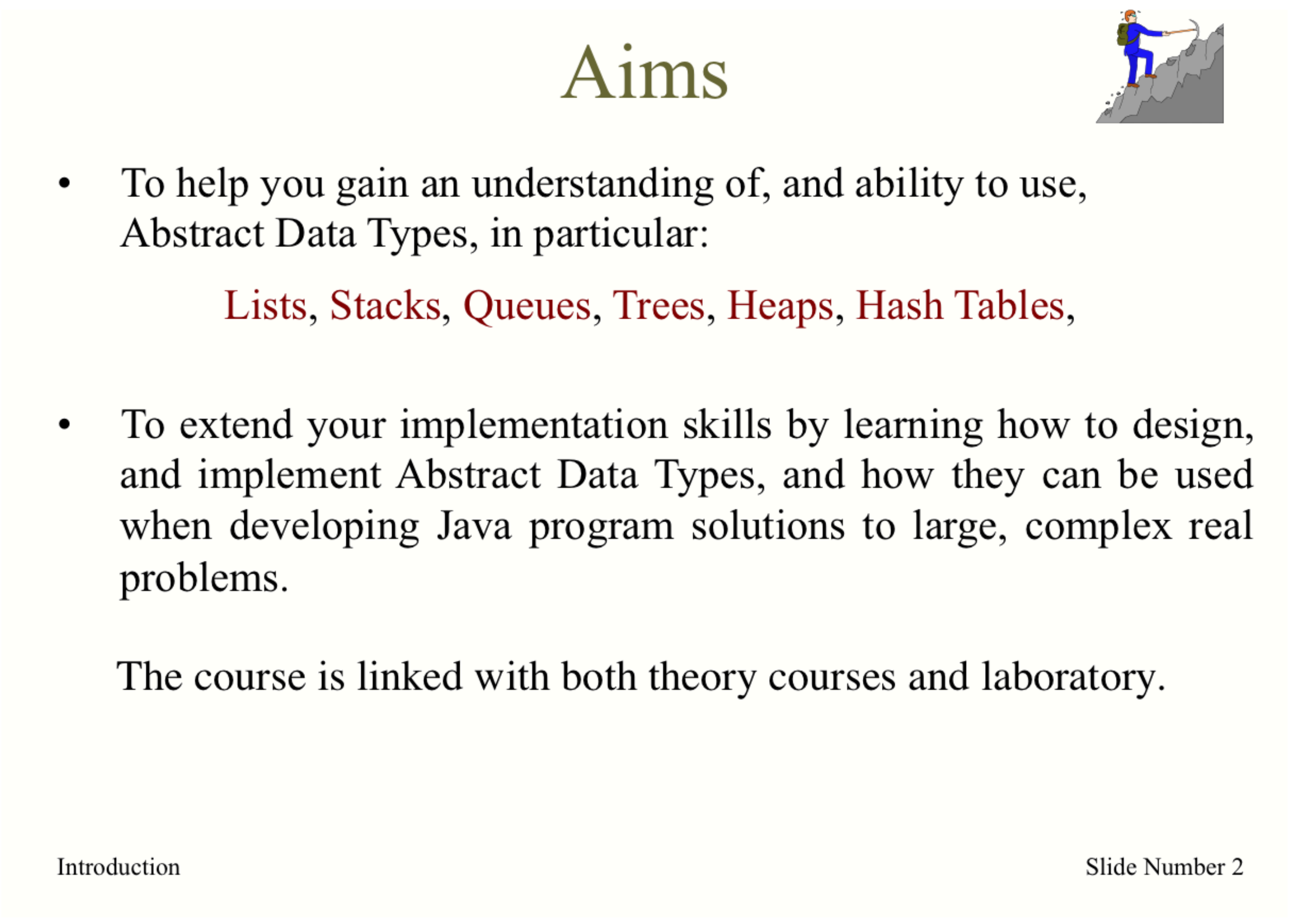


Aims

- To help you gain an understanding of, and ability to use, Abstract Data Types, in particular:

  Lists, Stacks, Queues, Trees, Heaps, Hash Tables,

- To extend your implementation skills by learning how to design, and implement Abstract Data Types, and how they can be used when developing Java program solutions to large, complex real problems.

The course is linked with both theory courses and laboratory.

Introduction Slide Number 2

This part of the course is about Abstract Data Types (ADT) and their role in the development of computing solutions for large and complex real problem domains. In particular, we will be looking at the most common ADTs, which are lists, stacks, queues, trees and Graphs; and study them from the point of view of how to specify them, how to implement them and how to use them **from first principles**. So you will not be using any of the existing analogous programming APIs like Java API structures (e.g. arrayList).

The **aims** of this second part of the course are:

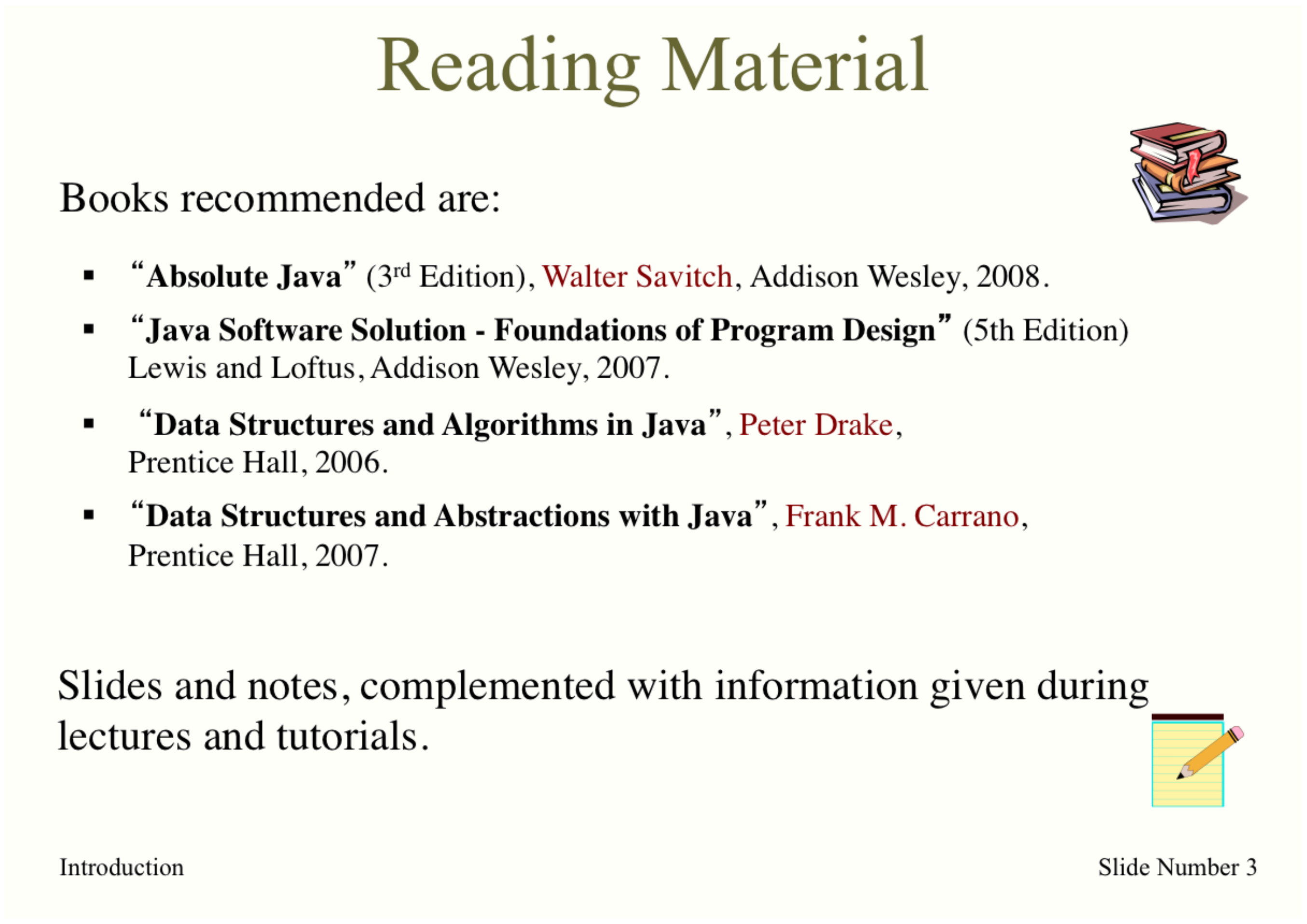
To help you gain a good understanding of, and ability to use, abstract data types; familiarise you with the above ADTs and the main operations that can be performed on these ADTs. We will refer to these operations as *access procedures* of the ADT.

To help you extend your programming skills by learning how to design and implement different data structure solutions for each of the above ADTs; and evaluate the practical benefits and limitations of different implementation choices.

Learning outcomes:

At the end of this part of the course, you will

- Understand basic principles, main features and operations of the above ADTs.
- Learn fundamental algorithms associated with these ADTs, including tree traversal and heapsort.
- Be able to develop Java implementations of these ADTs using different data structures approaches, and evaluate their differences.
- Be able to use ADT and related implementations in designing and implementing efficient solutions to given application problems.



Reading Material

Books recommended are:

- "**Absolute Java**" (3rd Edition), Walter Savitch, Addison Wesley, 2008.
- "**Java Software Solution - Foundations of Program Design**" (5th Edition) Lewis and Loftus, Addison Wesley, 2007.
- "**Data Structures and Algorithms in Java**", Peter Drake, Prentice Hall, 2006.
- "**Data Structures and Abstractions with Java**", Frank M. Carrano, Prentice Hall, 2007.

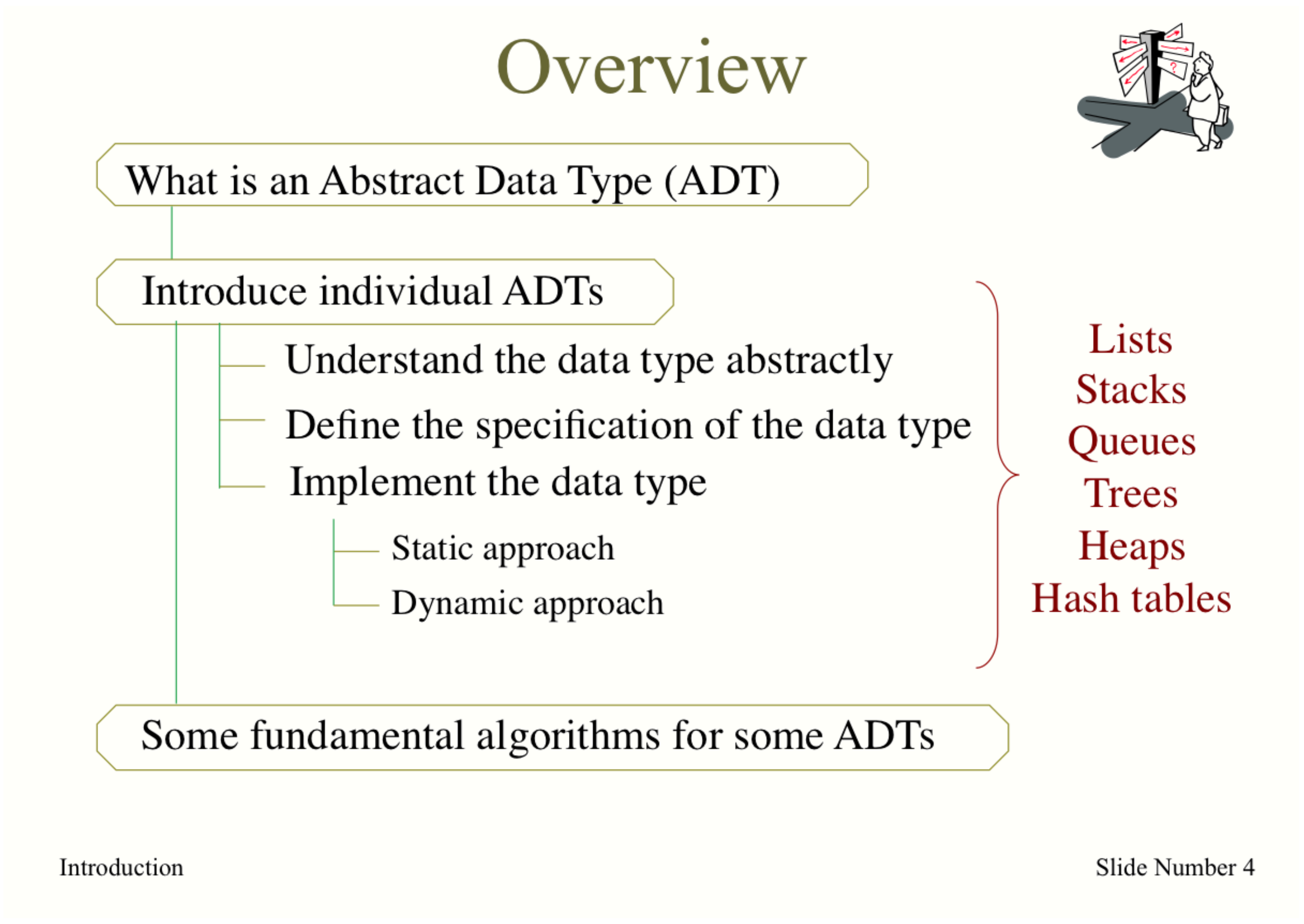
Slides and notes, complemented with information given during lectures and tutorials.

Introduction Slide Number 3

The first two books are well written text book for object oriented programming in general. They do not cover ADTs in detail, but the last few chapters provide a brief summary of some of the main ADTs that we are going to see during these lectures.

The third and forth books are text books on ADTs in Java. The last is, in particular, the main text book for this part of the course.

The slides will be available on my Web page at http://www.doc.ic.ac.uk/~ar3 and on CATE, together with tutorial sheets and model answers. There will be no assessed coursework for this module, as you will be already involved with few PPT Labs on Object-Oriented Programming in general and on ADTs in particular.



Overview

- What is an Abstract Data Type (ADT)
- Introduce individual ADTs
  - Understand the data type abstractly
  - Define the specification of the data type
  - Implement the data type
    - Static approach
    - Dynamic approach
- Some fundamental algorithms for some ADTs

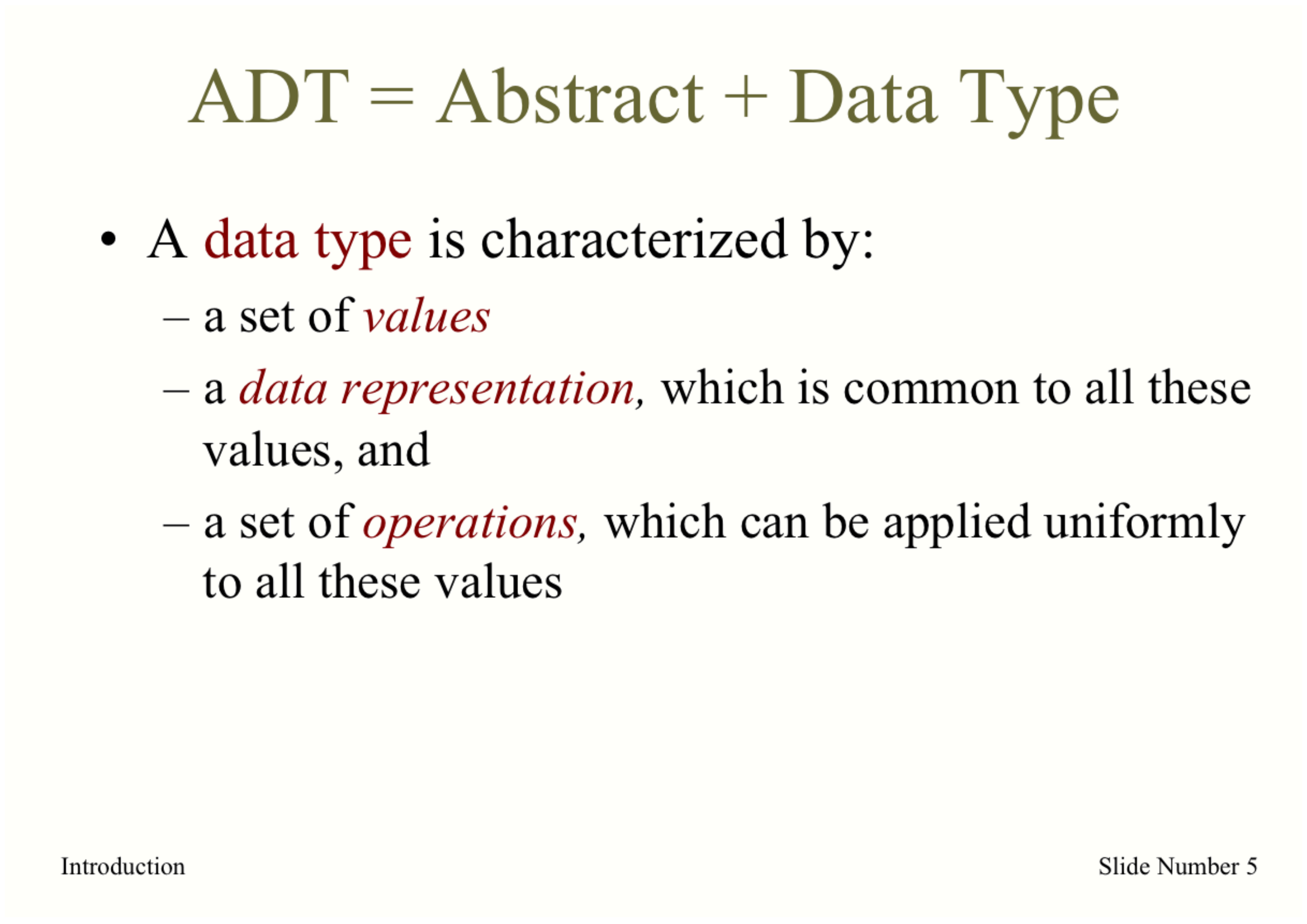
Lists
Stacks
Queues
Trees
Heaps
Hash tables

Introduction Slide Number 4

This is a brief overview of the main topics that we are going to cover.

We will begin with a general introduction of data types, abstraction, and definition of the main terminology used during these lectures, e.g. **data structure**, **data type**and **abstract data type.** We will then look into each of the main abstract data types listed above. For each of these abstract data types, we will see what the data type means in abstract terms, we will then define its specification and see some examples (also through your tutorial sheets) of how abstract data types can be used within the context of complex problems. Then we will see how the ADT can be implemented in Java, so as to meet its specification. In particular, we will look at two different approaches to ADT implementation: the static approach, which uses only arrays, and the dynamic approach, which deploys more complex data representations (i.e. data structures).

Finally, for some ADTs like trees, heaps and hash tables, we will look at some specific algorithms for handling them.



ADT = Abstract + Data Type

- A data type is characterized by:
  - a set of *values*
  - a *data representation,* which is common to all these values, and
  - a set of *operations,* which can be applied uniformly to all these values

Introduction Slide Number 5

Often a problem solution requires operations on data organised in specific ways. Example operations could be, for instance, " **addition**" to a list of words in lexicographic order, " **deletion**" of a file from a given tree structure of folders and sub-folders, or " **retrieval**" of the i-th inserted item in a stack.

Data abstraction is not just another data type but it's a means for specifying a required collection of structured data and think in terms of *what*operations you can apply to that collection independently of *how*you implement it. The notion of Abstract Data Type is therefore the result of applying abstraction to the notion of data type. **Let's see what we abstract away from a notion of Data Type when we define an ADT**.

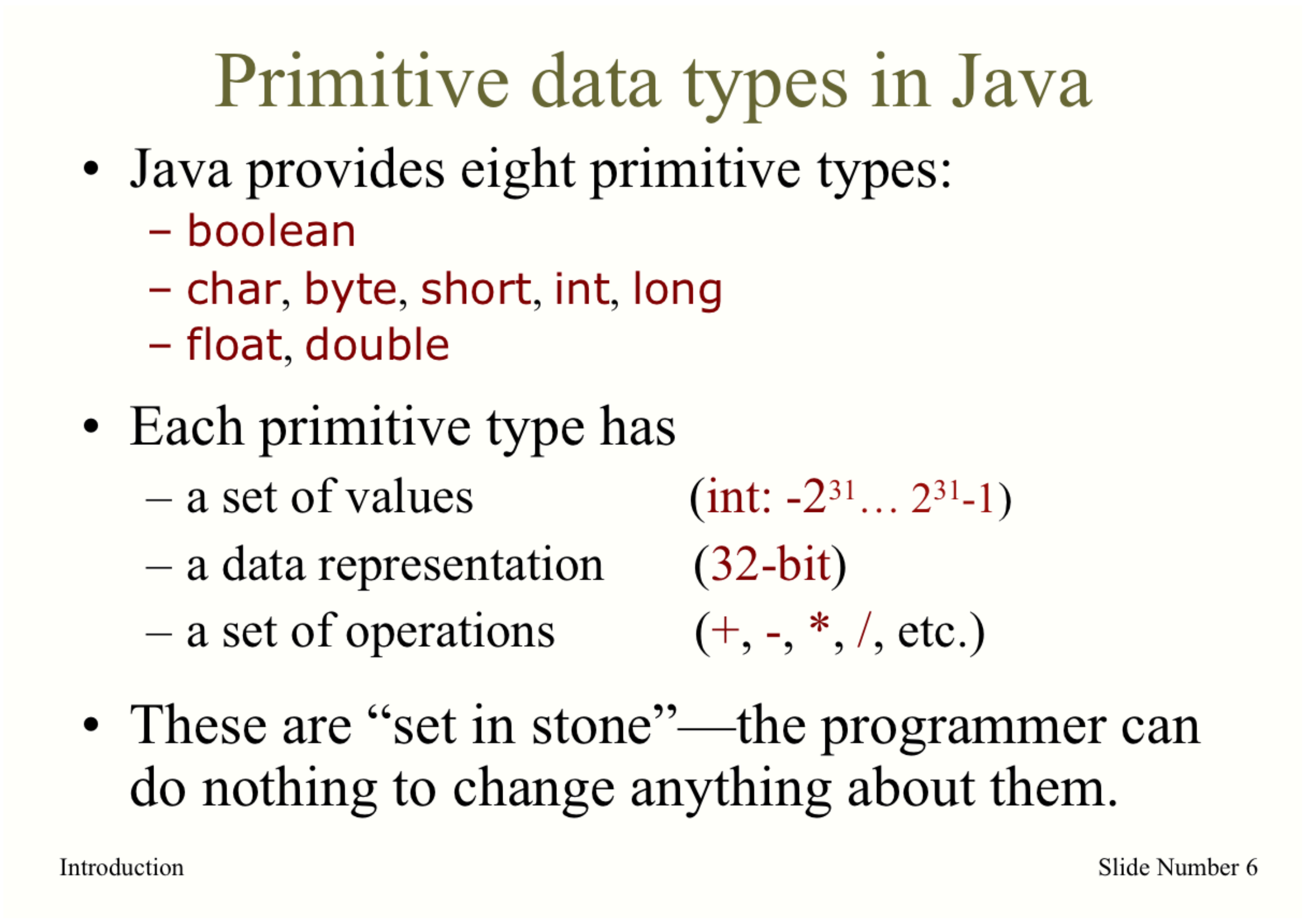
In the previous part of this course you have been introduced different data types. A Data Type is normally defined in terms of

(i) set of values that the given data type refers to (i.e. values that a variable of that data type is allowed to have),

(ii) data representation, which defines how data of the given data type are stored in memory during the execution of a program and

(iii) operations that can be applied to variables of the given data type.

These parameters depend on the particular data type and on whether the type is *primitive* or *reference-based*.

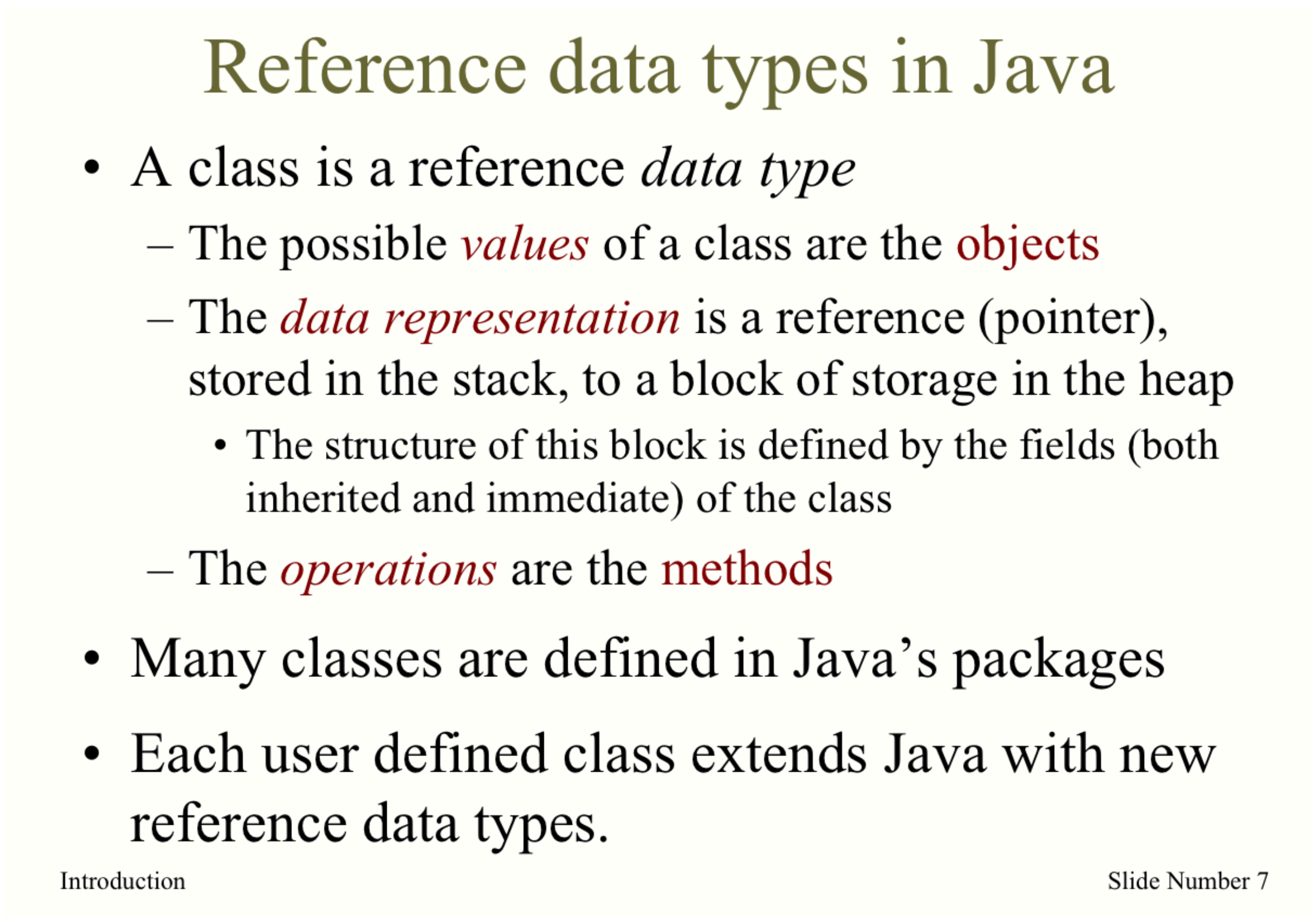

Primitive data types in Java

- Java provides eight primitive types:
  - boolean
  - char, byte, short, int, long
  - float, double
- Each primitive type has
  - a set of values (int: $-2^{31} \ldots 2^{31}-1$)
  - a data representation (32-bit)
  - a set of operations (+, -, *, /, etc.)
- These are "set in stone"—the programmer can do nothing to change anything about them.

Introduction Slide Number 6

Java includes the 8 primitive data types listed above. Each of them refers to a specific set of possible values, For instance, the **int** primitive type refers to all integers values from -2^{31} to $2^{31} – 1$. This is because the representation of integers data in Java is in terms of 32-bit binary code in the execution stack. Variables of primitive data are stored in the execution stack. The set of operations depend on the particular data type.

These types are the building blocks of any more complex data type that a programmer might need to define to solve a given problem. They are therefore predefined simple data types whose features cannot be changed by the user.



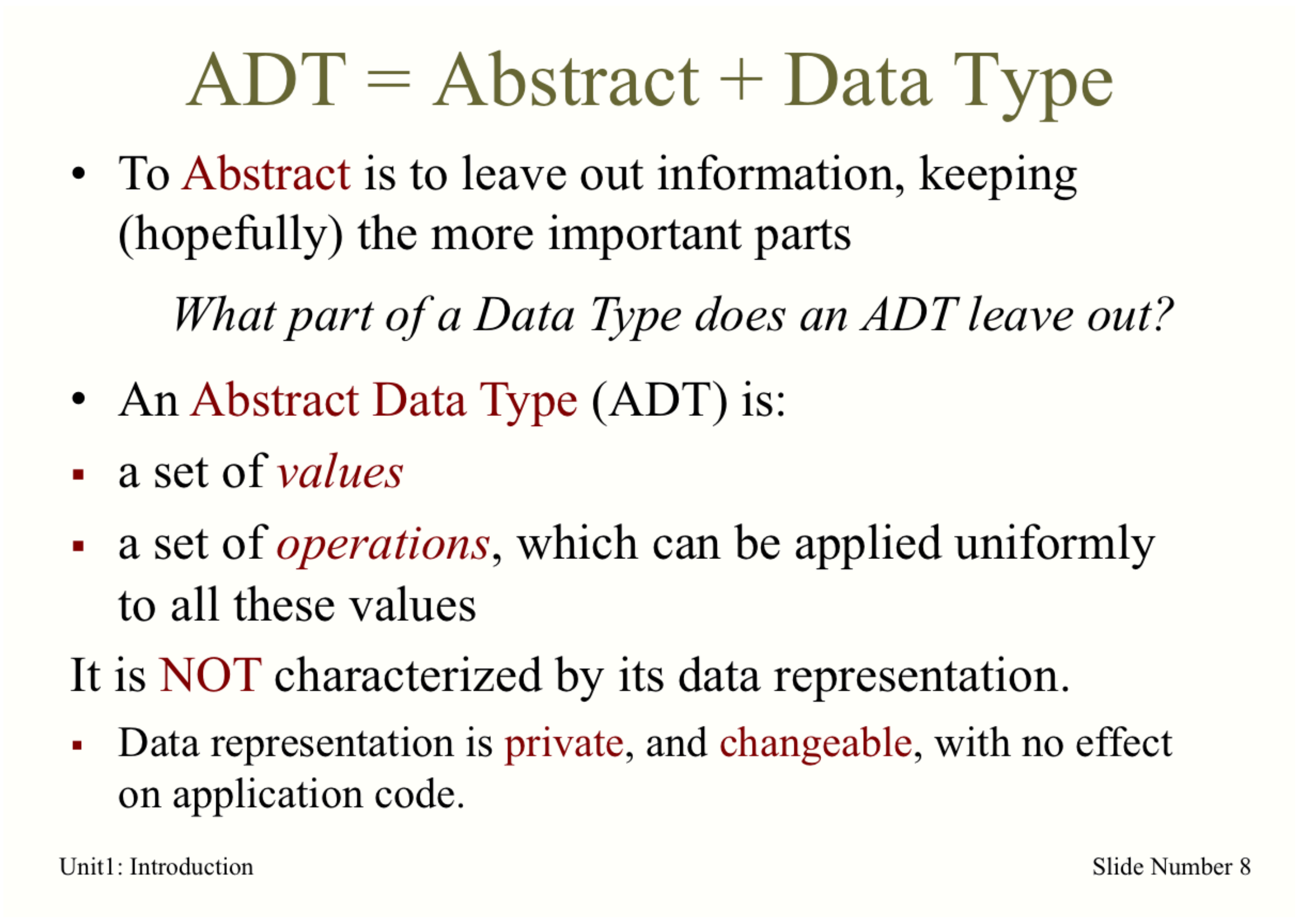
Reference data types in Java

- A class is a reference *data type*
  - The possible *values* of a class are the objects
  - The *data representation* is a reference (pointer), stored in the stack, to a block of storage in the heap
    - The structure of this block is defined by the fields (both inherited and immediate) of the class
  - The *operations* are the methods
- Many classes are defined in Java's packages
- Each user defined class extends Java with new reference data types.

Introduction Slide Number 7

Different is the case of *reference* data types. These referred to data with a certain structure. For each dynamically created value of a reference data type a reference variable is created and stored in the execution stack in order to point to the " **block" in the heap where the value is stored.** Examples of reference data types include class types, interfaces types and array types. Values of reference types are therefore references to objects (e.g. dynamically created instances of class types or array types). The data representation is *fixed* by the fields of the particular class type.

For predefined reference data type, such as **String and array**, the operations are the Java predefined methods provided by the Java API for these classes. User defined classes constitute new reference data types with their own operations (given by the methods defined in their respective classes). Their values are still references to the objects that are dynamically created.



ADT = Abstract + Data Type

- To Abstract is to leave out information, keeping (hopefully) the more important parts

  What part of a Data Type does an ADT leave out?

- An Abstract Data Type (ADT) is:
- a set of *values*
- a set of *operations*, which can be applied uniformly to all these values

It is NOT characterized by its data representation.

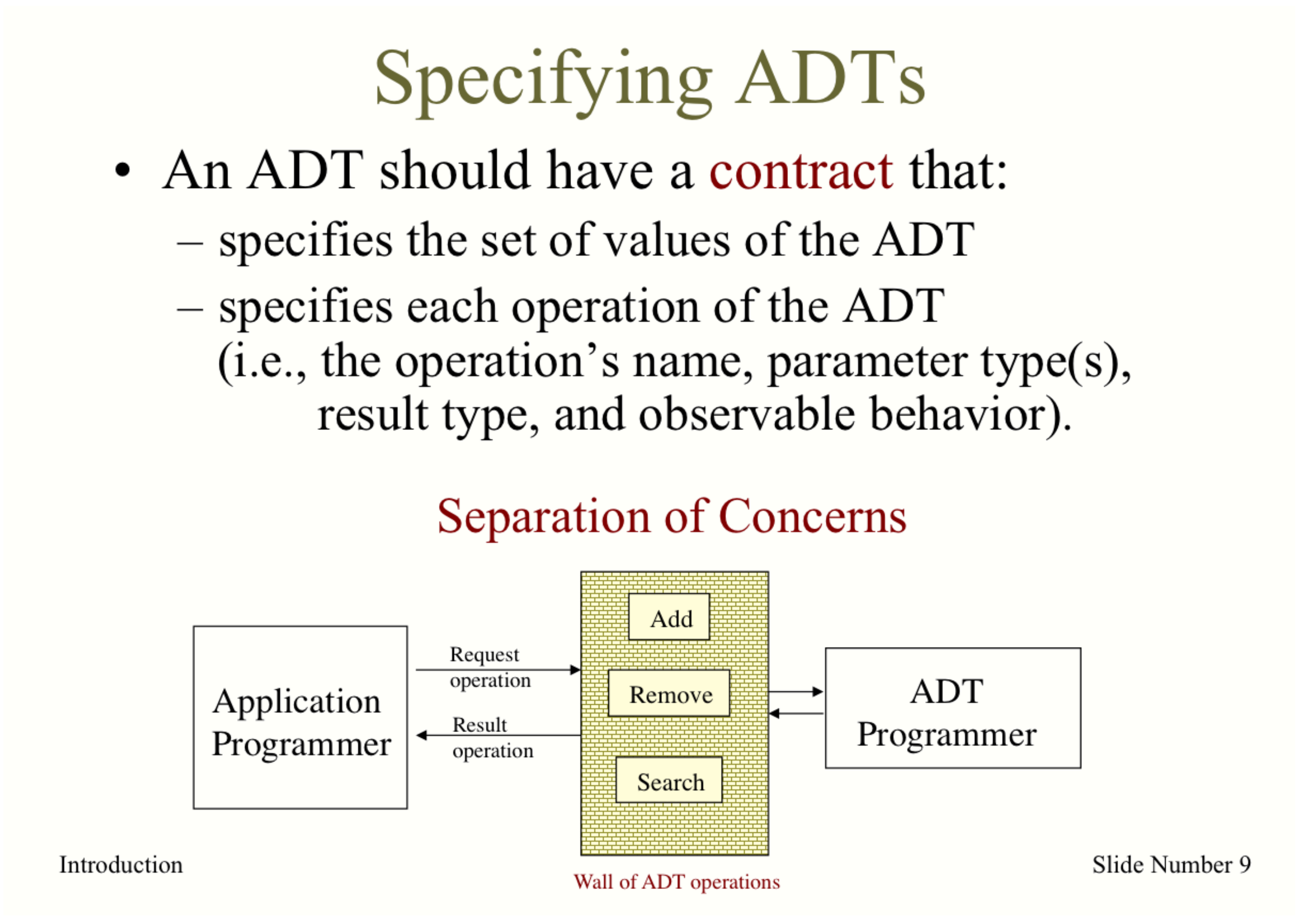
- Data representation is private, and changeable, with no effect on application code.

Unit1: Introduction Slide Number 8

With Abstract Data Type, **we define new data types by abstracting away their data representation**. ADT are even more complex data types that build upon primitive and reference data types. They are also reference data type as reference variables are created when object instances of an ADT are created (e.g. object instance of type List). Such object instances are pointed by their reference variables.

But an ADT definition does not have a specific data representation. It's a abstraction of the notion of data type in the sense that it is independent from the data representation used to store the data. The data representation is private to the ADT and is changeable, without impacting upon the application code that uses the given ADT.

To specify an ADT we need to define the set of values that the ADT can assume and the operations that are allowed to manipulate the data. For instance, in the case of an **ADT list of words in lexicographic order** , the set of values are possible lists of words in lexicographic order and operations include search the list for a given word, manipulate the list by adding a new word, deleting am existing word, etc..



Specifying ADTs

- An ADT should have a contract that:
  - specifies the set of values of the ADT
  - specifies each operation of the ADT (i.e., the operation's name, parameter type(s), result type, and observable behavior).

Separation of Concerns

Application Programmer
Request operation
Result operation
Add
Remove
Search
ADT Programmer
Wall of ADT operations

Introduction Slide Number 9

Defining an ADT means defining a contract between the application programmer and the ADT programmer. The contract needs to state the set of value and the specific operations, in terms of the operation name, parameters in input, the output type of the operation, and its own pre and post-conditions. Specification of the operation defines, for instance, the expected observable behavior of the collection of data once the operation has been performed.

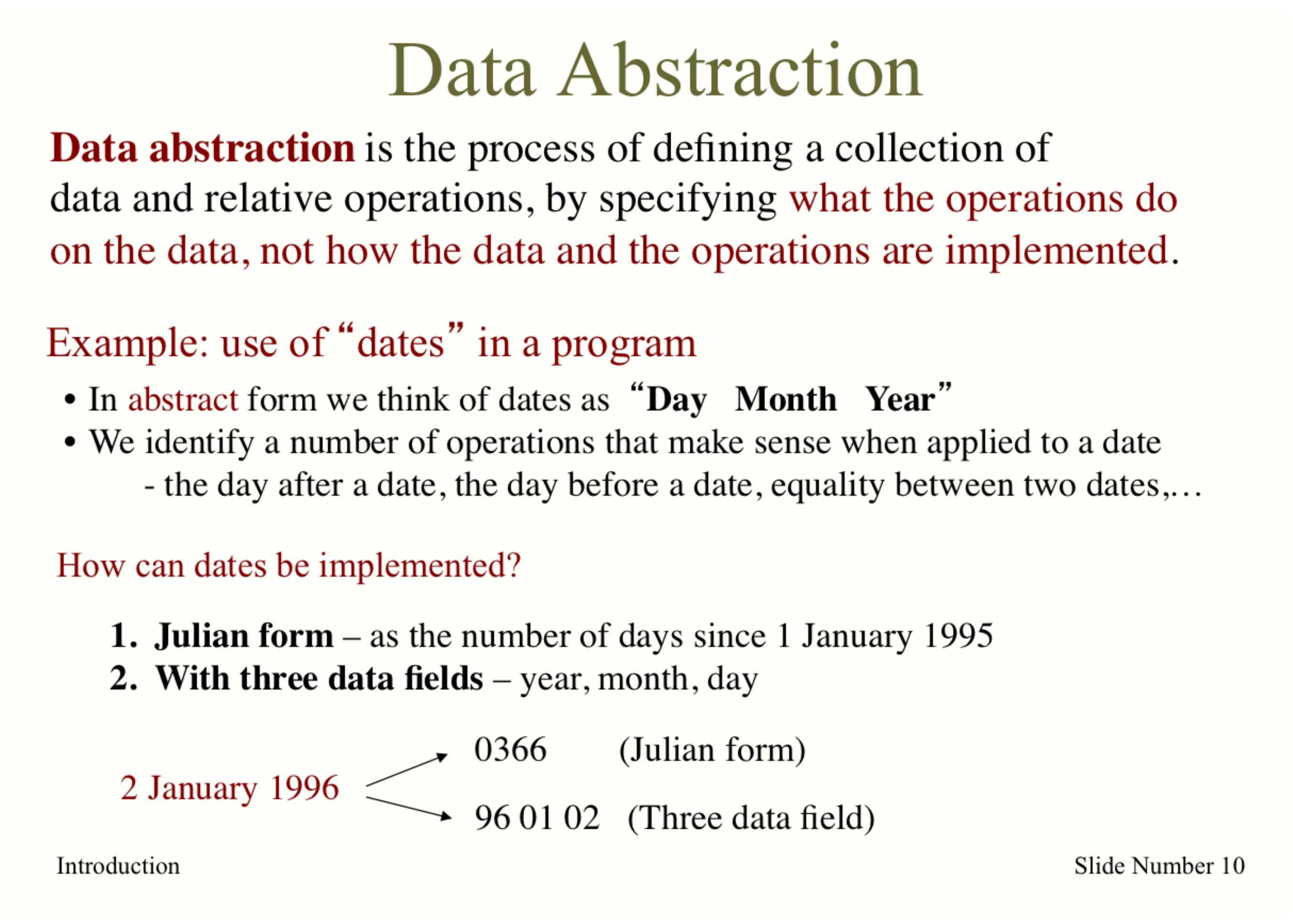
Using ADTs in complex programming task enables a clear separation of concerns between the application environment that uses the ADT and the implementation of the ADT Itself. The data representation is private to the implementation part of the ADT, so allowing the application environment to use the ADT without knowing how it is implemented. For example, in the case of searching a list for a given word the observable behavior from the application program point of view is the outcome of the search (i.e. whether the word is in the collection of data or not), which is independent from the specific algorithm used to search the word (bubble sort, binary chop, etc..)

It is a clear separation of concerns in that:

i) The ADT programmer is not concerned with what applications the ADT is used for. But he/she is concerned with making the operations efficient and correct, with respect to the set of values of the ADT and their specification.

ii) The application programmer is not concerned with how the ADT is implemented, but he/she wants just to use the ADT to get a job done.

This separation of concerns is essential for designing and implementing large, complex systems.



Data Abstraction

Data abstraction is the process of defining a collection of data and relative operations, by specifying what the operations do on the data, not how the data and the operations are implemented.

Example: use of "dates" in a program

- In abstract form we think of dates as "**Day Month Year**"
- We identify a number of operations that make sense when applied to a date
  - the day after a date, the day before a date, equality between two dates,…

How can dates be implemented?

1. **Julian form** – as the number of days since 1 January 1995
2. **With three data fields** – year, month, day

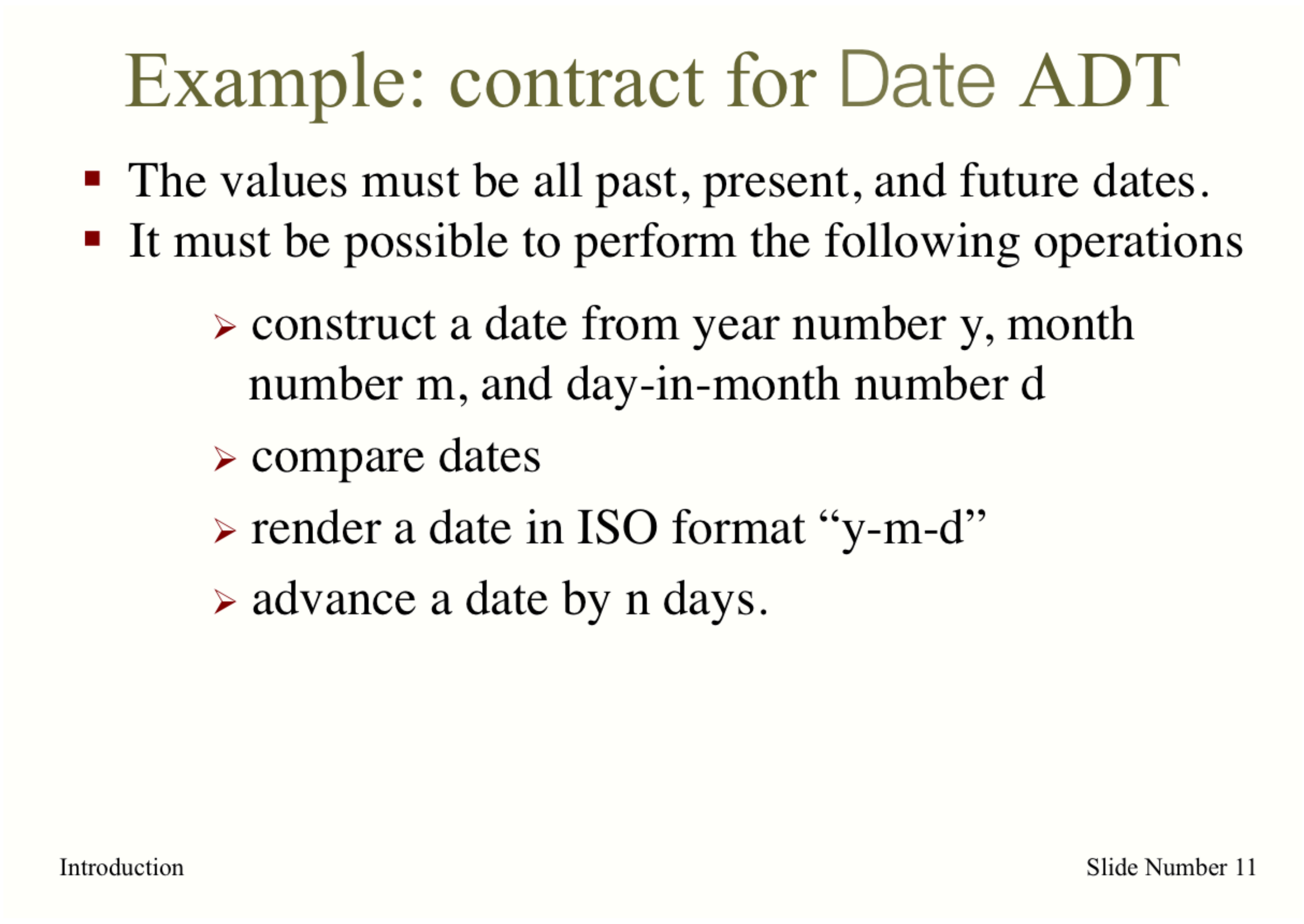
$$\begin{tikzcd} & \text{0366 (Julian form)} \\ \text{2 January 1996} \arrow[ur] \arrow[r] & \text{96 01 02 (Three data field)} \end{tikzcd}$$

Introduction Slide Number 10

In summary, **Data abstraction** is therefore the process of defining a collection of data and operations allowed on this collection. Helps you think in terms of *what*you can do to a collection of data independently of *how*you implement it. **Data abstraction allows you to think abstractly about data** ; to use in a program a given collection of data organised in a particular way, just by knowing what the specified operations can do to the data and without knowing how the data are stored and the operations implemented. It allows, therefore, to think about problem solutions by using only the abstract definitions of data types and therefore remaining independent from the particular implementation of the data collection. In this way changes over the implementation of the data collections will not affect the solution of the given initial problem.

Consider, for example, the use of a type date in a program. In abstract form, we think of a date as given by a Day, a Month, and a Year. In defining this data abstraction, we could just simply think of this common form of a date and specify a number of operations that make sense when applied to a date, as for instance the operation of next-day , the day before a given date, the date after a specified period before/ after a given date, whether two dates are the same, etc. We can completely ignore how the date is actually implemented.

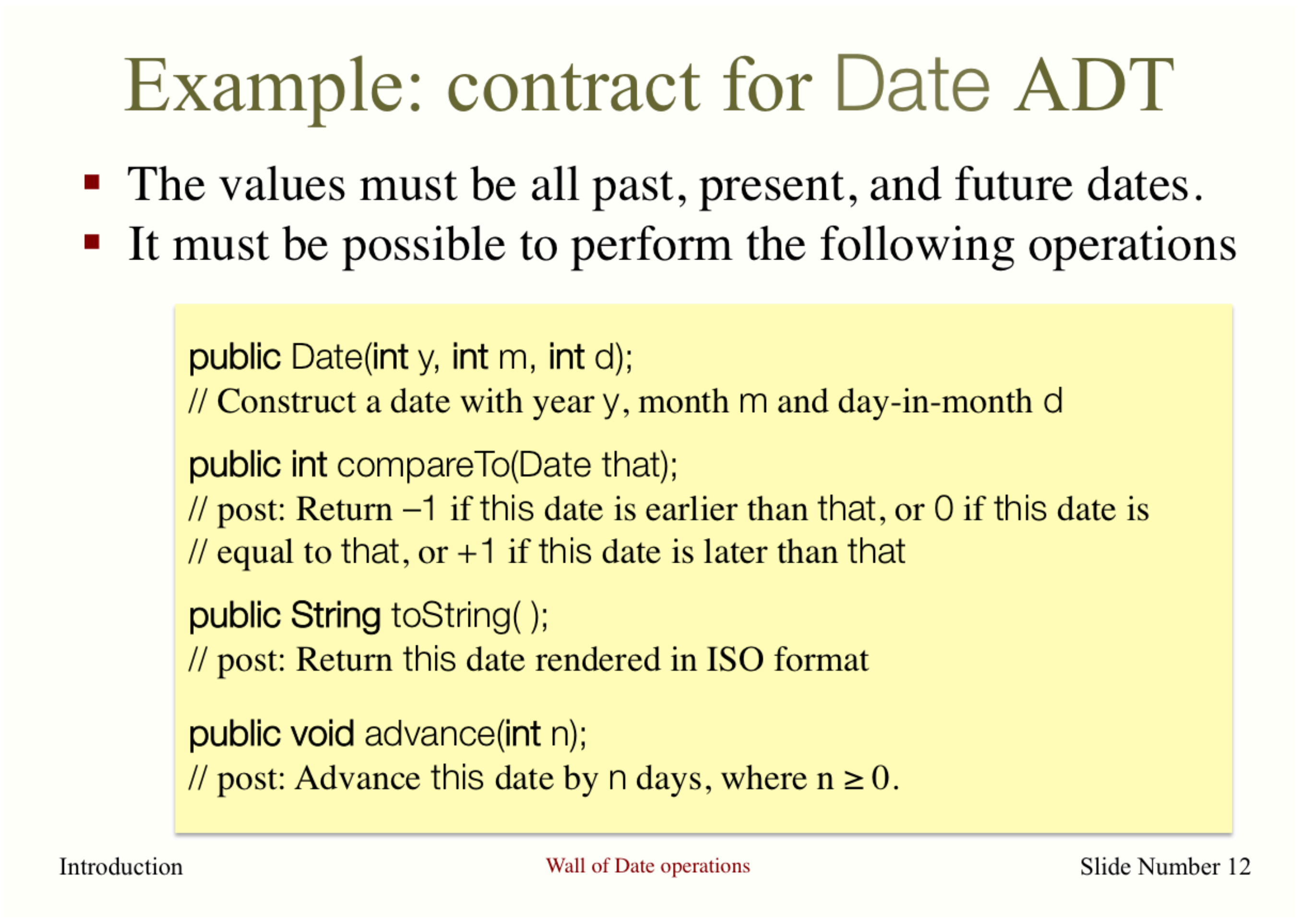
We illustrate in the next few slides the definition of this ADT.


Example: contract for Date ADT

- The values must be all past, present, and future dates.
- It must be possible to perform the following operations
  - construct a date from year number y, month number m, and day-in-month number d
  - compare dates
  - render a date in ISO format “y-m-d”
  - advance a date by n days.

Introduction Slide Number 11

Possible values of an ADT Date are all possible dates (past, present and future dates). Different operations on dates can be defined. Here are some examples of operations.

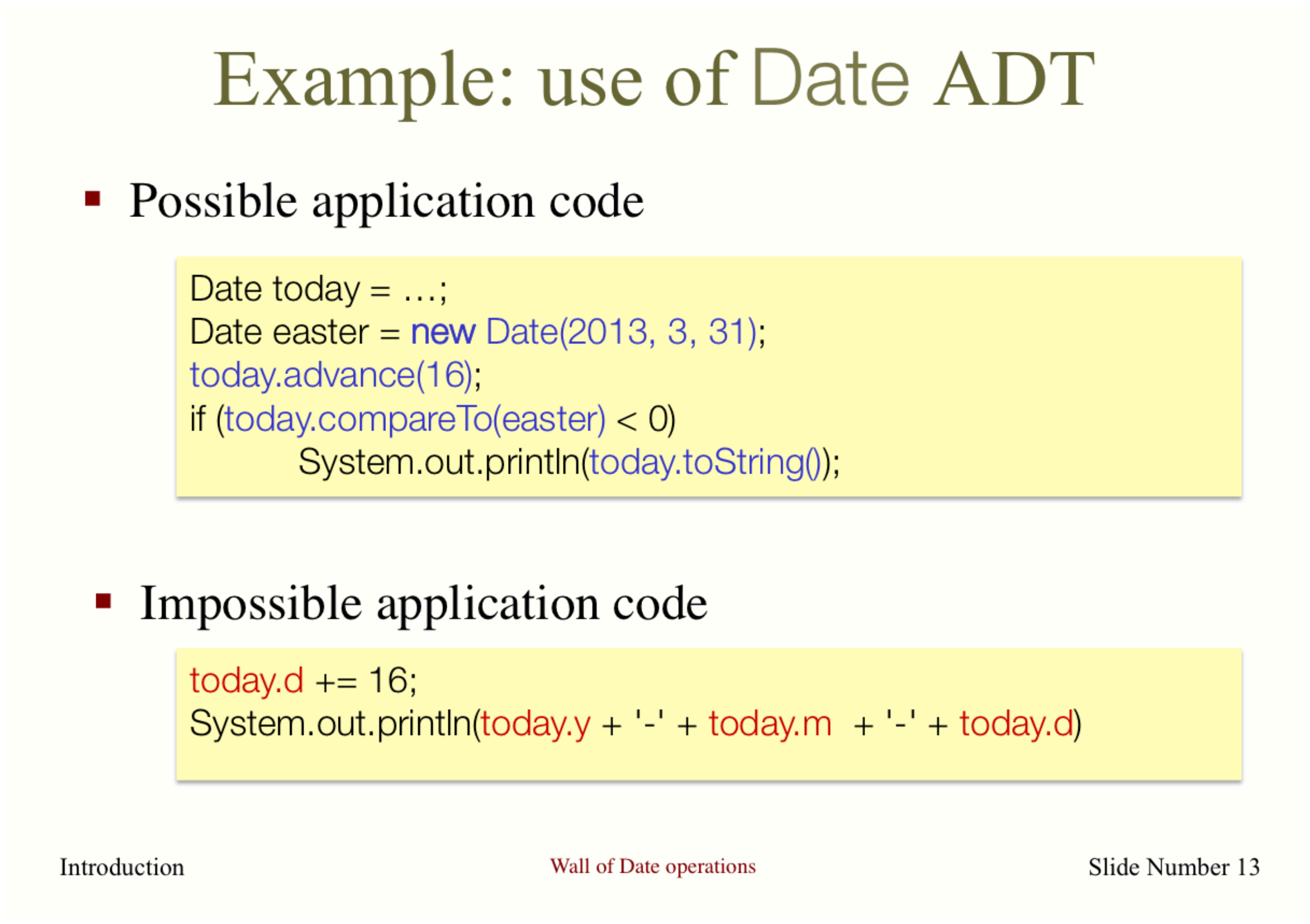

Example: contract for Date ADT

- The values must be all past, present, and future dates.
- It must be possible to perform the following operations

```
public Date(int y, int m, int d);
// Construct a date with year y, month m and day-in-month d

public int compareTo(Date that);
// post: Return –1 if this date is earlier than that, or 0 if this date is
// equal to that, or +1 if this date is later than that

public String toString( );
// post: Return this date rendered in ISO format

public void advance(int n);
// post: Advance this date by n days, where n ≥ 0.
```

Introduction Wall of Date operations Slide Number 12

The operations need to be specified in terms of type of input parameters, output parameters, and pre and post-conditions. Assuming that the notion of a data is given by day d, month m and year y. One of the operations is the construction of a data value of type Date (this is the operation given above). The other operations specify the operations given in the previous slide. As you can see the post-condition describe what the observable behavior from the point of view of the application environment that uses this operation. It does not refer to the algorithm used to implement these operations.



Example: use of Date ADT

- Possible application code

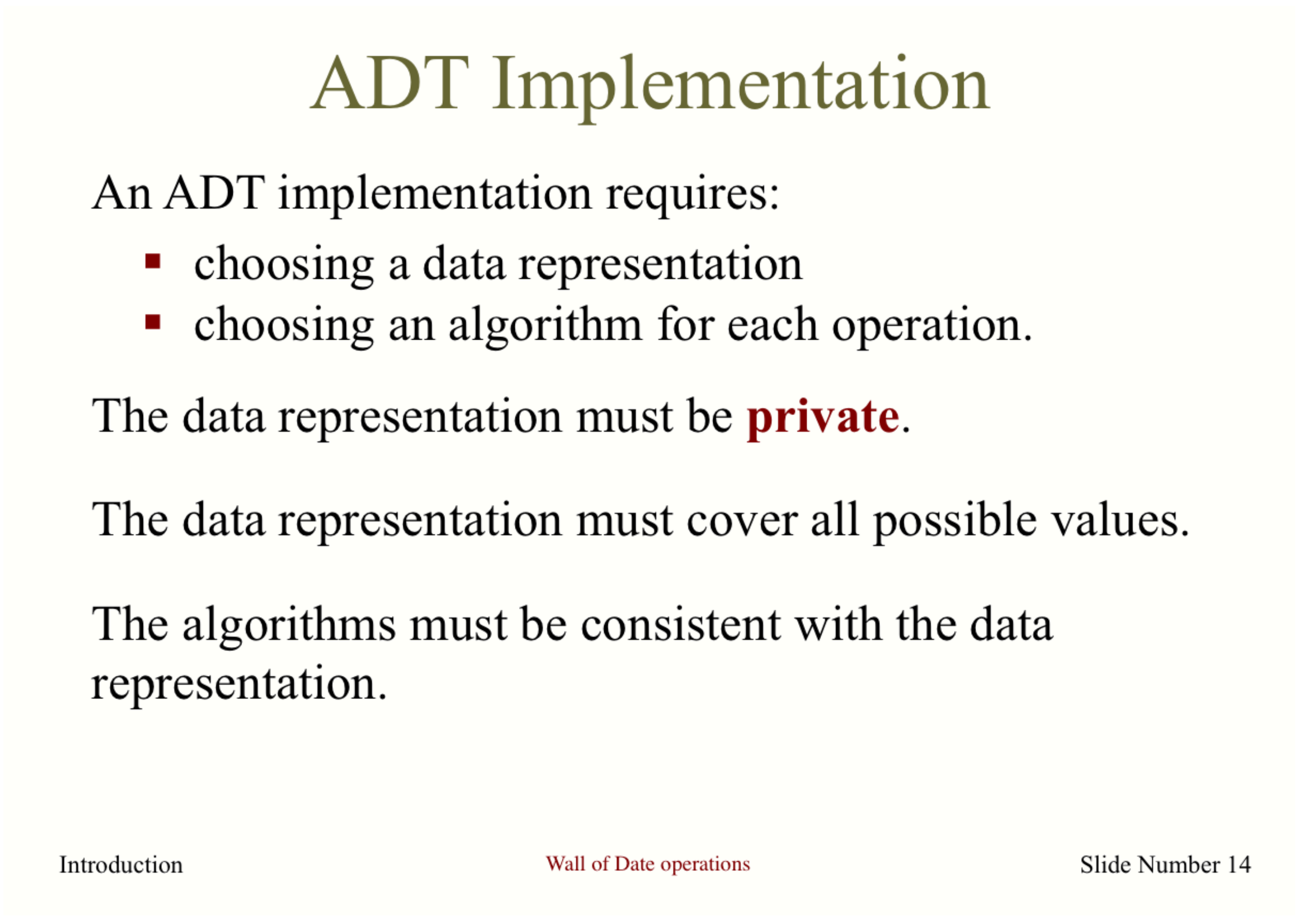
```
Date today = …;
Date easter = new Date(2013, 3, 31);
today.advance(16);
if (today.compareTo(easter) < 0)
        System.out.println(today.toString());
```

- Impossible application code

```
today.d += 16;
System.out.println(today.y + '-' + today.m  + '-' + today.d)
```

Introduction   Wall of Date operations   Slide Number 13

From the application environment, given the specification of the operations of an ADT Date, objects of type Date can be created and used. The above code shows that should be possible and what should not be allowed to do from the application program. A Date object can be created. It can be advanced of 16 days and the new date value can be viewed using the ISO format "y-m-d".
The application program should not be able to access directly the data structure used to store a Date (i.e. the value of d, m, and y directly).



ADT Implementation

An ADT implementation requires:

- choosing a data representation
- choosing an algorithm for each operation.

The data representation must be **private**.

The data representation must cover all possible values.

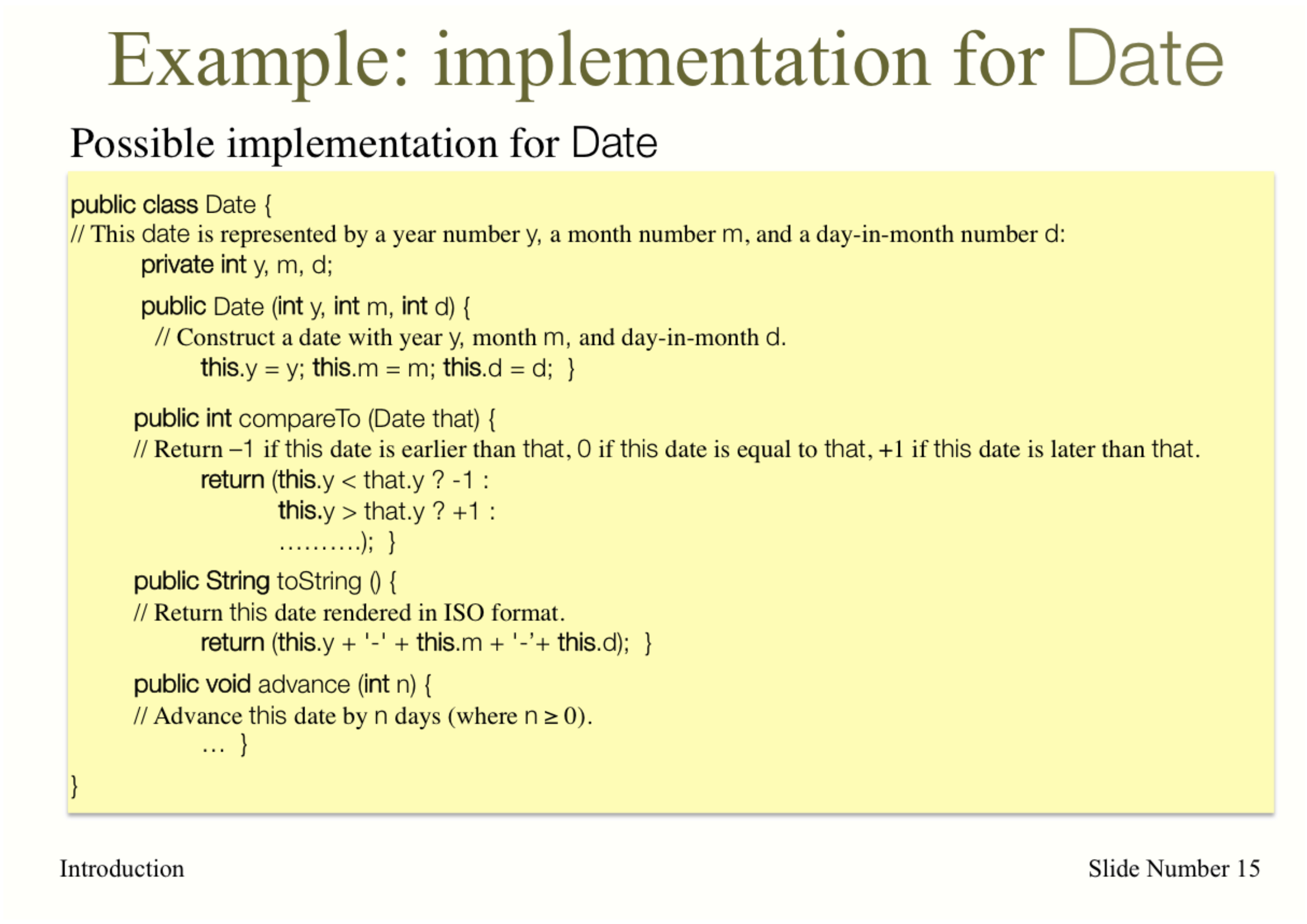
The algorithms must be consistent with the data representation.

Introduction Wall of Date operations Slide Number 14

From the ADT implementation point of view, different data representation can be chosen. An ADT implementer needs to fix the most efficient data representation and based on this, implement the ADT operations in a way that are correct with respect to their specification. The important feature of ADTs is that data representation has to be private to the code of the ADT .

In the case of the ADT Date, different data representations can be chosen. We could choose to implement it as a class with just one integer data field that gives the Julian representation of a date, i.e. as the number of elapsed days since a known start date (e.g. since 1 January 1995). Or it could be implemented as a class with three fields, called respectively Year, Month, Day. So, for example, for a given date 2 January 1996 , we would have the Julian form representation 0366 , or the three fields representation 96 01 02.

The implementation of the given ADT operations maybe more or less complex depending on the chosen data representation. We illustrate in the next few slides example implementations of the given operations.



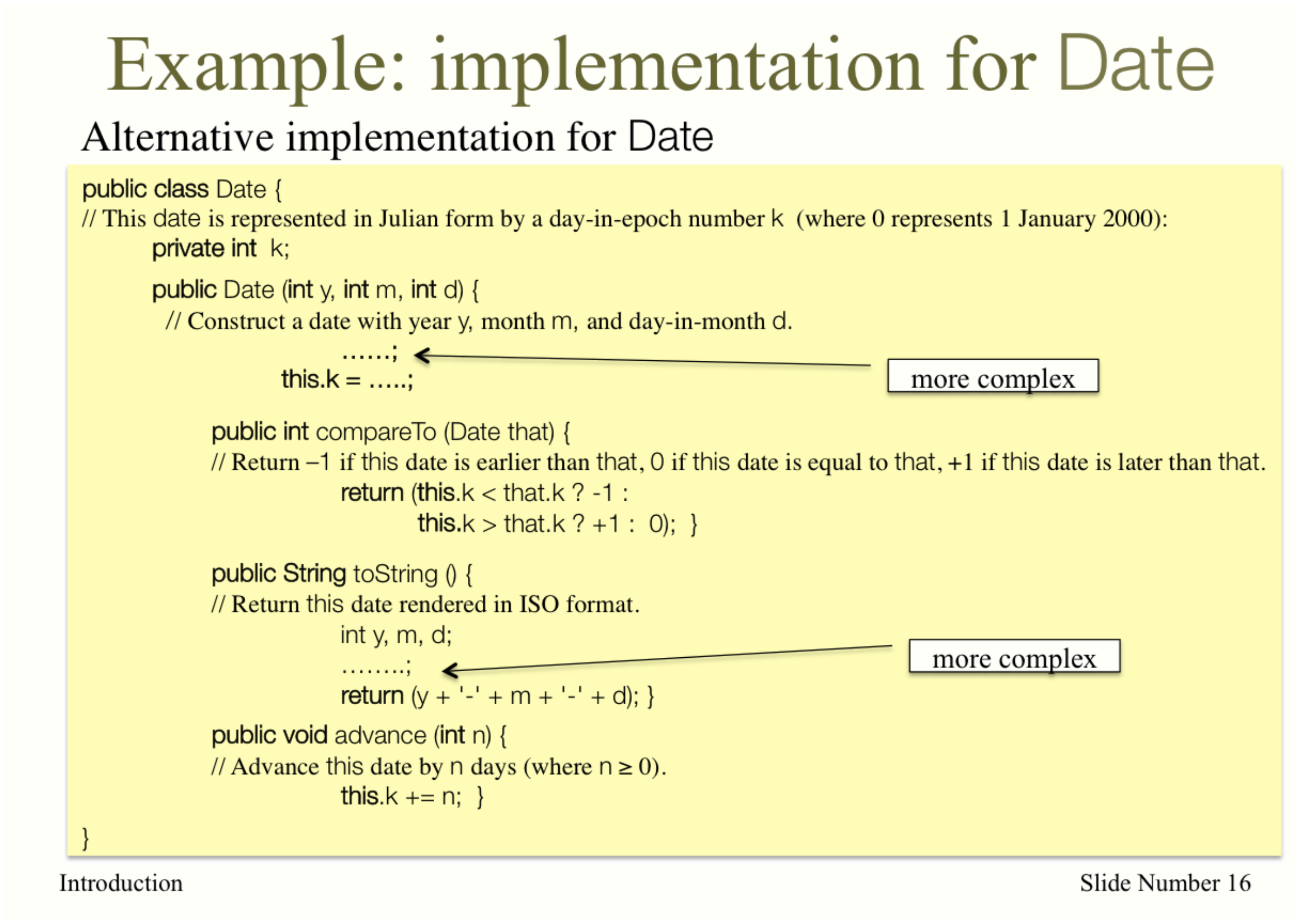
Example: implementation for Date

Possible implementation for Date

```
public class Date {
// This date is represented by a year number y, a month number m, and a day-in-month number d:
        private int y, m, d;
        public Date (int y, int m, int d) {
          // Construct a date with year y, month m, and day-in-month d.
                this.y = y; this.m = m; this.d = d;  }
       public int compareTo (Date that) {
       // Return –1 if this date is earlier than that, 0 if this date is equal to that, +1 if this date is later than that.
                return (this.y < that.y ? -1 :
                           this.y > that.y ? +1 :
                           ……….);  }
       public String toString () {
       // Return this date rendered in ISO format.
                return (this.y + '-' + this.m + '-'+ this.d);  }
       public void advance (int n) {
       // Advance this date by n days (where n ≥ 0).
                …  }
}
```

Introduction Slide Number 15

In this example we have chosen to use as data representation of a ADT Date, the three fields d, m and y. The constructor is therefore very simple, and so is the operation for render the value of a Date in the ISO Format. More complex are the implementations of the operation advance and compareTo as using this data representation the advance of a Date by n given days has to consider the cases when the month, the year or both have to advance as well.



Example: implementation for Date

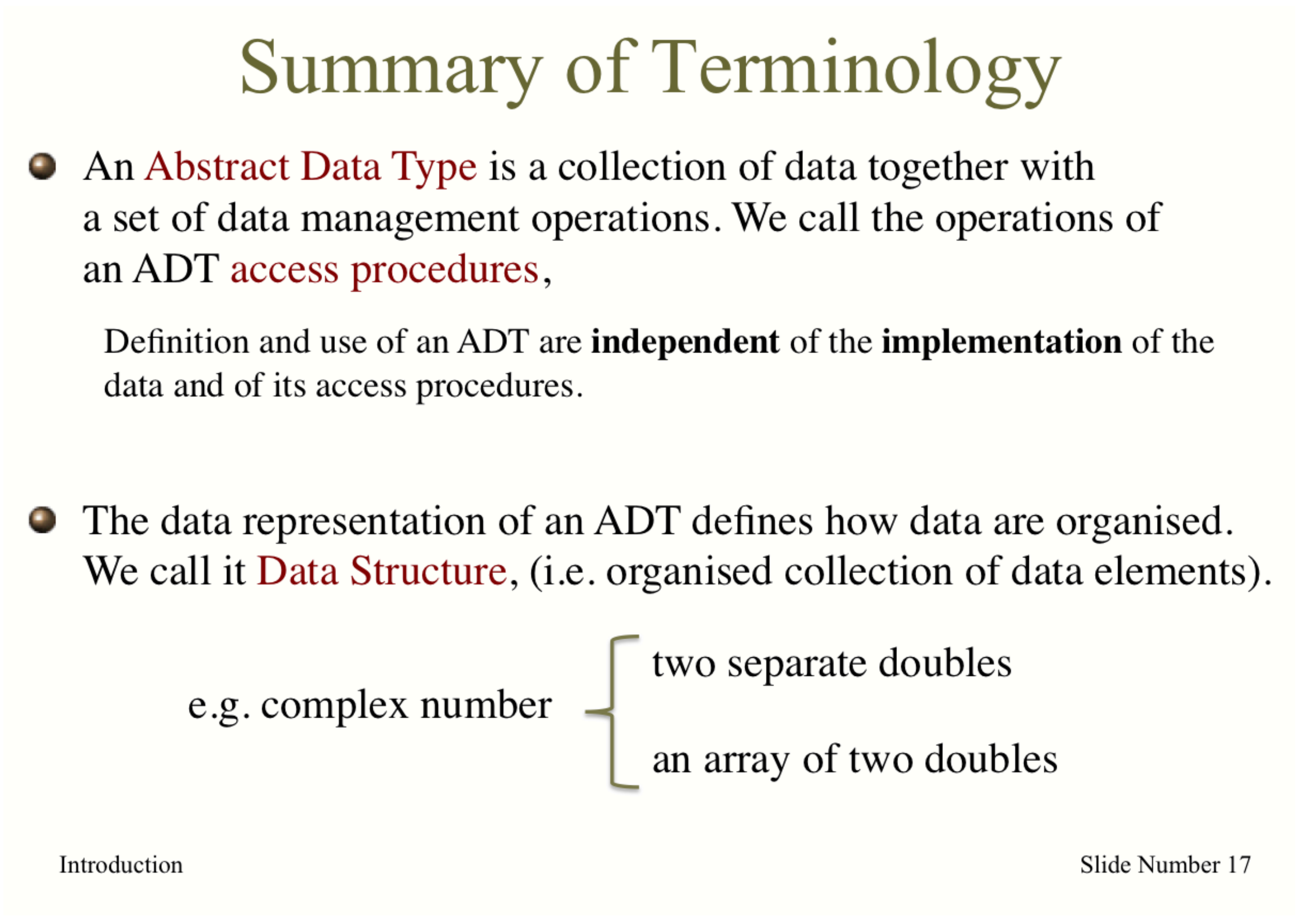
Alternative implementation for Date

```
public class Date {
// This date is represented in Julian form by a day-in-epoch number k  (where 0 represents 1 January 2000):
        private int  k;

        public Date (int y, int m, int d) {
          // Construct a date with year y, month m, and day-in-month d.
                      ……;   <—— more complex
                this.k = …..;

              public int compareTo (Date that) {
              // Return –1 if this date is earlier than that, 0 if this date is equal to that, +1 if this date is later than that.
                      return (this.k < that.k ? -1 :
                              this.k > that.k ? +1 :  0);  }

              public String toString () {
              // Return this date rendered in ISO format.
                      int y, m, d;
                      ……..;   <—— more complex
                      return (y + '-' + m + '-' + d); }
              public void advance (int n) {
              // Advance this date by n days (where n ≥ 0).
                      this.k += n;  }
}
```

Introduction   Slide Number 16

In this case, we have instead used the Julian form representation of the notion of Date. As you can see the construct and the operation for outputting the date in ISO format require now more complex algorithms, whereas the other two operations are much simpler.

The choice of the particular data representation depends very much on the use that the application program has to make of the ADT Date, where instead the application environment should be completely independent on the data representation. Different data representations can be chosen in different application environment context in order to maximize the efficiency of the application program.



Summary of Terminology

- An Abstract Data Type is a collection of data together with a set of data management operations. We call the operations of an ADT access procedures,

  Definition and use of an ADT are **independent** of the **implementation** of the data and of its access procedures.

- The data representation of an ADT defines how data are organised. We call it Data Structure, (i.e. organised collection of data elements).

  e.g. complex number { two separate doubles / an array of two doubles }

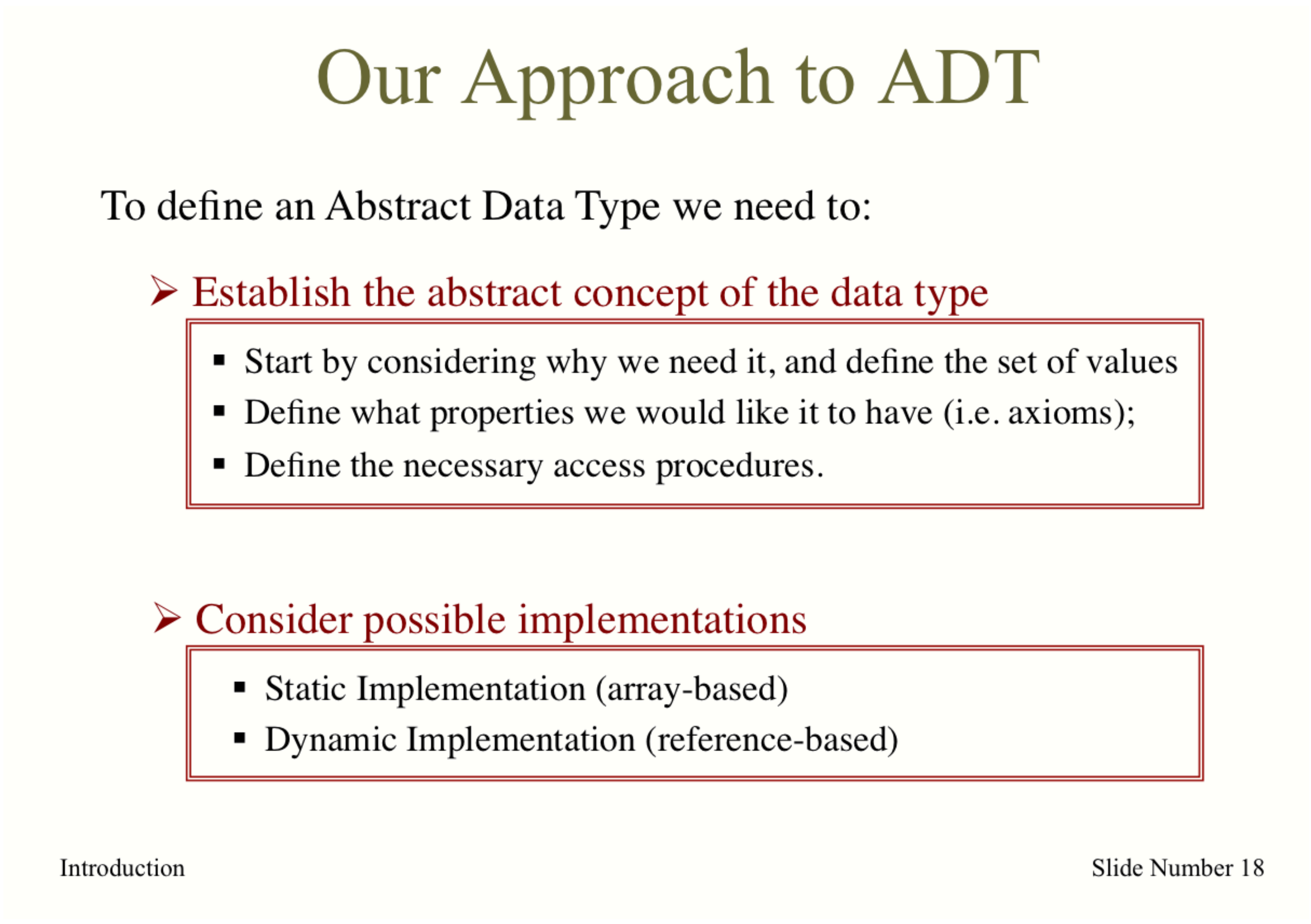
Introduction   Slide Number 17

We introduce here some basic terminology. An **Abstract Data Type** (ADT) is a collection of data together with a set of operations defined on that data. The description of an ADT must be rigorous enough to specify completely the effect that each of its operations has on the data, yet it must not specify how to store the data, nor how to implement the operations. Throughout this part of the course we will call the operation of an ADT *access procedures*.

When we implement an ADT we choose a particular data representation. Many kinds of data consist of multiple parts, organized (structured) in some way. We will therefore refer to these data representation as **data structure**. A data structure needs to be chosen in order to implement a given ADT.

Defining an ADT can be seen as building a wall (of operations) around a given data structure so that the interaction between the application program and the given data structure can happen via and controlled by only the operations provided by the ADT definition.

Whenever a program has to perform operations on complex data structures that are not directly supported by the programming language, then an abstract data type can be defined.



Our Approach to ADT

To define an Abstract Data Type we need to:

➢ Establish the abstract concept of the data type

- Start by considering why we need it, and define the set of values
- Define what properties we would like it to have (i.e. axioms);
- Define the necessary access procedures.

➢ Consider possible implementations

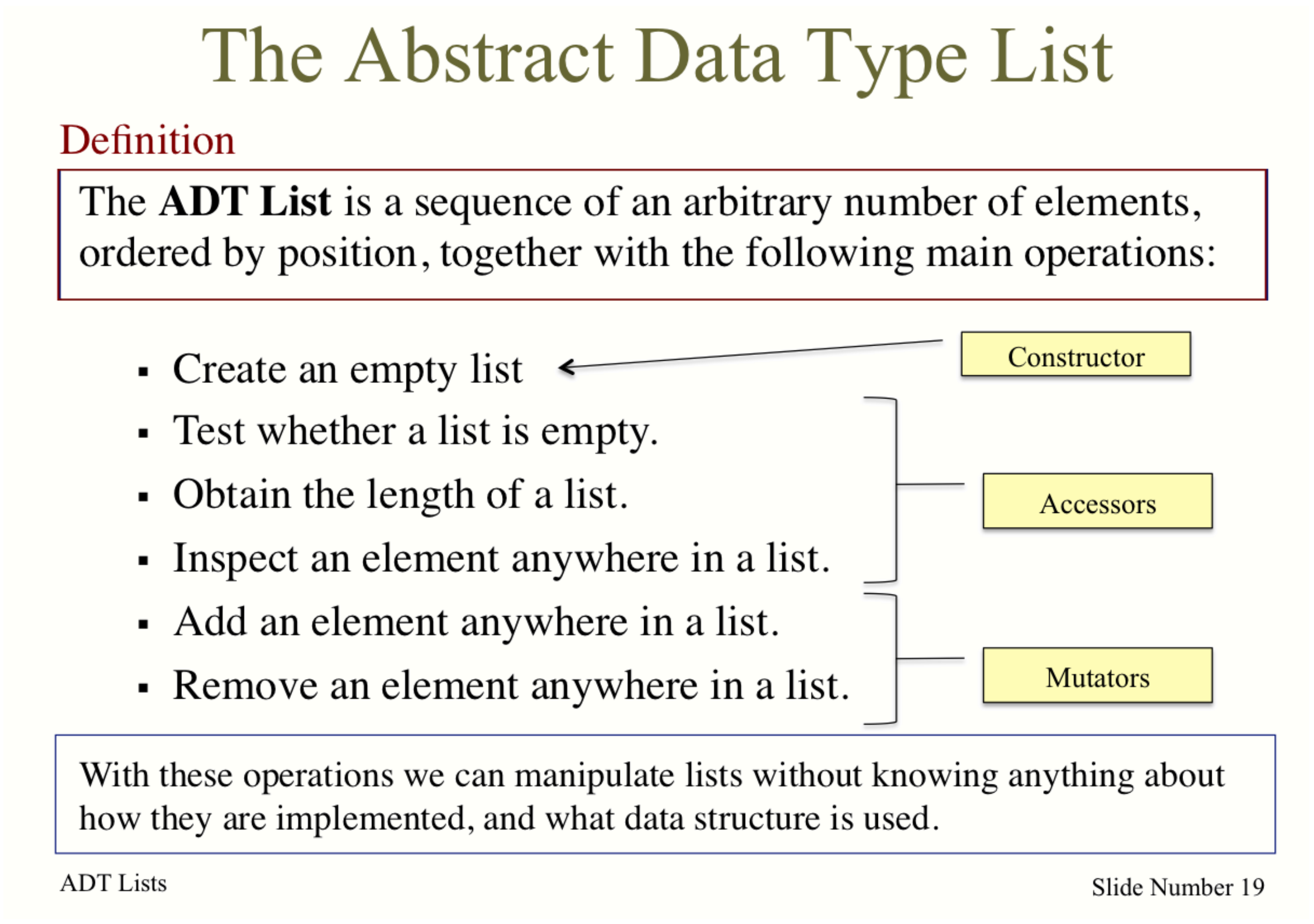
- Static Implementation (array-based)
- Dynamic Implementation (reference-based)

Introduction Slide Number 18

For the remainder of this course we will look at particular types of ADTs. Our general methodology for defining Abstract Data Types will be composed of two main parts: (a) the definition of a particular ADT, and (b) its implementation.

In (a) we will consider mainly why and when a particular ADT is needed, and what are its main properties and its main access procedures. We will refer to the properties of the ADT operation as **Axioms** .

As for the implementation, we will consider two different approaches, **static** and **dynamic** implementations, respectively. The word *static*means that the memory required by the data structure that support the storage and implementation of a particular abstract data type is allocated at compilation time, whereas the word *dynamic*means that the memory is allocated at run-time, as required by the underlying data structure. The data structures that we will consider for static implementations will mainly be based on arrays. These, in fact, cannot change their size or shape, once they have been declared. For the dynamic implementation we will use, instead, reference-based implementation.



The Abstract Data Type List

Definition

The **ADT List** is a sequence of an arbitrary number of elements, ordered by position, together with the following main operations:

- Create an empty list
- Test whether a list is empty.
- Obtain the length of a list.
- Inspect an element anywhere in a list.
- Add an element anywhere in a list.
- Remove an element anywhere in a list.

Constructor
Accessors
Mutators

With these operations we can manipulate lists without knowing anything about how they are implemented, and what data structure is used.

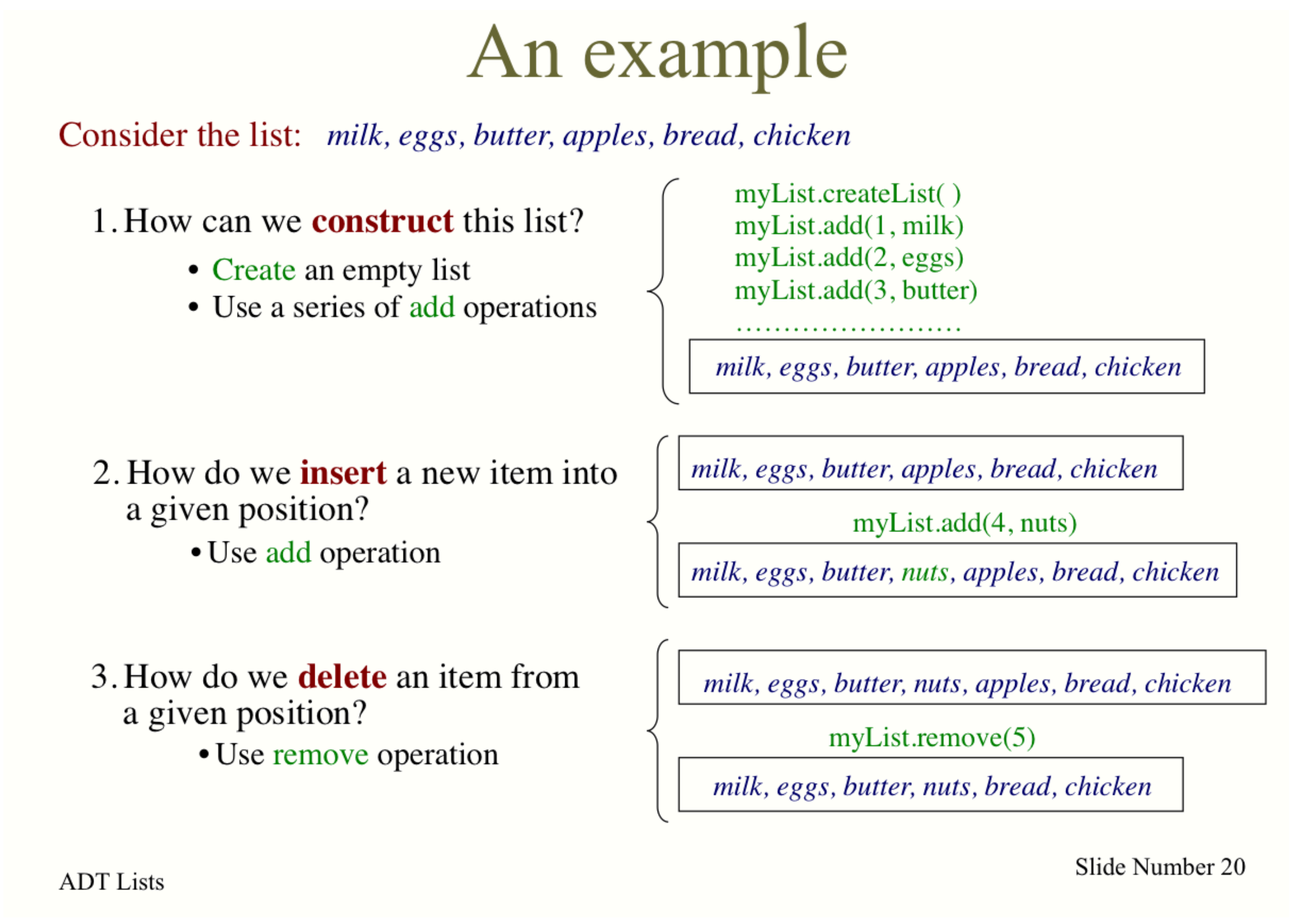
ADT Lists Slide Number 19

Let's start with considering the first type of ADT, a **list**. Think about a list that you might want to use, such as a list of important dates, or a list of appointments, or a list of graduated students in an academic year.

What are the main features of a list? Assuming that it is a neat one-column list, the elements in the list appear in a sequence. The list has one first element (the **head**) and one last element. Except for the first and the last, each element has one unique predecessor and one unique successor. So we can say that a list **is a linear sequence of elements**, where the elements are of the **same type**, and ordered by position (i.e. the first element in a list, the second element in a list, etc..). The elements do not need to be ordered by their value. The fact that a list is a **linear sequence** of items means simply that a list is a collection of elements that can be referenced by position number, and that given a position there is only one next element (if any) or next position.

To define a ADT list we need to fully specify the operations that can be performed on a list. For example, we might ask ourselves, where do we put new elements? To keep the list neat, we could just as well add the new element at the beginning of the list or add it at a certain position. We should be able to count the elements currently in a list, remove the element at a given position from the list, checking whether the list is empty, getting (e.g. retrieving) the element at a given position. In this slide, we have defined what we consider to be the main required access procedures of an ADT List. In general, the operations of an ADT can be grouped into *constructors* (operations that create an empty instance of an ADT), *accessors*, which are operations that allow to access to the elements in an ADT but do not change their value or the structure of the ADT, and *mutators* (or also called transformers) that change the elements in an ADT and the ADT structure, e.g. adding/deleting elements. These three groups have been highlighted above for the case of a List.

Different textbooks define different sets of access procedures for a List. Some might include, for instance, adding an element at the end of a list, make a list empty, replaces the element at position with a new element. However, all textbooks include the access procedures given in this slide. We will therefore consider these access procedures as the main core operations of an ADT List.



An example

Consider the list: *milk, eggs, butter, apples, bread, chicken*

1. How can we **construct** this list?
   - Create an empty list
   - Use a series of add operations

```
myList.createList( )
myList.add(1, milk)
myList.add(2, eggs)
myList.add(3, butter)
.......................
```

milk, eggs, butter, apples, bread, chicken

2. How do we **insert** a new item into a given position?
   - Use add operation

milk, eggs, butter, apples, bread, chicken

myList.add(4, nuts)

milk, eggs, butter, nuts, apples, bread, chicken

3. How do we **delete** an item from a given position?
   - Use remove operation

milk, eggs, butter, nuts, apples, bread, chicken

myList.remove(5)

milk, eggs, butter, nuts, bread, chicken

ADT Lists

Slide Number 20

To get an idea of how the operations of a List work, let's consider a grocery list

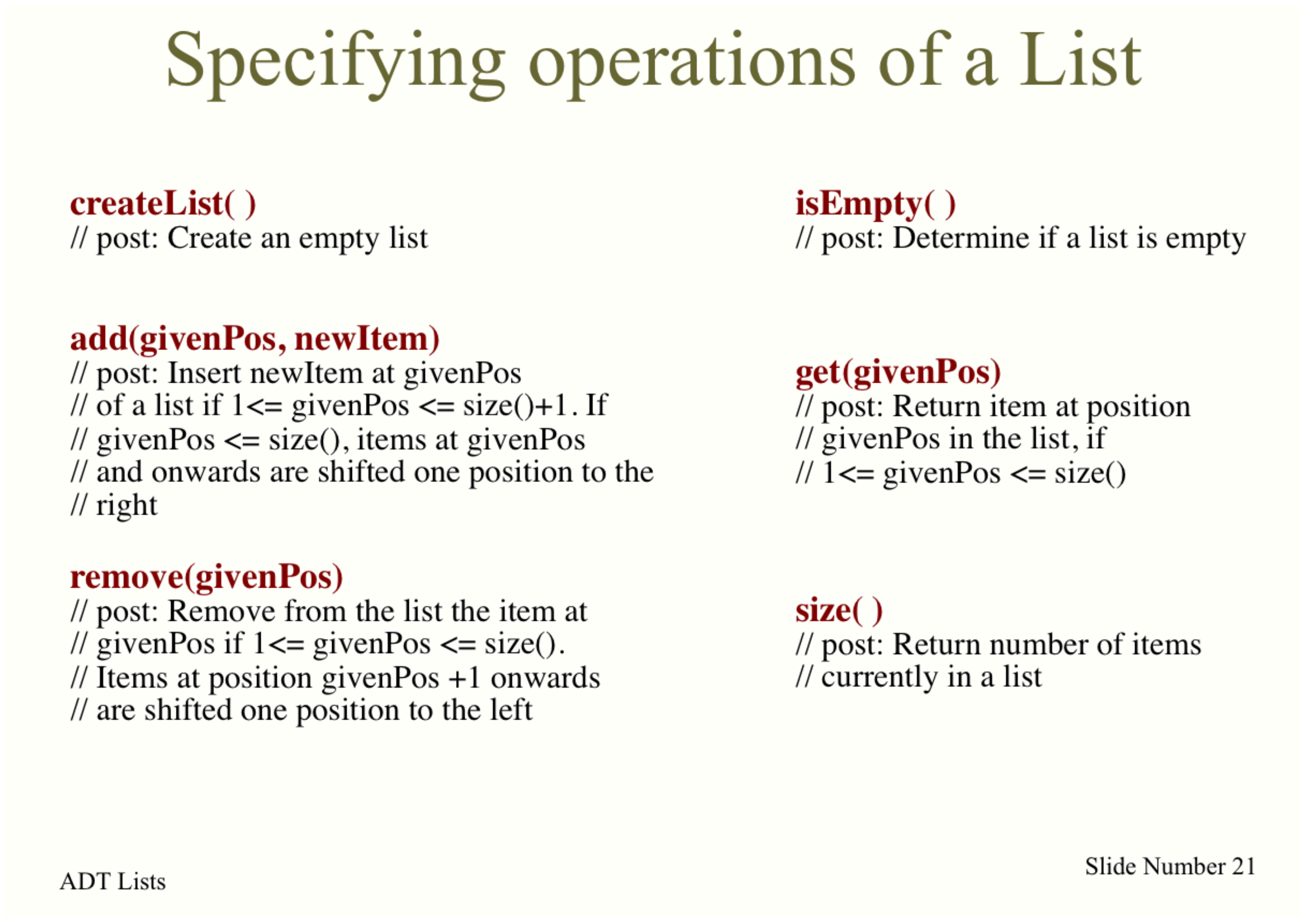
milk, eggs, butter, apples, bread, chicken

where *milk*is the first item on the list and so on. To begin, we consider how we would construct such a list, using the access procedures given in the previous slide. One way is first to create an empty list ***myList*** and then use a series of **add** operations to append successively the elements to the list, as described in the top right hand side of this slide. Note that **myList.O** indicates that the operation **O** is applied to the list object **myList**.

The operation **add** can insert a new element at any given position of the list, not just at the front or at the end. Therefore, if a new element is inserted at a position **i**, we would expect as observable behavior that all the elements starting from the one that is currently at position **i** and onwards are shifted one position to the right. So, for example, if we start with the list given at the top of this slide, and we perform the operation **myList.add(4, nuts)**, then the list **myList** would become *milk, eggs, butter, nuts, apples, bread, chicken.* All elementsthat had position greater than or equal to 4 before the addition, have now their position increased by 1 after the addition. This is shown in the right hand side of the slide.

Similarly, the deletion operation (or **remove**) should behave in a way that not empty spaces are left inside a List. After the deletion of an element from a given position **I**, we would expect as observable behavior that each element that was at a position greater than **i**is shifted one position to the left. For example, if **myList** is currently given by *milk, eggs, butter, nuts, apples, bread, chicken* , and we perform the operation **myList.remove(5)**, then the list **myList** would become *milk, eggs, butter, nuts, bread, chicken.* All elements that had position greater than 5 before the deletion, have now their position decreased by 1 after the deletion.

These descriptions specify only the effects of the operations addition and deletion of an element at a given position, not what underlying data structure for a List is used and how the operations are implemented.



Specifying operations of a List

createList( )
// post: Create an empty list

add(givenPos, newItem)
// post: Insert newItem at givenPos
// of a list if 1<= givenPos <= size()+1. If
// givenPos <= size(), items at givenPos
// and onwards are shifted one position to the
// right

remove(givenPos)
// post: Remove from the list the item at
// givenPos if 1<= givenPos <= size().
// Items at position givenPos +1 onwards
// are shifted one position to the left

isEmpty( )
// post: Determine if a list is empty

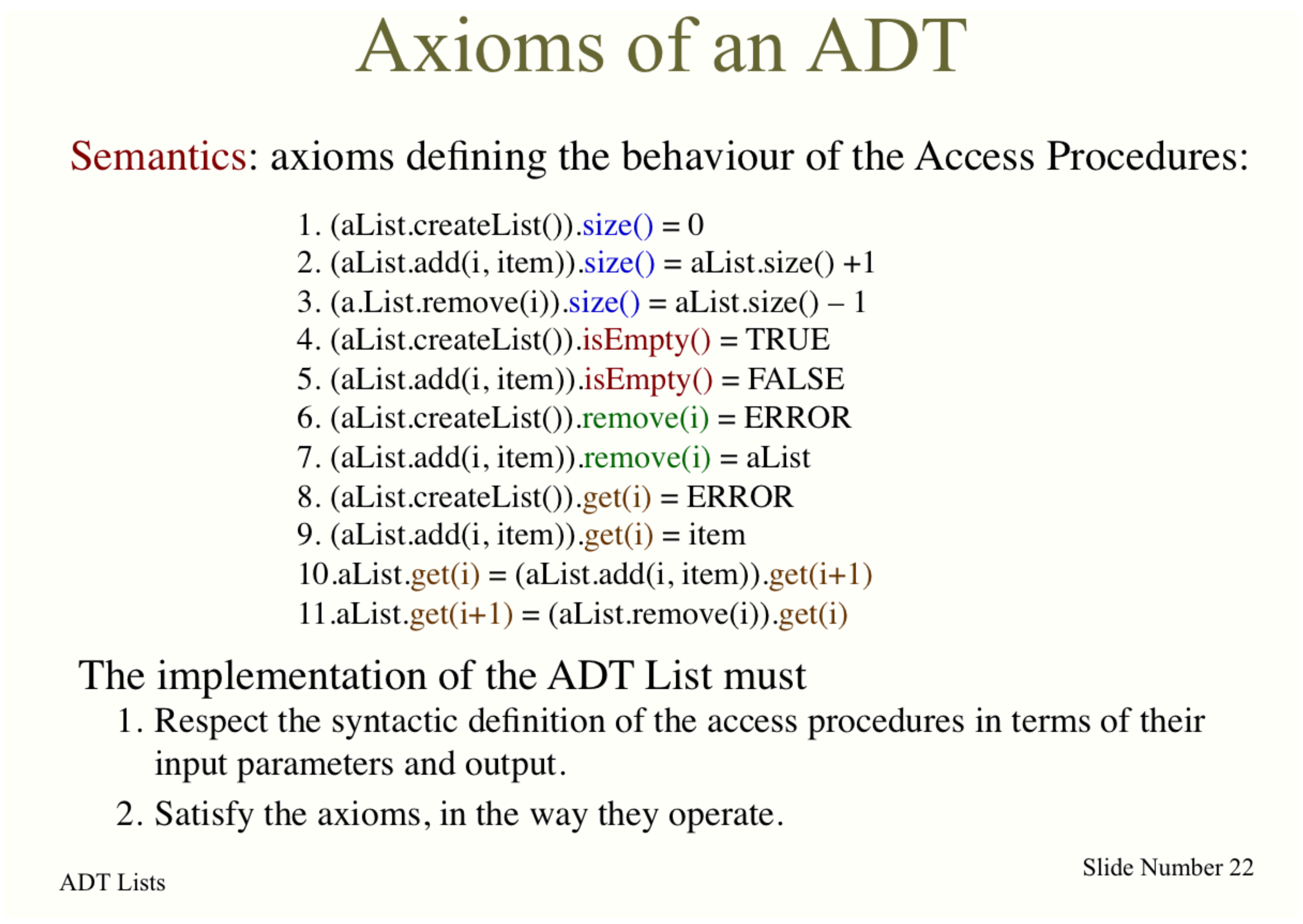
get(givenPos)
// post: Return item at position
// givenPos in the list, if
// 1<= givenPos <= size()

size( )
// post: Return number of items
// currently in a list

ADT Lists

Slide Number 21

This slide provides a nearly complete informal specification of the access procedures for List that we have introduced in the previous slides. In the next few slides I will show how access procedures are formally specified in Java.



Axioms of an ADT

Semantics: axioms defining the behaviour of the Access Procedures:

1. (aList.createList()).size() = 0
2. (aList.add(i, item)).size() = aList.size() +1
3. (a.List.remove(i)).size() = aList.size() – 1
4. (aList.createList()).isEmpty() = TRUE
5. (aList.add(i, item)).isEmpty() = FALSE
6. (aList.createList()).remove(i) = ERROR
7. (aList.add(i, item)).remove(i) = aList
8. (aList.createList()).get(i) = ERROR
9. (aList.add(i, item)).get(i) = item
10. aList.get(i) = (aList.add(i, item)).get(i+1)
11. aList.get(i+1) = (aList.remove(i)).get(i)

The implementation of the ADT List must

1. Respect the syntactic definition of the access procedures in terms of their input parameters and output.
2. Satisfy the axioms, in the way they operate.

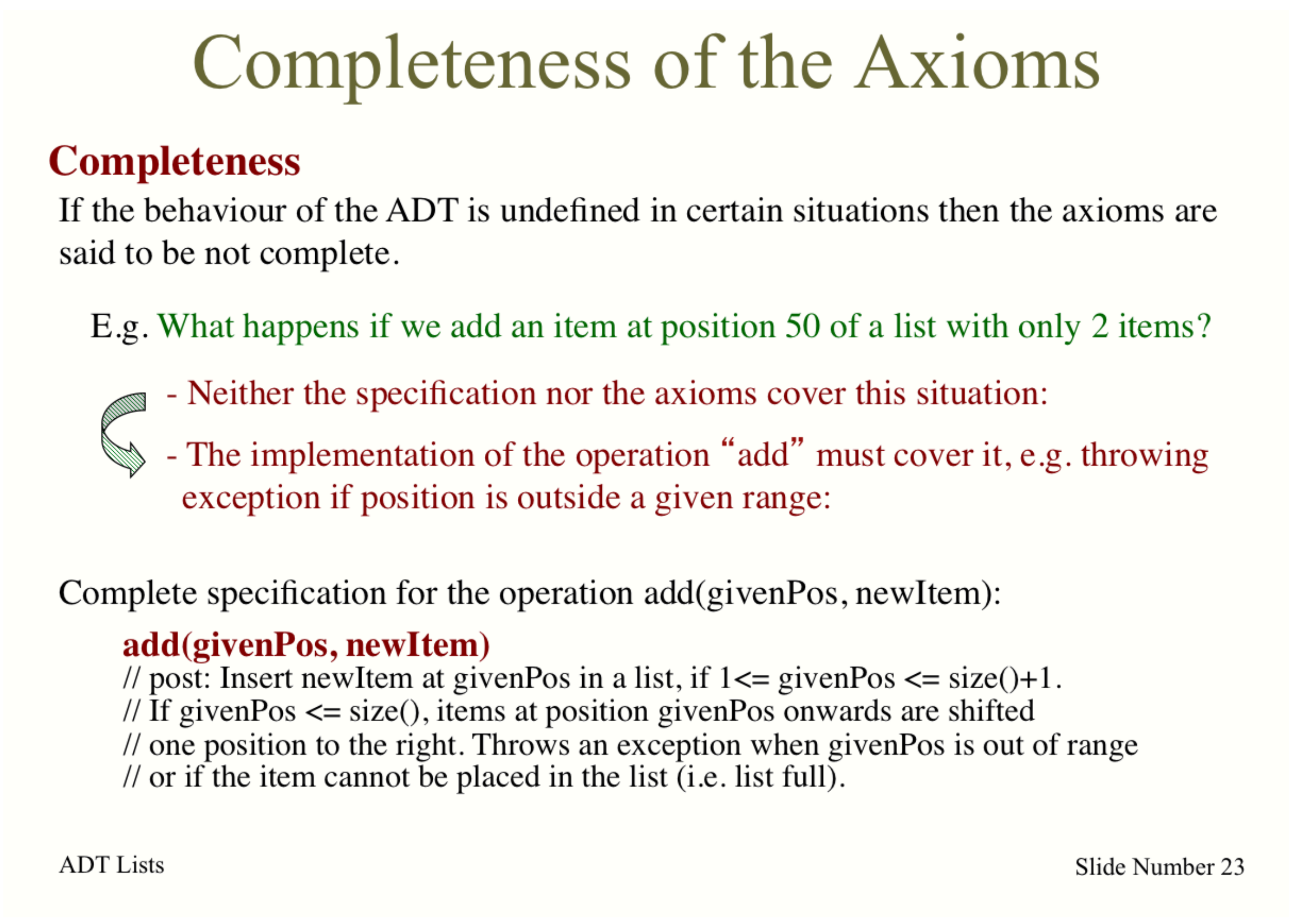
ADT Lists

Slide Number 22

The check the completeness of the specifications of an ADT it is often useful to define general properties that the access procedures will have to satisfy when applied to the ADT individually or in concatenation. This slide shows an example of axiom definitions of the properties that our List s access procedures have to satisfy when applied to any List.

For example, the statement *A newly created list is empty* is an axiom since it is true for all newly created lists. We can state this axiom succinctly in terms of the operations of an ADT List as follows: (aList.createList()).isEmpty() is true , which means that a list aList , which has just been created, is empty (refer to Slide 2 for an explanation of the notation used here). Note that the brackets denote the list obtained after the execution of the access procedure specified within the brackets.

An implementation of the ADT List must then take into account both the syntactic and semantic specification of the ADT. It will include the operations as they are syntactically defined, i.e. with their respective input parameters and result types. It will also implement the operations so that their behaviours satisfy the axioms for any possible list object they are applied to.



Completeness of the Axioms

Completeness

If the behaviour of the ADT is undefined in certain situations then the axioms are said to be not complete.

E.g. What happens if we add an item at position 50 of a list with only 2 items?

- Neither the specification nor the axioms cover this situation:
- The implementation of the operation "add" must cover it, e.g. throwing exception if position is outside a given range:

Complete specification for the operation add(givenPos, newItem):

add(givenPos, newItem)
// post: Insert newItem at givenPos in a list, if 1<= givenPos <= size()+1.
// If givenPos <= size(), items at position givenPos onwards are shifted
// one position to the right. Throws an exception when givenPos is out of range
// or if the item cannot be placed in the list (i.e. list full).

ADT Lists Slide Number 23

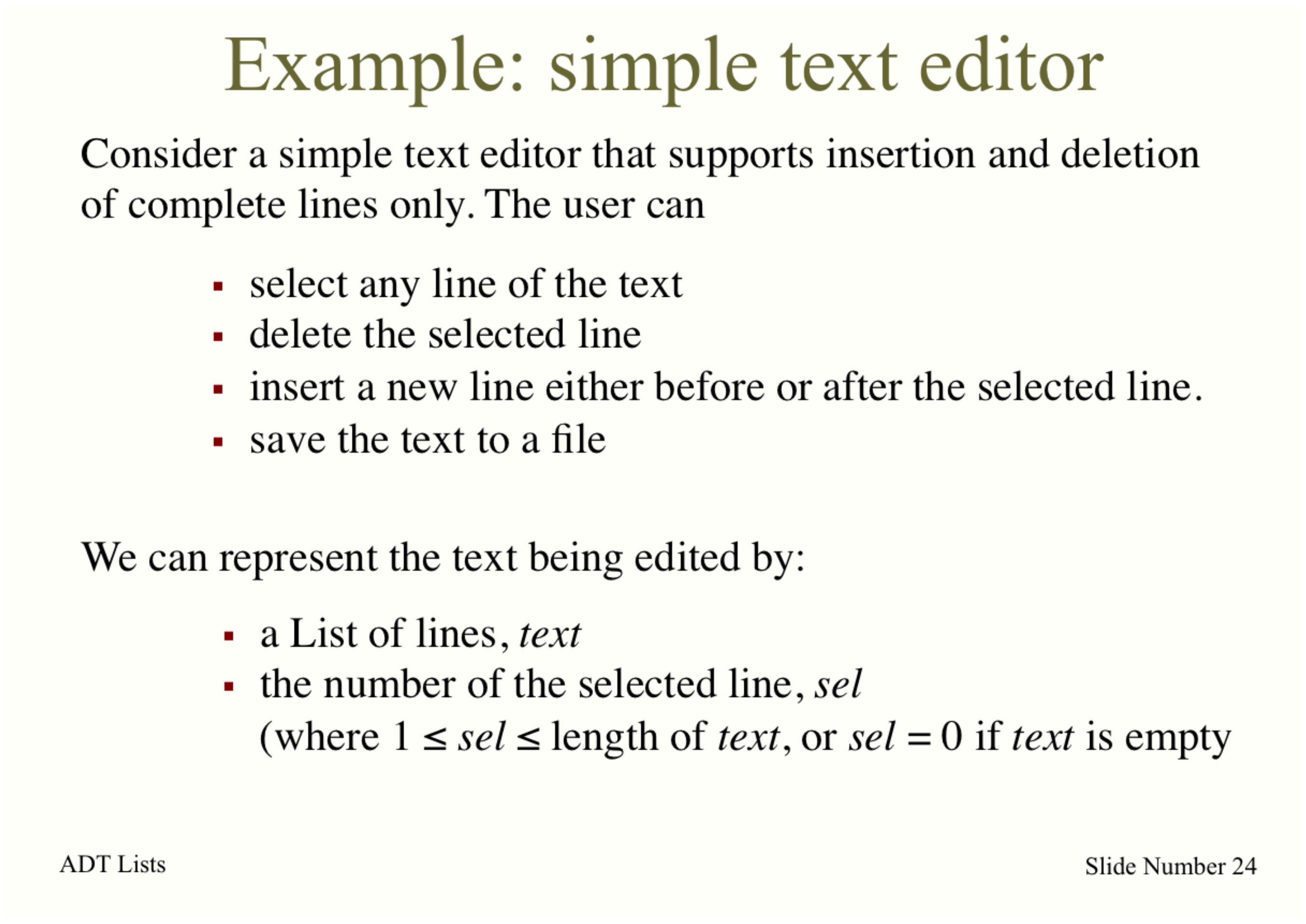
A set of axioms is not complete if the behaviour of the ADT is not defined in all possible situations in which ADT operations can be used. We have given here an example. Possible solutions are (i) to define special pre-conditions for the access procedure, which exclude the cases not covered by the axioms; (ii) define the implementation of the ADT access procedure in such a way that these cases are covered and the behaviour of the ADT is defined. An example of this second approach is given in this slide.

Similar examples of incompleteness can be identified for the operations remove and get , according to the axioms and the informal specification given in the previous slides. Their complete specification is given below:

remove(givenPos)
// post: Removes the item at givenPos of a list, if 1 <= givenPos <= size().
// If givenPos < size(), items at position givenPos + 1 and onwards are shifted one position to
// the left. Throws an exception when givenPos is out of range, or if list is empty.

get(givenPos)
//post: Returns the item at givenPos of a list, if 1 <= givenPos <= size( ). The list is
// left unchanged by this operation. Throws an exception if givenPos is out of range,
// or if the list if empty.

The post-conditions given here for the access procedures add, remove and get provide the complete specifications of these operation including also exception cases.


Example: simple text editor

Consider a simple text editor that supports insertion and deletion of complete lines only. The user can

- select any line of the text
- delete the selected line
- insert a new line either before or after the selected line.
- save the text to a file

We can represent the text being edited by:

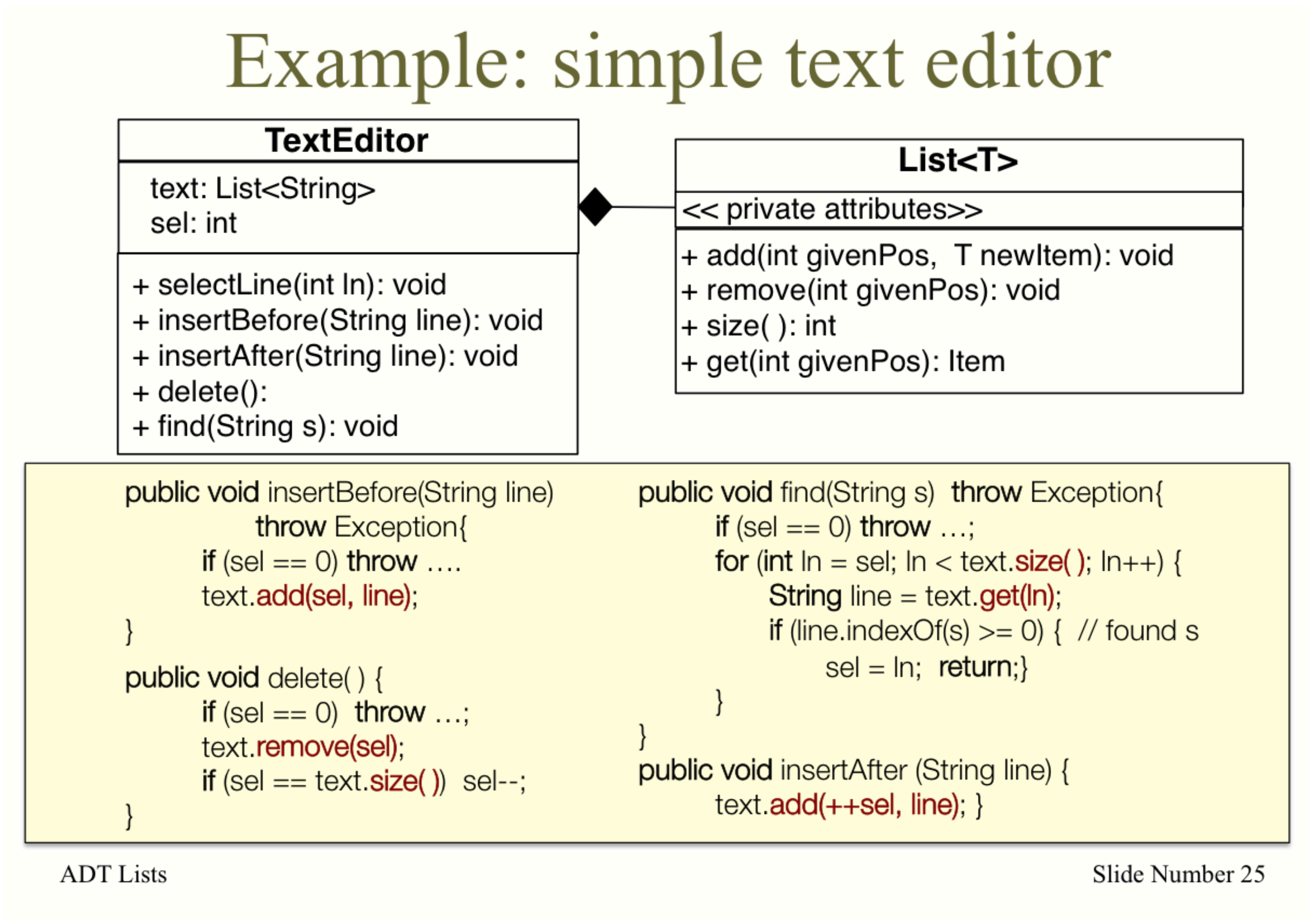
- a List of lines, *text*
- the number of the selected line, *sel* (where $1 \leq sel \leq$ length of *text*, or $sel = 0$ if *text* is empty

ADT Lists

Slide Number 24

Once the operations of an ADT have been specified, applications can be designed that access and manipulate the ADT's data, by using only the access procedures. A little example is considered here. This is the example of a simple text editor that can support insertion and deletion of complete lines only. It uses a List text where each element is a complete line. We can assume that the user can load text from a file or save text to a file (although these two operations are omitted here as not very relevant. The text editor has also an attribute sel because the user can select any line of the text by either directly using the mouse-click or by searching for the next line matching a given String. The selected line can be deleted and new lines can be inserted before or after the selected line. We show here how the methods of this program access the List of lines text and manipulate the text by using only the access procedures of the list.

Later we will show how to create an object List.



Example: simple text editor

TextEditor
text: List<String>
sel: int
+ selectLine(int ln): void
+ insertBefore(String line): void
+ insertAfter(String line): void
+ delete():
+ find(String s): void
List<T>
<< private attributes>>
+ add(int givenPos, T newItem): void
+ remove(int givenPos): void
+ size( ): int
+ get(int givenPos): Item

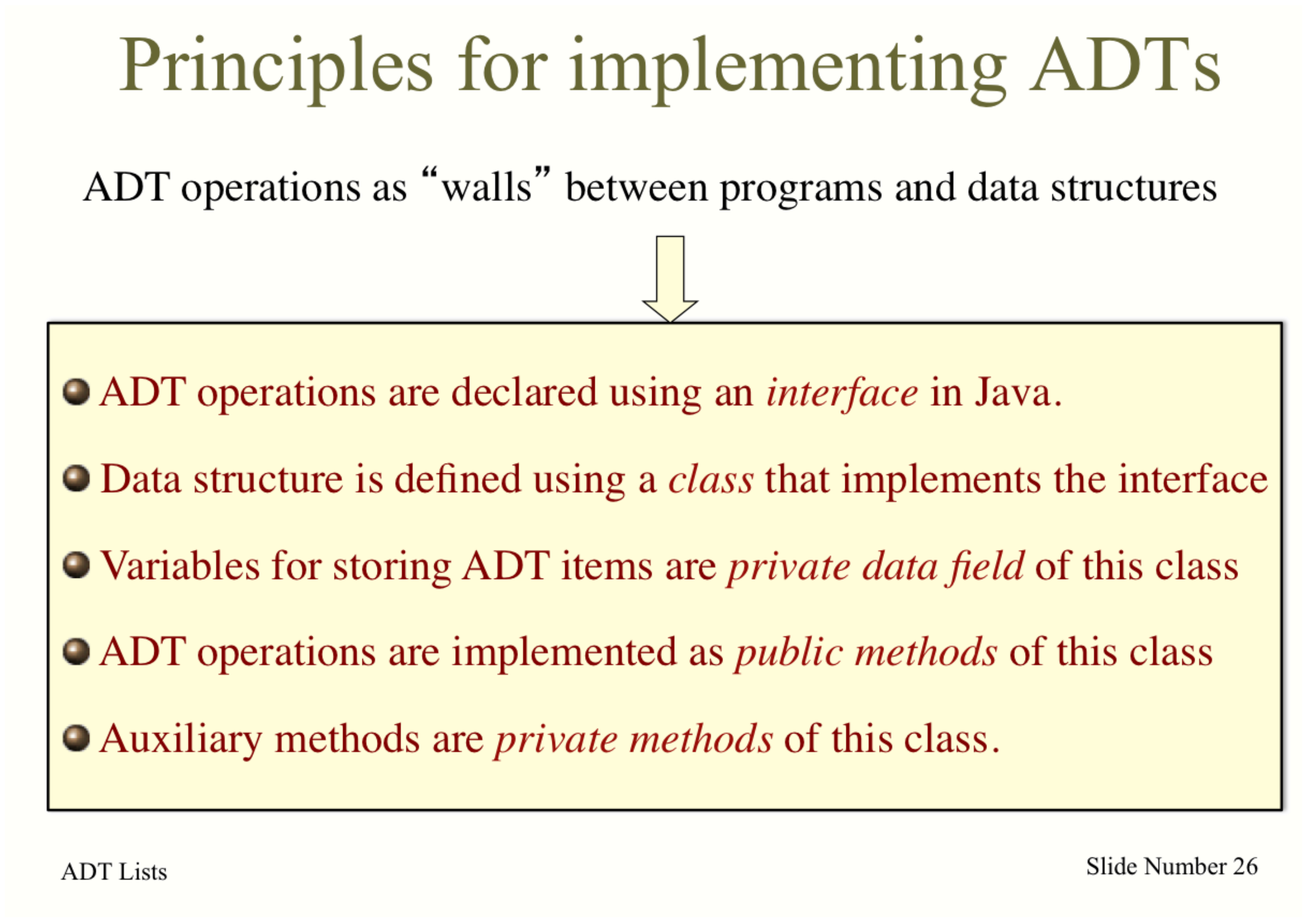
```
public void insertBefore(String line)
                throw Exception{
        if (sel == 0) throw ….
        text.add(sel, line);
}
public void delete( ) {
        if (sel == 0)  throw …;
        text.remove(sel);
        if (sel == text.size( ))  sel--;
}
```

```
public void find(String s)  throw Exception{
        if (sel == 0) throw …;
        for (int ln = sel; ln < text.size( ); ln++) {
              String line = text.get(ln);
              if (line.indexOf(s) >= 0) {  // found s
                    sel = ln;  return;}
        }
}
public void insertAfter (String line) {
        text.add(++sel, line); }
```

ADT Lists Slide Number 25

Once the operations of an ADT have been specified, applications can be designed that access and manipulate the ADT's data, by using only the access procedures. The above example shows the application program TextEditor using an ADT List. In the application program the user can select any line of the text by either directly (e.g., with a mouse-click), or by searching for the next line matching a given search string.

The four methods insetBefore, delete, find, insertAfter are example of methods of the application program that need to access data in the List ADT. As shown in the code above, they do that only through the access procedures of the List. So far we have in fact based the implementation of the application program only on the specification of the access procedures of the List without knowing anything about the particular implementation of the List.



Principles for implementing ADTs

ADT operations as "walls" between programs and data structures

- ADT operations are declared using an *interface* in Java.
- Data structure is defined using a *class* that implements the interface
- Variables for storing ADT items are *private data field* of this class
- ADT operations are implemented as *public methods* of this class
- Auxiliary methods are *private methods* of this class.

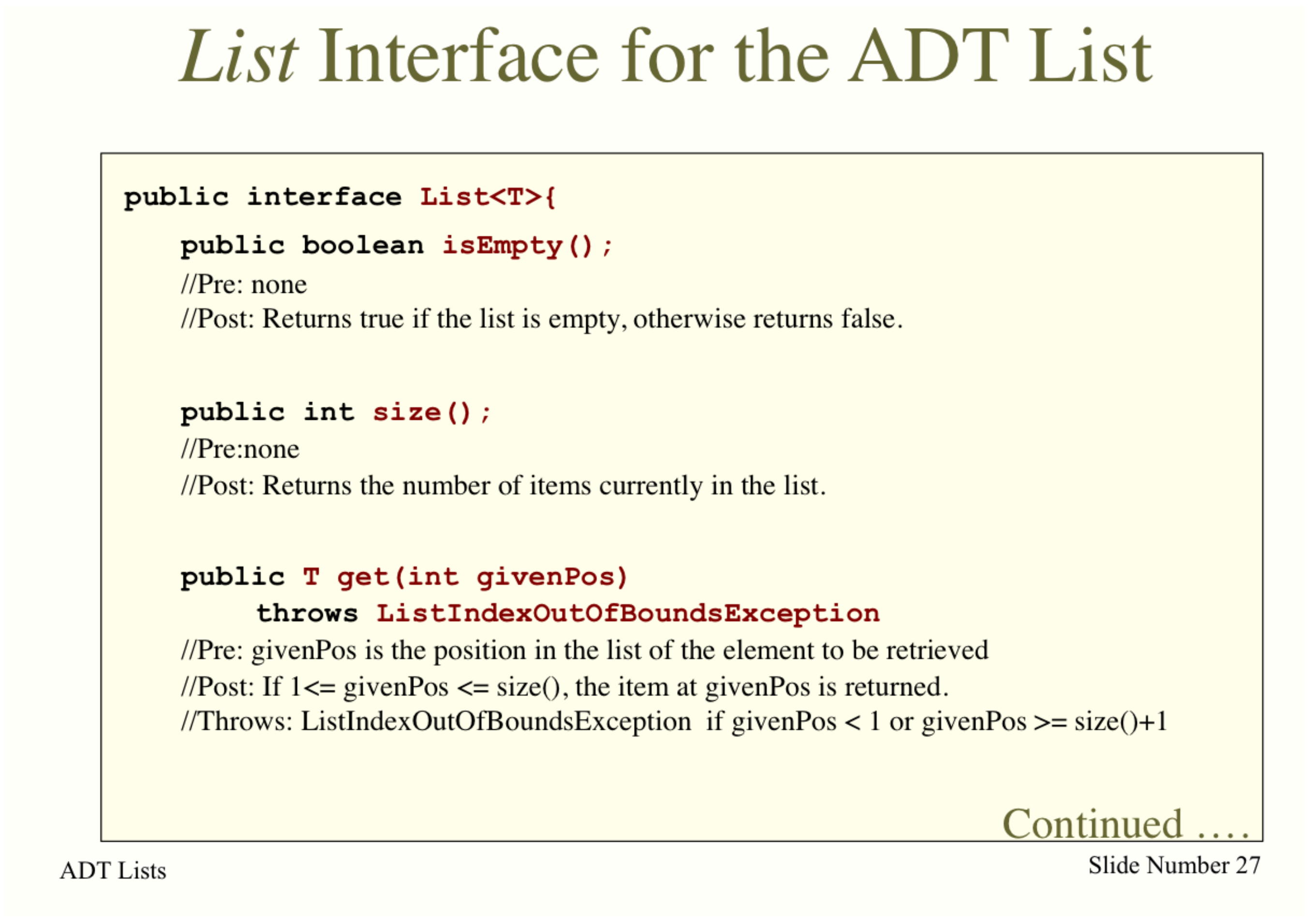
ADT Lists

Slide Number 26

In previous slides, we have said that ADT operations are like walls between any client program that uses an ADT and the actual data structure that implements the ADT. This general principle is reflected in the design of an ADT Java implementation. To implement an ADT, we define a Java interface, which provides just the declaration of the operations or access procedures of our ADT, and then device a class that implements this interface. This class **encapsulates** the **data structure** necessary to store the data of our ADT. Note that such class may use one or more other classes, as data fields according to how complex is the data structure chosen for representing the elements of the ADT.

Each ADT operation is implemented as a *public method* of the ADT class. For instance, let us assume for the sake of argument, that we are using an array as data structure of our ADT list. So the array can be declared as data field of the ADT class List. To support the principle of encapsulation of the data structure, we need to keep the array data structure hidden from the client program. This is done by declaring the array to be a *private* field in the class List, and the operations that access the elements in the array as *public methods.*

The implementation of the access procedures may require *auxiliary* methods. These are normally declared as *private methods* of the ADT class. An example is the private method is given when we see the implementation of a list.


List Interface for the ADT List

```
public interface List<T>{
    public boolean isEmpty();
    //Pre: none
    //Post: Returns true if the list is empty, otherwise returns false.

    public int size();
    //Pre:none
    //Post: Returns the number of items currently in the list.

    public T get(int givenPos)
         throws ListIndexOutOfBoundsException
    //Pre: givenPos is the position in the list of the element to be retrieved
    //Post: If 1<= givenPos <= size(), the item at givenPos is returned.
    //Throws: ListIndexOutOfBoundsException  if givenPos < 1 or givenPos >= size()+1
```

Continued ….

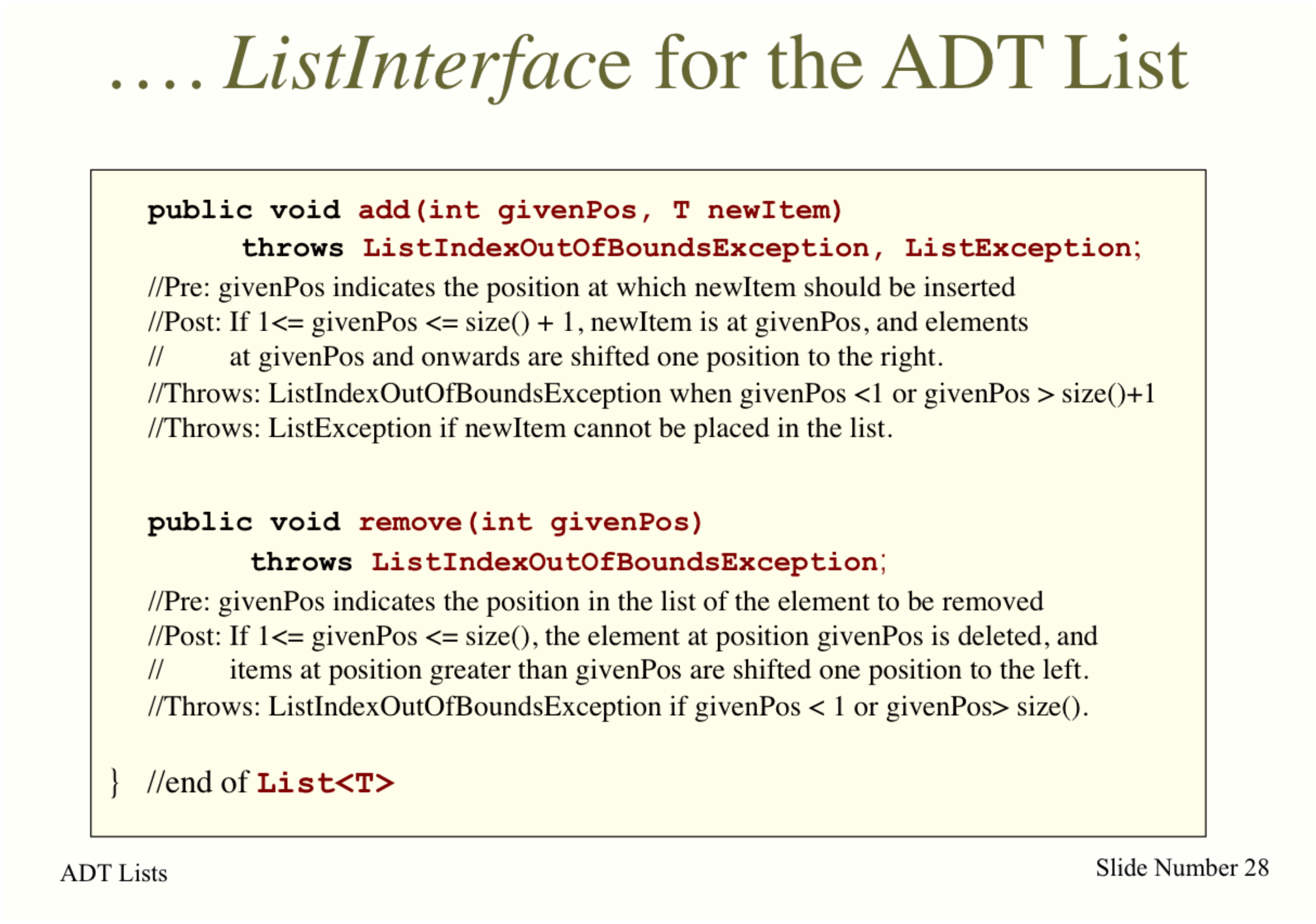
ADT Lists Slide Number 27

This is a possible specification of the ADT list (i.e. as we have mentioned in the previous unit a possible contract between the client/application program and the ADT implementation. The ADT operations can be declared using a Java interface.

In this slide we have defined a generic interface for the ADT list, called List<T>. The notion of a list and its operations have to be independent from the type of elements that are stored in the list. The use of a generic interface facilitates this independence, as the type of the elements included in the list will be defined dynamically at run-time. An alternative definition of this interface could be given, where the type of the items in the list is Object. In this second case the use of access procedures would require castings to the specific types of items that are stored in the list. The use of generic interface helps avoid casting. Any class created in Java can be used as a type of the elements in the list. The type of the elements in a list has to be declared at the moment of creation of a list. This will appear clear when we see the class that implements the array-based data structure for our ADT list.

Note: In this interface we haven't included the ADT access procedure createList(). This is because this access procedure is normally implemented by the constructor of the class that implements the ADT list.

Note: The interface List<T> includes also the two methods given in the next slide, which define the list operations "add" and "remove" respectively .

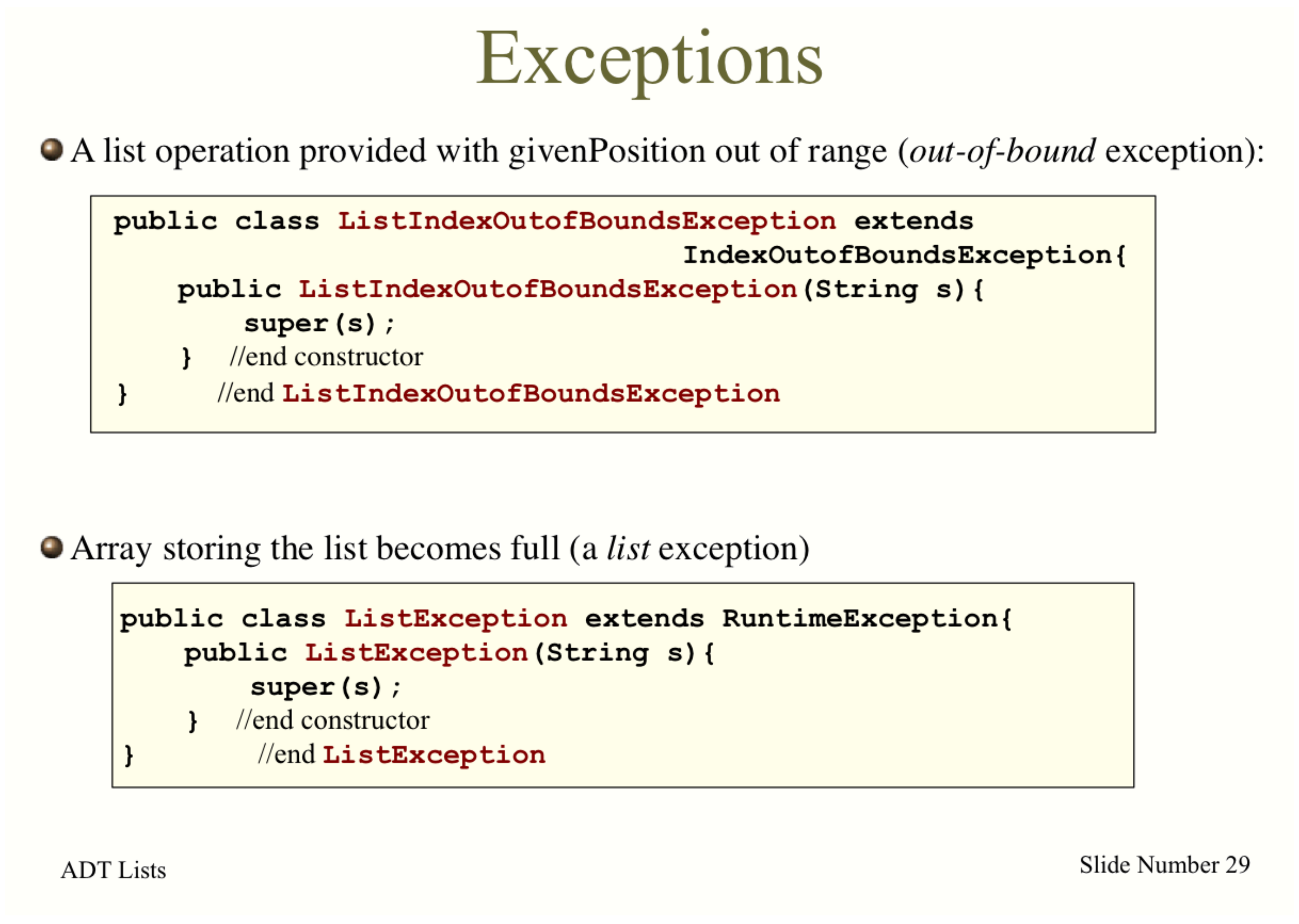


…. *ListInterface* for the ADT List

```
public void add(int givenPos, T newItem)
      throws ListIndexOutOfBoundsException, ListException;
//Pre: givenPos indicates the position at which newItem should be inserted
//Post: If 1<= givenPos <= size() + 1, newItem is at givenPos, and elements
//        at givenPos and onwards are shifted one position to the right.
//Throws: ListIndexOutOfBoundsException when givenPos <1 or givenPos > size()+1
//Throws: ListException if newItem cannot be placed in the list.


public void remove(int givenPos)
       throws ListIndexOutOfBoundsException;
//Pre: givenPos indicates the position in the list of the element to be removed
//Post: If 1<= givenPos <= size(), the element at position givenPos is deleted, and
//        items at position greater than givenPos are shifted one position to the left.
//Throws: ListIndexOutOfBoundsException if givenPos < 1 or givenPos> size().

}  //end of List<T>
```

ADT Lists

Slide Number 28



Exceptions

- A list operation provided with givenPosition out of range (*out-of-bound* exception):

```
public class ListIndexOutofBoundsException extends
                                             IndexOutofBoundsException{
     public ListIndexOutofBoundsException(String s){
          super(s);
     }    //end constructor
}        //end ListIndexOutofBoundsException
```

- Array storing the list becomes full (a *list* exception)

```
public class ListException extends RuntimeException{
     public ListException(String s){
          super(s);
     }    //end constructor
}          //end ListException
```

ADT Lists   Slide Number 29

Some of the list operations have as parameter a position value that refers to a particular data item in the list. The givenPos parameter can, however, be out of "range". Each of the three operations add, remove and get might be provided a position value that is out of range. In this case the operation has to throw an exception. You have seen in the first part of the course that in case of exceptions you have a choice from the point of view of the client program: either deal with the possible raise of an exception, or propagate the exception upwards. In the case of ADTs the latter is a better choice.

We have defined in this slide a class that implements the out-of-range index exception for lists extending the more general IndexOutOfBoundException class given by the Java API.

The second exception here, is useful mainly in the case of static implementation of a list. As we will see in the following slides, a static implementation of a list uses an array as underlying data structure. Therefore, in this case, we would need to guess a priori a maximum size of the list for the Java runtime environment to reserve a specific number of memory references for the elements of the array. So the exception ListException would occur whenever the array storing the list becomes full and we are trying to add new items to it. We have defined this exception with the class ListException, which extends the general RuntimeException of Java.

Note: Remember that RuntimeException is the Java superclass of all those exceptions that can be thrown and left unchecked during the normal operation of the Java Virtual machine. IndexOutOfBoundsException is instead an exception thrown to indicate that an index of some sort is out of range. It extends the RuntimeException, but applications normally use the subclass IndexOutOfBoundsException to indicate exceptions similar to an index out of range. This is why ListIndexOutOfBoundsException has been here defined as a subclass of IndexOutBoundsException instead of RuntimeException.