Object-Oriented Thinking and Introduction to Generics

Introduction to Programming and Computational Problem Solving: Accelerated Pace
CSE 11
Lecture 11
Announcements

• Assignments 2-4 upgrades due today, 11:59 PM

• Assignment 5 will be released today
  – Due May 15, 11:59 PM
Object-oriented thinking

• The advantages of object-oriented programming over procedural programming
• Classes provide more flexibility and modularity for building reusable software
• How to solve problems using the object-oriented paradigm
• Class design
Procedural programming vs object-oriented programming

• Procedural programming
  – Data and operations on data are separate
  – Requires passing data to methods

• Object-oriented programming
  – Data and operations on data are in an object
  – Organizes programs like the real world
    • All objects are associated with both attributes and activities
  – Using objects improves software reusability and makes programs easier to both develop and maintain
Procedural programming: method abstraction

• You can think of the method body as a black box that contains the detailed implementation for the method
Object-oriented programming: class abstraction and encapsulation

- **Class abstraction** means to separate class implementation from the use of the class
- The creator of the class provides a description of the class and lets the user know how the class can be used
  - The *class contract*
- The user of the class does not need to know how the class is implemented
- The detail of implementation is encapsulated and hidden from the user
  - *Class encapsulation*
  - A class is called an *abstract data type* (ADT)

Class implementation is like a black box hidden from the clients

Class Contract (Signatures of public methods and public constants)

Clients use the class through the contract of the class
Class abstraction and encapsulation

• For example, a class for a loan

<table>
<thead>
<tr>
<th>Loan</th>
<th>The annual interest rate of the loan (default: 2.5).</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The number of years for the loan (default: 1)</td>
</tr>
<tr>
<td></td>
<td>The loan amount (default: 1000).</td>
</tr>
<tr>
<td></td>
<td>The date this loan was created.</td>
</tr>
<tr>
<td>+Loan()</td>
<td>Constructs a default Loan object.</td>
</tr>
<tr>
<td>+Loan(annualInterestRate: double, numberOfYears: int, loanAmount: double)</td>
<td>Constructs a loan with specified interest rate, years, and loan amount.</td>
</tr>
<tr>
<td>+getAnnualInterestRate(): double</td>
<td>Returns the annual interest rate of this loan.</td>
</tr>
<tr>
<td>+getNumberOfYears(): int</td>
<td>Returns the number of the years of this loan.</td>
</tr>
<tr>
<td>+getLoanAmount(): double</td>
<td>Returns the amount of this loan.</td>
</tr>
<tr>
<td>+getLoanDate(): Date</td>
<td>Returns the date of the creation of this loan.</td>
</tr>
<tr>
<td>+setAnnualInterestRate(annualInterestRate: double): void</td>
<td>Sets a new annual interest rate to this loan.</td>
</tr>
<tr>
<td>+setNumberOfYears(numberOfYears: int): void</td>
<td>Sets a new number of years to this loan.</td>
</tr>
<tr>
<td>+setLoanAmount(loanAmount: double): void</td>
<td>Sets a new amount to this loan.</td>
</tr>
<tr>
<td>+getMonthlyPayment(): double</td>
<td>Returns the monthly payment of this loan.</td>
</tr>
<tr>
<td>+getTotalPayment(): double</td>
<td>Returns the total payment of this loan.</td>
</tr>
</tbody>
</table>

The creator of the class provides a description of the class and lets the user know how the class can be used.

The class contract
Class abstraction and encapsulation

- A class is designed for use by many different users (or customers or clients)
- To be useful in a wide range of applications, a class should provide a variety of ways for customization through properties, and constructors and methods that, together, are minimal and complete
Thinking in objects

• Procedural programming focuses on designing methods

• Object-oriented programming
  – Couples data and methods together into objects
  – Focuses on designing objects and operations on objects

• Object-orientated programming combines the power of procedural programming with an additional component that integrates data with operations into objects
Class relationships

• To design classes, one must understand the relationships among classes
  – Association
  – Aggregation
  – Composition
  – Inheritance (covered next lecture)
Association

• A general binary relationship that describes an activity between two classes

• For example
  – A student taking course is an association between the Student class and the Course class
  – A faculty member teaching a course is an association between the Faculty class and the Course class
Association

• Multiplicity
  – The number of objects of a class

• For example
  – Each student may take any number (*) of courses
  – Each course must have 5 to 60 students
  – Each course is taught by 1 faculty member
  – Each faculty member must teach 0 to 3 courses
Association

• In Java, associations can be implemented using data fields and methods
  – For example
    • A student takes a course
      addCourse method in Student class
      addStudent method in Course class
    • A faculty member teaches a course
      addCourse method in Faculty class
      setFaculty method in Course class
    • The Student class may store the courses a student is taking
      private Course[] courseList;
    • The Faculty class may store the courses a faculty member is teaching
      private Course[] courseList;

• There are many possible ways to implement association relationships
Aggregation

• Special form of association representing an owner-subject relationship
  – The owner object is called an aggregating object and its class is called an aggregating class
  – The subject object is called an aggregated object and its class is called an aggregated class

• Models has-a relationships
  – For example
    • A student has-a name
    • A student has-an address
Composition

• Aggregation between two objects is called *composition* if the existence of the aggregated object is dependent on the aggregating object
  – **Exclusive** ownership of the subject
  – The subject (i.e., aggregated object) cannot (conceptually) exist on its own
• For example
  – A book **has-a** page and when the book is destroyed, so is the page
  – A page has no meaning or purpose without the book
Aggregation and composition

• For example
  – When the student object is destroyed
    • Their name is destroyed (composition)
    • Their address is not destroyed (aggregation)

Each address is shared by up to 3 students
Aggregation and composition

• Usually represented as a data field in the aggregating class

public class Name {
  ...
}

public class Student {
  private Name name;
  private Address address;
  ...
}

public class Address {
  ...
}

Aggregated class  
Aggregating class  
Aggregated class
Aggregation between same class

- Aggregation may exist between objects of the same class
  - For example, a person may have a supervisor
    ```java
    public class Person {
        // The type for the data is the class itself
        private Person supervisor;
        ...
    }
    ```
  - For example, a person may have multiple supervisors
    ```java
    public class Person {
        // The type for the data is the class itself
        private Person[] supervisors;
        ...
    }
    ```
Aggregation or composition

• Warning: Since aggregation and composition relationships are represented using classes in similar ways, many texts do not differentiate them, calling both compositions
Class design and development

• For example, a class for a course

<table>
<thead>
<tr>
<th>Course</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-courseName: String</td>
<td>The name of the course.</td>
</tr>
<tr>
<td>-students: String[]</td>
<td>An array to store the students for the course.</td>
</tr>
<tr>
<td>-numberOfStudents: int</td>
<td>The number of students (default: 0).</td>
</tr>
<tr>
<td>+Course(courseName: String)</td>
<td>Creates a course with the specified name.</td>
</tr>
<tr>
<td>+getCourseName(): String</td>
<td>Returns the course name.</td>
</tr>
<tr>
<td>+addStudent(student: String): void</td>
<td>Adds a new student to the course.</td>
</tr>
<tr>
<td>+dropStudent(student: String): void</td>
<td>Drops a student from the course.</td>
</tr>
<tr>
<td>+getStudents(): String[]</td>
<td>Returns the students in the course.</td>
</tr>
<tr>
<td>+getNumberOfStudents(): int</td>
<td>Returns the number of students in the course.</td>
</tr>
</tbody>
</table>
public class TestCourse {
    public static void main(String[] args) {
        Course course1 = new Course("Data Structures");
        Course course2 = new Course("Database Systems");

        course1.addStudent("Peter Jones");
        course1.addStudent("Brian Smith");
        course1.addStudent("Anne Kennedy");

        course2.addStudent("Peter Jones");
        course2.addStudent("Steve Smith");

        System.out.println("Number of students in course1: ", course1.getNumberOfStudents());
        String[] students = course1.getStudents();
        for (int i = 0; i < course1.getNumberOfStudents(); i++)
            System.out.print(students[i] + ", ");

        System.out.println();
        System.out.println("Number of students in course2: ", course2.getNumberOfStudents());
    }
}
public class Course {
    private String courseName;
    private String[] students = new String[4];
    private int numberOfStudents;

    public Course(String courseName) {
        this.courseName = courseName;
    }

    public void addStudent(String student) {
        students[numberOfStudents] = student;
        numberOfStudents++;
    }

    public String[] getStudents() {
        return students;
    }

    public int getNumberOfStudents() {
        return numberOfStudents;
    }

    public String getCourseName() {
        return courseName;
    }

    public void dropStudent(String student) {
        // TODO
    }
}
Class design and development

• Use a UML class diagram to design the class
• Write a test program that uses the class
  – Developing a class and using a class are two separate tasks
  – It is easier to implement a class if you must use the class
• Implement the class
• Use Javadoc to document the class (contract)
Object-oriented thinking

• Classes provide more flexibility and modularity for building reusable software

• Class abstraction and encapsulation
  – Separate class implementation from the use of the class
  – The creator of the class provides a description of the class and let the user know how the class can be used
  – The user of the class does not need to know how the class is implemented
  – The detail of implementation is encapsulated and hidden from the user
Primitive data type values as objects

• A primitive data type is not an object
• But it can be wrapped in an object using a Java API wrapper class
  Boolean
  Character (**not** Char)
  Short
  Byte
  Integer (**not** Int)
  Long
  Float
  Double

Notes
• The wrapper classes do not have no-arg constructors
• The instances of all wrapper classes are immutable (i.e., their internal values cannot be changed once the objects are created)
## Integer and Double wrapper classes

<table>
<thead>
<tr>
<th>java.lang.Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>value: int</td>
</tr>
<tr>
<td>MAX_VALUE: int</td>
</tr>
<tr>
<td>MIN_VALUE: int</td>
</tr>
<tr>
<td>+Integer(value: int)</td>
</tr>
<tr>
<td>+Integer(s: String)</td>
</tr>
<tr>
<td>+byteValue(): byte</td>
</tr>
<tr>
<td>+shortValue(): short</td>
</tr>
<tr>
<td>+intValue(): int</td>
</tr>
<tr>
<td>+longValue(): long</td>
</tr>
<tr>
<td>+floatValue(): float</td>
</tr>
<tr>
<td>+doubleValue(): double</td>
</tr>
<tr>
<td>+compareTo(o: Integer): int</td>
</tr>
<tr>
<td>+toString(): String</td>
</tr>
<tr>
<td>+valueOf(s: String): Integer</td>
</tr>
<tr>
<td>+valueOf(s: String, radix: int): Integer</td>
</tr>
<tr>
<td>+parseInt(s: String): int</td>
</tr>
<tr>
<td>+parseInt(s: String, radix: int): int</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>java.lang.Double</th>
</tr>
</thead>
<tbody>
<tr>
<td>value: double</td>
</tr>
<tr>
<td>MAX_VALUE: double</td>
</tr>
<tr>
<td>MIN_VALUE: double</td>
</tr>
<tr>
<td>+Double(value: double)</td>
</tr>
<tr>
<td>+Double(s: String)</td>
</tr>
<tr>
<td>+byteValue(): byte</td>
</tr>
<tr>
<td>+shortValue(): short</td>
</tr>
<tr>
<td>+intValue(): int</td>
</tr>
<tr>
<td>+longValue(): long</td>
</tr>
<tr>
<td>+floatValue(): float</td>
</tr>
<tr>
<td>+doubleValue(): double</td>
</tr>
<tr>
<td>+compareTo(o: Double): int</td>
</tr>
<tr>
<td>+toString(): String</td>
</tr>
<tr>
<td>+valueOf(s: String): Double</td>
</tr>
<tr>
<td>+valueOf(s: String, radix: int): Double</td>
</tr>
<tr>
<td>+parseDouble(s: String): double</td>
</tr>
<tr>
<td>+parseDouble(s: String, radix: int): double</td>
</tr>
</tbody>
</table>
Wrapper classes

- Constructors
- Class Constants MAX_VALUE and MIN_VALUE
- Conversion Methods
Numeric wrapper class constructors

• You can construct a wrapper object either from a primitive data type value or from a string representing the numeric value
  – For example, the constructors for Integer and Double are
    
    ```java
    public Integer(int value)
    public Integer(String s)
    public Double(double value)
    public Double(String s)
    ```
Numeric wrapper class constants

- Each numerical wrapper class has the constants `MAX_VALUE` and `MIN_VALUE`
- `MAX_VALUE` represents the maximum value of the corresponding primitive data type
- For `Byte`, `Short`, `Integer`, and `Long`, `MIN_VALUE` represents the minimum byte, short, int, and long values
- For `Float` and `Double`, `MIN_VALUE` represents the minimum positive float and double values
Numeric wrapper class conversion methods

• Each numeric wrapper class implements the abstract methods `doubleValue`, `floatValue`, `intValue`, `longValue`, and `shortValue`
  – Defined in the abstract `Number` class (covered later in quarter)

• These methods “convert” objects into primitive type values
Numeric wrapper class
static valueOf methods

• The numeric wrapper classes have a useful class method valueOf(String s)
• This method creates a new object initialized to the value represented by the specified string
  – For example
    ```java
    Double doubleObject = Double.valueOf("12.4");
    Integer integerObject = Integer.valueOf("12");
    ```
Numeric wrapper class static parsing methods

• Each numeric wrapper class has two overloaded parsing methods to parse a numeric string into an appropriate numeric value based on 10 or any specified radix (e.g., 2 for binary, 8 for octal, 10 for decimal, 16 for hexadecimal)

  – For example

    Integer.parseInt("13") returns 13
    Integer.parseInt("13", 10) returns 13
    Integer.parseInt("1A", 16) returns 26
Automatic conversion between primitive types and wrapper class types

• Converting a primitive value to a wrapper object is called *boxing*

• Converting a wrapper object to a primitive value is called *unboxing*

• The Java compiler will automatically convert a primitive data type value to an object using a wrapper class (*autoboxing*) and vice versa (*autounboxing*), depending on the context
Automatic conversion between primitive types and wrapper class types

Integer[] intArray = {new Integer(2), new Integer(4), new Integer(3)};

Equivalent

Integer[] intArray = {2, 4, 3};

New JDK 1.5 boxing

Integer[] intArray = {1, 2, 3};
System.out.println(intArray[0] + intArray[1] + intArray[2]);

Autoboxing

Autounboxing
BigInteger and BigDecimal classes

• If you need to compute with very large integers or high precision floating-point values, you can use the BigInteger and BigDecimal classes in the java.math package

• Both are immutable

• Both extend the Number class and implement the Comparable interface (covered later in quarter)
BigInteger and BigDecimal classes

BigInteger a = new BigInteger("9223372036854775807");
BigInteger b = new BigInteger("2");
BigInteger c = a.multiply(b); // 9223372036854775807 * 2
System.out.println(c);

BigDecimal a = new BigDecimal(1.0);
BigDecimal b = new BigDecimal(3);
BigDecimal c = a.divide(b, 20, BigDecimal.ROUND_UP);
System.out.println(c);
String class

• The String class has 13 constructors and more than 40 methods
• A good example for learning classes and objects
Constructing strings

• Create from a string literal
  – Syntax
    ```java
    String newString = new String(stringLiteral);
    ```
  – Example
    ```java
    String message = new String("Welcome to Java");
    ```
  – Since strings are used frequently, Java provides a shorthand initializer for creating a string
    ```java
    String message = "Welcome to Java";
    ```

• Create from an array of characters
  – Syntax
    ```java
    String newString = new String(charArray);
    ```
  – where, for example
    ```java
    char[] charArray = {'U', 'C', 'S', 'D', ' ', 'C', 'S', 'E'};
    ```
Strings are immutable

- A String object is immutable (i.e., its contents cannot be changed once the string is created)
- The following code does not change the contents of the string

```java
String s = "Java";
s = "HTML";
```
Strings are immutable

```
String s = "Java";
s = "HTML";
```

![Diagram showing the immutability of strings](image)

After executing `String s = "Java";`

- `s` points to a `String` object for "Java"
- Contents cannot be changed

After executing `s = "HTML";`

- `s` points to a `String` object for "HTML"
- This string object is now unreferenced
Strings are immutable

String s = "Java";

s = "HTML";

After executing String s = "Java";

After executing s = "HTML";

This string object is now unreferenced

Contents cannot be changed
Interned strings

• Since strings are immutable and are frequently used, to improve efficiency and save memory, the Java Virtual Machine (JVM) uses a unique instance for string literals with the same character sequence

• Such an instance is called *interned*
Interned strings

• A new object is created if you use the `new` operator

• If you use the string initializer, no new object is created if the interned object is already created

```java
String s1 = "Welcome to Java";
String s2 = new String("Welcome to Java");
String s3 = "Welcome to Java";
System.out.println("s1 == s2 is " + (s1 == s2));
System.out.println("s1 == s3 is " + (s1 == s3));
```

`s1 == s2` is false
`s1 == s3` is true
## Replacing and splitting strings

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>replace(oldChar: char, newChar: char): String</code></td>
<td>Returns a new string that replaces all matching character in this string with the new character.</td>
</tr>
<tr>
<td><code>replaceFirst(oldString: String, newString: String): String</code></td>
<td>Returns a new string that replaces the first matching substring in this string with the new substring.</td>
</tr>
<tr>
<td><code>replaceAll(oldString: String, newString: String): String</code></td>
<td>Returns a new string that replace all matching substrings in this string with the new substring.</td>
</tr>
<tr>
<td><code>split(delimiter: String): String[]</code></td>
<td>Returns an array of strings consisting of the substrings split by the delimiter.</td>
</tr>
</tbody>
</table>
Replacing a string

• "Welcome".replace('e', 'A') returns a new string WALcomA
• "Welcome".replaceFirst("e", "AB") returns a new string WAB1come
• "Welcome".replace("e", "AB") returns a new string WAB1comAB
• "Welcome".replace("el", "AB") returns a new string WABcome
Splitting a string

• Split a string into an array of strings
  – For example, using # as a delimiter
    ```java
    String[] tokens = "CSE#8B#and#11#use#Java".split("#", 0);
    for (int i = 0; i < tokens.length; i++)
        System.out.print(tokens[i] + " ");
    ```
  – Displays CSE 8B and 11 use Java
Matching, replacing, and splitting by patterns

• You can match, replace, or split a string by specifying a pattern
  – For example
    "Java".equals("Java");
    "Java".matches("Java");

• This is an extremely useful and powerful feature known as *regular expression*
  – https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html#sum
  – https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/regex/Pattern.html#sum
Convert character and numbers to strings

• The String class provides several static `valueOf` methods for converting a character, an array of characters, and numeric values to strings

• These methods have the same name `valueOf` with different argument types `char`, `char[]`, `double`, `long`, `int`, and `float`
  – For example, to convert a `double` value to a string, use `String.valueOf(5.44)`
    • The return value is string consists of characters `'5'`, `'.'`, `'4'`, and `'4'`
    • Compare with `String s = 5.44 + "";`
StringBuilder and StringBuffer classes

• The StringBuilder and StringBuffer classes are alternatives to the String class
• In general, a StringBuilder or StringBuffer can be used wherever a string is used
• StringBuilder and StringBuffer are more flexible than String
• You can add, insert, or append new contents into a string buffer, whereas the value of a String object is fixed once the string is created
## StringBuilder constructors

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>StringBuilder()</code></td>
<td>Constructs an empty string builder with capacity 16.</td>
</tr>
<tr>
<td><code>StringBuilder(capacity: int)</code></td>
<td>Constructs a string builder with the specified capacity.</td>
</tr>
<tr>
<td><code>StringBuilder(s: String)</code></td>
<td>Constructs a string builder with the specified string.</td>
</tr>
</tbody>
</table>
### Modifying strings in the builder

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>append(data: char[])</code>: StringBuilder</td>
<td>Appends a char array into this string builder.</td>
</tr>
<tr>
<td><code>append(data: char[], offset: int, len: int): StringBuilder</code></td>
<td>Appends a subarray in data into this string builder.</td>
</tr>
<tr>
<td><code>append(v: aPrimitiveType)</code>: StringBuilder</td>
<td>Appends a primitive type value as a string to this builder.</td>
</tr>
<tr>
<td><code>append(s: String)</code>: StringBuilder</td>
<td>Appends a string to this string builder.</td>
</tr>
<tr>
<td><code>delete(startIndex: int, endIndex: int)</code>: StringBuilder</td>
<td>Deletes characters from startIndex to endIndex.</td>
</tr>
<tr>
<td><code>deleteCharAt(index: int)</code>: StringBuilder</td>
<td>Deletes a character at the specified index.</td>
</tr>
<tr>
<td><code>insert(index: int, data: char[], offset: int, len: int): StringBuilder</code></td>
<td>Inserts a subarray of the data in the array to the builder at the specified index.</td>
</tr>
<tr>
<td><code>insert(offset: int, data: char[])</code>: StringBuilder</td>
<td>Inserts data into this builder at the position offset.</td>
</tr>
<tr>
<td><code>insert(offset: int, b: aPrimitiveType)</code>: StringBuilder</td>
<td>Inserts a value converted to a string into this builder.</td>
</tr>
<tr>
<td><code>insert(offset: int, s: String)</code>: StringBuilder</td>
<td>Inserts a string into this builder at the position offset.</td>
</tr>
<tr>
<td><code>replace(startIndex: int, endIndex: int, s: String)</code>: StringBuilder</td>
<td>Replaces the characters in this builder from startIndex to endIndex with the specified string.</td>
</tr>
<tr>
<td><code>reverse()</code>: StringBuilder</td>
<td>Reverses the characters in the builder.</td>
</tr>
<tr>
<td><code>setCharAt(index: int, ch: char)</code>: void</td>
<td>Sets a new character at the specified index in this builder.</td>
</tr>
</tbody>
</table>
### java.lang.StringBuilder

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+toString(): String</code></td>
<td>Returns a string object from the string builder.</td>
</tr>
<tr>
<td><code>+capacity(): int</code></td>
<td>Returns the capacity of this string builder.</td>
</tr>
<tr>
<td><code>+charAt(index: int): char</code></td>
<td>Returns the character at the specified index.</td>
</tr>
<tr>
<td><code>+length(): int</code></td>
<td>Returns the number of characters in this builder.</td>
</tr>
<tr>
<td><code>+setLength(newLength: int): void</code></td>
<td>Sets a new length in this builder.</td>
</tr>
<tr>
<td><code>+substring(startIndex: int): String</code></td>
<td>Returns a substring starting at startIndex.</td>
</tr>
<tr>
<td><code>+substring(startIndex: int, endIndex: int): String</code></td>
<td>Returns a substring from startIndex to endIndex-1.</td>
</tr>
<tr>
<td><code>+trimToSize(): void</code></td>
<td>Reduces the storage size used for the string builder.</td>
</tr>
</tbody>
</table>
The ArrayList class

• You can create an array to store objects, but the array’s size is fixed once the array is created
• Java provides the ArrayList class that can be used to store an unlimited number of objects
The ArrayList class

```
java.util.ArrayList<E>

+ArrayList()  
+add(o: E) : void  
+add(index: int, o: E) : void 
+clear(): void  
+contains(o: Object): boolean  
+get(index: int) : E  
+indexOf(o: Object) : int  
+isEmpty(): boolean  
+lastIndexOf(o: Object) : int  
+remove(o: Object): boolean  
+size(): int  
+remove(index: int) : boolean  
+set(index: int, o: E) : E
```

Creates an empty list.
Appends a new element o at the end of this list.
Adds a new element o at the specified index in this list.
Removes all the elements from this list.
Returns true if this list contains the element o.
Returns the element from this list at the specified index.
Returns the index of the first matching element in this list.
Returns true if this list contains no elements.
Returns the index of the last matching element in this list.
Removes the element o from this list.
Returns the number of elements in this list.
Removes the element at the specified index.
Sets the element at the specified index.
The ArrayList class

• ArrayList is known as a generic class with a generic type E
• You can specify a concrete type to replace E when creating an ArrayList
• For example
  – The below statements creates an ArrayList used to store strings and assigns its reference to variable cities
    ```java
    ArrayList<String> cities = new ArrayList<String>();
    ArrayList<String> cities = new ArrayList<>();
    ```
# Comparing arrays and ArrayList

<table>
<thead>
<tr>
<th>Operation</th>
<th>Array</th>
<th>ArrayList</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating an array/ArrayList</td>
<td>String[] a = new String[10]</td>
<td>ArrayList&lt;String&gt; list = new ArrayList&lt;&gt;();</td>
</tr>
<tr>
<td>Accessing an element</td>
<td>a[index]</td>
<td>list.get(index);</td>
</tr>
<tr>
<td>Updating an element</td>
<td>a[index] = &quot;London&quot;;</td>
<td>list.set(index, &quot;London&quot;);</td>
</tr>
<tr>
<td>Returning size</td>
<td>a.length</td>
<td>list.size();</td>
</tr>
<tr>
<td>Adding a new element</td>
<td></td>
<td>list.add(&quot;London&quot;);</td>
</tr>
<tr>
<td>Inserting a new element</td>
<td></td>
<td>list.add(index, &quot;London&quot;);</td>
</tr>
<tr>
<td>Removing an element</td>
<td></td>
<td>list.remove(index);</td>
</tr>
<tr>
<td>Removing an element</td>
<td></td>
<td>list.remove(Object);</td>
</tr>
<tr>
<td>Removing all elements</td>
<td></td>
<td>list.clear();</td>
</tr>
</tbody>
</table>
Array to/from ArrayList

• Creating an ArrayList from an array of objects

```java
String[] array = {"red", "green", "blue"};
ArrayList<String> list = new ArrayList<>(Arrays.asList(array));
```

• Creating an array of objects from an ArrayList

```java
String[] array1 = new String[list.size()];
list.toArray(array1);
```
Useful methods in `java.util.Collections`

- Maximum element in `ArrayList`
  `java.util.Collections.max`
- Minimum element in `ArrayList`
  `java.util.Collections.min`
- Sort an `ArrayList`
  `java.util.Collections.sort`
- Shuffle an `ArrayList`
  `java.util.Collections.shuffle`
Next Lecture

• Inheritance