Announcements

- Assignment 5 has been posted

Review

- Variables whose type is a class are references
- Alias of a variable is a different variable which refers to the same object
- Aliases can be used to represent shared references/shared information
- Question: When are two references (of the same type) equivalent?
- Reference equality
  - Pointer equality
  - Structural equality
    - Shallow
    - Deep

Review

- Two references are *pointer-equal* if they are aliases (i.e. they refer to the same object)
- Two references are *structurally-equal* if the attributes of the objects they refer to are equal
  - Two references are *shallowly-equal* if the attributes of the objects they refer to are pointer-equal
    * Two references are *deeply-equal* if the attributes of the objects they refer to are structurally-equal
- Copying or cloning: creating a *different* (i.e. not pointer-equal) object which is structurally-equal to the original
  - Copy $\left\{ \begin{array}{l} \text{Shallow} \\ \text{Deep} \end{array} \right.$
  - A copy is shallow if it is shallowly-equal to the original
  - A copy is deep if it is deeply-equal to the original

Parameter passing by reference vs. by value

- A programming language that has methods, procedures, and/or functions can pass arguments to the function in several ways:
  - Passing parameters by value: The arguments received by the function are a copy (usually shallow) of the actual arguments.
  - Passing parameters by reference: The arguments received by the function are aliases of the actual arguments.

- In Java, primitive data types are passed by value, but all user-defined data types are passed by reference.

Passing parameters by reference

```
class A {
  int x;
}
class B {
  static void m(A u)   // By ref
  {
    u.x++;
  }
  static void t(int x)   // By val
  {
    x++;
  }
}
```

Passing parameters by reference

```
class Test {
  void p()
  {
    A q = new A();
    q.x = 3;
    B.m(q);
    System.out.println(q.x);
    B.t(q.x);
    System.out.println(q.x);
  }
}
```

The null reference

- A variable whose type is a class is initialised to `null`.

- If a variable whose type is a class is not assigned an object (constructed with new,) and we try to access its attributes or methods, then a run-time error, called a "null-pointer exception" will occur.

- In the following example, if method `r` is called, a null pointer exception will occur:

```
class A { int x; }
class B {
  static void p(A u)
  {
    u.x = 7;   // Null pointer exception
  }
  static void r()
  {
    A v;  // v == null
    p(v);
  }
}
```

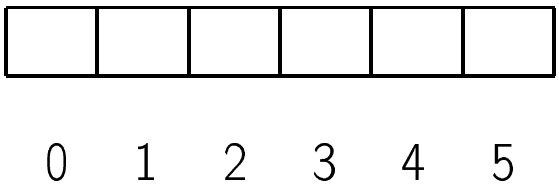
The null reference (contd.)

- We can avoid these errors by using an explicit check for a valid reference:

```
class A { int x; }
class B {
  static void p(A u)
  {
    if (u != null)
      u.x = 7;
  }
  static void r()
  {
    A v;  // v == null
    p(v);
  }
}
```

Arrays

- An *array* is an indexed sequence of variables of the same type. By indexed we mean that the variables are consecutive in memory and each of them has an index, with 0 being the first, 1 the second, and so on.



- Each variable in the array is called a *position*, a *cell* or a *slot*, and as any variable, it can contain a value.

- Arrays are declared as follows:

  type `[]` *name* `;`

- Where *type* is any data type (primitive or user-defined).

Arrays (contd.)

- For example an array of integers called `mylist` which is declared as

```
int[] mylist;
```

- In an array declaration *type[]* is the type of the array, and *type* is its *base type*. (An array of integers is not the same as a single integer.)

- Arrays can have as base type a class.

- For example, if we have a class Mouse then an array of mice is declared as:

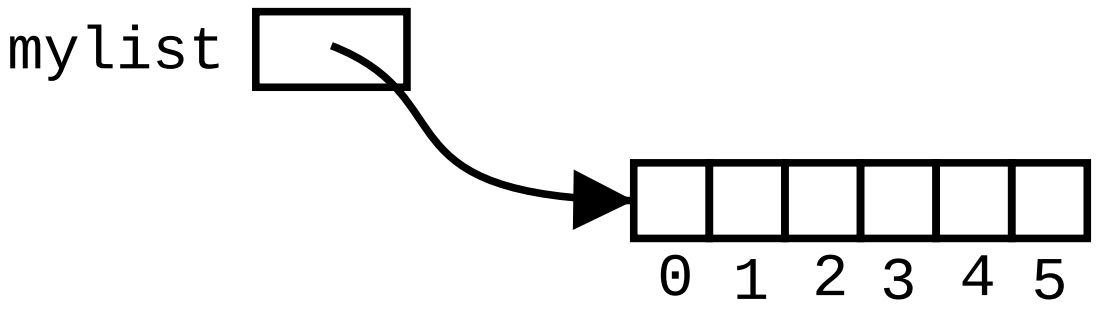
```
Mouse[] mouse_list;
```

Arrays (contd.)

- But declaring an array does not create the array itself, only a reference.
- To create an array we use the new keyword.

```
mylist = new int[6];
```

- Where the variable mylist is actually a reference to the aray itself



Array access

- To access individual elements of an array we use the indexing operator [.]: If variable is a reference to an array, and *number* is a positive integer, or 0, then the position *number* can be accessed by

  variable[*number*]

- For example `mylist[0]` refers to the first position of mylist, `mylist[1]` to the second, `mylist[2]` to the third, and so on.

- To write a value in the array, we can use the assignment operator:

  variable[*number*] = *expression*;

- Where *expression* must be of the same type as the base type of the array.

Processing arrays

- Processing arrays is a generalization of processing strings.
- `a[i]` is analogous to `s.charAt(i)`, but only for reading the `i`-th, not for writing: `charAt` cannot be used for modifying a string. This is: `s.charAt(i) = expr;` is illegal syntax.
- Use loops to traverse an array.
- The length of an array `a` can be obtained by the expression `a.length`
- This is independent of the number of slots that hold a value

Example 1

- Finding the minimum number in an array

```
static double find_min(double[] a)
{
  int index;
  double minimum;
  index = 0;
  minimum = 999999999.9;
  while (index < a.length) {
    if (a[index] < minimum) {
      minimum = a[index];
    }
    index++;
  }
  return minimum;
}
```

Example 2

- Returning the index where the minimum is located

```
static int find_min(double[] a)
{
  int index, min_index;
  double minimum;
  index = 0;
  min_index = 0;
  minimum = a[0];
  while (index < a.length) {
    if (a[index] < minimum) {
      minimum = a[index];
      min_index = index;
    }
    index++;
  }
  return min_index;
}
```

Processing arrays: safety

- Since arrays are references, it is often useful to check whether they are null or not before using them, to avoid null-pointer exceptions.

- If the array has as base type a class, it is also useful to check that each slot which will be processed or accessed is not null.

- For example:

```
class A { int x; }
class B {
  static void m(A[] list)
  {
    if (list != null) {
      for (int i = 0; i < list.length; i++) {
        if (list[i] != null) {
          list[i] = 2 * i;
        }
      }
    }
  }
```

Initializing arrays

- If we have a class

```
class B {
  int n;
  B(int x) { n = x; }
}
```

- and somewhere else we declare and create an array

```
B[] list = new B[7];
```

- Then all the slots in the array will be initialized to `null`. This is, the constructor for `B` will not be called. If we want an object created in each slot, we have to do it explicitely:

```
for (int i=0; i < list.length; i++)
  list[i] = new B(3);
```

Initializing arrays

- Arrays can be initialized with default values using the syntax:

```
type[] var = { expr1, expr2, ..., exprn };
```

Where each *expri* is of type *type*.

- For example:

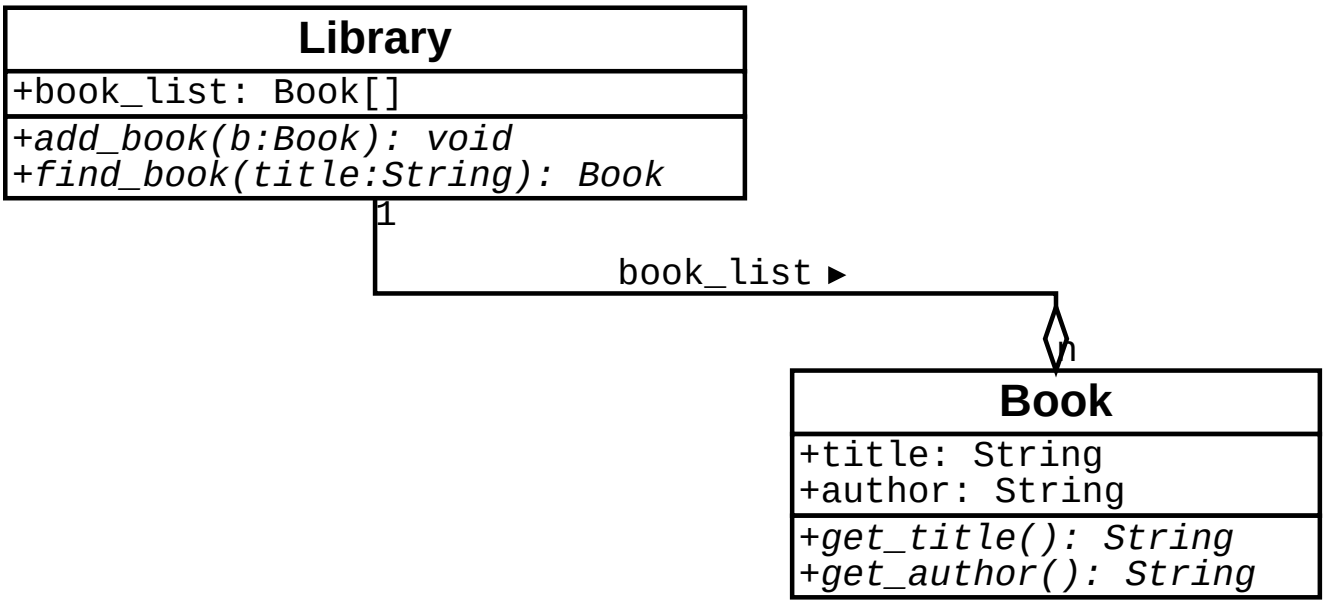
```
int[] a = { 1, 1, 2, 3, 5 };
Z[] u = { new Z(), new Z() };
```

Array applications

- Library: Book database
- Problem: Create a database of books, which supports the operations of adding a new book, and searching for a book by title.
- Analysis:
  - Identify objects and classes:
    * Individual books
    * A library: book database
  - Relationships
    * Each book *has a* title and an author
    * A book database *has a* list of books
  - Operations/Interactions/Behaviour
    * Adding books to a database
    * Searching for a book in a database

Array applications (contd.)

- Design
  - Class diagram



Array applications (contd.)

- Design (contd.)
  - Adding a book m:

  1. Set book_list[i] to m, where i is the next available slot

  So we need to remember the next available slot. We can do this by adding a new attribute to the database which is the index of the next available cell. Hence the add book operation should now be:

  1. If i < length of book_list: (where i is the next available slot)
     (a) Set book_list[i] to m,
     (b) increase i by 1

Array applications (contd.)

- Design (contd.)
  - Finding a book with title t:

1. For each element book_list[i] which is not null and such that i<the length of the list:
   (a) If the title ot book_list[i] is equal to t then
      i. return book_list[i]
2. If not found, return null

Array applications (contd.)

```
class Book {
  private String title, author;
  public Book(String t, String d)
  {
    title = t;
    author = d;
  }
  public String title() { return title; }
  public String author() { return author; }
  public Book clone()
  {
    return new Book(this.title, this.author);
  }
}
```

Array applications (contd.)

```
class Library {
  private Book[] book_list;
  private int next_available;
  public int number_of_books;

  public Library(int max_capacity)
  {
    book_list = new Book[max_capacity];
    next_available = 0;
    number_of_books = 0;
  }

  // Continues below...
```

```
    public void add_book(Book m)
    {
      if (next_available < book_list.length) {
        book_list[next_available] = m;
           // or m.clone();
        next_available++;
        number_of_books++;
      }
    }
    public Book find_book(String title)
    {
      int index = 0;
      while (index < number_of_books) {
        Book m = book_list[index];
        String t = m.title();
        if (t.equals(title)) {
          return m;
        }
        index++;
      }
      return null;
    }
  } // End of Library
```

Array applications (contd.)

- Second version of find

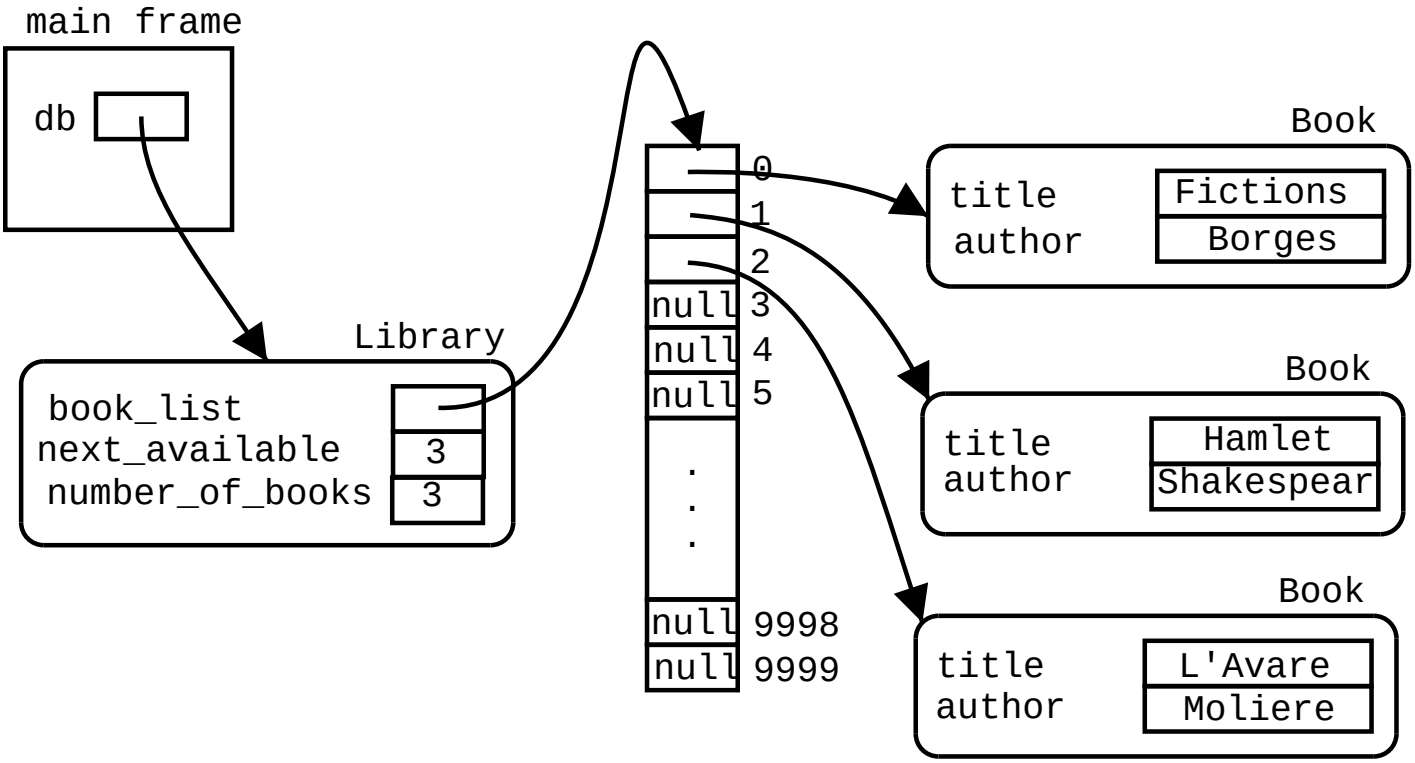
```
public Book find_book(String title)
{
  int index;
  index = 0;
  while (index < number_of_books) {
    if (book_list[index].title().equals(title)) {
      return book_index[index];
    }
    index++;
  }
  return null;
}
```

Array applications (contd.)

- Using the database

```
public class Test {
  public static void main(String[] args)
  {
    Library db = new Library(10000);
    Book m;
    m = new Book(“Fictions”,“Borges”);
    db.add_book(m);
    m = new Book(“Hamlet”,“Shakespeare”);
    db.add_book(m);
    m = new Book(“L\’Avare”,“Moliere”);
    db.add_book(m);
    Book k = db.find_book(“Hamlet”);
    System.out.println(k.author());
  }
}
```

Array applications (contd.)



Array applications (contd.)

- Deleting elements from the database

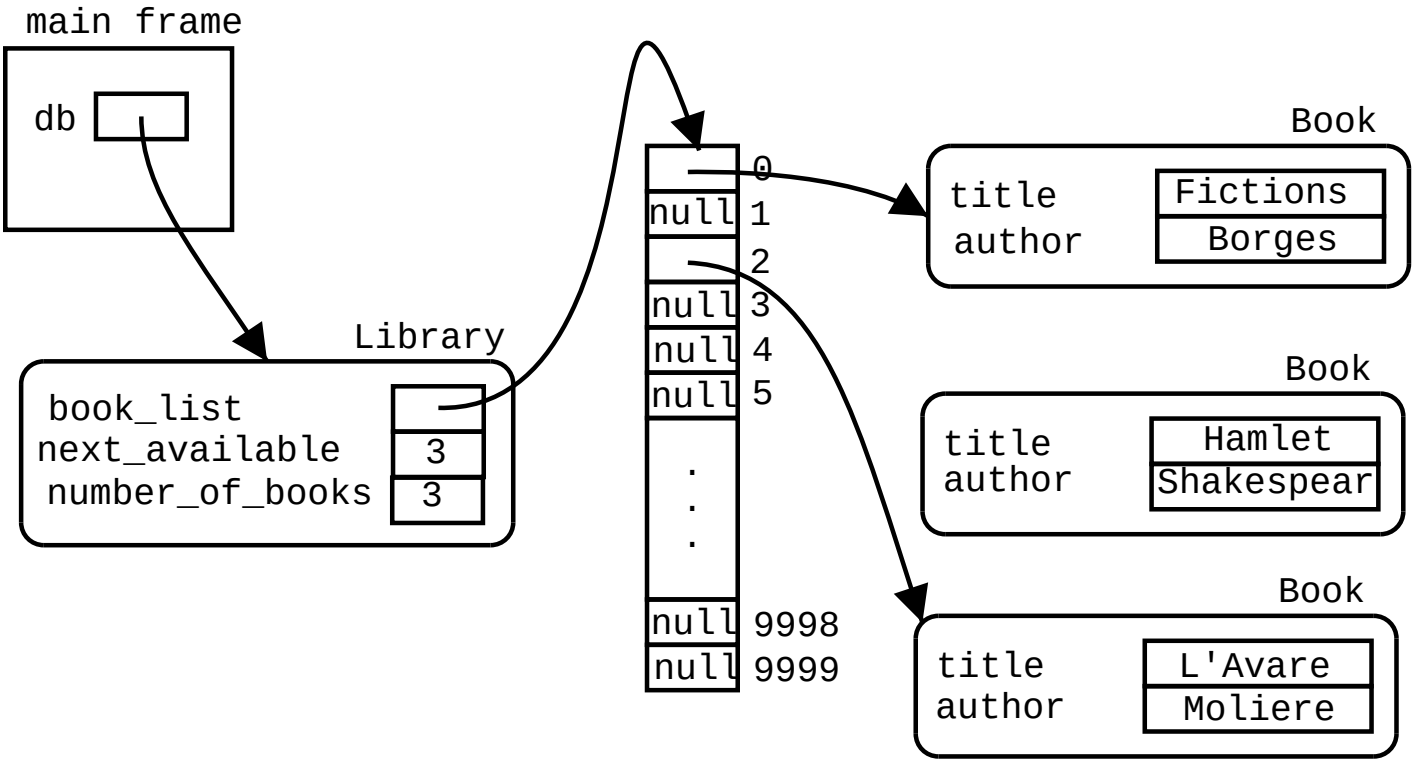
```
public int book_index(String title)
{
  for (int i=0; i < book_list.length; i++) {
    Book m = book_list[i];
    if (m != null && m.title().equals(title)) {
      return i;
    }
  }
  return -1;
}
public void delete_book(String title)
{
  int i = book_index(title);
  if (i != -1) {
    book_list[i] = null;
    number_of_books--;
  }
}
```

Array applications (contd.)

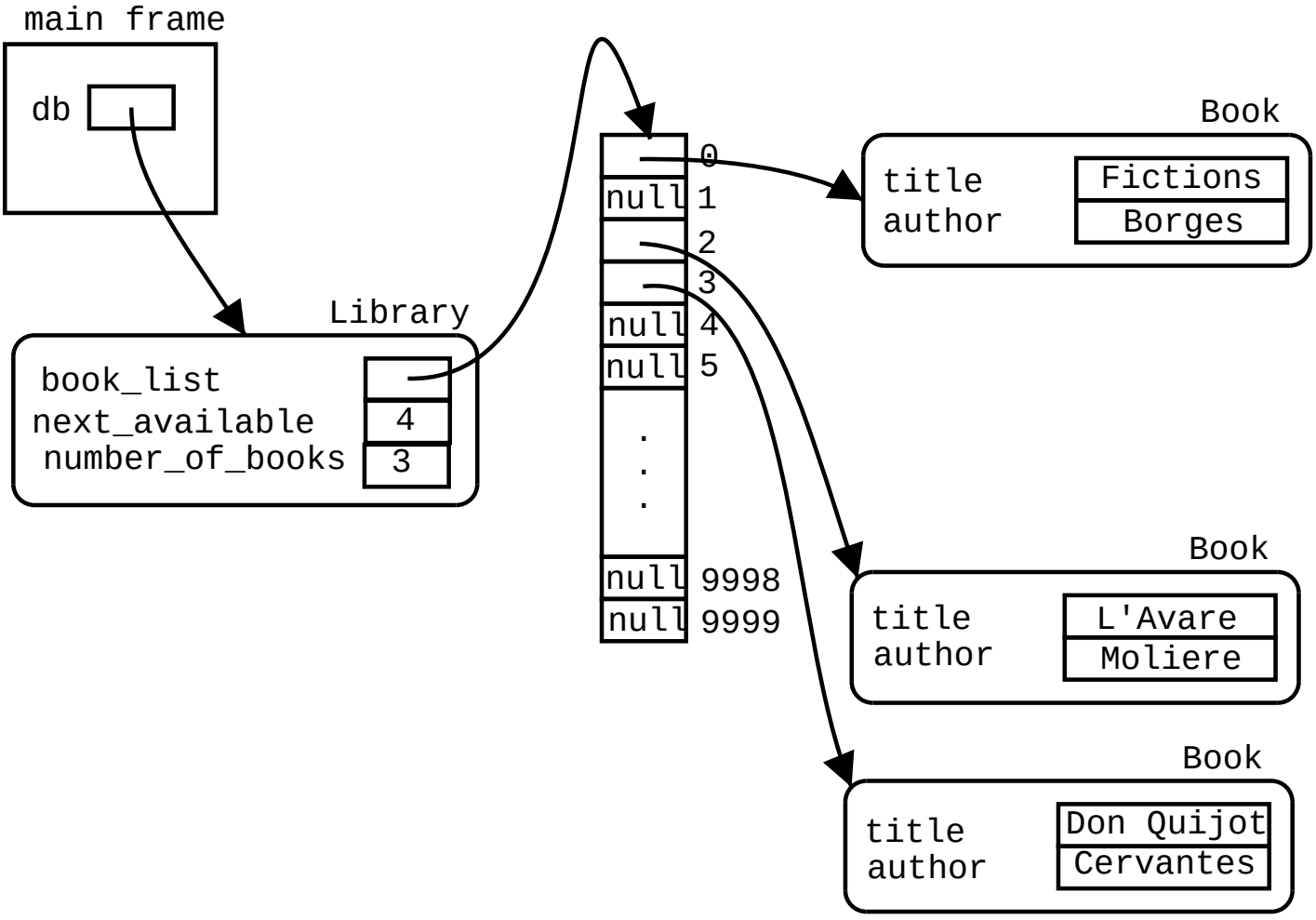
- But there's a problem: holes!

```
public class Test {
  public static void main(String[] args)
  {
    Library db = new Library(10000);
    Book m;
    m = new Book(“Fictions”,“Borges”);
    db.add_book(m);
    m = new Book(“Hamlet”,“Shakespeare”);
    db.add_book(m);
    m = new Book(“L\’Avare”,“Moliere”);
    db.add_book(m);
    db.delete_book(“Hamlet”);
    m = new Book(“Don Quijote”,“Cervantes”);
    db.add_book(m);
  }
}
```

Array applications (contd.)



Array applications (contd.)



Array applications (contd.)

- New algorithm for adding a book m:

1. Find an available slot i in book_list

2. Set book_list[i] to the book m

Array applications (contd.)

- Implementation

```
public void add_book(Book m)
{
  // Find an empty slot
  int index = 0;
  while (index < book_list.length
      && book_list[index] != null) {
    index++;
  }
  // Store the book
  if (index < book_list.length) {
    book_list[index] = m;
    number_of_books++;
  }
}
```

Array applications (contd.)

```
class Library {
  private Book[] book_list;
  public int number_of_books;

  public Library(int max_capacity)
  {
    book_list = new Book[max_capacity];
    number_of_books = 0;
  }

  public void add_book(Book m)
  {
    int index = 0;
    while (index < book_list.length
        && book_list[index] != null) {
      index++;
    }
    if (index < book_list.length) {
      book_list[index] = m;
      number_of_books++;
    }
  }
```

Array applications (contd.)

```
public int book_index(String title)
{
  for (int i=0; i < book_list.length; i++) {
    Book m = book_list[i];
    if (m != null && m.title().equals(title)) {
      return i;
    }
  }
  return -1;
}

public void delete_book(String title)
{
  int i = book_index(title);
  if (i != -1) {
    book_list[i] = null;
    number_of_books--;
  }
}
```

Array applications (contd.)

```
  public Book find_book(String title)
  {
    int i = book_index(title);
    if (i != -1) return book_list[i];
    return null;
  }
} // End of Library
```

Optimized Book database

Idea: instead of looking for an available cell each time we add a book, modify the delete method so that when we delete a book, move the last book of the list to the cell which just openned. This way, the array is not fragmented. This is, there are no holes, and all books are all grouped toghether at the beginning of the array.

```
public class Library {
  private Book[] book_list;
  private int next_available;

  public Library(int max_capacity)
  {
    book_list = new Book[max_capacity];
    next_available = 0;
  }
  // Continues below...
```

Optimized Book database

```
public void add_book(Book m)
{
  if (next_available < book_list.length) {
    book_list[next_available] = m;
    next_available++;
  }
}

public int book_index(String title)
{
  for (int i=0; i < book_list.length; i++) {
    Book m = book_list[i];
    if (m != null && m.title().equals(title)) {
      return i;
    }
  }
  return -1;
}
```

Optimized Book database

```
  public void delete_book(String title)
  {
    int i = book_index(title);
    if (i != -1) {
      book_list[i]=book_list[next_available-1];
      book_list[next_available - 1] = null;
      next_available--;
    }
  }
  public Book find_book(String t)
  {
    int i = book_index(t);
    if (i != -1) return book_list[i];
    return null;
  }
  public int number_of_books()
  {
    return next_available;
  }
}
```

Growing arrays

- An array has a finite and fixed amount of memory.
- In some applications we don't know a priori how much memory we need.
- C/C++ allow to grow arrays at will: big data-safety problem.
- Java does not allow to grow arrays directly, but we can simulate it indirectly:
- Growing arrays:
  - Whenever the array of interest fills up, a new, bigger array is created,
  - ...and the values of the old array are copied (shallowly) into the new array.
- Or, use class `ArrayList` or `Vector` from the standard library.

The Vector and ArrayList classes

- Two classes which encapsulate growing arrays
- The two provide essentially the same functionality, but have a slightly different underlying implementation.
- Vector has methods

```
void setElementAt(Object o, int index)
Object elementAt(int index)
int size()
boolean contains(Object o)
int index_of(Object o)
// ... etc
```

- ArrayList has methods

```
Object get(int index)
void set(int index, Object o)
void add(Object o)
int size()
// ... etc
```

The Vector and ArrayList classes

```
public class Library {
  private ArrayList book_list;

  public Library()
  {
    book_list = new Vector();
  }
  public void add_book(Book m)
  {
    book_list.add(m);
  }
  // ...
}
```

Growing arrays

- Change algorithm for adding a movie m:

1. Find first available cell

2. If an available cell is found:

   (a) Store m in that cell

3. Otherwise:

   (a) Grow the array (copying contents of the old to the new)
   (b) Find the first available cell in the new array (guarranteed to exist.)
   (c) Store m in that cell

Growing arrays

```
// In class Library
private void grow_array(int n)
{
  int new_capacity = book_list.length + n;
  Book[] new_list = new Book[new_capacity];
  int i = 0;
  while (i < Book_list.length) {
    new_list[i] = book_list[i]; // shallow copy
    i++;
  }
  book_list = new_list; // Update list reference
}
```

The method is private to ensure encapsulation so that only MovieDatabase objects can grow the movie lists.

The end