CSE 1325: Object-Oriented Programming

Lecture 06

Strings, ArrayLists, Docs, and Class Relationships

Mr. George F. Rice

george.rice@uta.edu

Office Hours:
Prof Rice 11:00 Tuesday and Thursday in ERB 336
For TAs see this web page

Descartes walks into a bar.
The bartender asks, "Want a beer?" He replies, "I think not!" and disappears.

Today's Topics

- More on Strings
- ArrayList and arrays
- Using Java Documentation
- Class Relationships
  - Association (Class)
  - Dependency
  - Aggregation
  - Composition
  - Inheritance
- Unified Modeling Language
  - Relationships on Class Diagrams (inline throughout)

Text Matters

- All data can be represented by text
  - Books, articles, web pages
  - Tables of structured information (e.g., XML, JSON)
  - Email, SMS, social media
  - Graphics (e.g., vector formats)
  - Software code(!)
  - Binary data (e.g., uuencode, uudecode)
  - All languages and way too many emojis
- Text is very portable (except that annoying \n, \r, \r\n thingie)
- Text is easily created and edited by your choice of text editor

Java Options for Representing Text

- Old C-style(ish) strings (byte or char array)
  - `byte[] btext = {'H', 'e', 'l', 'l', 'o'};  // equivalent to C char[]`
    `for(byte b : btext) System.out.print((char) b);`
  - A char is 2 bytes in Java, though – we're asking for trouble with byte[ ]!
  - byte[ ] is primitive and rarely done except in unusual I/O cases (embedded code)
  - char[ ] is sometimes helpful in select circumstances only
- String (an *immutable* class!)
  - `String text = "Hello"; System.out.println(text);`
  - `String text = new String(btext); // Convert byte array to a String`
  - But *immutable* – once created, a String object can never change
  - Still, the most common representation for text
- StringBuilder (a *mutable* class!)
  - `StringBuilder sb = new StringBuilder("Hello");`
    `System.out.println(sb);`
  - More efficient than String for lots of appends and inserts
- StringBuffer – StringBuilder for threads (see Lecture 24)

Some Useful String Methods
(1 of 5)

```
public class StringDemo {
    public static void main(String[] args) {
        // Special case for initialization of Strings, but could use
        //   String s = new String("I am a Java programmer!");
        String s = "I am a Java programmer!";

        // Java 16+ includes Text Blocks for initializating multiline String objects.
        // Note the use of triple-double-quotes and indents to encode the sentence.

        String humpty = """
                        "When I use a word," Humpty Dumpty said,
                        in rather a scornful tone, "it means just what I
                        choose it to mean - neither more nor less.""";

        // Special case for concatenating String objects using +, but could use
        //     s.concat(" characters");
        // The length() method returns the number of characters in the String
        System.out.println("'" + s + "' is " + s.length() + " characters");
```

```
ricegf@antares:~/dev/202108/09/code_from_slides$ java StringDemo
'I am a Java programmer!' is 23 characters
```

Some Useful String Methods
(2 of 5)

```
// We can do a lot of comparisons using String methods!
String first = "george";
String last = "rice";
if(first.startsWith("ge")) System.out.println(first + " starts with ge");
if(first.endsWith("ge")) System.out.println(first + " ends with ge, too!");

String compared = " equals ";
if (first.compareTo(last) < 0) compared = " less than ";
if (first.compareTo(last) > 0) compared = " greater than ";
System.out.println(first + " is" + compared + last);

// We can also ignore case
String sarcasm = "GeOrGe";
compared = " equals ";
if (first.compareToIgnoreCase(sarcasm) < 0) compared = " less than ";
if (first.compareToIgnoreCase(sarcasm) > 0) compared = " greater than ";
System.out.println(first + " is" + compared + sarcasm + " (ignoring case)");

// We could also use first.equals(last) and first.equalsIgnoreCase(sarcasm)
if(first.equalsIgnoreCase(sarcasm))
    System.out.println(first + " equalsIgnoreCase " + sarcasm);
```

```
george starts with ge
george ends with ge, too!
george is less than rice
george is equal to GeOrGe (ignoring case)
george equalsIgnoreCase GeOrGe
```

Some Useful String Methods
(3 of 5)

```
// In addition to using System.printf, you can format a String variable
String format = "I am %d years old!";
int age = 25;
String s2 = String.format(format, age);
System.out.print(s2);

// Use valueOf to convert a string to a numeric class type (Integer or Double)
String sage = "37";
age = Integer.valueOf(sage);
System.out.printf(format, age);

// We can iterate over a String's chars using toCharArray()
// Note that a Java char only covers the most common 65,536 code points (ahem)
//   of the 154,998 defined (as of Unicode 16.0) and 1,114,112 possible.
//   Handling the other 16 planes of 16-bit code points is unfortunately hard[1]
//   in Java but fortunately will NOT be on the exam!
for(char c : sage.toCharArray()) System.out.println(c);

// Alternately, we can access characters with an index
for(int i=0; i<sage.length(); ++i)
    System.out.println(sage.charAt(i));
```

```
I am 25 years old!
I am 37 years old!
3
7
3
7
```

[1] See ExtendedCodePoints.java for an example.

Some Useful String Methods
(4 of 5)

```
// We can also take substrings starting at the first index
//   and (optionally) ending at the last index-1
System.out.println(s.substring(5) + " " + s.substring(0, 4));

// We can remove whitespace from both ends of the String and convert
//   to upper (toUpperCase) or lower (toLowerCase) case
String spacey = "  I do object-oriented!   \n";
System.out.println(spacey.trim() + " I said, " + spacey.trim().toUpperCase());

// Searching within a String is also supported
System.out.println("In '" + s + "', 'Java' starts at " + s.indexOf("Java"));
System.out.println("The last 'am' in '" + s
                 + "' starts at " + s.lastIndexOf("am"));
String filename = "Readme.txt";
int dot = filename.lastIndexOf(".");
System.out.println("In " + filename
    + ", the name is '" + filename.substring(0, dot)
    + "' and the extension is '" + filename.substring(dot+1) + "'");
```

```
a Java programmer! I am
I do object-oriented! I said, I DO OBJECT-ORIENTED!
In 'I am a Java programmer!', 'Java' starts at 7
The last 'am' in 'I am a Java programmer!' starts at 17
In Readme.txt, the name is 'Readme' and the extension is 'txt'
```

Some Useful String Methods
(5 of 5)

```
        // We can also split a String into an array around a given character or regex
        // String sep is an Old Programmer's Trick for inserting commas between
        //   list elements but not at the beginning or end
        String pathname = "/home/ricegf/Documents/resume.odt";
        System.out.print("In " + pathname + ", the directories and filename are \n  ");
        String sep = "'";
        for(String dir : pathname.split("/")) {
            System.out.print(sep + dir);
            sep = "', '";
        }
        System.out.println("'");

        // Or we can replace a substring
        System.out.println(s.replace("Java", "good"));

        // We can't insert strings into a String (although we could into a StringBuilder)
        // It's easier to split and concatenate anyway
    }
}
```

```
I do object-oriented! I said, I DO OBJECT-ORIENTED!
In 'I am a Java programmer!', 'Java' starts at 7
The last 'am' in 'I am a Java programmer!' starts at 17
In Readme.txt, the name is 'Readme' and the extension is 'txt'
In /home/ricegf/Documents/resume.odt, the directories and filename are
  '', 'home', 'ricegf', 'Documents', 'resume.odt'
I am a good programmer!
ricegf@antares:~/dev/202108/09/code_from_slides$
```

Reversing a String

- How would you reverse a String?
  - We can iterate over chars in the String
  - + concatenates
  - So something like this?

```
// Reverse using String concatenation in a loop
public static String ReverseWithNewString(String s) {
    String result = "";
    for(char c : s.toCharArray()) result = c + result;
    return result;
}
```

- But String is *immutable* – it cannot change
  - So we're creating a new String for *every char*!

Reverse via StringBuilder

- A StringBuilder object is *mutable*
  - AND it has a reverse() method already!
  - Is it worth converting a String to a StringBuilder to reverse it and convert back to a String?

```
// Reverse by converting to StringBuilder, reverse(), and toString()
public static String ReverseWithStringBuilder(String s) {
    StringBuilder sb = new StringBuilder(s);
    return sb.reverse()
             .toString();
    // OR return (new StringBuilder(s)).reverse().toString();
}
```

Hey, look – method chaining!

- Don’t pontificate – demonstrate!
  - Or as Linus Torvolds said, “Show me the code!”

String vs StringBuilder

```
public static void main(String[] args) {
    // Try using String concatenation
    long scStartTime = System.nanoTime();
    String scString = ReverseWithNewString(args[0]);
    long scElapsedTime = System.nanoTime() - scStartTime;

    // Try using StringBuffer
    long sbStartTime = System.nanoTime();
    String sbString = ReverseWithStringBuilder(args[0]);
    long sbElapsedTime = System.nanoTime() - sbStartTime;

    // Print results
    if(!sbString.equals(scString))
        System.err.println("Reversed string mismatch!"
            + "\n  sb = " + sbString
            + "\n  sc = " + scString);
    System.out.printf("String concatenation took %12d nanoseconds\n",
            scElapsedTime);
    System.out.printf("StringBuilder       took %12d nanoseconds\n",
            sbElapsedTime);
    System.out.println(
        ((scElapsedTime < sbElapsedTime)
             ? "String concatenation is faster by "
             : "StringBuilder is faster by ")
      + 100 * Math.abs(scElapsedTime - sbElapsedTime)
        / Long.min(scElapsedTime, sbElapsedTime)
      + "%"
    );
```

StringBuildervsString.java

Here's how to time and compare methods!

And the Winner Is...

- StringBuilder by a country mile!

```
ricegf@antares:~/dev/202301/06/code_from_slides$ java StringBuilderVsString
String concatenation took     15576490 nanoseconds
StringBuilder         took        28243 nanoseconds
StringBuilder is faster by 55051%
ricegf@antares:~/dev/202301/06/code_from_slides$
```

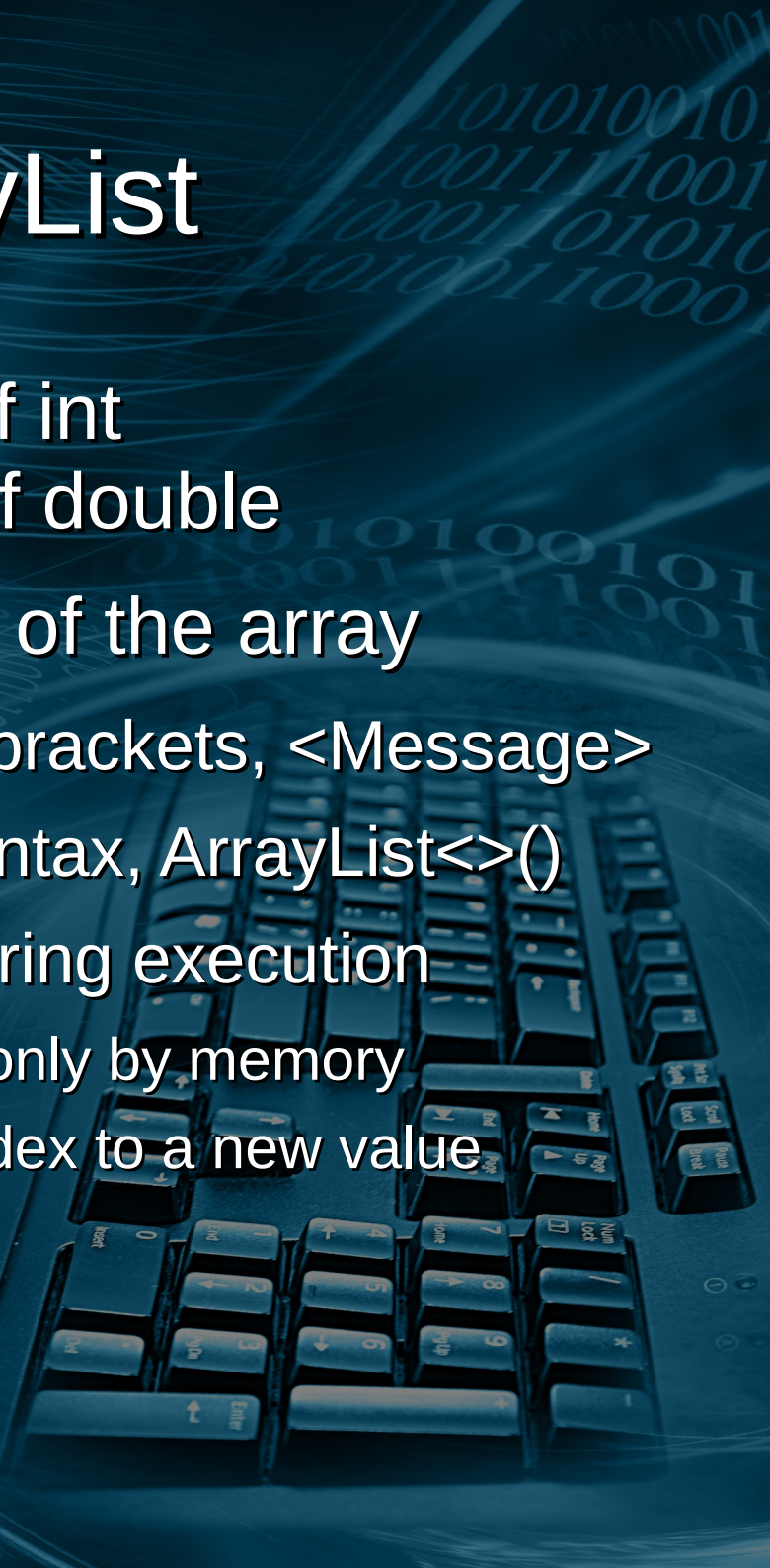
- Lessons to learn from this
  - Any serious String manipulation should be done with a StringBuilder object instead
  - Check StringBuilder for optimized algorithms – you probably aren't the first to need a transformation

Summary of Java String Methods

```
// Comparisons
int s.length()                                  // Number of chars
bool first.equals(last)                         // Compare by chars (NEVER use ==)
bool first.equalsIgnoreCase(sarcasm)            //   Same ignoring upper / lower case
int  first.compareTo(last)                      // negative if <, 0 if equals, positive if >
int  first.compareToIgnoreCase(sarcasm)         //   Same ignoring upper / lower case
bool first.startsWith("ge")                     // Compare first few chars
bool first.endsWith("ge")                       // Compare last few chars

// Conversions
String   s1 + s2 + i                            // Concatenation (toString() is implied!)
String   String.format(format, age)             // To String using sprintf format
int      Integer.valueOf(sage)                  // From String to an int
String   spacey.trim()                          // Remove whitespace from beginning and end
String   spacey.toUpperCase()                   // Convert all chars to upper case
String   spacey.toLowerCase()                   // Convert all chars to lower case

// Chars and substrings
char[]   sage.toCharArray()                     // Convert entire String to array of char
char     sage.charAt(i)                         // Char at subscript i
String   s.substring(0, 5) + s.substring(5)     // Chars 0 to 4 then 5 to end

// Search, replace, and splits
int      s.indexOf("Java")                      // Subscript where first "Java" starts
int      s.indexOf("Java", 42)                  //   Same for first "Java" after index 41
int      s.lastIndexOf("Java")                  //   Same for last "Java" (or -1 if none)
String   s.replace("Java", "good")              // Substitute "good" for all "Java"
String[] pathname.split("/")                    // Create array of Strings between all "/"
```

Java ArrayList

- Integer is the class version of int
  Double is the class version of double
- ArrayList is the class version of the array
  - The type is specified in angle brackets, <Message>
  - Instantiation uses a special syntax, ArrayList<>()
  - The size can freely change during execution
    - Add (append) elements limited only by memory
    - Set (replace) any element by index to a new value
    - Remove any element by index



Comparing Arrays with ArrayList

Declaring and Initializing

ArrayListDemo.java

```
import java.util.ArrayList; // Unlike arrays, ArrayList must be imported for "bare" use

public class ArrayListDemo {
    public static void main(String[] args) {

        // Declaring an array and array list is similar
        Integer[] array;         // Same as C syntax - could also be int
        ArrayList<Integer> ali;  // MUST be class type such as Integer, NOT int

        // In Java, BOTH must be initialized using new
        array = new Integer[10];   // Size must be specified when allocated
        ali   = new ArrayList<>(); // Size need NOT be declared, can change dynamically!
                                   // Also note the empty <> - Java knows ali's type!

        // Filling the array and array list
        for(int i=0; i<array.length; ++i) {
            array[i] = (int) (100.0 * Math.random()); // Must specify the subscript
            ali.add((int) (100.0 * Math.random()));   // add method appends to the list
        }

        // Continued next slide
```

Comparing Arrays with ArrayList
Size and Subscripts

```
        // Continued from previous slide

        // Printing the size of the array and array list
        System.out.println("Array size is "     + array.length); // fixed property
        System.out.println("ArrayList size is " +  ali.size()); // changeable method

        // Printing the array and array list using indexing
        for(int i=0; i<ali.size(); ++i) {
            System.out.printf("  array[%d] = %2d, ", i, array[i]);   // subscript
            System.out.printf("ali.get(%d) = %2d\n", i, ali.get(i)); // get method
        }

        // Or use the for-each loop on either
        for(int i : array) System.out.printf(" %2d", i);
        System.out.println();

        for(int i : ali)   System.out.printf(" %2d", i);
        System.out.println();
    }
}
```

Other useful ArrayList methods include **`set(index, value)`** to overwrite an existing element, **`remove(index)`** to delete an element, **`clear()`** to remove all elements, and **`indexOf(value)`** to get the index of the first **`value`** in the ArrayList or -1 if not there

Using Java Documentation

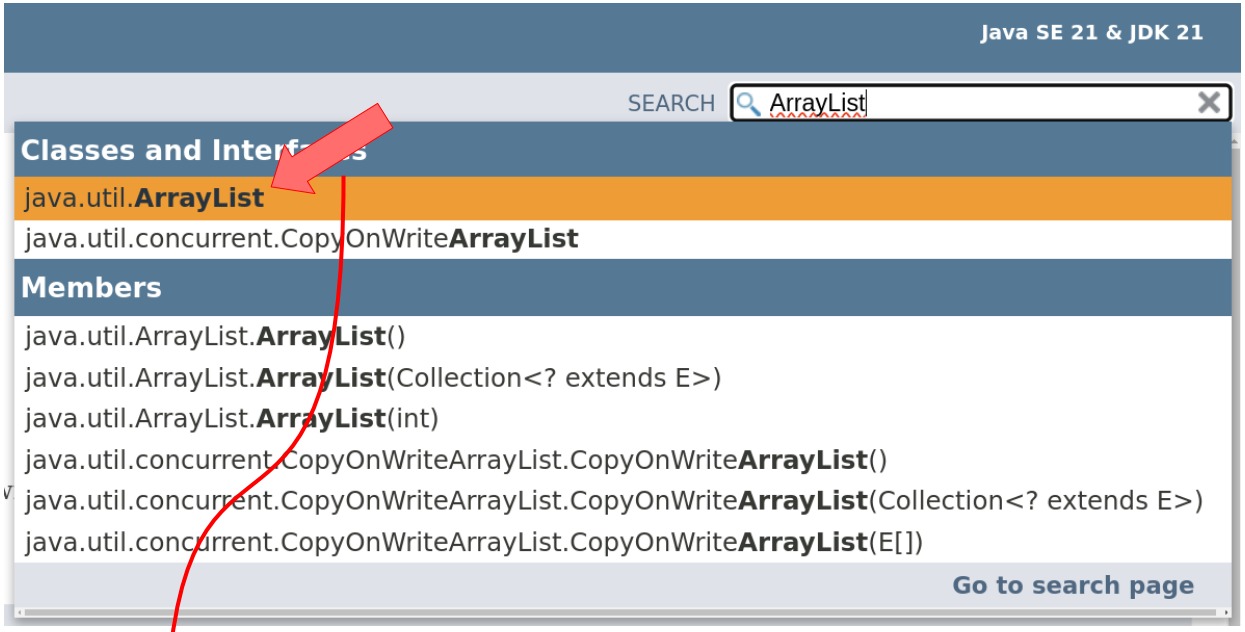
- What does ArrayList's **`indexOf`** method return?

Go to the JDK 21 web docs
https://docs.oracle.com/en/java/javase/21/

Select API Documentation
in the upper left

Enter the name of the class
for which you need info
next to SEARCH in upper right

Select the best match!



Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method	Description
void	`add(int index, E element)`	Inserts the specified element at the specified position in this list.
boolean	`add(E e)`	Appends the specified element to the end of this list.
boolean	`addAll(int index, Collection<? extends E> c)`	Inserts all of the elements in the specified collection into this list, starting at the specified position.
boolean	`addAll(Collection<? extends E> c)`	Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's Iterator.
void	`addFirst(E element)`	Adds an element as the first element of this collection (optional operation).

Using Java Documentation

- What does ArrayList's **`indexOf`** method return?

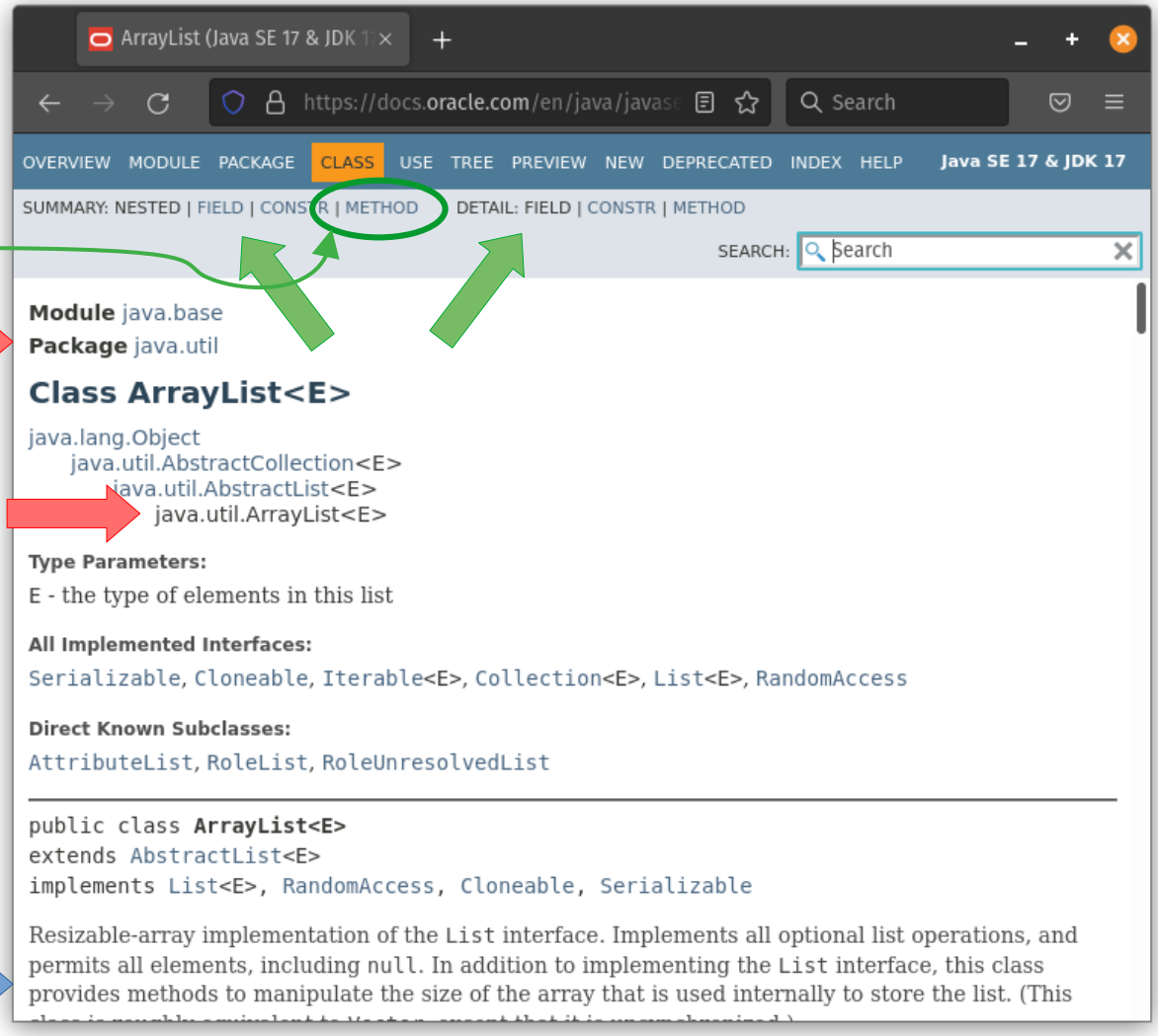
Summary takes you to spec overviews, while **Detail** takes you deep.

Since **`indexOf`** is a method, we select **Summary: Method**

Package tells how to import

```
import java.util.ArrayList;
```

Examples are often found starting here

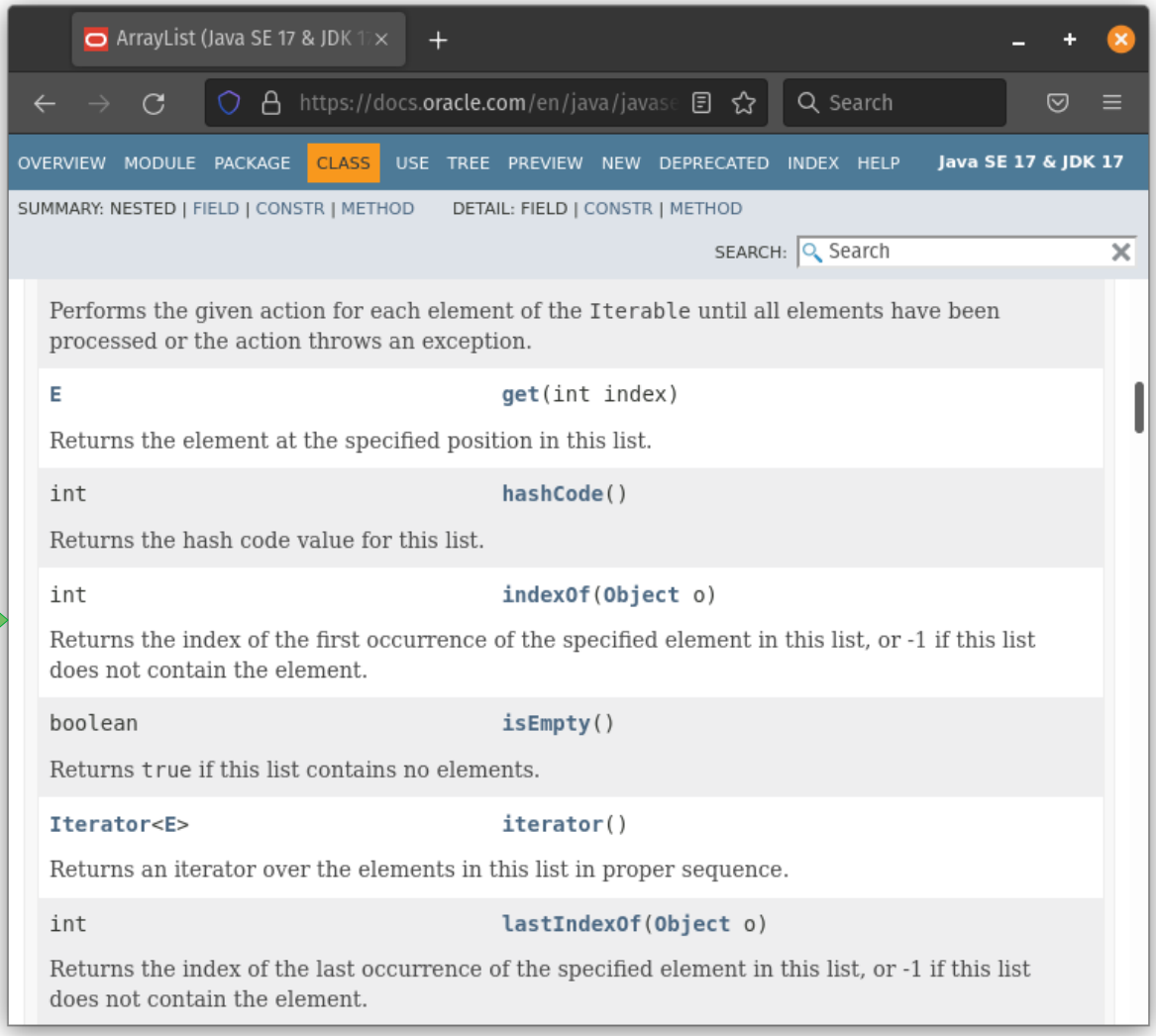


Using Java Documentation

- What does ArrayList's **`indexOf`** method return?

int indexOf(Object o) tells you the return type, and the text summarizes its meaning.

Want more? Click the return or parameter type for details on their respective classes – or click **`indexOf`** for even more on this method.



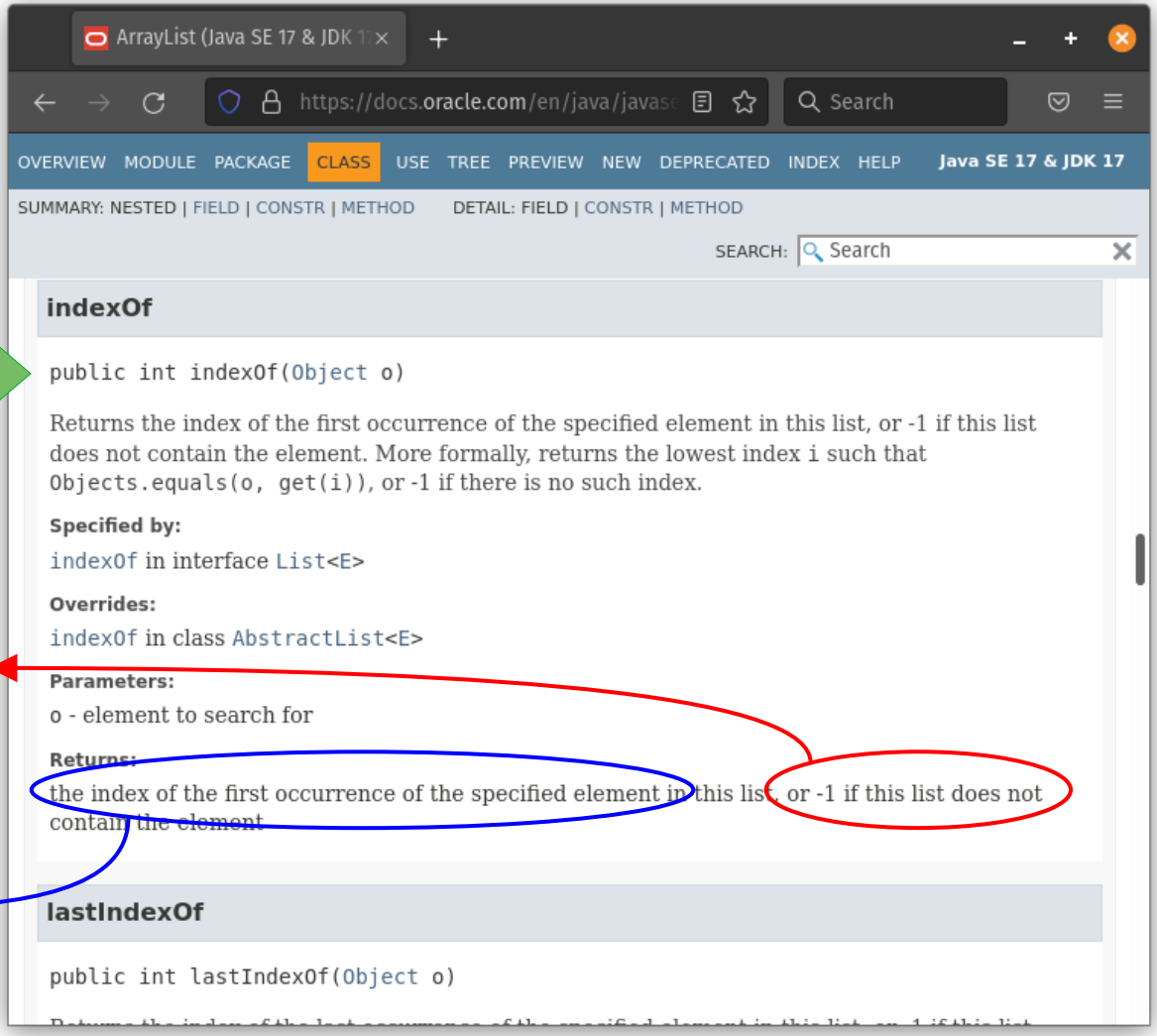
Using Java Documentation

- What does ArrayList's **`indexOf`** method return?

Even more!

Do NOT copy this spec into your code! Use the method name *on an ArrayList object* with a *parameter* of the matching type, and handle the *return value*.

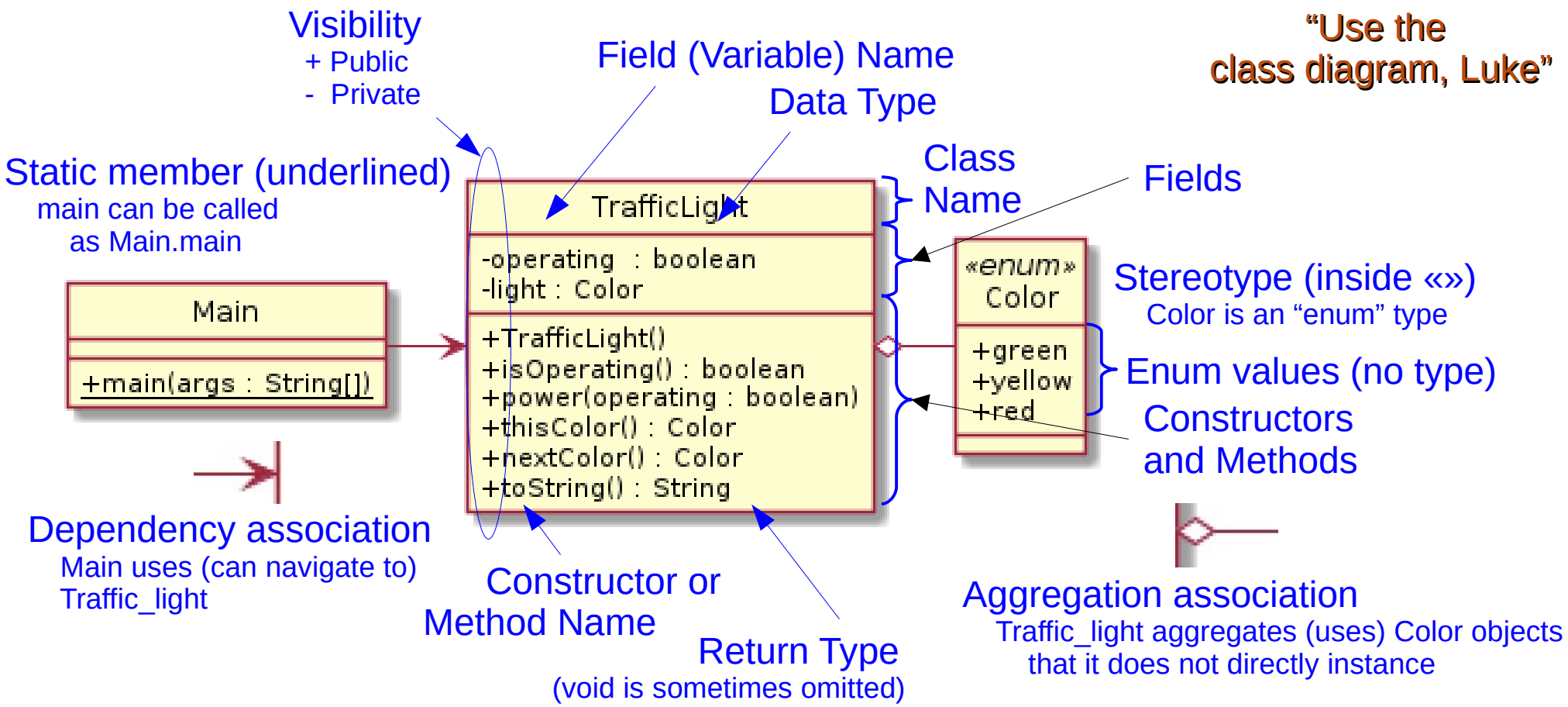
```
ArrayList<String> als =
  new ArrayList<>();
// Fill als with text
String key = "cse1325";
if(als.indexOf(key) == -1)
  System.out.println(
    "als has no " + key);
else System.out.println(
    "als has " + key
  + " at index"
  + als.indexOf(key));
```



Review

UML Class Diagram

"Use the class diagram, Luke"



The class diagram is your "battle map" for implementation

Simply (ahem) write the classes, fields, and methods in Java as shown

Classes Share Relationships

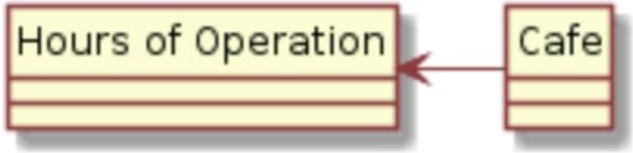
- To date, our classes have (mostly) stood alone
- Most non-trivial programs have numerous classes which interact in interesting ways
  - They reference and depend on each other
  - They aggregate or composite into larger classes
  - They reuse data and methods
- These interactions are based on *relationships*

While relationships are more easily understood in the UML, we'll also look at some equivalent Java implementations.

UML Class Relationships
Association

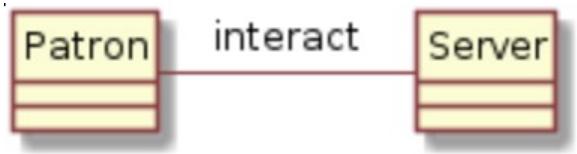
- A line simply indicates a general association
  - One class has a relationship to another class

General

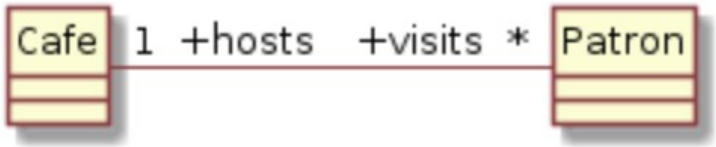


Directional
(points to the included class)

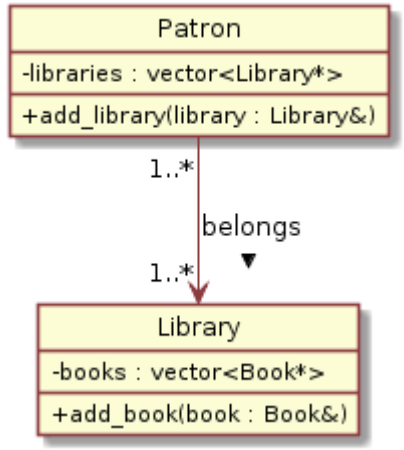
  - The association can be named



  - Each class may include multiplicity and role

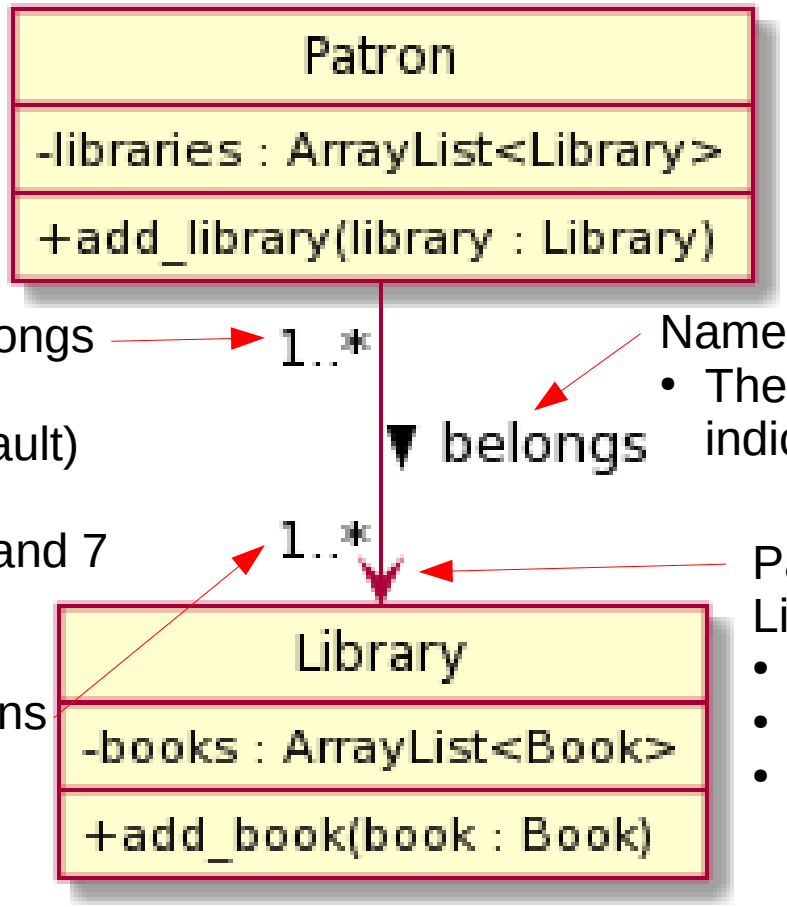


Association



- **Association** requires
  - The two classes are otherwise unrelated
    - Patrons and libraries are unrelated other than the specified interaction of membership (“belongs”)
  - The classes may be involved in any number of associations
    - A Patron may also interact with Books, as may a Library
  - The classes have independent lifecycle
    - Neither constructs or destructs the other
  - Each class may or may not reference the other
    - In this case, Patron references Library, but not vice versa (Yes, a Library SHOULD know its Patrons! Work with me here...)

Association



Multiplicity – A Patron belongs to *many* Libraries

- 1 means exactly 1 (default)
- 5 means exactly 5
- 3..7 means between 3 and 7
- 1..* means 1 or more
- 0..1 means 0 or 1

A Library has many Patrons

Named – "Patron *belongs to* Library"

- The is optional, but when present indicates the direction to read

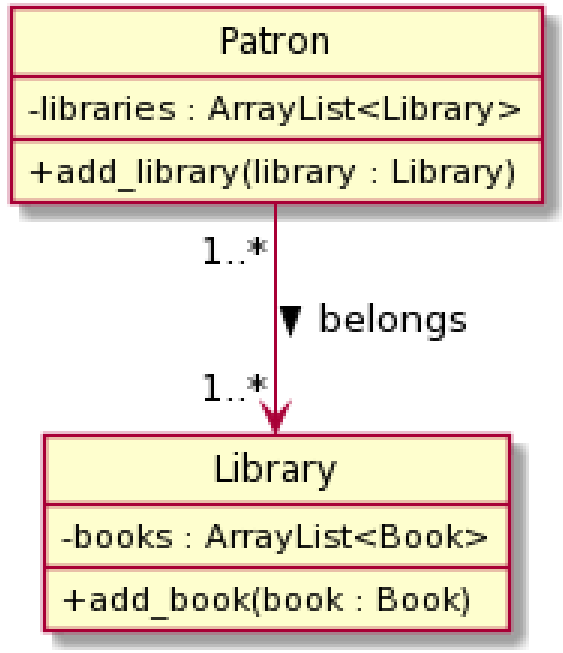
Patron references Library, but Library does not reference Patron

- Directional (one arrow)
- Non-directional (no arrows)
- Bi-directional (both arrows)

Association in Java

```java
import java.util.ArrayList;

public class Patron {
  // Association: Each Patron knows to which Library they belong
  public void add_library(Library library) {libraries.add(library);}

  private ArrayList<Library> libraries;
}
```

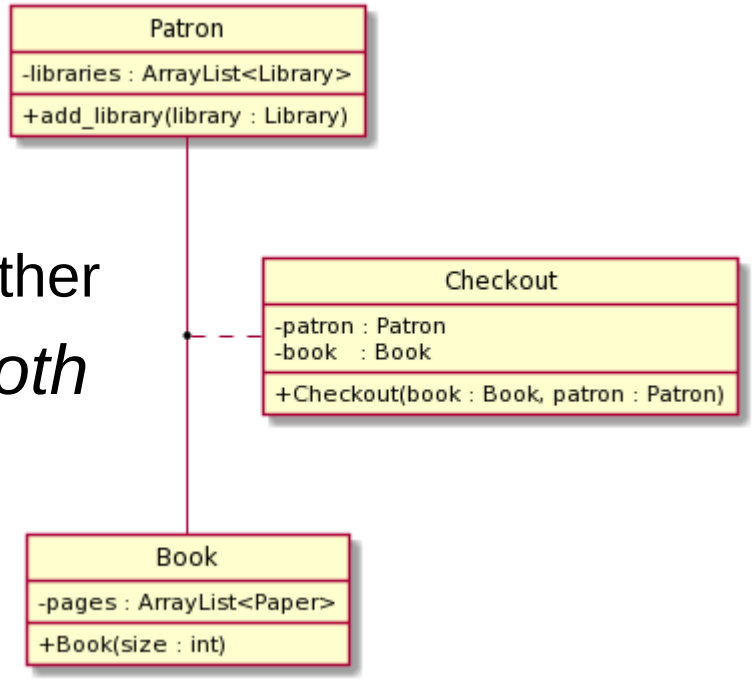
```java
import java.util.ArrayList;

public class Library {
  public void add_book(Book book) {books.add(book);}

  private ArrayList<Book> books;
}
```



Note: A Java **ArrayList** is the class version of an array.
You can append new elements with the **add(E)** method, replace elements with **set(int, E)**, remove elements with **remove(int)**, remove all with **clear()**, subscript with the **get(int)** method, and get the number of elements using the **size()** method, among many others.

UML Class Relationships
Association Class

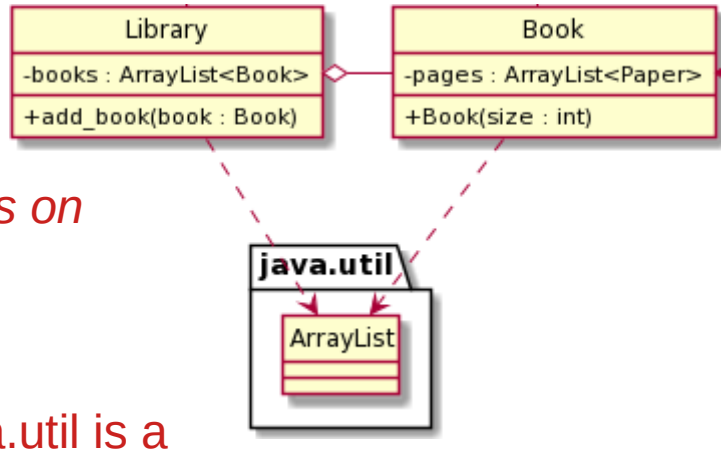
- **Association Class** requires
  - Same as Association, except
    - The classes do not reference each other
  - Instead, a third class references *both*
    - In this case, the Checkout class references *both* the Patron and Book class, creating the association




```
public class Checkout {
  // Association Class: Checkout associates a Book with a Patron
  public Checkout(Patron patron, Book book) {
      this.patron = patron;
      this.book = book;
  }

  private Patron patron;
  private Book book;
}
```

UML Class Relationships
Dependency

- **Dependency** shows that one class depends in some way on another
  - Library and Book depend on ArrayList
- The classes have no fields of the other type
  - Rather, they *use* the other type in some way



Library *depends on* ArrayList

java.util is a "package" containing ArrayList

```
// Dependency: on ArrayList
import java.util.ArrayList;

public class Library {
  public void add_book(Book book) {
    books.add(book);
  }

  private ArrayList<Book> books;
}
```

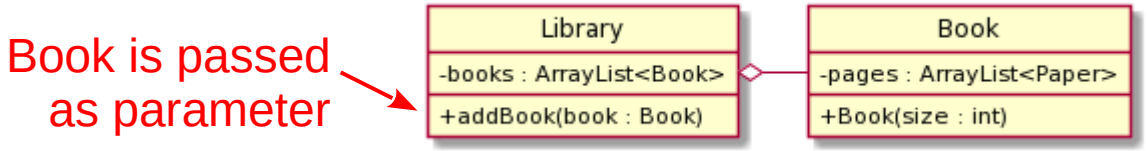
Aggregation vs Composition

- Both usually mean a field of the other's type
  - If Book is composed of Pages, Book will have a Page[ ], ArrayList<Page>, or similar field
- **Compositions** contain and manage their fields **`new` keyword**
  - Constructs the object referenced by the field (this may require a special "copy constructor" for the field type)
  - Responsible for deleting the object (the garbage collector handles this in Java, but it's important in C++)
- **Aggregations** reference external data **passed as parameter**
  - Referenced objects are constructed (and deleted) elsewhere
  - Fields are usually set to a reference passed as a parameter, for example in the constructor

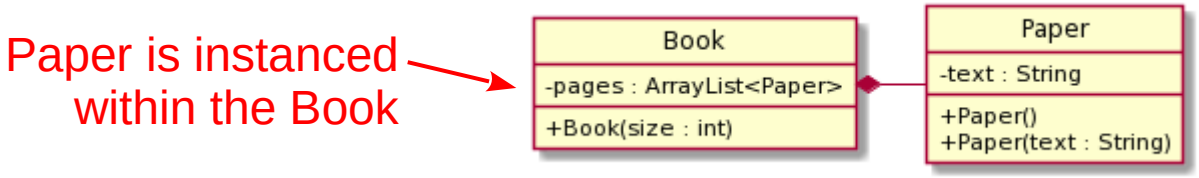
UML Class Relationships

Aggregation and Composition

- **Aggregation** shows a class being comprised of one or more other classes by *reference*
  - The library includes many books, which may come and go without affecting the existence of either

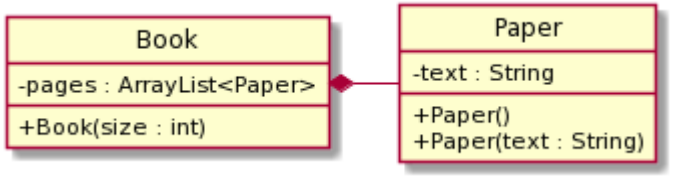


- **Composition** shows that the existence of the class depends on the composite class
  - Books are composed of many pages of Paper, and ceases to exist if the pages are removed



Composition

- **Composition** requires
  - The instance is a member of the class
    - Pages is a field of Book
  - The instance can only belong to one class at a time
    - Each Paper instance is part of the Book's allocated memory
    - Though other classes may have a reference to it
  - The instance is managed by the class
    - It is constructed when the class is constructed or when requested via a method
    - It is deleted when the class is deleted
    - No external code manages the instance's existence
  - The instance has no reference to the class
    - A page cannot modify the rest of the book

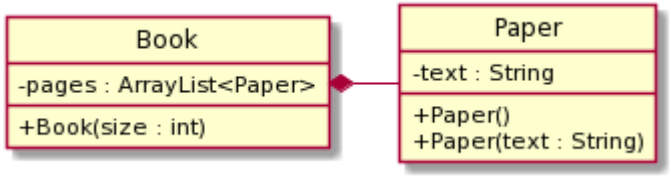


(Diamond is next to the compositor)

Different from aggregation

Composition in Java

```java
public class Paper {
  public Paper() {this("");}
  public Paper(String text) {this.text = text;}

  private String text;
}
```

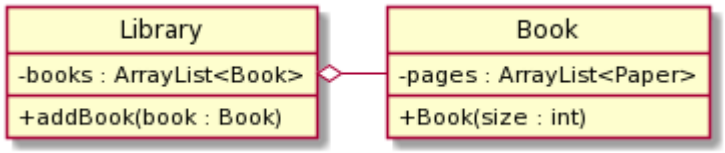



```java
import java.util.ArrayList;

public class Book {
  public Book(int size) {
      // Composition: The paper is contained entirely within the Book
      for(int i=0; i<size; ++i) pages.add(new Paper());
  }
  private ArrayList<Paper> pages;
}
```

Paper is instanced within the Book

Aggregation

- **Aggregation** requires
  - The instance is a member of the class
    - Books is a field of Library



(Diamond is next to the aggregator)

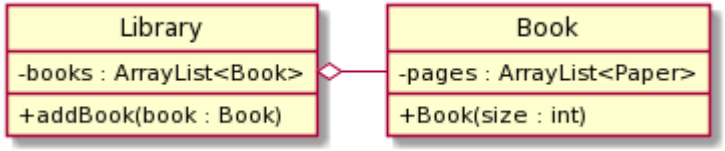
  - The instance can belong to more than one class at a time
    - Each Book instance may be part of several Libraries' allocated memory
  - The instance is NOT managed by the class

Different from composition

    - It is constructed independently of when the class is constructed
    - It is deleted independently of when the class is deleted
    - The instance's existence is managed elsewhere
  - The instance has no reference to the class
    - A book cannot modify the rest of the library

Aggregation in Java

```java
import java.util.ArrayList;

public class Book {
  public Book(int size) {
      // Composition: The paper is contained entirely within the Book
      for(int i=0; i<size; ++i) pages.add(new Paper());
  }
  private ArrayList<Paper> pages;
}
```

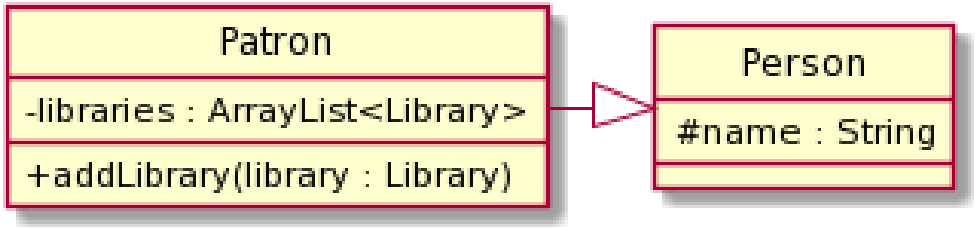



```java
import java.util.ArrayList;

public class Library {
  // Aggregation: Each Book is created and exists external to Library
  public void addBook(Book book) {books.add(book);}

  private ArrayList<Book> books;
}
```

Book is passed as parameter

UML Class Relationships
Inheritance

- Inheritance shows an “is a” relationship
  - A Patron “is a” Person



- This one is very important
  - You can build complex non-Java software without inheritance (though Java requires it for “Interfaces”)
  - But inheritance helps greatly with *some* types of software, such as graphical libraries

Inheritance
with People

- When you inherit from an ancestor, you acquire (many of) their assets.
  - Some may be redirected by a will
- When a class inherits from an ancestor class, it acquires (many of) its methods and fields (also called attributes).
  - Some may be redirected by keyword directives

"Assets"

The Heir

The Ancestor

```
class You extends RichAuntAala {
```

The "Will"

Inheritance
with Classes

- **Inheritance** – Reuse and extension of fields and method implementations from another class

- The original class is called the **superclass** (e.g., Exception)
- The extended class is called the **subclass** (e.g., BadArea)

"Assets"

The Heir — The Ancestor

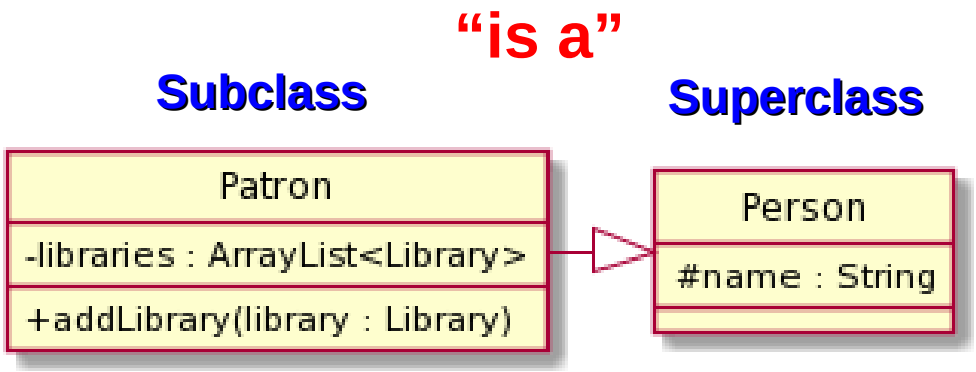
```
class BadArea extends Exception {
```

Subclass — **Superclass**

The "Will"

Inheritance in the UML and Java

Inheritance is an “is a” relationship. That is, a Patron “is a” Person.
If you can say “is a” about a relationship, that relationship may well be inheritance!




```
class Person {
```

Patron *inherits* name from Person. <u>Both</u> classes can access name in their respective methods.

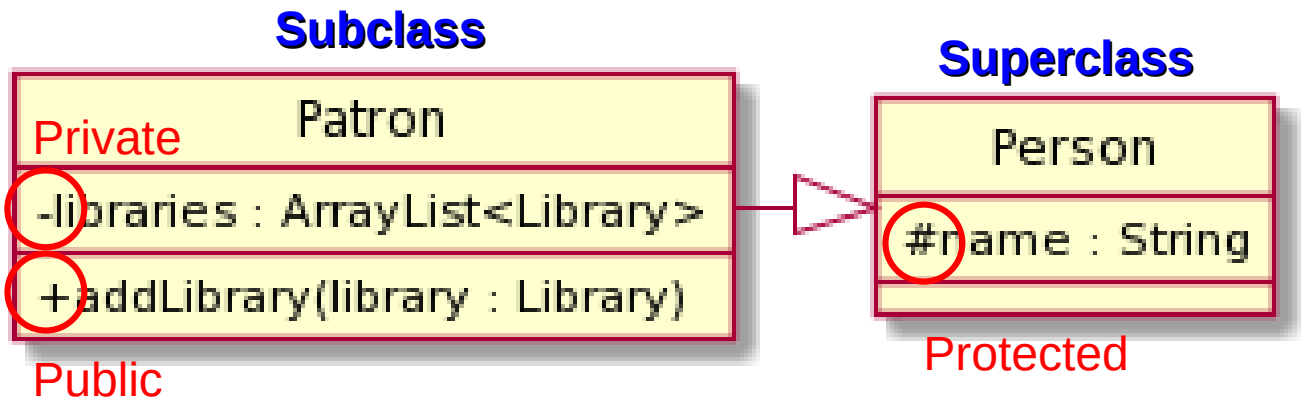
```
class Patron extends Person {
```

In the UML, an open-arrow line is drawn from the subclass to the superclass.

In Java, the superclass follows the class name separated by the keyword “extends”.

Protected Class Members

Often, the *sub*class needs to access members it inherited from the *super*class.
But making those inherited members public would enable them to also be accessed from main() and other methods.

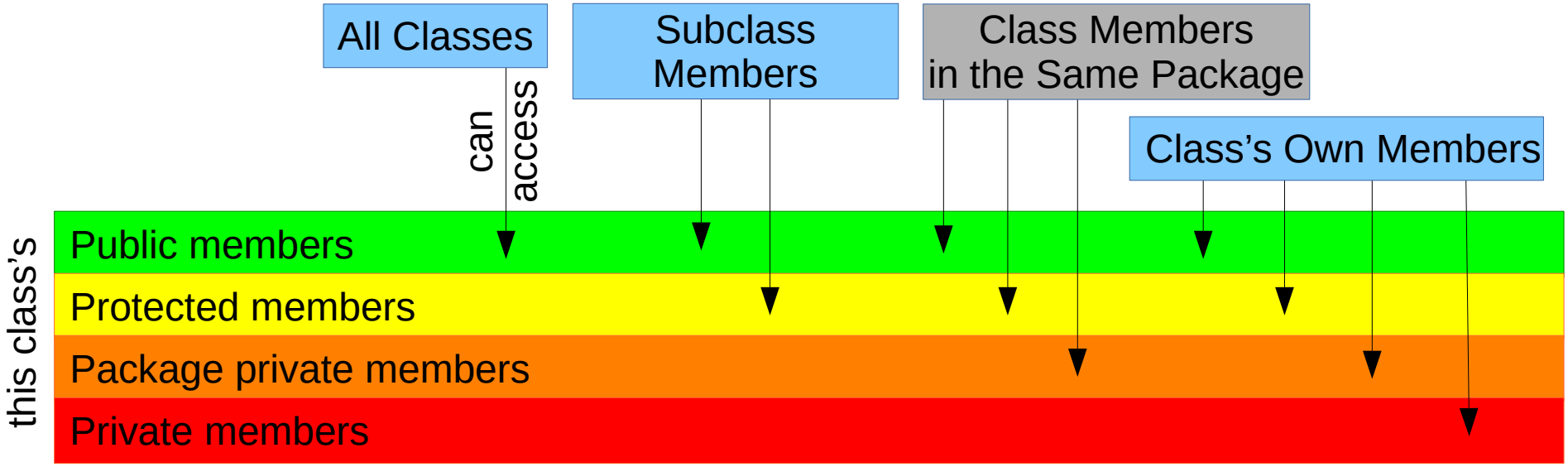


We need a middle ground - “only classes derived from me will have access to this member”. We call that middle ground “protected”, represented in the UML with ‘#’.

A protected member is accessible by subclasses, but is NOT accessible outside the class hierarchy*

* Java also gives package-private permissions to protected fields, which we’ll discuss later

Java Access Model

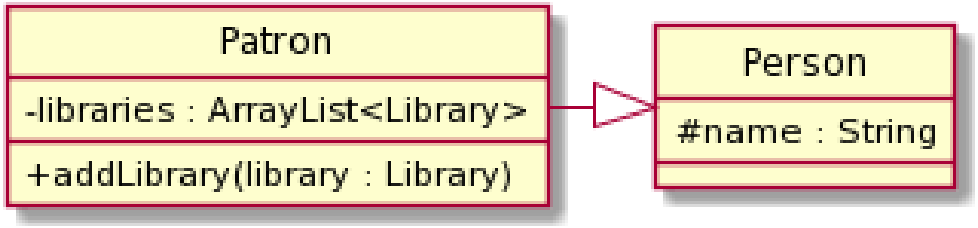


- A class or a class member (field, method, or class) can be
  - **Public** – **Anyone** can access this member
  - **Protected** – **Only class members and subclass members** can access this member*
  - **Package Private** (no modifier) **– Only class members within the same package** can access this member (we'll discuss later)
  - **Private** – **Only class members** can access this member

*Unlike most languages, Java also permits other classes in the same package to access protected members. Don't do this.

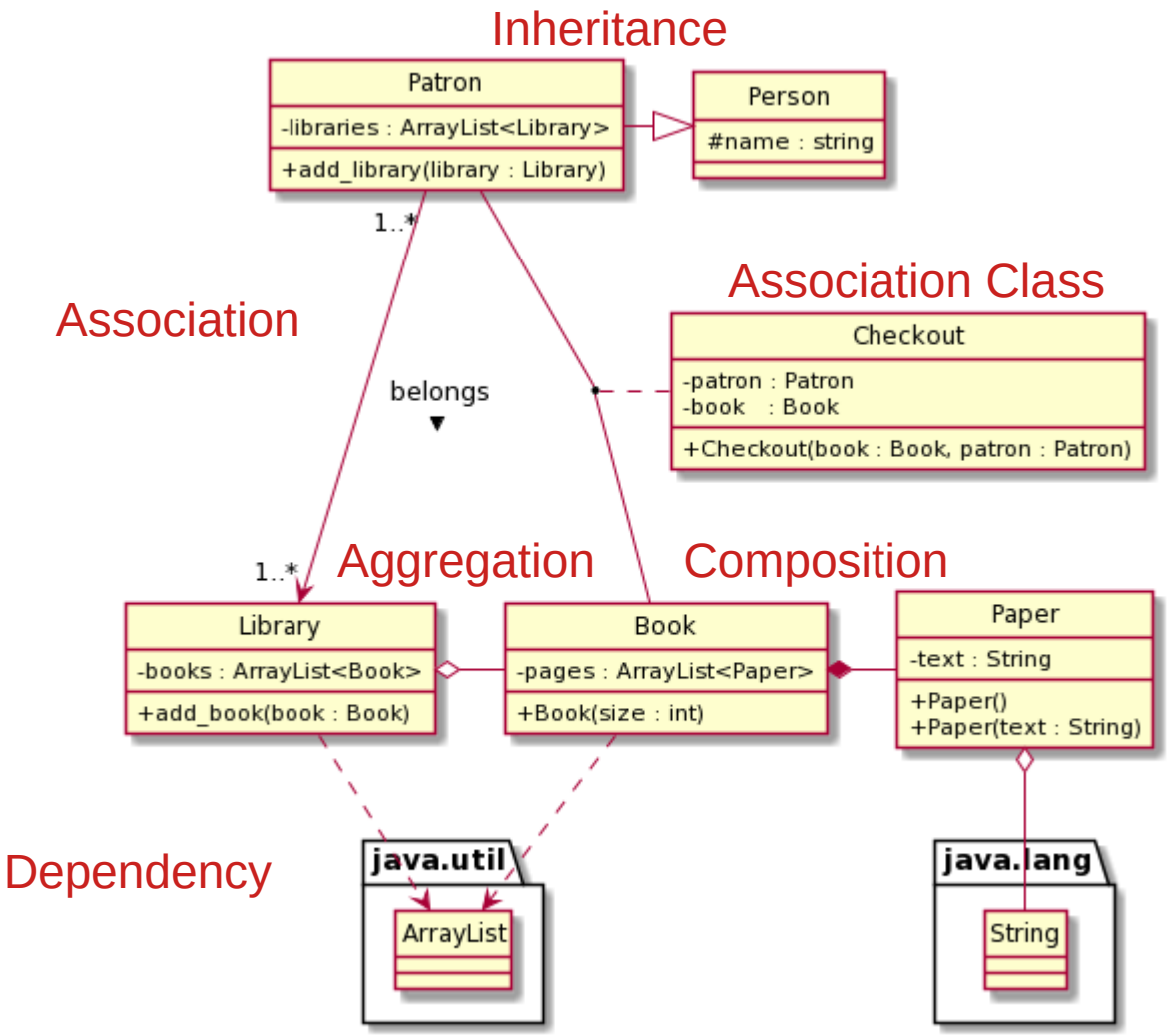
UML Class Relationships
Inheritance

- Inheritance is significantly more complex than the other relationships



- As a result, we'll focus solely on and dig much deeper into inheritance in the next lecture

UML Relationships Summary



This URL generates the above diagram:

http://www.plantuml.com/plantuml/uml/PP9Dxjem4CNtFiM8RiA7etQBgWZAgX8B90vGJH8hk77io7QgKihT6u-TDF3V63F-pPit3mxEe_L3fvXhfUxHOWULGkUEtbjP3bvyhuo-oZy2FhERh0LKqbPAC4OKd6LfqTxXmO13QKphO0z7Q_5-biv__JPM2W06My2w_X60B1mZ59xMx9c4w6jKwR4HhoeNx8MDCiHbBlZPWzIU54waS17M6HqqFe76BW3EMwcR_qCkw6r4E2XoBjz6fNqMdbMiPEbpf1EpC7-wYQerWm_Aj06DsQTvLAEj8UtD0BPpqpAGtNllOWYbGDwz-Er7uXgtWoszPElvzI9E2ZrMBJO2Vk8lp1NgXfn6tOXHuVc17RnK2tOE2Xu0KqDATVKbppShuBoOhkSvib8eeni7nGJplqNwmTX46Hwp2GTiUDoNPR5X5_ylaETpRX7CysBXQX9xBRCOdU5yokmWT3rqwXy0

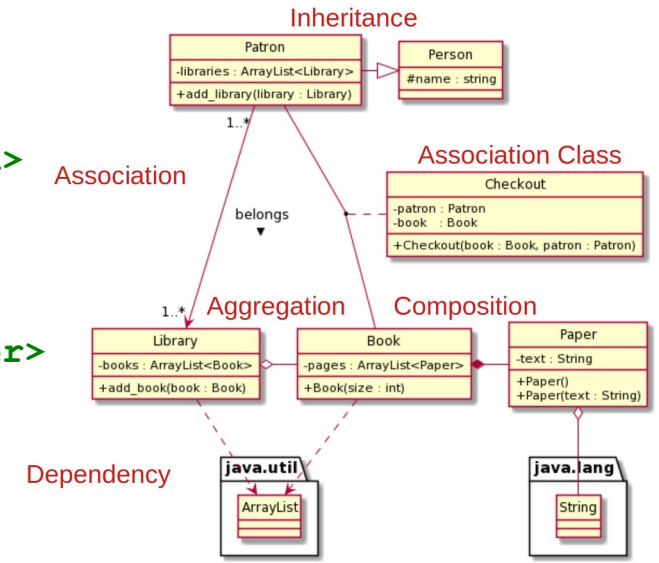
UML Relationship Summary

Relationship	UML	# per Class	Membership	Lifecycle	Directional	Java Implementation	C++ Implementation
Dependency	<--	Any	None	Independent	Uni- or Bi-	Import or local reference	Include or ::
Association	<--- or ---	Any	Either	Either	Uni- or Bi-	Attribute	Attribute, &, or *
Composition	◆---	One	Yes	Managed	Uni-	Attribute (internal instance)	Attribute
Aggregation	◇---	Any	None	Independent	Uni- or Bi-	Attribute (external instance)	& or *
Association Class	---	Any	None	Independent	3-Way	3rd Class: Attribute	3rd Class: & or * other 2
Inheritance	◁---	One	None	Related	Uni-	extends Superclass	: public Base
Interface	◁---	Any	None	Related	Uni-	implements Interface	: public Base1, public Base2

We'll discuss Inheritance and Interfaces next week!

For Interested Students, here's the PlantUML Code for
UML Relationships Class Diagram

```
@startuml
skinparam classAttributeIconSize 0
hide circle

‘ Classes

class Library {
 - books : ArrayList<Book>
 + addBook(book : Book)
}

class Book {
 - pages : ArrayList<Paper>
 + Book(size : int)
}

Class Person {
 # name : string
}

class Patron {
 - libraries : ArrayList<Library>
 + addLibrary(library : Library)
}

class Checkout {
  - patron : Patron
  - book   : Book
  + Checkout(book : Book, patron : Patron)
}
```




```
class Paper {
 -text : String
 +Paper()
 +Paper(text : String)
}

package java.lang {
  class String {
  }
}
package java.util {
  class ArrayList {
  }
}

‘ Relationships

Patron "1..*" --> "1..*"
        Library : belongs >
Patron -|> Person
Library o- Book
Book *- Paper
(Patron, Book) .. Checkout
Book ..> vector
Library ..> vector
Paper --o string

@enduml
```

http://plantuml.com/class-diagram
http://www.plantuml.com/plantuml/uml/

What We Learned Today

- String Operations
  - Useful String methods
  - String (immutable) vs StringBuilder (mutable)
- ArrayList vs arrays
- Using Java Documentation
- Class relationships and their Java and UML implementation
  - Association
  - Dependency
  - Aggregation
  - Composition
  - Inheritance