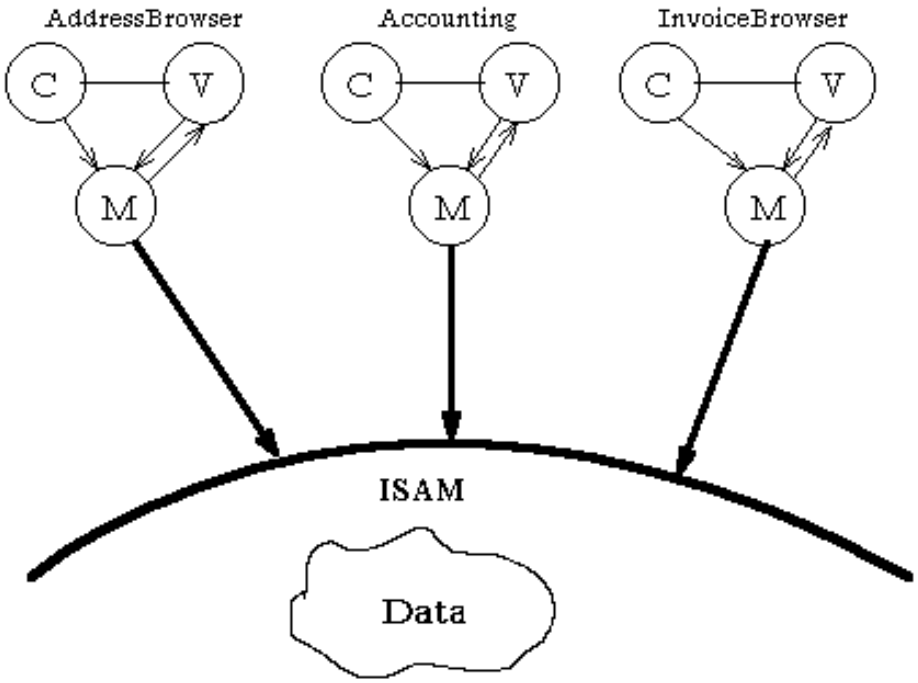
ISAM Toolbox
Programmers' Manual
Version 7.7.1



ISAM Toolbox is a VisualWorks contribution of Georg Heeg eK

Georg Heeg eK
Baroper Str. 337
44227 Dortmund
Germany

info@heeg.de
Phone +49 231 9 75 99 0
Fax +49 231 9 75 99 20

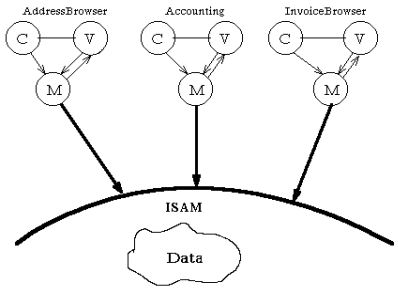
www.heeg.de

1.0 Introduction

One basic key towards integration of applications is the issue of data management. Applications of different kinds must be able to work with the same data without the need of knowing the structure of the other applications.

The data management facilities introduced by the Collection classes to Smalltalk80 allow either 'in core' collections of arbitrary complex data or collections of 'simple' data like characters or bytes on files. However, real life-size applications require data management facilities that can hold complex data on files. This leads to ad-hoc solutions which are both inefficient and non-portable.

One commonly known solution of this issue is the introduction of data management through indexed files called the **I**ndexed **S**equential **A**ccess **M**ethod **ISAM**. These ISAM files support portability, efficient memory utilization and data security.



2.0 Using ISAM

In this chapter we introduce the programming interface to ISAM. We do not list the complete implemented protocol but only the part that will be used by an applications programmer. Other protocol defined in the ISAM classes is likely to change between versions and should not be used by applications.

2.1 How Works ISAM?

In general ISAM collections define several orderings (called **indices**) on the same data which may be accessed separately. These indices are implemented by sorted collections that store **keys** which hold some relevant portion of data from the **records**. The records are normally stored in a file. The application may switch between the defined indices and may use them like ordinary collections.

2.1.1 What means ISAM?

ISAM is a short hand for Indexed Sequential Access Method which means that it allows both indexed and sequential access of data

Indexed means, that one can access the records through some known portion of data, the key.

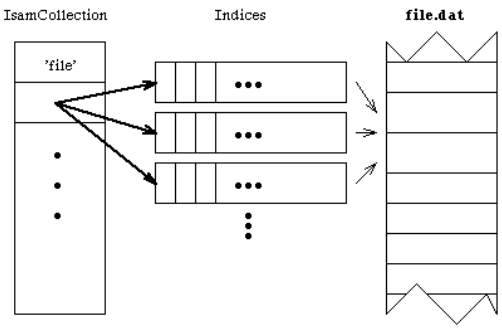
Sequential means that it is possible to access the data sequentially, i.e. one after the other.

The ordering of the records may be either physical or logical. A logical ordering is given by the used key and the physical ordering is given by the sequence of the records in the records file.

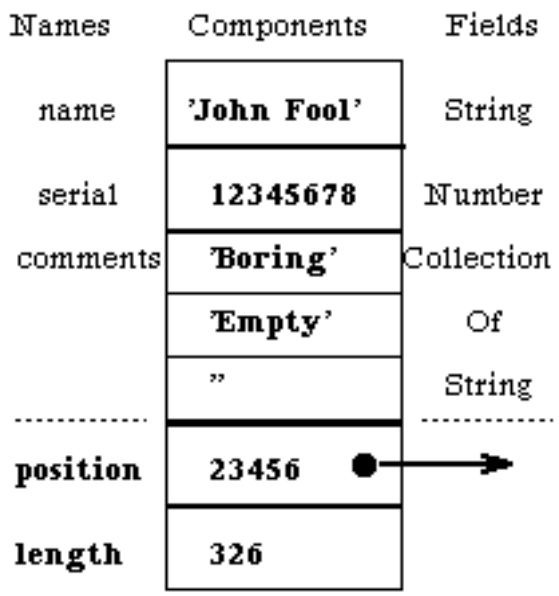
Example: Consider records made up from a name, a serial number and a collection of comments. Keys which hold the name part of such records order them by this name part.

2.1.2 Organization of Keys and Records in ISAM

The ISAM data is organized by an instance of **IsamCollection**. This IsamCollection implements the interface to applications. It contains a dictionary of indices associated with their key classes. Each index (which is an instance of **IsamIndex**) contains one key instance of its key class for each of the records in the records file. Thus all indices contain the same number of keys. The keys point to positions of their records in the records file.



The keys and records are instances of their key and record classes. These classes are subclasses of **IsamItem**.



IsamItems contain the record position and length and named **components** which are typed through **fields**. The fields are installed upon creation of the key or record class and are responsible for type checking, reading and writing of instance components.

For a more detailed description of these classes see chapters 2.2 for IsamItem and 2.3 for IsamCollection.

2.1.3 Developing an ISAM Application

As pointed out in the introduction to this chapter, IsamCollections work like ordinary collections. Thus they implement collect, detect, select, reject and so on. All these accessing methods work for both the currently selected index and for the collection of records. Furthermore IsamCollections allow indexed access to record data through a key.

These similarities to ordinary collections support rapid prototyping. The application can be developed and tested with ordinary collections and may be adapted to ISAM afterwards. The major difference to ordinary collection is the need for typed keys and records. Therefore the design of keys and record classes is an important step while developing an ISAM application.

There are some points of interest in designing the fields and the access to the indices. As the indices are kept 'in core' while the IsamCollection is in use, one should carefully select the fields of the records that should be kept in keys. Creating too large keys result both in inefficient memory utilization and bad performance.

Another constraint on key design is the type of application. For instance an application that offers a browser on most of the record fields requires keys for each of the browser list views.

The developer should find a suiting balance between keeping fields in keys and accessing the record for fields. The both counterpoints are defining keys for all fields (thus keeping the all records 'in core') and accessing the records through the record file only.

2.2 Implementation of Keys and Records

2.2.1 Declaring an IsamItem

All record and keys used in IsamCollections have to be declared as subclasses to IsamItem. The declaration is simply done like subclass declaration. However, instead of instance variables the user declares the names and types of the fields. The name and type must be separated by a '**=**'.

Example:

```
Heeg.Isam defineClass: #DemoRecord
        superclass: #{Heeg.Isam.IsamItem}
        indexedType: #objects
        private: false
        instanceVariableNames: ''
        classInstanceVariableNames: ''
        imports: ''
        category: ''
        attributes: #(#(#fields
                'name=PlatformString
                serial=Integer
                comments=CollectionOfString '))
```

In subclasses of such record or key declarations the types of the fields must be identical to the types of the fields with the same name in the superclass.

Example:

```
Heeg.Isam defineClass: #DemoKey
        superclass: #{Heeg.Isam.DemoRecord}
        indexedType: #objects
        private: false
        instanceVariableNames: ''
        classInstanceVariableNames: ''
        imports: ''
        category: ''
        attributes: #(#(#fields 'name serial '))
```

This declares a key to DemoRecord that holds the name and serial number only. The types are PlatformString and Integer as in DemoRecord.

Once the declaration is accepted the accessing and type check methods are automatically implemented. For maintenance purposes some class variables are declared which are filtered out in the browser (you won't see them but they exist).

Please note that for this special treatment of instance variable declaration no normal instance variables can be declared in subclasses of IsamItem. This would make no sense anyway, as only the components are written to a file upon closing the IsamCollection.

It is not enforced that key classes must be subclasses of their record classes and it is not prohibited to implement additional protocol to the record and key classes. However, do not change the automatically generated protocol unless you really know what you do. The least hazard that may happen to you is that this protocol is thrown away upon recompiling the class.

2.2.1.1 Implemented Types

Each component of a record or a key has an associated subclass instance of IsamField which describes the type of the component. These IsamFields have subclasses for each possible field type and have the following generic names: Isam<Typename>Field, i.e. **IsamStringField**.

The following types are implemented :

Integer	any Integer
Text	any Text with string encoded in MS CP 1252
UTF8Text	any Text with string encoded in UTF8
String	any String encoded in MS CP 1252
Symbol	any Symbol based on PlatformString
UTF8String	any String encoded in UTF8
UTF8Symbol	any Symbol encoded in UTF8
CollectionOfString	any number of strings encoded in MS CP 1252
CollectionOfUTF8String	any number of strings encoded in UTF8
CollectionOfInteger	any number of Integers

There might be other type fields declared after this manual was created. Check all subclasses of IsamField for a complete list of component types.

There are no limitations for the size of those fields or the number of elements in the collection fields except the available memory.

We recommend using the UTF8 classes for any future development. Non UTF-8 classes are basically for backward compatibility. Be aware ISAM Toolbox is more than 20 years old and there are databases life with ISAM toolbox since those days.

2.2.2 Creating an IsamItem

Once the records and keys are declared one can create empty instances with the message **new**. This method does an initialization of the IsamItem, so if you would like to initialize the record, add your initialization code to the class.

2.2.3 Using IsamItems

Normally records and keys are created from scratch and the application fills in the values for the components or the application requests an item from the IsamCollection which returns a fully specified item. The user does not need to hassle with the items except for accessing the items components.

The following sections describe some useful messages that each IsamItem understands.

2.2.3.1 Accessing Components

After compiling the declaration of the record or key the following methods for accessing the components are implemented:

For each field exist a method named like the field to read the component and a method named like the field with one argument to write the component. While writing a value in the component a type check is done to assure database integrity.

Example:

The following access messages are implemented for the DemoKey above:

name
name: aString

serial
serial: aNumber

2.2.3.2 Copying Components

These messages may be used to copy components between items.

copyPositionFrom: anItem
copies the record position in the record file and the record length to self.

copyFrom**: anItem**
copies all those components from anItem to self that are declared in my class. Provide an error notifier if anItem does not contain all my fields.

copyTo: anItem
copies all my components to anItem. anItem must contain all my fields.

copyAllFrom: anItem
copies both the position and the components from anItem

Example:

*Using the above declaration of DemoRecord and DemoKey the statement **aDemoKey copyFrom: aDemoRecord** copies both the components name and serial from the demo record to the demo key*.

2.2.3.3 Comparing IsamItems

IsamItems are ordered lexically over their components in the order they were declared. If two keys or records share the same components then their record position (if given) determines their ordering.

IsamItem implements two different kinds of comparison. The first kind fails to compare two items if they are not of the same class. The second kind tries to compare components common to both items.

The following methods compare items of the same class:

< anItem
> anItem
= anItem

The following methods compare items of different classes. All fields in the receiver must be shared by the argument.

less: anItem
greater: anItem
equal: anItem

*Example: An instance A of **DemoKey** is **less:** an instance B of **DemoKey** if*

a. *name of A is less than name of B or*
b. *name of A is equal to name of B and serial of A is less than serial of B.*

2.3 Programming

Once the record and key classes are declared, a new instance of IsamCollection may be created that manages instances of the record class. One can add and delete indices associated with the new key classes and of course one may add, delete and retrieve records.

2.3.1 Maintaining an IsamCollection

The usual tasks are to create a new IsamCollection and to load and save ISAM data to files.

2.3.1.1 Creating an IsamCollection

To create a new IsamCollection one has to specify the filename of the files, the record class and a collection of key class symbols. The records file gets the extension **.dat** and the index file gets the extension **.idx**.

The first key class given will be called the primary index. This is the default index and cannot be erased.

IsamCollection on: filename records: aRecordClass keys: aCollectionOfKeyClassSymbols
creates an instance that stores instances of aRecordClass in a binary file named filename.dat and has indices for each of the key classes given. The primary key class given will be selected.

Example:

demoCollection := IsamCollection on: 'demo' records: DemoRecord keys: #(DemoKey***)***
creates an IsamCollection that stores instances of DemoRecord and has one index over DemoKey instances.

2.3.1.2 Saving and Writing

loadFrom: filename
reads the index data from a file filename.idx, opens the record file filename.dat and selects the primary index. If the index file is invalid then the indices are rebuild from the records (see chapter Security). This may take some time.

saveTo: filename
saves the index data to the file filename.idx and closes the record file filename.dat. The record data is copied if the old and the new filename differ.

close
closes the record file.

2.3.1.3 General Accessing

The special handling of the records file leads to an increasing amount of inaccessible file space which was occupied by old records.

streamFragmentation
returns the number of unused bytes.

fragmentation
return the unused / used bytes ratio.

compact
compresses the records file by removing the inaccessible records. This method should not be interrupted! A temporary copy of the compressed records file might be found in **compact.tmp** in the current directory.

size
returns the number of records stored in the records file.

2.3.2 Indices

The indices store keys for each record in the record stream. One may add or remove such indices and access these defined indices.

addIndex: keyClass
This method defines a new index over the records with keys of keyClass. For each currently available record one instance of keyClass is added to this new index. Thus adding an index to a large IsamCollection may take quite a while. The new index does not become the current index.

removeIndex: keyClass
This method removes the index associated with keyClass from the IsamCollection. The current index can be removed but vanishes only after a change of the current index. The primary index cannot be erased.

These methods can be used to access and select existing indices.

as: keyClass
Returns a copy of the IsamCollection with a new current index associated with keyClass.

currentIndex
Returns the sorted collection of keys for the current index. This should be used very carefully as changes to just one of the indices will probably crash ISAM.

indexKey
Returns an empty key instance for the current index.

selectedIndex: keyClass
Makes the index associated with keyClass the new current index.

2.3.3 Safety

The subject of safety is very important for database systems. It is even more important for ISAM Toolbox as the indices are kept 'in core' while the changing record data is updated on a file. Therefore the record data written to the record file contains validation information and records are always appended to the record file. This assures that the indices can be recovered from the record file.

Furthermore IsamCollection carries an **inFlux** flag which signals that ISAM is in a critical operation. If the **inFlux** flag is detected then some operation was not finished properly and the index data may be corrupted.

2.3.3.1 Crash Recovery

The indices may be rebuilt from the record information with the method **rebuildIndices**. This method scans the complete record file and adds keys for each valid record found in this file.

2.3.3.2 Integrity Checks

There are two increasing possibilities to check the system integrity.

checkIndicesForOverlappings
: This method checks the indices for overlapping. This means test if one of the keys overlaps with one of the other keys.

checkIntegrity
: This method checks the indices for overlapping and then checks the key components if they contain the same information as their records. This check takes some time but if it succeeds you can be sure that all your records are accessible and no record will be (partially) damaged when changing other records.

2.3.4 Using Keys and Records

In the following chapters item denotes either a full specified record or a matching key, record denotes a full specified record (instance of the record class declared in the creation of this IsamCollection) and key denotes an instance of either the record class or an index key class which is full specified (has all components filled with valid data) and has a valid record position attached.

2.3.4.1 Adding and Removing

Adding and removing of data is only admissible for records. The indices must not be changed separately as they all must contain the same number of keys with references to the same collection of records.

add: aRecord
: This method adds the record to the record stream and adds instances of the key classes to the indices.

rewrite: aRecord
: This method rewrites the record. This means it first removes the old version and then adds the new one.

rewrite: item with: aRecord
: Rewrites the record specified by item with aRecord.

remove: item ifAbsent: aBlock
: Removes the record specified by item. If no record is found, evaluate aBlock.

2.3.4.2 Accessing

As explained in the introductory chapters, ISAM allows both indexed and sequential access. The following two chapters describe both access methods implemented in ISAM Toolbox.

2.3.4.2.1 Indexed Accessing

These methods implement indexed access to records and keys in the IsamCollection.

keyAt: item
: Determines which index corresponds to item and searches the matching key in this index. Returns nil if no index or no key was found.

keyAtPosition: position
: Searches in the current index for a key with the given record position. Return nil if no such key was found.

keyOf: index atPosition: position
Searches the collection of keys index for a key with the given record position. Return nil if no such key was found.

allKeysAt: item
Returns all keys that match item and have different record positions.

recordAtKey: key
Returns the record associated with key.

recordAtPosition: position
Returns the record at the position in the record stream.

recordAt: item
This is the most general method to access a record by some portion of data. If item contains a record position then read the record directly. Otherwise search for a key matching item. If one was found read its associated record. If none was found (item has no matching index or no matching key) then search all records sequentially for this item. If nothing was found, return nil.

allRecordsAt: item
Returns all records that match item.

includesItem: item
True if the IsamCollection contains a record that matches item.

Normally the developer will need only recordAt: . He has either some data for which he needs a matching record or he has a key (gained from enumerating an index for instance) and wants to read its associated record.

2.3.4.2.2 Sequential Accessing

IsamCollections can be enumerated like ordinary collections. However, one may enumerate over the current index or over the whole collection of records (which is rather costly).

The following methods are implemented. Please note that some more standard enumeration methods work but are implemented in the class Collection.

The species of IsamCollection is OrderedCollection which means that the collect, select and reject methods return instances of OrderedCollection.

do: aBlock
Enumerate over the current index. This method serves as a 'primitive' for all the standard enumeration methods.

recordsDo: aBlock
Enumerate over all records. Note! this method reads all records from the record stream in the logical order defined by the current index.

collectRecords: aBlock
Collect over all records.

selectRecords: aBlock
Select over all records.

rejectRecords: aBlock
Reject over all records.

3. A Useful Example

This chapter describes a small but useful example application that is based on ISAM.

The application is a small bibliography which stores the entries and their keywords in an ISAM database.

3.1 BibInf Manual

3.1.1 What does BibInf ?

BibInf stores bibliography notes in an ISAM database.

The notes are organized with categories and may be searched by title, author and year. They are entered in plain text and parsed by the application.

Example:

John Fool
My boring life without ISAM
Junk Press Publishers
Lowdown, 1989
This book is not worth reading

3.1.2 About Entries and Categories

An entry consists of a title, the author's name, the year, the publisher and the place where the paper/book was published. Furthermore it may contain an arbitrary length comment and an arbitrary number of categories.

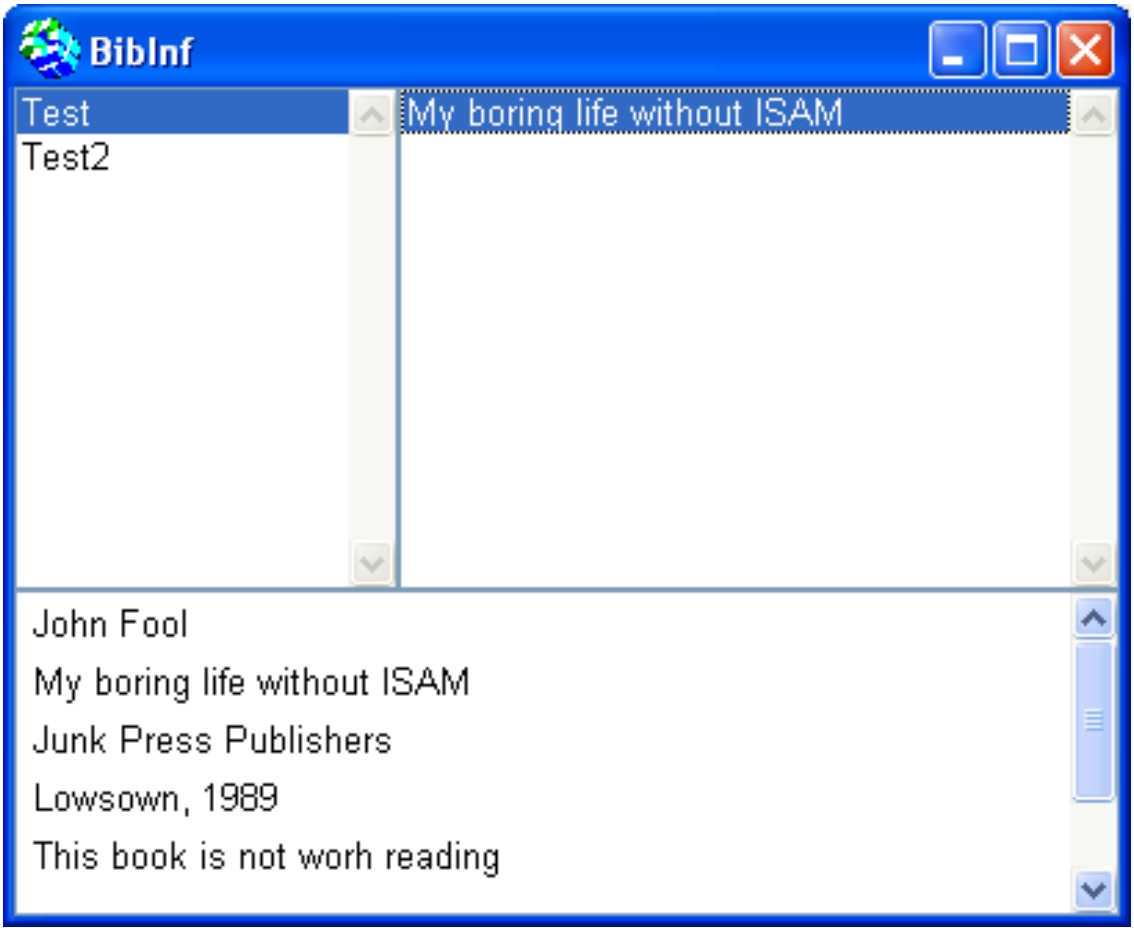
These categories are simple texts that structure the collection of entries (like class and method categories).

3.1.3 Views and Menus

The BibInf application may be started with

BibInf open: filename.

This opens a small browser with three sub views.



The first sub view contains the categories, the second the entries stored under the selected category and the third is used for entry editing.

3.1.3.1 Category View

The category view is used to view, add and delete categories. Additionally it contains menu entries for loading and saving the ISAM data.

- **add category** This adds a new category to the ISAM data..
- **remove category** This removes the currently selected category.
- **load** Loads the data from a user specified file.
- **save** Writes the current data to a user specified file.
- **inspect** This opens an inspector on the **IsamCollection**.

Note that ISAM appends the suffix **.dat** and **.idx** to the filename given above. If your file system supports only one suffix per filename (e.g. MS-DOS or Atari ST) then don't enter a filename with a suffix.

3.1.3.2 Entry View

The entry view is used to display, delete and search entries.

- **show title** This shows the title of the selected entries only.
- **show authors** This shows the authors names of the selected entries.
- **show all** This shows the authors' names, the title and the year.
- **remove** This removes the selected entry.
- **search title** This asks the user for a search string (with * and # wildcards) and displays the found entries with matching titles in the entry view.
- **search author** This searches for matching author names.
- **search year** This searches for matching years.
- **add to category** This adds the selected entry to another category.

3.1.3.3 Input View

The input view is a **TextView** with the normal text editing features.

The entry format is:

Author
Title
Publisher
Place, Year

The rest of the entry is stored as a comment and is not parsed.

3.2 Objects used in BibInf

The first step in designing an ISAM application is to develop an idea what records and keys are needed.

The BibInf application certainly needs a record to hold all components of an entry. Additionally it needs a key which stores the author's name, the title of the entry and his year as these are the parameters used in queries.

To access the records through their categories one needs a key which holds all categories of an entry.

```
Object ()
        IsamItem ('recordPosition' 'recordLength' )
                BibEntry ()
                        BibCategoryKey ()
                        BibKey ()
```

3.2.1 BibEntry

The records of BibInf are instances of class BibEntry.

```
Heeg.Isam defineClass: #BibEntry
        superclass: #{Heeg.Isam.IsamItem}
        indexedType: #objects
        private: false
        instanceVariableNames: ''
        classInstanceVariableNames: ''
        imports: ''
        category: 'Bibliography'
        attributes: #(#(#fields
                'title=PlatformString authors=PlatformString year=Integer
                publisher=PlatformString place=PlatformString
                comment=PlatformString category=CollectionOfString '))
```

- BibEntry knows how to parse an entry. This is done with the methods in category parsing.

3.2.2 BibKey

The instances of BibKey are used to access the search and display criteria of records.

```
Heeg.Isam defineClass: #BibKey
        superclass: #{ Heeg.Isam.BibEntry}
        indexedType: #objects
        private: false
        instanceVariableNames: ''
        classInstanceVariableNames: ''
        imports: ''
        category: 'Bibliography'
        attributes: #(#(#fields
                'title=PlatformString authors=PlatformString
                year=Integer '))
```

3.2.3 BibCategoryKey

The instances of BibCategoryKey are used to access the category component of records.

```
Heeg.Isam defineClass: #BibCategoryKey
        superclass: #{Heeg.Isam.BibEntry}
        indexedType: #objects
        private: false
        instanceVariableNames: ''
        classInstanceVariableNames: ''
        imports: ''
        category: 'Bibliography'
        attributes: #(#(#fields 'category=CollectionOfString '))
```

4. Adding new Types

An IsamField describes one of the fields (components) in an IsamItem. It is used for type-checking the component and for reading and writing the component in a binary format.

To add a new type of IsamItem fields, define a subclass of IsamField. The name must be of the form Isam<typename>Field (e.g. IsamIntegerField) where typename is the name that is given in an IsamItem component declaration after the '='.

Currently the following subclasses of IsamField exist:

IsamField (abstract)
- **IsamIntegerField**
- **IsamStringField (abstract)**
  - **IsamPlatformStringField**
  - **IsamUtf8StringField**
- **IsamTextField**
- **IsamCollectionField (abstract)**
  - **IsamCollectionOfStringField**
  - **IsamCollectionOfIntegerField.**

After it is declared, the new type may be used in any IsamItem declaration.

The following protocol must be implemented by every IsamField.

check: value
Return true if value is valid for this type of field.

size: value
Return the size (in bytes) of the binary representation of this value.

readFrom: byteStream
Read a value of my type from the stream.

write: value to: byteStream
Write the value to the byte stream.

The following methods are predefined to support new IsamItem field types.

readIntegerFrom:

readStringFrom:

writeInteger:to:

writeString:to:

IsamFields that describe types for which no simple comparison **=**, **<** and **>** exist should redefine **equal:to:**, **greater:than** and **less:than:**. This is done in the collection fields for instance.

For a description of the used binary format see the next chapter.

5. Data-Formats

5.1 Binary Formats

These are the formats of the binary representations of values and items.

5.1.1 Item Format

The IsamItem subclass instances have the following binary representation:

1. a validation-tag
2a. recordfile position
2b. item length
3. the binary representations of the components.

The validation tag is a single byte which indicates if the found record is in use or obsolete. See class variables **ValidTag** and **InvalidTag** of class **IsamItem** for the current bitmask.

The values 2a and 2b are used in keys in the index file only and are not present in records. They are written as binary representations of their integer value.

5.1.2 Binary Formats of Fields

View this list of field formats (binary representations of the components) as preliminary. A full list of formats may be extracted from the subclasses of **IsamItem**.

5.1.2.1 Integer

Format for an integer:

1. byte : sign and size
2. - n-th byte: the bytes of the integer.

The first byte is positive for a positive integer and negative for a negative integer. The other 7 bits denote the size of this integer, which is the number of bytes. (e.g. a number less than 2^8n has size n) Thus integers are restricted to 2^ 1016 (127 bytes)

5.1.2.2 PlatformString

A PlatformString consists of the binary representation for its size and its bytes.

5.1.2.4 Text

A Text consists of the binary representations of the sizes of the runs and the values (should be the same normally) and the binary representations of the bytes of the runs and the values. Thus a text of n characters has a binary representation with 2n+2 integers.

5.1.2.5 Collections

A collection is simply made up of the binary representation of the size of the collection and of the binary representations of the elements of the collection.

5.2 Index File

The index file consists of the binary representations of the following components:

1. Name of the record class
2. The name of the primary index class
3. The first free position in the records file
4. The fragmentation in bytes of the records file
5. Every index consisting of
   5.1. Name of the key class
   5.2. Size of the index
   5.3. Each key

5.3 Record File

The records file is a sequence of records (written without record position and length). These records might be marked as invalid and are skipped when rebuilding the indices. The number of bytes occupied by invalid records is recorded in the variable **streamFragmentation**. These invalid records are not reused and must be reclaimed with **compact** occasionally. ISAM always appends new records to the records file. To avoid frequent extensions of the records file (which are expensive) ISAM pre-allocates some disk space for the records. The variable **streamEnd** points at the first unused byte in this free space.

This odd format serves two purposes. First it is much faster than keeping a free list of unused space in the records file and extending the file frequently. Second it is safer. Common data crashes we experienced while working with former versions of ISAM were:

- overlapping records. This is now impossible as records always use new space.
- lost records when the machine crashed and the records and directory entries were not flushed. In the current version we overwrite pre-allocated records only.
- invalid index structure due to interrupted changes. This is checked with the **inFlux** flag.
- inaccessible records due to lost index data. Now the indices are obsolete and may be rebuilt from the records when needed.

6. Historical Remarks

ISAM Toolbox has been developed in the late 1980ies on Smalltalk-80 version 2.3. It first use was on the Atari platform. It has been designed to be fast and stable as Atari computers were very slow (68000 processor with 8 MHz clock rate) and had no hard disk buffering, RAID or other modern features.

Used for more than 20 years, no loss of data has been reported.