this.booklist.contains(b3)
      == false;
}
```

```
interface ILoAuthor{
  // to compute the size of this list
  int size();

  // is the given author in this list?
  boolean contains (Author that);
}

class MTLoAuthor implements ILoAuthor{
  MTLoAuthor() {}

  int size(){ return 0; }

  boolean contains (Author that){
    return false;
  }
}

class ConsLoAuthor implements ILoAuthor{
  Author first;
  ILoAuthor rest;

  ConsLoAuthor(Author first,
               ILoAuthor rest){
    this.first = first;
    this.rest = rest;
  }

/* TEMPLATE:
..this.first..               - Author
..this.rest..                - ILoAuthor
..this.rest.size()..           - int
..this.rest.contains(Author).. - boolean
*/
  int size(){
    return 1 + this.rest.size();
  }

  boolean contains (Author that){
    return this.first.same(that)
        || this.rest.contains(that);
  }
}
class Examples{
  Examples () {}

  Author a1 = new Author("DB", 1956);
  Author a2 = new Author("StEx", 1900);
  Author a3 = new Author("MF", 1972);

  ILoAuthor mtauthors =
            new MTLoAuthor();
  ILoAuthor authorlist =
            new ConsLoAuthor(a1,
            new ConsLoAuthor(a2,
                             mtauthors));

  boolean testSize1 =
    mtauthors.size() == 0;
  boolean testSize2 =
    authorlist.size() == 2;

  boolean testContains1 =
    this.mtauthors.contains(this.a1)
      == false;
  boolean testContains2 =
    this.authorlist.contains(this.a2)
      == true;
  boolean testContains3 =
    this.authorlist.contains(a3)
      == false;
}
```

Figure 1: Class definitions for lists of books and authors

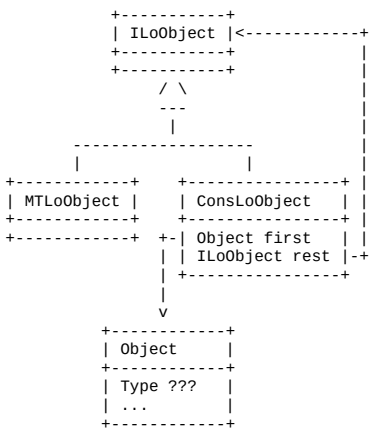


Figure 2: A class diagram lists of books and authors

data — and see that the examples of lists of books and of the lists of authors are the same as before. We now compar the methods. The method *size*() does not refer at all to the any instances of objects contained in the list - indeed, the code was already identical in the two original lists. We now look at the similarities and differences between the two implementations of the method *contains*.

```
class ConsLoBook implements ILoBook{

/* TEMPLATE:
..this.first..               - Book
..this.rest..                - ILoBook
..this.rest.size()..         - int
..this.rest.contains(Book).. - boolean
*/

  boolean contains (Book that){
    return this.first.same(that)
        || this.rest.contains(that);
  }
}
```

```
class ConsLoAuthor implements ILoAuthor{

/* TEMPLATE:
..this.first..               - Author
..this.rest..                - ILoAuthor
..this.rest.size()..           - int
..this.rest.contains(Author).. - boolean
*/

  boolean contains (Author that){
    return this.first.same(that)
        || this.rest.contains(that);
  }
}
```

Figure 3: The method contains for lists of books and authors

The two methods differ only in type of argument consumed by the method *contains*. If we replace the type by *Object*, the super type of both *Book* and *Author* everything is almost OK. The only problem is that the class *Object* does not provide the method *same*. Not only that. The method argu-

ment in the two cases is again different.

We need a method

```
boolean same(Object obj)
```

that is defined for any object we wish to look for in the list. We can do this for the classes *Book* and *Author* by having both classes implement a common interface *ISame*:

```
interface ISame{
  // is this the same as the given object?
  boolean same(Object obj);
}
```

We will first look at how this knowledge is leveraged inside of the definition of the method *contains*, then return to completing the definitions of the classes *Book* and *Author*.

We first have to indicate inside the method *contains* that the field *this.first* is an instance of a class that defines the method boolean *same*(*Object obj*). To do so, we use *cast*:

```
  boolean contains (Object that){
    return ((ISame)this.first).same(that)
        || this.rest.contains(that);
  }
```

We enclose in parentheses the type we wish to *cast* to, and enclose in another set of parentheses the cast directive and the expression that should be cast to that type: ((*ISame*)*this.first*). The entire expression now represents a value of the type *ISame* and can invoke any methods declared in the *ISame* interface. The compiler will replace then at run time look for a method with the signature boolean *same*(*Object obj*). If we construct the list with an object that does not implement the *ISame* interface, the program will fail at the runtime with the *ClassCastException*.

We now look at how the method boolean *same*(*Object obj*) is implemented in the classes *Book* and *Author*:

```
class Book implements ISame{
  String title;
  int year;

  Book(String title, int year){
    this.title = title;
    this.year = year;
  }

  boolean same(Object obj){
    if (obj instanceof Book)
      return this.same((Book)obj);
    else
      return false;
  }

  boolean same(Book that){
    return
      this.title.equals(that.title)
    && this.year == that.year;
  }
}
```

```
class Author implements ISame{
  String name;
  int year;

  Author(String name, int year){
    this.name = name;
    this.year = year;
  }

  boolean same(Object obj){
    if (obj instanceof Author)
      return this.same((Author)obj);
    else
      return false;
  }

  boolean same(Author that){
    return
      this.name.equals(that.name)
    && this.year == that.year;
  }
}
```

We have two different methods named *same* overloaded with two different types of arguments. Inside the method *contains* the expected method for comparison only knows that the argument is of the type *Object*. As the result, the compiler invokes the method *same*(*Object obj*). Each of the two classes, *Book*/*Author* then first verifies that the argument is indeed another *Book*/*Author*, and if true, it invokes the original *same* method that compares the corresponding fields.

The tests are the same as before:

```
class Examples{
  Examples () {}

      Book b1 = new Book("DVC", 2003);
      Book b2 = new Book("LPP", 1942);
      Book b3 = new Book("HtDP", 2001);

      ILoObject mtbooks =
                new MTLoObject();
      ILoObject booklist =
                new ConsLoObject(b1,
                new ConsLoObject(b2,
                                 mtbooks));

      boolean testSizeBook1 =
        mtbooks.size() == 0;
      boolean testSizeBook2 =
        booklist.size() == 2;

      boolean testContainsBook1 =
        this.mtbooks.contains(this.b1) ==
          false;
      boolean testContainsBook2 =
        this.booklist.contains(this.b2) ==
          true;
      boolean testContainsBook3 =
        this.booklist.contains(b3) ==
          false;
```

```
      Author a1 = new Author("DB", 1956);
      Author a2 = new Author("StEx", 1900);
      Author a3 = new Author("MF", 1972);

      ILoObject mtauthors =
                new MTLoObject();
      ILoObject authorlist =
                new ConsLoObject(a1,
                new ConsLoObject(a2,
                                 mtauthors));

      boolean testSizeAuthor1 =
        mtauthors.size() == 0;
      boolean testSizeAuthor2 =
        authorlist.size() == 2;

      boolean testContainsAuthor1 =
        this.mtauthors.contains(this.a1) ==
          false;
      boolean testContainsAuthor2 =
        this.authorlist.contains(this.a2) ==
          true;
      boolean testContainsAuthor3 =
        this.authorlist.contains(a3) ==
          false;
```

The class diagram for our complete solution is shown in figure 4:
We summarize what we have learned:

- We replace the list of *Book*, *Author*, or some other class by a list of *Object*. We refer to the class *Book*, or *Author*, etc. as the *target* classes and to the classes that implement the list structure as list classes.

```
interface ILoObject
class MTLoObject implements ILoObject
class ConsLoObject implements ILoObject
```

- If a method in the list class hierarchy requires an argument of the *target* type, we replace if with the argument of the type *Object*.

```
interface ILoObject{
  boolean contains(Object obj);
}
```

- If the method in the list classes invokes a method that is to be implemented by our *target* class, we proceed as follows:

```
class CosnLoObject ... {
  boolean contains(Object obj){
    ... this.first.same(that) ...
  }
}
```

  - Define an common interface (we will call it *Icommon*) that contains only the desired method. If the method consumes an object of the type of the *target* class, change it to the type *Object*.

```
interface ISame {
  boolean same(Object obj);
}
```

  - Make every *target* class implement the *Icommon* interface. If the method consumes an object of the *Object* class that previously was of the type *target* class, the method will have two clauses:
    * If the argument is *instanceof* the *target* class, invoke the original method, casting the argument to the *target* class. The method is now overloaded — with two variants of the argument type - the original one of the *target* type and the type *Object*.
    * Else, the method invocation is invalid and produces an error. You decide whether it should just be false for methods that produce boolean value, or whether the method should **throw** an exception (probably the *ClassCastException*.

```
class Book implements ISame{
...
  boolean same(Object obj){
    if (obj instanceof Book)
      return this.same((Book)obj);
    else
      throw new ClasCastException(
        "Cannot compare a Book with other object");
  }
...
}
```

  - In the list classes cast the *target* object to the type of the *Icommon* interface.

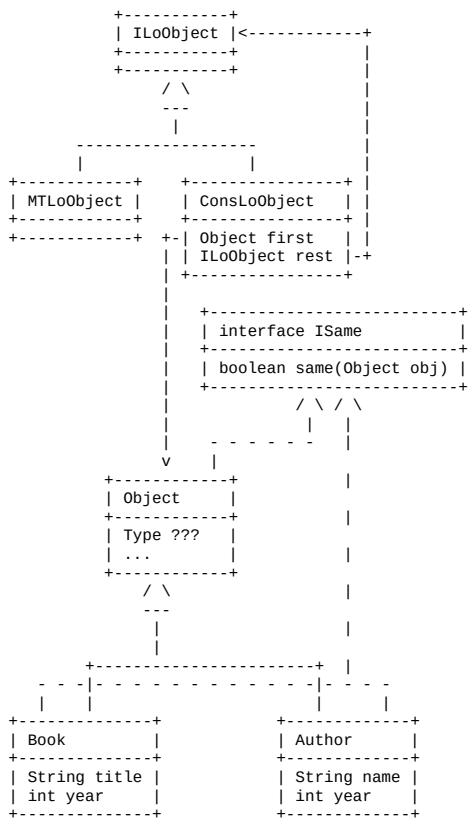


Figure 4: A class diagram for a list of objects