GIUSEPPE TURINI

CS-102 COMPUTING AND ALGORITHMS 2
LESSON 02
DATA ABSTRACTION - THE WALLS

DATA ABSTRACTION AND PROBLEM SOLVING WITH JAVA, **CHAP 4, 197-240**.

HIGHLIGHTS

Abstract Data Types (ADTs)

- Modularity, Procedural Abstraction, and Information Hiding
- Abstract Data Types vs Data Structures, Data Abstraction, and Axioms

The ADT List

- The ADT List Description and Operations
- The ADT List Axioms and the ADT Sorted List

Implementing Abstract Data Types

Java Classes Revisited

- Object-Oriented Programming (OOP) and Encapsulation
- Classes, Data Fields, Methods, Constructors, and the Java Garbage Collector
- Inheritance and Object Equality

Java Interfaces and Java Exceptions and Java Packages

Implementing the ADT List using Arrays

ABSTRACT DATA TYPES 1

MODULARITY

Keeps the complexity of a large program manageable by systematically controlling the interaction of its components. Isolates errors and eliminates redundancies.

A **modular program** is:

- easier to write,
- easier to read, and
- easier to modify.

Note: Generally speaking, modules should be logically separate pieces of your program. In object-oriented programming (OOP), usually modules are defined via classes and their relationships.

ABSTRACT DATA TYPES 2

PROCEDURAL ABSTRACTION

Separates the purpose and use of a module from its implementation.

*"The principle that **any operation that achieves a well-defined effect can be treated by its users as a single entity**, despite the fact that the operation may actually be achieved by some sequence of lower-level operations."*
John Daintith. *A Dictionary of Computing.* 2004.

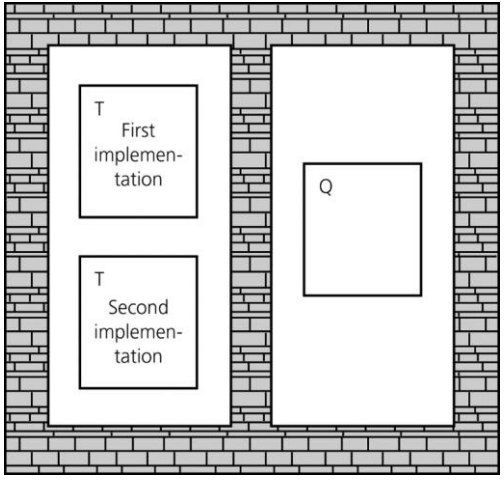
The **specifications of a module** should:

- detail how the module behaves, and
- identify details that can be hidden within the module.

ABSTRACT DATA TYPES 3

INFORMATION HIDING

Hides certain implementation details within a module.
Makes these details inaccessible from outside the module.

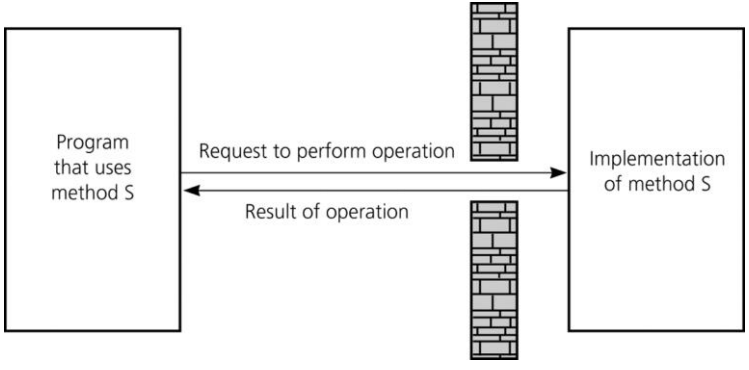


ISOLATED TASKS

Implementation of task **T** does not affect task **Q**.

THE ISOLATION OF MODULES IS NOT TOTAL

Specifications of methods govern how they interact with each other.



ABSTRACT DATA TYPES 4

TYPICAL OPERATIONS ON DATA

- **Add data** to a data collection.
- **Remove data** from a data collection.
- **Query the data** in a data collection.

DATA ABSTRACTION

Think **what you can do** to a collection of data independently of **how you do** it.
Allows development of a data structure in relative isolation from the rest of the code.
A natural extension of **procedural abstraction**.

ABSTRACT DATA TYPES 5

ABSTRACT DATA TYPE (ADT)

An ADT is composed of:

- a **collection of data**, and
- a **set of operations** on that data.

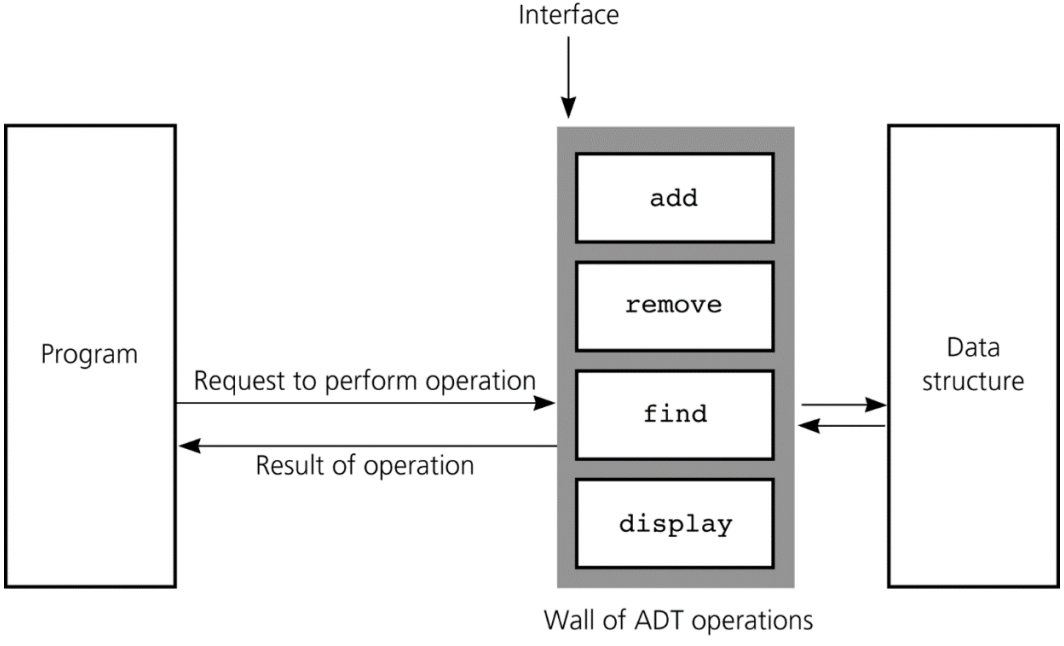
ADT Specifications: What ADT operations do, not how to implement them.
ADT Implementation: It includes choosing a particular data structure.
Data Structure: Construct defined to store a collection of data (e.g. arrays).

Abstract data types (ADTs) and data structures are not the same!

ABSTRACT DATA TYPES 6

DATA ABSTRACTION

Results in a wall of ADT operations which isolates data structures from the program that accesses the data stored in these data structures.



ABSTRACT DATA TYPES 7

DESIGNING AN ABSTRACT DATA TYPE

The design of an ADT should evolve naturally during the problem-solving process:

What data does a problem require?

What operations does a problem require?

Note: For complex ADTs, operations behaviors are specified by **axioms** (math rules).

THE ADT LIST 1

THE ADT LIST: DESCRIPTION

Each item (except first and last) has: 1 **unique predecessor**, and 1 **unique successor**.
The first item is called **head** or front, and does not have a predecessor.
The last item is called **tail** or end, and does not have a successor.

THE ADT LIST: OPERATIONS

1. create an empty list: **aList.createList()**
2. check if a list is empty: **aList.isEmpty()**
3. determine the number of items in a list: **aList.size()**
4. add an item at a given position in the list: **aList.add(i, x)**
5. remove the item at a given position in the list: **aList.remove(i)**
6. remove all the items from the list: **aList.removeAll()**
7. retrieve the item at a given position in the list: **aList.get(i)**

THE ADT LIST 2

Note: List items are referenced by their position within the list.

THE ADT LIST: SPECIFICATIONS OF THE OPERATIONS

- define the contract for the ADT list,
- do not specify how to store the list or how to perform the operations.

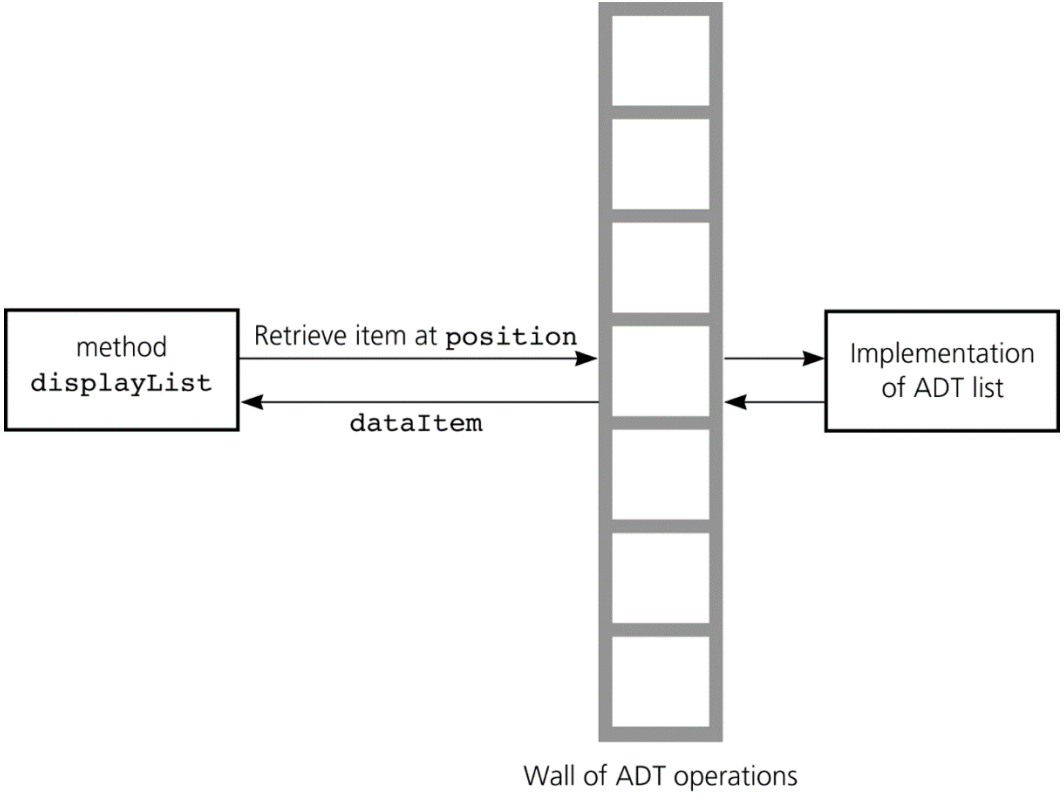
ADT operations can be used without knowing their implementation.

THE ADT SORTED LIST

- maintains items in sorted order,
- inserts and deletes items by their values, not by their positions!

THE ADT LIST 3

The wall between the method **displayList** and the implementation of the **ADT list**.



THE ADT LIST 4

Example: Axioms to specify the behavior of operations of the **ADT list**:

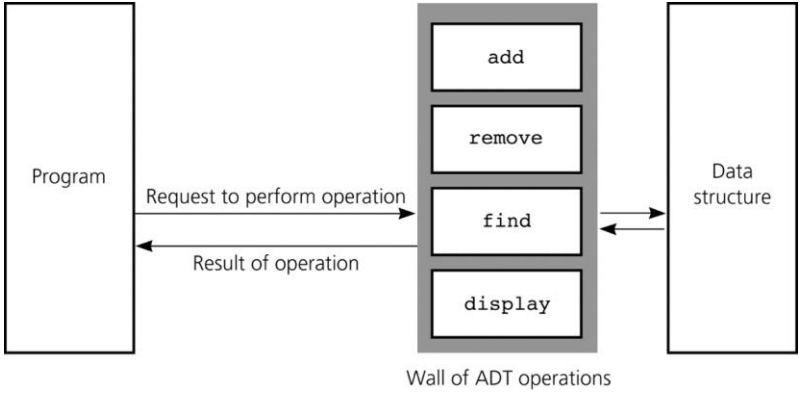
- Axiom: **(aList.createList()).size() == 0**
- Axiom: **(aList.add(i, x)).size() == aList.size() + 1**
- Axiom: **(aList.remove(i)).size() = aList.size() – 1**
- Axiom: **(aList.createList()).isEmpty() = true**
- Axiom: **(aList.add(i, x)).isEmpty() = false**
- Axiom: **(aList.createList()).remove(i) = error**
- Axiom: **(aList.add(i, x)).remove(i) = aList**
- Axiom: **(aList.createList()).get(i) = error**
- Axiom: **(aList.add(i, x)).get(i) = x**
- Axiom: **aList.get(i) = (aList.add(i, x).get(i + 1)**
- Axiom: **aList.get(i + 1) = (aList.remove(i)).get(i)**

IMPLEMENTING ABSTRACT DATA TYPES 1

Choose the data structure to represent the data of the ADT, considering:

- the details of the operations of the ADT, and
- the context in which the operations will be used.

Implementation details hidden behind a wall of ADT operations. This means that a program will only be able to access the data structure using the ADT operations.

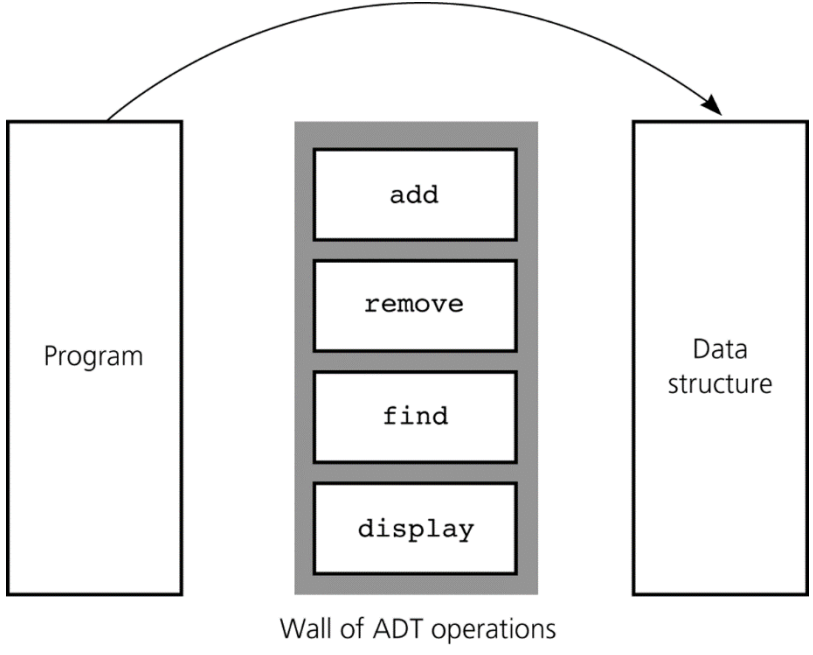


IMPLEMENTING ABSTRACT DATA TYPES 2

VIOLATING THE WALL OF ADT OPERATIONS

In a **non**-object-oriented implementation, both the **data structure** and the **ADT operations** are distinct pieces…

In this case the data structure is hidden only if the program using the ADT does not look over the wall!



Object-oriented languages provide a way to enforce the wall of an ADT!

JAVA CLASSES REVISITED 1

OBJECT-ORIENTED PROGRAMMING (OOP)

Object-oriented programming (OOP) views **a program** not as a sequence of actions but **as a collection of components called objects.**

PRINCIPLES OF OBJECT-ORIENTED PROGRAMMING

- **Encapsulation:** objects combine data and operations.
- **Inheritance:** classes can inherit properties from other classes.
- **Polymorphism:** objects can determine proper operations at run time.

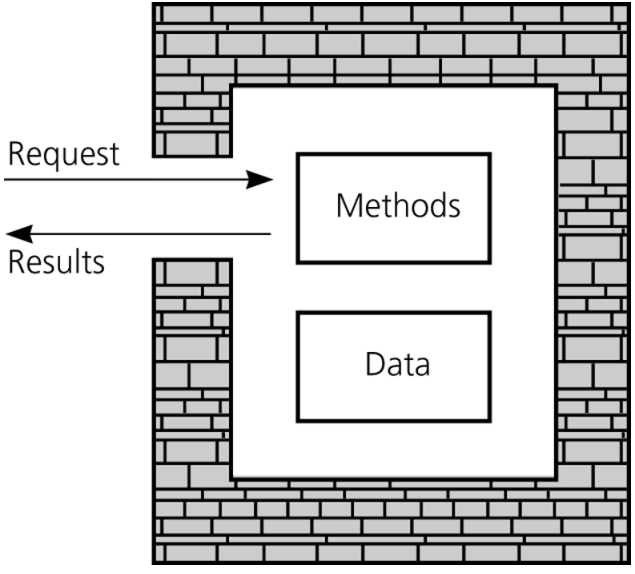
See: docs.oracle.com/javase/tutorial/java/iandi/subclasses
See: docs.oracle.com/javase/tutorial/java/iandi/polymorphism

JAVA CLASSES REVISITED 2

ENCAPSULATION

The **packing of data and functions** into a single element (i.e. an object):

- it is a principle of OOP;
- can enforce the walls of an ADT;
- hides the ADT implementation.



Encapsulation combines the ADT data with its operations to form an object.

JAVA CLASSES REVISITED 3

DESCRIPTION OF A JAVA CLASS

In Java, **a class is a new data type** whose instances are objects, and including the following class members:

- **data fields** (should almost always be private),
- **methods**.

ACCESS CONTROL OF A JAVA CLASS

- A **class member** with no access modifier is **package-private**, otherwise it can be configured as **private**, **public**, or **protected**.
- A **class** with no access modifier is **package-private**, otherwise it can be configured as **public**.

See: docs.oracle.com/javase/tutorial/java/javaoo/accesscontrol

A program or module that uses a class is usually referred as a client of a class.

JAVA CLASSES REVISITED 4

JAVA CLASS CONSTRUCTORS

The **class constructor** is a method that creates and initializes new instances (i.e. objects) of a class. It has the same name as the class, no return type, and may have parameters. The class constructor allocates the memory required to store an object.
A class can have more than one constructor. A constructor with no parameters is usually called the **default constructor**.

If a class has no constructor,
the Java compiler generates automatically a default constructor for it!

See: docs.oracle.com/javase/tutorial/java/javaoo/constructors

JAVA CLASSES REVISITED 5

JAVA GARBAGE COLLECTOR

The **garbage collector (GC)** destroys all objects that a program no longer needs (i.e. references), so there is **no need of class destructors** to deallocate memory.

A GC works in this way:

1. when a program no longer references an object, the **Java Runtime Environment (JRE)** marks it for garbage collection;
2. periodically, the JRE executes a method that free the memory used by these marked objects making this space available for future use;
3. if when an object is destroyed, other tasks beyond memory deallocation are necessary, **you can define a finalize method for a class**.

See: en.wikipedia.org/wiki/garbage_collection_(computer_science)

JAVA CLASSES REVISITED 6

INHERITANCE

In OOP, inheritance is when a class (**derived class** or **subclass**) is based on another class (**base class** or **superclass**), using the same implementation (**inheriting**) or specifying an implementation to maintain the same behavior (realizing an **interface**). Inheritance is a mechanism for **code reuse** and to allow **independent extensions** of the original software via **public classes** and **interfaces**.

Note: The relationships of classes through inheritance results in a **hierarchy**.
Note: Use **extends** keyword in base class to inherit from a superclass.
Note: Use **super** keyword in base class constructor to call the superclass constructor.
Note: Remember that Java allows **only 1 superclass!**

See: docs.oracle.com/javase/tutorial/java/iandi/subclasses

JAVA CLASSES REVISITED 7

OBJECT EQUALITY

Objects should be not compared directly with the **==** operator. To compare objects you should use or implement the method **equals**.

equals is a method of the **Object** class, and in its default implementation compares 2 objects checking their references.

However, a **custom implementation of the method equals** can be developed (via **method overriding**) to check objects data for equality.

See: docs.oracle.com/javase/tutorial/java/nutsandbolts/op2
See: docs.oracle.com/javase/8/docs/api/java/lang/object
See: docs.oracle.com/javase/tutorial/java/iandi/override

JAVA INTERFACES 1

An interface specifies methods and constants but supplies no implementation, and it should be used to specify a desired common behavior.
The Java API has many predefined interfaces (e.g. **java.util.Collection**).

See: docs.oracle.com/javase/8/docs/api/java/util/collection
See: docs.oracle.com/javase/tutorial/java/iandi/createinterface

A class that implements an interface must:

- include an **implements** clause, and
- provide implementations of the methods of the interface.

To define an interface:

- use the keyword **interface** instead of **class** in the header, and
- provide only method specifications and constants in the interface definition.

JAVA INTERFACES 2

Defining an interface:

```
public interface MyInterface {
   public final int f1 = 0;
   public void method1();
   public int method2( int a, int b ); }
```

Defining a class that implements an interface:

```
public class MyClass implements MyInterface {
   public void method1() {}
   public int method2( int a, int b ) { return a + b; } }
```

JAVA INTERFACES 3

OBJECT COMPARISON AND INTERFACES

We can only compare objects that are "comparable". In Java this means that their class needs to implement the **java.lang.Comparable interface** containing the method **compareTo** that the object class needs to implement to perform the comparison.

See: docs.oracle.com/javase/8/docs/api/java/lang/comparable

```
// The SimpleSphere class implements the interface Comparable (Java API).
public class SimpleSphere implements java.lang.Comparable<Object> {
   public int compareTo( Object rhs ) { // Note: "rhs" stands for right-hand side.
      SimpleSphere other = (SimpleSphere) rhs; // Note: possible ClassCastException!
      if( radius == other.getRadius() ) { return 0; } // This is equal to "other".
      else if( radius < other.getRadius() ) { return -1; } // This is less than "other".
      else { return 1; } } // This is greater than "other".
}
```

JAVA EXCEPTIONS 1

An **exception is a mechanism for handling an error during execution**. A method indicates that a runtime error has occurred by "throwing" an exception. In your code, you can handle a runtime error by "catching" the exception thrown by the method causing the error during execution.

CATCHING EXCEPTIONS

To catch exceptions, code that might throw an exception is enclosed in a **try** block:

```
try { ... }
```

Right after a **try** block, one or more **catch** blocks can be used to handle the error:

```
catch( ExceptionClass e ) { ... }
```

JAVA EXCEPTIONS 2

TYPES OF EXCEPTIONS: CHECKED EXCEPTIONS

Instances of classes that are subclasses of the **java.lang.Exception** class, these exceptions must be handled locally or explicitly thrown from the method. Are used in situations where the method has encountered a **serious problem**.

See: docs.oracle.com/javase/8/docs/api/java/lang/exception

TYPES OF EXCEPTIONS: RUNTIME EXCEPTIONS

Instances of classes that are subclasses of the **java.lang.RuntimeException** class, these exception are not required to be caught locally or explicitly thrown. These exceptions are usually used when the **error is not critical**.

See: docs.oracle.com/javase/8/docs/api/java/lang/runtimeexception

JAVA EXCEPTIONS 3

THROWING EXCEPTIONS

A **throw** statement is used to throw an exception:

```
   throw new MyCustomException( "My custom message." );
```

DEFINING CUSTOM EXCEPTIONS

To define custom exception classes representing specific runtime error types:

```
import java.lang.RuntimeException;
import java.lang.String;

public class MyCustomException extends RuntimeException {
   public MyCustomException( String s ) { super(s); } // Constructor.
}
```

JAVA PACKAGES 1

Java packages provide a way to group related classes together.

You must use the same name for both
a Java class and the file that contains that class.

You must use the same name for both
a package and the directory that contains all the classes in that package.

JAVA PACKAGES 2

To create a package, place a **package** statement before each class in the package:

```
package MyPackage;
public class MyClass {} // Note: public class available to all clients of the package.
```

To make the class available to the clients of the package, use the keyword **public** (access modifier) before the class declaration. If there is no access modifier, the class is available only to other classes in the package (i.e. **package-private**).

```
package MyPackage;
class MyClass {} // Note: no access modifier means the class is package-private.
```

Note: When a class is publicly available within a package, it can also be used by other classes, even those appearing in other packages.

JAVA PACKAGES 3

A package can contain other packages as well, and in this case the directory hierarchy should be consistent. In fact **the package name consists of the hierarchy of package names, separated by periods.**

```
package MyDrawingPackage.MyShapePackage;
import java.lang.Object;
```

To use a package in a program we need to use the **import** statement. In particular, you can use the **asterisk notation** to indicate to the Java compiler that you might use any class in that package.

```
import java.io.*;
import java.io.DataStream;
```

JAVA PACKAGES 4

CLASSES WITH NO EXPLICIT PACKAGE

If you omit the package declaration from the source file for a class, the class is added to a **default unnamed package**. If all the classes in a group are declared this way, they are all considered to be within this same default unnamed package and hence do not require an **import** statement.

However, **if you are developing a package, and you want to use a class that is contained in the default unnamed package, you will need to import the class**. In this case, since the package has no name, the class name itself is sufficient in the import statement.

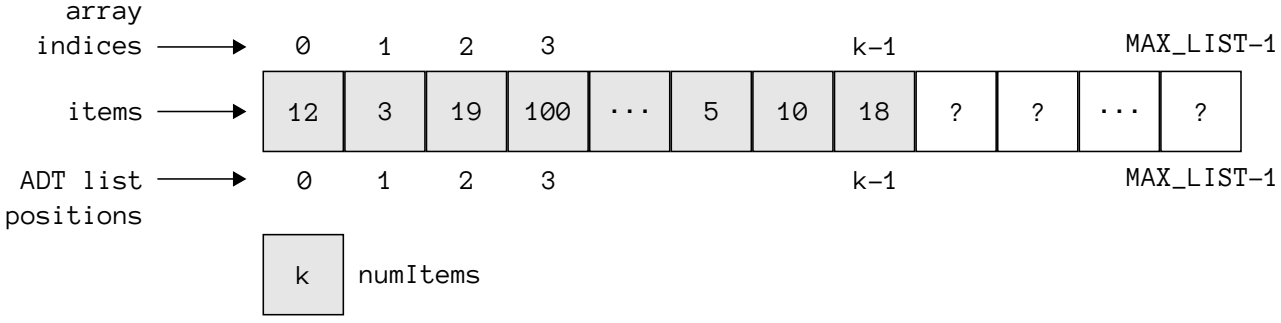
```
import MyClass;
```

IMPLEMENTING THE ADT LIST USING ARRAYS 1

FIRST SKETCH OF AN ARRAY-BASED ADT LIST

The following is a sketch of an array-based implementation of the ADT list:

- all the items of the list are stored in an array **items**,
- the **kth item** will be stored in **items[k]**,
- **max size** of the array is a fixed value **MAX_LIST** (i.e. **physical size**),
- **current number of items** in the list stored in **numItems** (i.e. **logical size**).

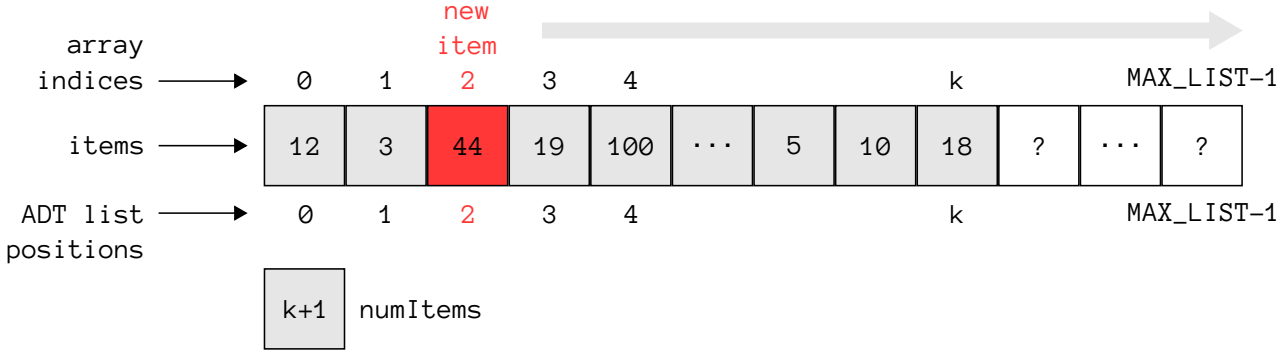


IMPLEMENTING THE ADT LIST USING ARRAYS 2

INSERTION OF AN ITEM INTO THE ARRAY-BASED ADT LIST

To insert a new item at a given position **p** in the array of list items, you must:

1. perform a **shift to the right** of the items from position **p** on, and
2. perform the **insertion of the new item** in the newly created opening (at **p**).

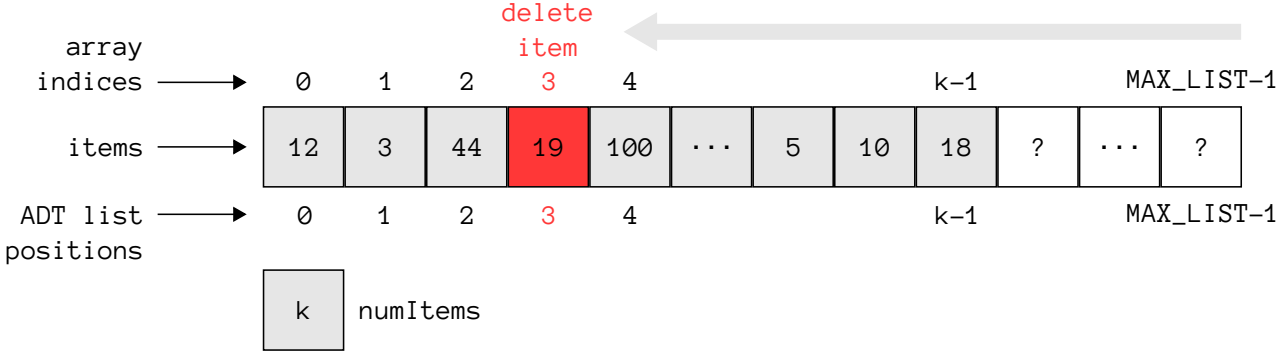


IMPLEMENTING THE ADT LIST USING ARRAYS 3

DELETION OF AN ITEM FROM THE ARRAY-BASED ADT LIST

To delete an item from the list, you must not only to erase the target item but also to remove the gap created by the removal. Therefore:

1. at first, **delete the item** from the array, and
2. then **shift left all the items on the right side** of the newly created gap.



IMPLEMENTING THE ADT LIST USING ARRAYS 4

LIST INDEX OUT OF BOUNDS EXCEPTION

```
// Exception used for an out-of-bounds list index.
public class ListIndexOutOfBoundsException extends IndexOutOfBoundsException {
   // Constructor.
   public ListIndexOutOfBoundsException( String s ) { super(s); } }
```

LIST EXCEPTION

```
// Exception used when the array storing the list becomes full.
public class ListException extends RuntimeException {
   // Constructor.
   public ListException( String s ) { super(s); } }
```

IMPLEMENTING THE ADT LIST USING ARRAYS 5

LIST INTERFACE

```
// Interface providing the specifications for the ADT list operations.
public interface ListInterface {

   public boolean isEmpty(); // Determine whether a list is empty.
   public int size(); // Detemines the length of a list.
   public void removeAll(); // Deleted all the items from the list.

   // Adds an item to the list at position index.
   public void add(int i, Object o) throws ListIndexOutOfBoundsException, ListException;

   // Retrieves a list item by position.
   public Object get(int i) throws ListIndexOutOfBoundsException;

   // Deletes an item from the list at a given position.
   public void remove(int i) throws ListIndexOutOfBoundsException;

}
```

IMPLEMENTING THE ADT LIST USING ARRAYS 6

LIST ARRAY BASED A

```
// Array-based implementation of the ADT list.
public class ListArrayBased implements ListInterface {
   private static final int MAX_LIST = 50; // Maximum (physical) size of the list.
   private Object items[]; // An array of list items.
   private int numItems; // Number of items (logical size) of the list.

   // Default constructor.
   public ListArrayBased() { items = new Object[ MAX_LIST ]; numItems = 0; }

   public boolean isEmpty() { return ( numItems == 0 ); }

   public int size() { return numItems; }

   public void removeAll() {
      // Creates a new array, and marks old array for garbage collection.
      items = new Object[ MAX_LIST ];
      numItems = 0; }
```

IMPLEMENTING THE ADT LIST USING ARRAYS 7

LIST ARRAY BASED B

```
// ...
public void add( int index, Object item )
   throws ListIndexOutOfBoundsException, ListException {
      if( numItems > MAX_LIST ) {
         throw new ListException( "ListException on add." ); }
      if( ( index >= 0 ) && ( index <= numItems ) ) {
         // Insert new item by right shifting all items at position >= index.
         for( int pos = numItems; pos > index; pos-- ) {
            items[ pos ] = items[ pos-1 ]; }
         items[ index ] = item; // Insert new item.
         numItems++; }
      else {
         throw new ListIndexOutOfBoundsException("ListIndexOutOfBoundsException"); } }
```

IMPLEMENTING THE ADT LIST USING ARRAYS 8

LIST ARRAY BASED C

```
   // ...
   public Object get( int index ) throws ListIndexOutOfBoundsException {
      if( ( index >= 0 ) && ( index < numItems ) ) { return items[ index ]; }
      else {
         throw new ListIndexOutOfBoundsException("ListIndexOutOfBoundsException"); } }

   // ...
   public void remove( int index ) throws ListIndexOutOfBoundsException {
      if( ( index >= 0 ) && ( index < numItems ) ) {
         // Delete item by left shifting all items at position > index.
         for( int pos = index+1; pos <= numItems; pos++ ) {
            items[ pos-1 ] = items[ pos ]; }
         numItems--; }
      else {
         throw new ListIndexOutOfBoundsException("ListIndexOutOfBoundsException"); } }

}
```