



Advanced Object-Oriented Programming

CPT204 – Lecture 10
Erick Purwanto



Xi'an Jiaotong-Liverpool University

西交利物浦大學

CPT204 Advanced Object-Oriented Programming

Lecture 10

**ADT, Rep Independence,
Interface, Inheritance, Iterable**

Welcome !

- Welcome to Lecture 10 !
- In this lecture we are going to
 - learn about Abstract Data Types (ADT) and Representation Independence
 - learn about Interface, Inheritance, and Iterability

Part 1 : ADT and Rep Independence

- The first part of the lecture introduces two ideas:
 - Abstract data types
 - Representation independence
- We look at a powerful idea called **abstract data types**, which enable us to *separate* how we use a data structure in a program from the particular form of the data structure itself
- Abstract data types address a particularly dangerous problem: clients making assumptions about the type's *internal representation*
 - we'll see why this is dangerous and how it can be avoided
 - we'll also discuss the classification of operations and some principles of good design for abstract data types

What Abstraction Means

Abstract data types are an instance of a general principle in software engineering, which goes by many names:

- **Abstraction.** Omitting or hiding low-level details with a simpler, higher-level idea
- **Modularity.** Dividing a system into components or modules, each of which can be designed, implemented, tested, reasoned about, and reused separately from the rest of the system
- **Encapsulation.** Building walls around a module (a hard shell or capsule) so that the module is responsible for its own internal behavior, and bugs in other parts of the system can't damage its integrity
- **Information hiding.** Hiding details of a module's implementation from the rest of the system, so that those details can be changed later without changing the rest of the system
- **Separation of concerns.** Making a feature (or "concern") the responsibility of a single module, rather than spreading it across multiple modules

User-Defined Types (1)

- In the early days of computing, a programming language came with built-in types (such as integers, booleans, strings, etc.) and built-in procedures, e.g., for input and output
 - users could define their own procedures: that's how large programs were built
- A major advance in software development was the idea of abstract types: *that one could design a programming language to allow user-defined types, too*
- This idea came out of the work of many researchers, notably **Dahl** (the inventor of the Simula language), **Hoare** (who developed many of the techniques we now use to reason about abstract types), **Parnas** (who coined the term information hiding and first articulated the idea of organizing program modules around the secrets they encapsulated), and **Barbara Liskov** and **John Guttag**, who did seminal work in the specification of abstract types, and in programming language support for them
 - **Barbara Liskov** earned **the Turing Award**, computer science's equivalent of the Nobel Prize, for her work on abstract types !!

User-Defined Types (2)

- The key idea of data abstraction is that a type is *characterized by the operations you can perform on it*
 - a number is something you can add and multiply; a string is something you can concatenate and take substrings of; a boolean is something you can negate, and so on
- In a sense, users could already define their own types in early programming languages: you could create a record type date, for example, with integer fields for day, month, and year
 - but what made abstract types new and different was the focus on operations: the user of the type would not need to worry about how its values were actually stored, in the same way that a programmer can ignore how the compiler actually stores integers

User-Defined Types (3)

- In Java, as in many modern programming languages, the separation between built-in types and user-defined types is a bit blurry
 - the classes in `java.lang`, such as `Integer` and `Boolean` are built-in; whether you regard all the collections of `java.util` as built-in is less clear (and not very important anyway)
 - Java complicates the issue by having primitive types that are *not* objects
 - the set of these types, such as `int` and `boolean`, cannot be extended by the user

Classifying Types and Operations

Review

- Types, whether built-in or user-defined, can be classified as mutable or immutable
- The objects of a mutable type can be changed: that is, they provide operations which when executed cause the results of other operations on the same object to give different results
 - so `Date` is mutable, because you can call `setMonth` and observe the change with the `getMonth` operation
 - but `String` is immutable, because its operations create new `String` objects rather than changing existing ones
 - sometimes a type will be provided in two forms, a mutable and an immutable form: `StringBuilder`, for example, is a mutable version of `String` (although the two are certainly not the same Java type, and are not interchangeable)

Operations of Abstract Type (1)

The operations of an abstract type are classified as follows:

- **Creators** create new objects of the type;
A creator may take an object as an argument, but *not* an object of the type being constructed
- **Producers** create new objects from old objects of the type;
The `concat` method of `String`, for example, is a producer: it takes two strings and produces a new one representing their concatenation
- **Observers** take objects of the abstract type and return objects of a different type;
The `size` method of `List`, for example, returns an `int`
- **Mutators** change objects;
The `add` method of `List`, for example, mutates a list by adding an element to the end

Operations of Abstract Type (2)

- We can summarize these distinctions schematically like this:
 - creator : $t^* \rightarrow T$
 - producer : $T^+, t^* \rightarrow T$
 - observer : $T^+, t^* \rightarrow t$
 - mutator : $T^+, t^* \rightarrow \text{void} \mid t \mid T$
- These show informally the shape of the signatures of operations in the various classes
 - each T is the abstract type itself; each t is some other type
 - the $+$ marker indicates that the type may occur *one or more* times in that part of the signature
 - and the $*$ marker indicates that it occurs *zero or more* times
 - \mid indicates or

Operations of Abstract Type (3)

- For example, a producer may take two values of the abstract type T , like `String.concat()` does:
 - `concat : String × String → String`
- Some observers take zero arguments of other types t , such as:
 - `size : List → int`
- And other observers take several, such as:
 - `regionMatches : String × boolean × int × String × int × int → boolean`

Operations of Abstract Type (4)

- A creator operation is often implemented as a constructor, like `new ArrayList()`
 - But a creator can simply be a static method instead, like `Arrays.asList()`
- A creator *implemented as a static method* is often called a **factory method**
- The various `String.valueOf` methods in Java are other examples of creators implemented as factory methods

Operations of Abstract Type (5)

- Mutators are *often* signaled by a void return type
 - a method that returns void must be called for some kind of *side-effect*, since otherwise it doesn't return anything
- But **not all** mutators return void
 - for example, `Set.add()` returns a boolean that indicates whether the set was actually changed
 - also, in Java's graphical user interface toolkit, `Component.add()` returns the object itself, so that multiple `add()` calls can be chained together

Abstract Data Type Examples (1)

Here are some examples of abstract data types, along with some of their operations, grouped by kind

- **int** is Java's primitive integer type
 - int is immutable, so it has *no* mutators
 - creators: the numeric literals 0, 1, 2, ...
 - producers: arithmetic operators +, -, ×, ÷
 - observers: comparison operators ==, !=, <, >
 - mutators: none (it's immutable)

Abstract Data Type Examples (2)

- **List** is Java's list type

List is mutable

List is also an interface, which means that other classes provide the actual implementation of the data type

These classes include ArrayList and LinkedList

- creators: ArrayList and LinkedList constructors, Collections.singletonList
- producers: Collections.unmodifiableList
- observers: size, get
- mutators: add, remove, addAll, Collections.sort

Abstract Data Type Examples (3)

- **String** is Java's string type

String is immutable

- creators: String constructors
- producers: concat, substring, toUpperCase
- observers: length, charAt
- mutators: none (it's immutable)

In-Class Quiz 1

- `Map.keySet()` is a method from the Java library
 - below is the method's Javadoc documentation, look at its signature
- What kind of operation of an abstract data type is it?

keySet

`Set<K> keySet()`

Returns a `Set` view of the keys contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa. If the map is modified while an iteration over the set is in progress (except through the iterator's own `remove` operation), the results of the iteration are undefined. The set supports element removal, which removes the corresponding mapping from the map, via the `Iterator.remove`, `Set.remove`, `removeAll`, `retainAll`, and `clear` operations. It does not support the `add` or `addAll` operations.

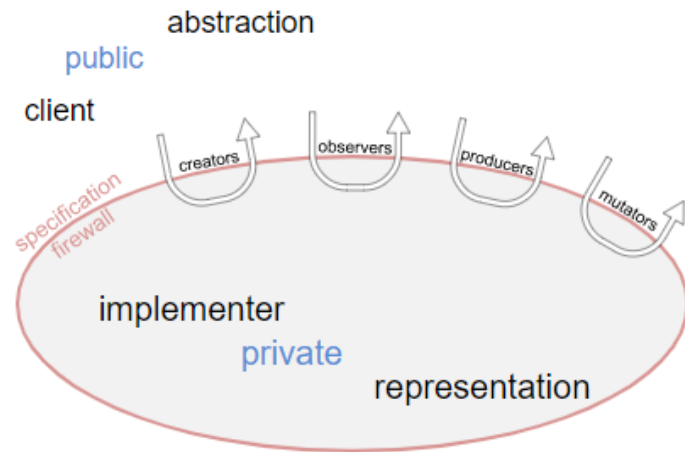
Returns:

a set view of the keys contained in this map

- creator
- producer
- observer
- mutator

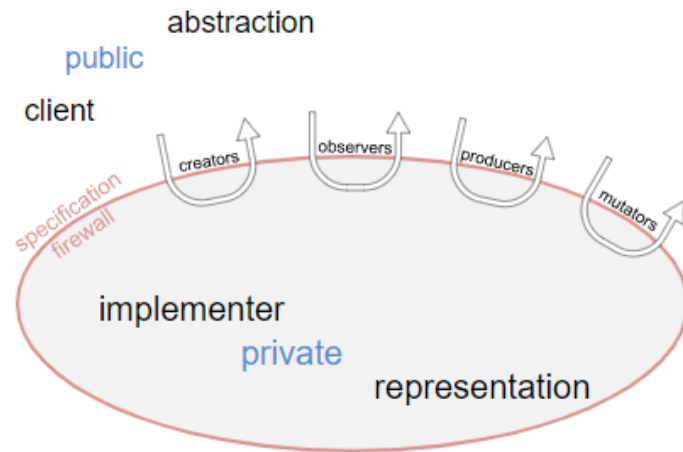
An Abstract Type is Defined by Its Operations

- The essential idea here is that **an abstract data type is defined by its operations**
 - the set of operations for a type T , along with their specifications, fully characterize what we mean by T
- So, for example, when we talk about the `List` type, what we mean is **not** a linked list or an array or any other specific data structure for representing a list
- Instead, the `List` type is a set of opaque values — the possible objects that can have `List` type — that satisfy the specifications of all the operations of `List`: `get()`, `size()`, etc
 - the values of an abstract type are opaque in the sense that a client can't examine the data stored inside them, except as permitted by operations



Abstraction vs Representation

- Expanding our metaphor of a specification firewall, you might picture values of an abstract type as *hard shells*, *hiding* not just the implementation of an individual function, but of a set of related functions (the operations of the type) and the data they share (the private fields stored inside values of the type)
- The operations of the type constitute its **abstraction**
 - this is the **public** part, visible to clients who use the type
- The fields of the class that implements the type, as well as related classes that help implement a complex data structure, constitute a particular **representation**
 - this part is **private**, visible only to the implementer of the type



Designing an Abstract Type (1)

Designing an abstract data type involves choosing good operations and determining how they should behave;

Here are a few rules of thumb:

- It's better to have **a few, simple operations** that can be combined in powerful ways, rather than lots of complex operations
- Each operation should have a well-defined purpose and should have a **coherent** behavior rather than a multitude of special cases
 - we probably *shouldn't* add a sum operation to List, for example
 - it might help clients who work with lists of integers, but what about lists of strings? or nested lists? all these special cases would make sum a hard operation to understand and use

Designing an Abstract Type (2)

- The set of operations should be **adequate** in the sense that there must be enough to do the kinds of computations clients are likely to want to do
 - a good test is to check that *every property* of an object of the type can be *extracted*
 - for example, if there were no get operation, we would not be able to find out what the elements of a list are
 - basic information should not be inordinately difficult to obtain
 - for example, the size method is not strictly necessary for List, because we could apply get on increasing indices until we get a failure, but this is inefficient and inconvenient

Designing an Abstract Type (3)

- The type may be *generic*: a list or a set, or a graph, for example
 - or it may be *domain-specific*: a street map, an employee database, a phone book, etc
- But it **should not mix generic and domain-specific features**
 - A Deck type intended to represent a sequence of playing cards shouldn't have a generic add method that accepts arbitrary objects like integers or strings
 - Conversely, it wouldn't make sense to put a domain-specific method like dealCards into the generic type List
- Next, it should be **representation independent**

Representation Independence

- Critically, a good abstract data type (ADT) should be **representation independent**
- This means that the use of an abstract type is independent of its ***representation*** : the ***actual data structure*** or ***data fields*** or ***instance variable*** used to implement it, so that changes in representation have no effect on code outside the abstract type itself
 - for example, the *operations offered by List* are *independent* of whether the list is represented as *a linked list* or as *an array*
- You won't be able to change the representation of an ADT at all unless its operations are fully specified with preconditions and postconditions, so that clients know what to depend on, and you know what you can safely change

Example: Different Representations for Strings (1)

- Let's look at a simple abstract data type to see what representation independence means and why it's useful
- The **MyString** type below has far fewer operations than the real Java String, and their specs are a little different, but it's still illustrative
- Here are the specs for the ADT:

```
/** MyString represents an immutable sequence of characters. */
public class MyString {

    //////////////// Example of a creator operation ////////////////
    /** @param b a boolean value
     *  @return string representation of b, either "true" or "false" */
    public static MyString valueOf(boolean b) { ... }
```

Example: Different Representations for Strings (2)

```
////////// Examples of observer operations //////////  
/** @return number of characters in this string */  
public int length() { ... }  
  
/** @param i character position (requires  $0 \leq i < \text{string length}$ )  
 * @return character at position i */  
public char charAt(int i) { ... }  
  
////////// Example of a producer operation //////////  
/** Get the substring between start (inclusive) and end (exclusive).  
 * @param start starting index  
 * @param end ending index. Requires  $0 \leq \text{start} \leq \text{end} \leq \text{string length}$ .  
 * @return string consisting of charAt(start)...charAt(end-1) */  
public MyString substring(int start, int end) { ... }  
}
```

no mutator operation. why not?

Testing MyString (1)

- These public operations and their specifications are the only information that a client of this data type is allowed to know
- Following the test-first programming paradigm, in fact, the first client we should create is a test suite that exercises these operations according to their specs
- At the moment, however, writing test cases that use `assertEquals` directly on `MyString` objects wouldn't work, because we don't have an equality operation defined on `MyString`
 - We'll talk about how to implement ***equality*** carefully in next lecture
 - For now, the only operations we can perform with `MyStrings` are the ones we've defined above: `valueOf`, `length`, `charAt`, and `substring`
 - Our tests have to limit themselves to those operations

Testing MyString (2)

- For example, here's one test for the valueOf operation:

```
MyString s = MyString.valueOf(true);  
assertEquals(4, s.length());  
assertEquals('t', s.charAt(0));  
assertEquals('r', s.charAt(1));  
assertEquals('u', s.charAt(2));  
assertEquals('e', s.charAt(3));
```

- Next, let's look at a simple representation for MyString:
just an array of characters, which is exactly the length of the string with no extra room at the end

Representation of MyString: Array of Characters (1)

- Here's how that internal representation would be declared, as an instance variable within the class:

```
private char[] a;
```

- With that choice of representation, the operations would be implemented in a straightforward

```
public static MyString valueOf(boolean b) {  
    MyString s = new MyString();  
    s.a = b ? new char[] { 't', 'r', 'u', 'e' }  
           : new char[] { 'f', 'a', 'l', 's', 'e' };  
    return s;  
}  
  
public int length() {  
    return a.length;  
}  
  
public char charAt(int i) {  
    return a[i];  
}  
  
public MyString substring(int start, int end) {  
    MyString that = new MyString();  
    that.a = new char[end - start];  
    System.arraycopy(this.a, start, that.a, 0, end - start);  
    return that;  
}
```

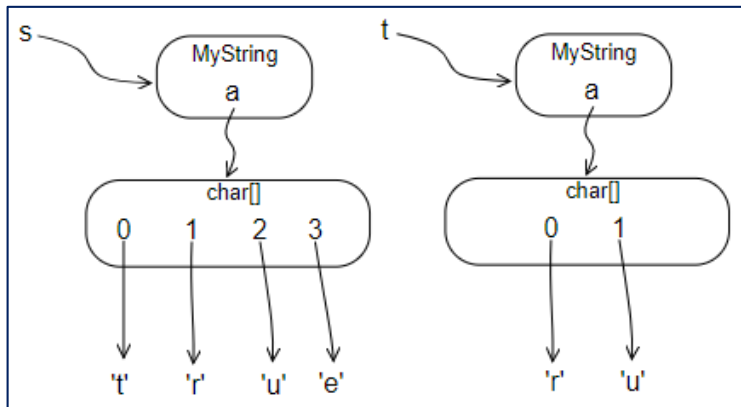
The ? ... : ... syntax is called the ternary conditional operator and it's a shorthand if-else statement

Question to ponder: Why don't charAt and substring have to check whether their parameters are within the valid range? What do you think will happen if the client calls these implementations with illegal inputs?

Representation of MyString: Array of Characters (2)

- Here's a snapshot diagram showing what this representation looks like for a couple of typical client operations:

```
MyString s = MyString.valueOf(true);  
MyString t = s.substring(1,3);
```



Representation of MyString: Array of Characters (3)

- One problem with this implementation is that it's passing up an opportunity for performance improvement
 - Because this data type is immutable, the substring operation *doesn't really have to copy characters out into a fresh array*
 - It could just point to the original MyString object's character array and keep track of the start and end that the new substring object represents
 - The String implementation in some versions of Java do this!

Representation of MyString: Array of Characters (4)

- To implement this optimization, we could change the *internal representation* and *operations* to:

```
private char[] a;
private int start;
private int end;

public static MyString valueOf(boolean b) {
    MyString s = new MyString();
    s.a = b ? new char[] { 't', 'r', 'u', 'e' }
           : new char[] { 'f', 'a', 'l', 's', 'e' };
    s.start = 0;
    s.end = s.a.length;
    return s;
}

public int length() {
    return end - start;
}

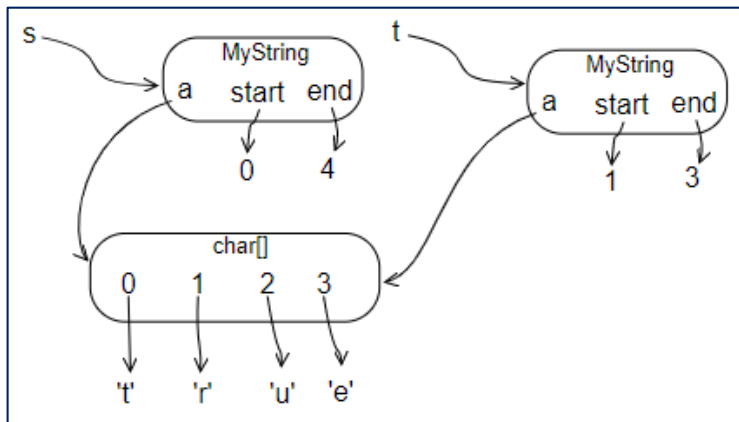
public char charAt(int i) {
    return a[start + i];
}

public MyString substring(int start, int end) {
    MyString that = new MyString();
    that.a = this.a;
    that.start = this.start + start;
    that.end = this.start + end;
    return that;
}
```


Representation of MyString: Array of Characters (5)

- Now the same client code produces a very different internal structure:

```
MyString s = MyString.valueOf(true);  
MyString t = s.substring(1,3);
```



Representation of MyString: Array of Characters (6)

- Because MyString's existing clients ***depend only*** on the specs of its public methods, not on its private fields
 - we can make this change without having to inspect and change all that client code
 - that's the power of *representation independence* !

Realizing ADT Concepts in Java (1)

- Let's summarize some of the general ideas we've discussed in this first part of the lecture, which are applicable in general to programming in any language, and their specific realization using Java language features
- The point is that there are several ways to do it, and it's important to both understand the big idea, like a creator operation, and different ways to achieve that idea in practice
- We'll also include three items that haven't yet been discussed so far, with notes about them in the next slide...

Realizing ADT Concepts in Java (2)

ADT concept	Ways to do it in Java	Examples
Abstract data type	Class	<code>String</code>
	Interface + class(es)	<code>List</code> and <code>ArrayList</code>
	Enum	<code>DayOfWeek</code>
Creator operation	Constructor	<code>ArrayList()</code>
	Static (factory) method	<code>List.of()</code>
	Constant	<code>BigInteger.ZERO</code>
Observer operation	Instance method	<code>List.get()</code>
	Static method	<code>Collections.max()</code>
Producer operation	Instance method	<code>String.trim()</code>
	Static method	<code>Collections.unmodifiableList()</code>
Mutator operation	Instance method	<code>List.add()</code>
	Static method	<code>Collections.copy()</code>
Representation	<code>private</code> fields	

Realizing ADT Concepts in Java (3)

ADT concept	Ways to do it in Java	Examples
Abstract data type	Class	<code>String</code>
	Interface + class(es)	<code>List</code> and <code>ArrayList</code>
	Enum	<code>DayOfWeek</code>
Creator operation	Constructor	<code>ArrayList()</code>
	Static (factory) method	<code>List.of()</code>
	Constant	<code>BigInteger.ZERO</code>
Observer operation	Instance method	<code>List.get()</code>
	Static method	<code>Collections.max()</code>
Producer operation	Instance method	<code>String.trim()</code>
	Static method	<code>Collections.unmodifiableList()</code>
Mutator operation	Instance method	<code>List.add()</code>
	Static method	<code>Collections.copy()</code>
Representation	<code>private</code> fields	

Defining an abstract data type using an interface + class(es). We've seen `List` and `ArrayList` as an example, and we'll discuss **interface** in **Lecture Part 2** next

Defining an abstract data type using an enumeration (enum). Enums are ideal for ADTs that have a small fixed set of values, like the days of the week Monday, Tuesday, etc

Using a constant object as a creator operation. This pattern is commonly seen in immutable types, where the *simplest* or *emptiest* value of the type is simply a public constant, and producers are used to build up more complex values from it

Testing an Abstract Data Type (1)

- We build a test suite for an abstract data type by creating tests for each of its operations
- These tests inevitably interact with each other
- The only way to test creators, producers, and mutators is by calling observers on the objects that result, and likewise, the only way to test observers is by creating objects for them to observe

Testing an Abstract Data Type (2)

- Here's how we might partition the input spaces of the four operations in our `MyString` type:

```
// testing strategy for each operation of MyString:
//
// valueOf():
//   true, false
// length():
//   string len = 0, 1, n
//   string = produced by valueOf(), produced by substring()
// charAt():
//   string len = 1, n
//   i = 0, middle, len-1
//   string = produced by valueOf(), produced by substring()
// substring():
//   string len = 0, 1, n
//   start = 0, middle, len
//   end = 0, middle, len
//   end-start = 0, n
//   string = produced by valueOf(), produced by substring()
```

Testing an Abstract Data Type (3)

- Then a compact test suite that covers all these partitions might look like:

```
@Test public void testValueOfTrue() {
    MyString s = MyString.valueOf(true);
    assertEquals(4, s.length());
    assertEquals('t', s.charAt(0));
    assertEquals('r', s.charAt(1));
    assertEquals('u', s.charAt(2));
    assertEquals('e', s.charAt(3));
}

@Test public void testValueOfFalse() {
    MyString s = MyString.valueOf(false);
    assertEquals(5, s.length());
    assertEquals('f', s.charAt(0));
    assertEquals('a', s.charAt(1));
    assertEquals('l', s.charAt(2));
    assertEquals('s', s.charAt(3));
    assertEquals('e', s.charAt(4));
}
```


Testing an Abstract Data Type (4)

```
@Test public void testEndSubstring() {
    MyString s = MyString.valueOf(true).substring(2, 4);
    assertEquals(2, s.length());
    assertEquals('u', s.charAt(0));
    assertEquals('e', s.charAt(1));
}

@Test public void testMiddleSubstring() {
    MyString s = MyString.valueOf(false).substring(1, 2);
    assertEquals(1, s.length());
    assertEquals('a', s.charAt(0));
}

@Test public void testSubstringIsWholeString() {
    MyString s = MyString.valueOf(false).substring(0, 5);
    assertEquals(5, s.length());
    assertEquals('f', s.charAt(0));
    assertEquals('a', s.charAt(1));
    assertEquals('l', s.charAt(2));
    assertEquals('s', s.charAt(3));
    assertEquals('e', s.charAt(4));
}

@Test public void testSubstringOfEmptySubstring() {
    MyString s = MyString.valueOf(false).substring(1, 1).substring(0, 0);
    assertEquals(0, s.length());
}
```

Testing an Abstract Data Type (5)

- Notice that each test case typically calls a few operations that make or modify objects of the type (creators, producers, mutators) and some operations that inspect objects of the type (observers)
- As a result, each test case covers parts of several operations

Part 2 : Inheritance

- In the second part of the lecture, we discuss Inheritance, Interface, and Iterable
- You have previously learned about Inheritance in CSE105
 - Please review those CSE105 materials and labs
 - We are also going to very quickly review it ...

SLList and ARList

- Recall our two list implementations

```
public class ARList<T>{  
    public ARList()  
    public void addLast(T item)  
    public T getLast()  
    public T get(int i)  
    public int size()  
    public T delLast()  
}
```

```
public class SLList<T>{  
    public SLList()  
    public SLList(T item)  
    public void addFirst(T item)  
    public void addLast(T item)  
    public T getFirst()  
    public T getLast()  
    public T get(int i)  
    public int size()  
    public T delLast()  
}
```

SLList and ARList

- Recall our two list implementations
 - notice there are same method signatures that appear on both classes

```
public class ARList<T>{  
    public ARList()  
    public void addLast(T item)  
    public T getLast()  
    public T get(int i)  
    public int size()  
    public T delLast()  
}
```

```
public class SLList<T>{  
    public SLList()  
    public SLList(T item)  
    public void addFirst(T item)  
    public void addLast(T item)  
    public T getFirst()  
    public T getLast()  
    public T get(int i)  
    public int size()  
    public T delLast()  
}
```

A Method Taking SLList

- Suppose you wrote a method taking SLList of String in a class StrLib as follows
 - Now, you also want a similar method that takes ARList of String as well...

```
public static String longestStr(SLList<String> list) {  
    int max = 0;  
    for (int i = 0; i < list.size(); i++) {  
        String longestString = list.get(max);  
        String thisString = list.get(i);  
        if (thisString.length() > longestString.length()) {  
            max = i;  
        }  
    }  
    return list.get(max);  
}
```

A Method Taking ARList

- You could create another copy of it, and make a little modification
 - You just did **method overloading** : same name, different parameter/signature!

```
public static String longestStr(ARList<String> list) {  
    int max = 0;  
    for (int i = 0; i < list.size(); i++) {  
        String longestString = list.get(max);  
        String thisString = list.get(i);  
        if (thisString.length() > longestString.length()) {  
            max = i;  
        }  
    }  
    return list.get(max);  
}
```

Method Overloading

- **Method overloading** : multiple methods with same name, but different parameters
 - Java will figure out which method is going to get called

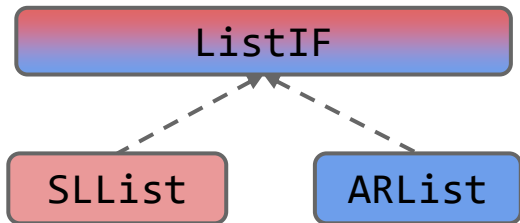
```
public static String longestStr(SLList<String> list) {  
    ...  
}  
  
public static String longestStr(ARList<String> list) {  
    ...  
}
```


Not a Good Idea

- While overloading works, it is a *bad* idea in this specific case of `longestStr`, for the following reasons:
 - Code is identical
 - Won't work for future lists
 - If you create a `NewList` class, you will have to make a third method
 - Harder to maintain
 - For example, suppose you find a bug in one of the methods
 - You fix it in the `SLList` version, but may forget to do it in the `ARList` version

Is-A Relationships in Java

- SLList and ARList are both clearly some kind of a list
 - SLList is-a list
 - ARList is-a list
- Expressing this in Java is a two-step process:
 - Step 1: Define *a reference type* called **interface**, let's call this ListIF
 - Step 2: Specify that SLList and ARList are **implementation** of that type



Step 1: Defining ListIF Interface

- We use the new keyword **interface** instead of class to define a ListIF interface
 - Interface is a specification of **what** a ListIF is able to do, **not how** to do it

```
public interface ListIF<T> {  
    public void addLast(T item);  
    public T getLast();  
    public T delLast();  
    public T get(int i);  
    public int size();  
}
```

end method signature with semicolon

every ListIF should be able to do get(i)

every ListIF should be able to do size()

Step 2: Implementing ListIF Interface

- Use the **implements** keyword to tell the Java compiler that SLList and ARList are implementations of ListIF

```
public class ARList<T> implements ListIF<T> {  
    ...  
    public void addLast(T item) {  
        ...  
    }  
}
```

One Method to rule them all



- We can now modify our longestStr method to work on either kind of list

```
public static String longestStr(ListIF<String> list) {  
    int max = 0;  
    for (int i = 0; i < list.size(); i++) {  
        String longestString = list.get(max);  
        String thisString = list.get(i);  
        if (thisString.length() > longestString.length()) {  
            max = i;  
        }  
    }  
    return list.get(max);  
}
```

Method Overriding

- If a subclass has a method with the ***exact same signature*** as in the superclass, we say the subclass **overrides** the method

```
public interface ListIF<T> {  
    ...  
    public void addLast(T item);  
    ...  
}
```

```
public class ARList<T> implements ListIF<T> {  
    ...  
    public void addLast(T item) {  
        ...  
    }  
}
```

we say ARList overrides addLast

Method Overriding vs Overloading

- Overriding: same signature
- Overloading: same method name, different signature
- Another example:

```
public interface Animal {  
    public void makeNoise();  
}
```

```
public class Dog implements Animal {  
    public void makeNoise(Dog d) {  
        ...  
    }  
}
```

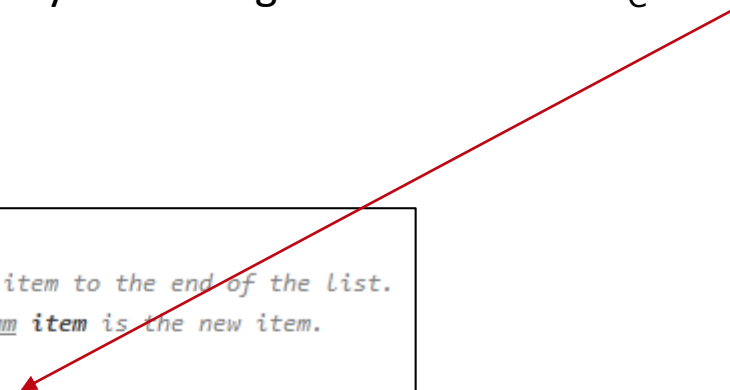
```
public class Pig implements Animal {  
    public void makeNoise() {  
        System.out.print("oink");  
    }  
}
```

makeNoise is **overloaded**,
it behaves differently when
taking a Dog

Pig **overrides** makeNoise

@Override Annotation (1)

- Tag every overriding method with the @Override annotation

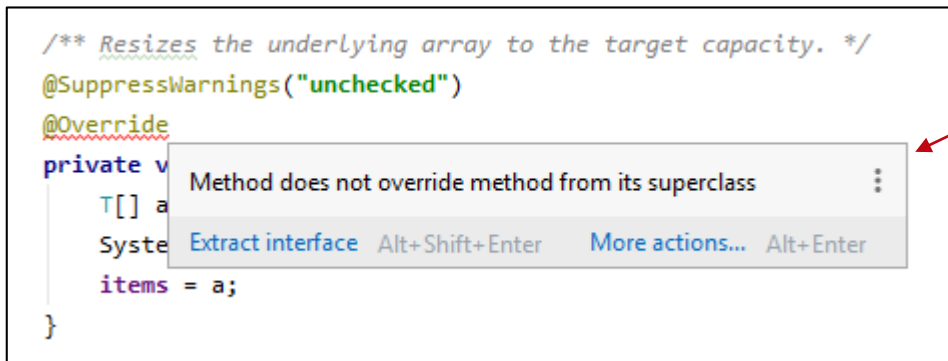


```
/**
 * Adds item to the end of the list.
 * @param item is the new item.
 */
@Override
public void addLast(T item) {
    if (size == items.length) {
        resize( capacity: 2 * size);
    }
    items[size] = item;
    size++;
}
```


@Override Annotation (2)

- Tag every overriding method with the @Override annotation
 - The only effect of this tag is that the code **won't** compile if it is **not** actually an overriding method

```
/** Resizes the underlying array to the target capacity. */
@SuppressWarnings("unchecked")
@Override
private void resize(int targetCapacity) {
    T[] a = (T[]) new Object[targetCapacity];
    System.arraycopy(items, 0, a, 0, items.length);
    items = a;
}
```



for example, adding it to resize
will generate compile error

Why @Override ?

- Why use @Override?
 - protects against typos
 - if you tag a method with @Override, but it isn't actually overriding anything, you'll get a compile error
e.g. `public void addLats(T item)`
 - otherwise, when you want to use the (not) overridden method of the subclass, the one that is actually used is the method of the superclass
e.g. `addLats` is a whole new method
 - reminds programmer/code maintainer that method definition came from somewhere higher up in the inheritance hierarchy

Interface Inheritance

- What we have seen so far in this lecture is Interface Inheritance, where we specify the capabilities of a subclass using the implements keyword
 - Interface: The list of all method signatures
 - Inheritance: The subclass *inherits* the interface from a superclass
 - The interface specifies what the subclass can do, but not how
 - Subclasses **must override all** of these methods!
 - it will fail to compile otherwise
- Interface inheritance is a powerful tool for generalizing code
 - `StrLib.longestStr` works on `SLList`, `ARList`, and even lists that have not yet been invented!

Instantiation Example

```
public static void main(String[] args) {  
    ListIF<String> list1 = new SLList<String>();  
    list.addLast("a");  
}
```

- An SLList object is created and its address is stored in the list1 variable of type ListIF
 - this is accepted by the compiler because SLList implements ListIF
- Then, the string "a" is added into the SLList object by addLast of SLList
 - because addLast of ListIF is overridden by addLast of SLList

Implementation Inheritance

- Another type of inheritance is called **Implementation Inheritance**
- Interface inheritance: subclass inherits signatures, but NOT implementation
 - Implementation inheritance: subclass inherits signatures AND implementation
- Inside an interface, we use the **default** keyword to specify a method that subclasses should inherit from that interface
 - For example, let's add *a default print() method* to ListIF interface

Default Method

- Default method example:

```
public interface ListIF<T> {  
  
    public void addLast(T item);  
    public T getLast();  
    public T dellLast();  
    public T get(int i);  
    public int size();  
  
    default public void print() {  
        for (int i = 0; i < size(); i++) {  
            System.out.print(get(i) + " ");  
        }  
    }  
}
```

we can use the methods defined above in default print method

this print method will be inherited by both SLList and ARList. **Is this an efficient implementation for them?**

Default Method

- Default method example:

```
public interface ListIF<T> {  
  
    public void addLast(T item);  
    public T getLast();  
    public T delLast();  
    public T get(int i);  
    public int size();  
  
    default public void print() {  
        for (int i = 0; i < size(); i++) {  
            System.out.print(get(i) + " ");  
        }  
    }  
}
```

we can use the methods defined above in default print method

because of this `get(i)`, this print implementation is efficient for `ARList`, but **not** for `SLList`!

let's override it in `SLList`!

Overriding Default Method

- This time by overriding, we are not only implement the method, but really **re-implement** the superclass's method
 - In SLList:

```
@Override
public void print() {
    Node p = sentinel.next;
    while (p != null) {
        System.out.print(p.item + " ");
        p = p.next;
    }
}
```

overriding default print
with a more efficient code

Recalling the use of @Override annotation

- Suppose you don't annotate print with @Override, and you misspell print

```
public void prnit() {  
    Node p = sentinel.next;  
    while (p != null) {  
        System.out.print(p.item + " ");  
        p = p.next;  
    }  
}
```

the code still compiles, but when you call print on objects of SLList, the default print will be called !

Which Method Selected

- Recall that if X is a superclass of Y, then an X variable can hold a reference to a Y

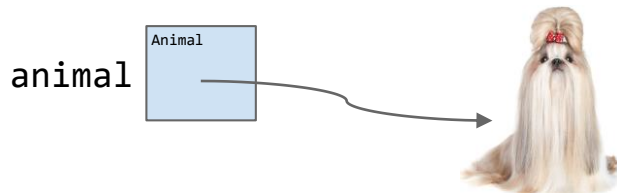
```
public static void main(String[] args) {  
    ListIF<String> list2 = new SLList<String>();  
    list2.addLast("abc");  
    list2.print();  
}
```

- Which print method will run when the code above executes?
 - SLList.print(), and not ListIF.print()
 - How does it work?
 - Before we can answer, we need to understand static and dynamic type

Static Type vs Dynamic Type

- Every variable in Java has a ***compile-time*** type = **static** type
 - This is the type specified at declaration and it *never* changes!
- Variables also have a ***run-time*** type = **dynamic** type
 - This is the type specified *at instantiation* (e.g. when using new)
 - Equal to the type of the object *being* pointed at

```
public static void main(String[] args) {  
    Animal animal;  
    animal = new Dog();  
}
```

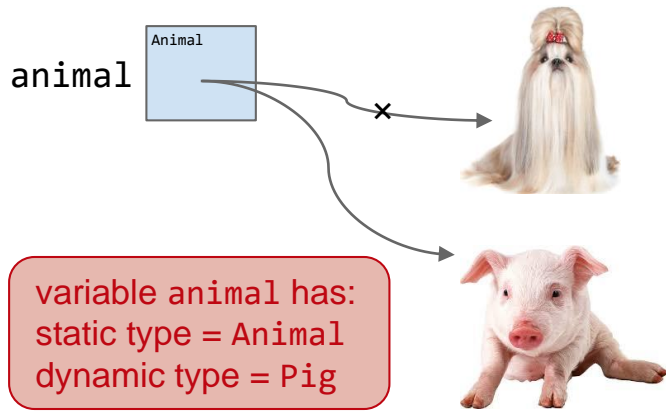


variable animal has:
static type = Animal
dynamic type = Dog

Static Type vs Dynamic Type

- Every variable in Java has a **compile-time** type = **static** type
 - This is the type specified at declaration and it *never* changes!
- Variables also have a **run-time** type = **dynamic** type
 - This is the type specified *at instantiation* (e.g. when using new)
 - Equal to the type of the object *being* pointed at

```
public static void main(String[] args) {  
    Animal animal;  
    animal = new Dog();  
    animal = new Pig();  
}
```



Dynamic Method Selection for Overridden Methods

- Suppose we call a method `m()` of an object using a variable with:
 - static type `X` and dynamic type `Y`
- First, the compiler records the `X`'s method `m()`, to be used in *run-time*
- At *run-time*, **if** `Y` overrides the method `m()`, **then** `Y`'s method `m()` is used *instead*
 - this is known as **dynamic method selection**

Dynamic Method Selection for Print()

- Suppose we call a method `m()` of an object using a variable with:
 - static type `X` and dynamic type `Y`
- First, the compiler records the `X`'s method `m()`, to be used in *run-time*
- At *run-time*, **if** `Y` overrides the method `m()`, **then** `Y`'s method `m()` is used *instead*
 - this is known as **dynamic method selection**
- Therefore, `print()` that belongs to `SLList` is used:

```
public static void main(String[] args) {  
    ListIF<String> list2 = new SLList<String>();  
    list2.addLast("abc");  
    list2.print();  
}
```

A Puzzle

- What will be printed by those four lines?

```
public interface Animal {  
    default void hello(Animal a) {  
        print("hello animal"); }  
    default void sniff(Animal a) {  
        print("sniff animal"); }  
    default void cool(Animal a) {  
        print("cool animal"); }  
}
```

```
public class Dog implements Animal {  
    void sniff(Animal a) {  
        print("sniff dog"); }  
    void cool(Dog d) {  
        print("cool dog"); }  
}
```

```
Animal a = new Dog();  
Dog d = new Dog();  
a.hello(d);  
a.sniff(d);  
d.cool(d);  
a.cool(d);
```

stop!

try to guess and write your answers
in a piece of paper!

In-Class Quiz 2.1

- What will be printed by those four lines?

```
public interface Animal {  
    default void hello(Animal a) {  
        print("hello animal"); }  
    default void sniff(Animal a) {  
        print("sniff animal"); }  
    default void cool(Animal a) {  
        print("cool animal"); }  
}
```

```
public class Dog implements Animal {  
    void sniff(Animal a) {  
        print("sniff dog"); }  
    void cool(Dog d) {  
        print("cool dog"); }  
}
```

```
Animal a = new Dog();  
Dog d = new Dog();  
a.hello(d);  
a.sniff(d);  
d.cool(d);  
a.cool(d);
```

What is the output of this line ?

In-Class Quiz 2.2

- What will be printed by those four lines?

```
public interface Animal {  
    default void hello(Animal a) {  
        print("hello animal"); }  
    default void sniff(Animal a) {  
        print("sniff animal"); }  
    default void cool(Animal a) {  
        print("cool animal"); }  
}
```

```
public class Dog implements Animal {  
    void sniff(Animal a) {  
        print("sniff dog"); }  
    void cool(Dog d) {  
        print("cool dog"); }  
}
```

```
Animal a = new Dog();  
Dog d = new Dog();  
a.hello(d);  
a.sniff(d);  
d.cool(d);  
a.cool(d);
```

What is the output of this line ?

In-Class Quiz 2.3

- What will be printed by those four lines?

```
public interface Animal {  
    default void hello(Animal a) {  
        print("hello animal"); }  
    default void sniff(Animal a) {  
        print("sniff animal"); }  
    default void cool(Animal a) {  
        print("cool animal"); }  
}
```

```
public class Dog implements Animal {  
    void sniff(Animal a) {  
        print("sniff dog"); }  
    void cool(Dog d) {  
        print("cool dog"); }  
}
```

```
Animal a = new Dog();  
Dog d = new Dog();  
a.hello(d);  
a.sniff(d);  
d.cool(d);  
a.cool(d);
```

What is the output of this line ?

In-Class Quiz 2.4

- What will be printed by those four lines?

```
public interface Animal {  
    default void hello(Animal a) {  
        print("hello animal"); }  
    default void sniff(Animal a) {  
        print("sniff animal"); }  
    default void cool(Animal a) {  
        print("cool animal"); }  
}
```

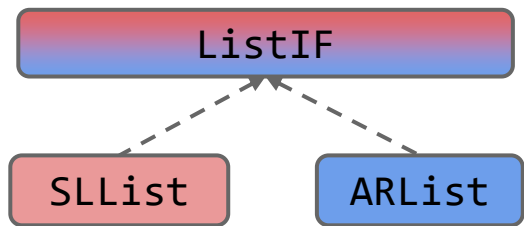
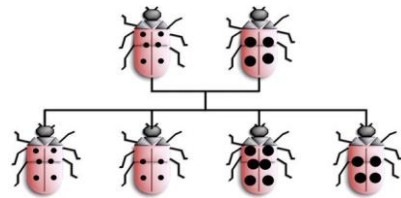
```
public class Dog implements Animal {  
    void sniff(Animal a) {  
        print("sniff dog"); }  
    void cool(Dog d) {  
        print("cool dog"); }  
}
```

```
Animal a = new Dog();  
Dog d = new Dog();  
a.hello(d);  
a.sniff(d);  
d.cool(d);  
a.cool(d);
```

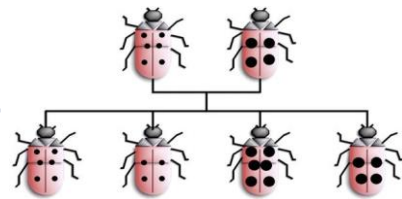
What is the output of this line ?

Implementation Inheritance 2

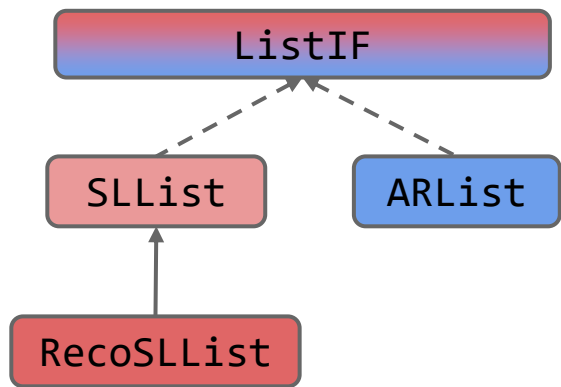
- If you want one class to be a subclass of another class (instead of an interface), you use the keyword **extends**



Implementation Inheritance 2

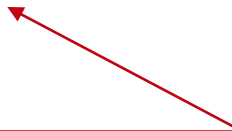


- If you want one class to be a subclass of another class (instead of an interface), you use the keyword **extends**
- For example, you want to build an SLList that
 - Recollects all items that have been deleted by delLast
 - Has an additional method printDelItems(), which prints all deleted items
- Let's call it RecoSLList, and
 - make it to be a subclass of SLList



Extends

- Recall that because of extends, RecoSLList inherits all members of SLList:
 - All instance and static variables
 - All methods
 - All nested classes
- Constructors are **not** inherited



even though inherited, the private members will not be accessible !

```
public class RecoSLList<T> extends SLList<T> {  
    . . .  
  
    public void printDelItems() {  
        . . .  
    }  
}
```

next, we will implement it
you may want to **try it on your own** first!

Storing Deleted Items

- To recollect and print the deleted items, we need store them

```
SLList<T> deletedItems;  
  
/** Prints deleted items. */  
public void printDelItems() {  
    deletedItems.print();  
}
```

let us choose to store them in an SLList!
yes, we can do that!

we can then use the print method
we implement last time

next, we want that deleted items get
automatically added into deletedItems.
how do we do that?

Overriding delLast

- You want to add another step to delLast, so that it stores the deleted item as well

```
@Override
public T delLast() {
    T x = super.delLast();
    deletedItems.addLast(x);
    return x;
}
```

first, you want to do "normal" deletion, however if you copy paste delLast code from SLList, you cannot access the private member variables

so, you can use **super** to call the delLast that belongs to SLList, the *superclass*

and then, store the deleted item

we still need to do one more thing to our class
can you guess what?

Constructor for RecoSLList 1

- We use the constructor to initialize our new member variable: `deletedItems`

```
public RecoSLList() {  
    deletedItems = new SLList<T>();  
}
```

Constructor for RecoSLList 2

- We use the constructor to initialize our new member variable: `deletedItems`

```
public RecoSLList() {  
    deletedItems = new SLList<T>();  
}
```

- In Java, every subclass constructor must call one of the superclass's constructor
 - if we don't, like we did above, Java will add the superclass's default constructor, so what we have is actually:

```
public RecoSLList() {  
    super();  
    deletedItems = new SLList<T>();  
}
```

SLList default constructor will initialize sentinel node and size which RecoSLList also inherited !

Constructor for RecoSLList 3

- Alternatively, you may want to overload the constructor with another constructor taking 1 element
 - in this case, you have to specifically call the constructor of the superclass that takes 1 element
 - otherwise, `super()` will be added and called instead, and that item is lost forever!

```
public RecoSLList(T x) {  
    super(x);  
    deletedItems = new SLList<T>();  
}
```

call `SLList(T x)` instead of `SLList()`

Last week on Iterator

- Last week, we have learned a design pattern for iterating elements using **another class** separated from the one (the data structure) we are iterating
 - we pass the data structure into the class using the constructor
 - we use method hasNext to check whether there are still elements to iterate
 - we use method Next to
 - retrieve the element
 - move to the next element in the data structure

This week on Iterator

- Last week, we have learned a design pattern for iterating elements using **another class** separated from the one (the data structure) we are iterating
 - we pass the data structure into the class using the constructor
 - we use method hasNext to check whether there are still elements to iterate
 - we use method Next to
 - retrieve the element
 - move to the next element in the data structure
- Now, we want use use the **same class** to give us an iterator that can iterate itself!
 - we also want to make that data structure class to be *iterable* using the enhanced for loop (also known as for-each loop)

Making your data structure iterable

- Three steps to make your data structure class DS can be iterated using the enhanced for loop:
 1. The DS must implement an `iterator()` method that return an `Iterator` object
 2. The `Iterator` must implement the methods `next()` and `hasNext()`
 3. You have to tell the world that DS has implemented 1

Making your data structure iterable

- Three steps to make your data structure class DS can be iterated using the enhanced for loop:
 1. The DS must implement an `iterator()` method that return an `Iterator` object
 - so, since we only have one public class, we have to use **private inner class `DSIterator`** to implement the **`Iterator` interface**, and then `iterator()` can instantiate it and return its object
 2. The `Iterator` must implement the methods `next()` and `hasNext()`
 - you have done this last week!
 3. You have to tell the world that DS has implemented 1
 - we use standard interface for this: **`Iterable` interface**

let's make our `ARList` iterable as an example!

Step 1

- The DS must implement an `iterator()` method that return an `Iterator` object

```
import java.util.Iterator;
```

Iterator is defined here, need to import it

```
/**  
 * Make an iterator  
 */  
public Iterator<T> iterator() { return new ARListIterator(); }
```

the method `iterator()` needs to return an `Iterator`

we will have to define this inner class,
and it has to implement the `Iterator` interface,
so that its object is indeed an `Iterator`

Step 2

- The Iterator must implement the methods hasNext() and next()
 - ARListIterator is-an Iterator, implement this interface

```
private class ARListIterator implements Iterator<T> {  
    private int index;  
  
    public ARListIterator() { index = 0; }  
  
    @Override  
    public boolean hasNext() { return index < size; }  
  
    @Override  
    public T next() {  
        T nextItem = items[index];  
        index++;  
        return nextItem;  
    }  
}
```

we have learned about this last week

being an inner class inside ARList,
it has direct access to ARList
member variables

being an Iterator, ARListIterator
must have hasNext() and next(),
override them!

Step 3

- You have to tell the world that DS has implemented 1
 - without this step 3, you can already use the "naked" enhanced for loop

```
// naked enhanced for loop
Iterator<Integer> iter = intList.iterator();
while(iter.hasNext()) {
    System.out.println(iter.next());
}
```

Step 3

- You have to tell the world that DS has implemented 1
 - without this step 3, you can already use the "naked" enhanced for loop

```
// naked enhanced for loop
Iterator<Integer> iter = intList.iterator();
while(iter.hasNext()) {
    System.out.println(iter.next());
}
```

- but to be able to **use enhanced for loop** you need *another interface*:

```
public class ARList<T> implements ListIF<T>, Iterable<T> {
```

now you can do:

```
// enhanced for loop / for-each loop
for(int item : intList) {
    System.out.println(item);
}
```

to be an Iterable, must implement iterator()

we tell Java that we have implemented it,
so Java can use it to do enhanced for loop

Thank you for your attention !

- In this lecture, you have learned:
 - about ADTs: their operations, specs, good ones, and to test ones
 - to use representation independence so that the implementation of an ADT can change without requiring changes from its clients
 - to use interface to define data structure operations
 - to differentiate overriding versus overloading a method
 - to reuse codes from parent class
 - to make user-defined data structure iterable
- Please continue to Lecture Quiz 10 and Lab 10:
 - to do Lab Exercise 10.1 - 10.3, and
 - to do Exercise 10.1 - 10.3