INHERITANCE AND POLYMORPHISM

Inheritance

- *Object-oriented programming allows you to define new classes from existing classes. This is called inheritance.*
- *Inheritance enables you to define a general class (i.e., a superclass) and later extend it to more specialized classes (i.e., subclasses)*
- You use a class to model objects of the same type.
- Different classes may have some common properties and behaviors, which can be generalized in a class that can be shared by other classes.
- You can define a specialized class that extends the generalized class.
- The specialized classes inherit the properties and methods from the general class.

Superclass and subclass

- a class extended from another class is called a *subclass*, and main class is called a *superclass*
- A superclass is also referred to as a *parent class* or a *base class* and a subclass as a *child class*, an *extended class*, or a *derived class*
- A subclass inherits accessible data fields and methods from its superclass and may also add new data fields and methods
  - **public class** Circles **extends** SimpleGeometricObject
- keyword **extends** tells the compiler that the **Circle** class extends the **GeometricObject** class
- a subclass is not a subset of its superclass, a subclass usually contains more information and methods than its superclass

Superclass and subclass (Contd…)

- Private data fields in a superclass are not accessible outside the class. Therefore, they cannot be used directly in a subclass
- Inheritance is used to model the is-a relationship. A subclass and its superclass must have the is-a relationship
- Some programming languages allow you to derive a subclass from several classes. This capability is known as *multiple inheritance*
- Java does not allow multiple inheritance.

Using the **super** Keyword

- *keyword* super *refers to the superclass and can be used to invoke the superclass's methods and constructors*
- subclass inherits accessible data fields and methods from its superclass

Calling Superclass Constructors

- Unlike properties and methods, the constructors of a superclass are not inherited by a subclass
- They can only be invoked from the constructors of the subclasses using the keyword **super**
- syntax to call a superclass's constructor is:
  - **super**(), or **super**(parameters);
- statement **super()** invokes the no-arg constructor of its superclass
- Statement **super(arguments)** invokes the superclass constructor that matches the **arguments**
- statement **super()** or **super(arguments)** must be the first statement of the subclass's constructor
  - this is the only way to explicitly invoke a superclass constructor

Constructor Chaining

- constructor may invoke an overloaded constructor or its superclass constructor
- If neither is invoked explicitly, the compiler automatically puts **super()** as the first statement in the constructor
- constructing an instance of a class invokes the constructors of all the superclasses along the inheritance chain
- When constructing an object of a subclass, the subclass constructor first invokes its superclass constructor before performing its own tasks
- If the superclass is derived from another class, the superclass constructor invokes its parent-class constructor before performing its own tasks.
- This process continues until the last constructor along the inheritance hierarchy is called.
- This is called *constructor chaining*

Calling Superclass Methods

- keyword **super** can also be used to reference a method other than the constructor in the superclass
  - super.method(parameters);

Overriding Methods

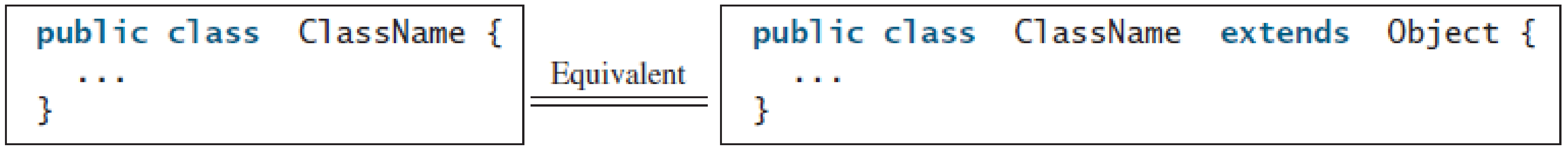
- *To override a method, the method must be defined in the subclass using the same signature and the same return type as in its superclass*
- Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass
- This is referred to as *method overriding*
- An instance method can be overridden only if it is accessible, thus a private method cannot be overridden
- If a method defined in a subclass is private in its superclass, the two methods are completely unrelated
- Like an instance method, a static method can be inherited
- a static method cannot be overridden
- If a static method defined in the superclass is redefined in a subclass, the method defined in the superclass is hidden
- hidden static methods can be invoked using the syntax **SuperClassName.staticMethodName**

Overriding vs. Overloading

- *Overloading means to define multiple methods with the same name but different signatures*
- *Overriding means to provide a new implementation for a method in the subclass*
- To override a method, the method must be defined in the subclass using the same signature and the same return type
- Overridden methods are in different classes related by inheritance
- overloaded methods can be either in the same class or different classes related by inheritance
- Overridden methods have the same signature and return type
- overloaded methods have the same name but a different parameter list
- To avoid mistakes, you can use a special Java syntax, called *override annotation*, to place **@Override** before the method in the subclass

Object Class and Its toString() Method

- *Every class in Java is descended from the* **java.lang.Object** *class*
- If no inheritance is specified when a class is defined, the superclass of the class is **Object** by default
- Following two definitions are same:



```
public class  ClassName {
  ...
}
```

Equivalent

```
public class  ClassName  extends  Object {
  ...
}
```

Object Class and Its **toString()** Method (Contd…)

- signature of the **toString()** method is:
  - **public** String toString()
  - Invoking **toString()** on an object returns a string that describes the object.
- By default, it returns a string consisting of a class name of which the object is an instance, an at sign (@), and the object's memory address in hexadecimal

Polymorphism

- *Polymorphism means that a variable of a supertype can refer to a subtype object*
- A class defines a type.
- A type defined by a subclass is called a *subtype*
- A type defined by its superclass is called a *supertype*
- inheritance relationship enables a subclass to inherit features from its superclass with additional new features
- subclass is a specialization of its superclass
- every instance of a subclass is also an instance of its superclass, but not vice versa

Polymorphism (contd…)

- An object of a subclass can be used wherever its superclass object is used.
- This is commonly known as *polymorphism*
- In simple terms, polymorphism means that a variable of a supertype can refer to a subtype object.

Dynamic Binding

- *A method can be implemented in several classes along the inheritance chain*
- *JVM decides which method is invoked at runtime*
- A method can be defined in a superclass and overridden in its subclass
- The type that declares a variable is called the variable's *declared type*
- A variable of a reference type can hold a **null** value or a reference to an instance of the declared type
- instance may be created using the constructor of the declared type or its subtype
- *actual type* of the variable is the actual class for the object referenced by the variable
- Which method is invoked by object is determined by object's actual type. This is known as *dynamic binding*

Dynamic Binding (Contd…)

- Suppose an object **o** is an instance of classes **C1**, **C2**, . . . ,**Cn-1**, and **Cn**, where **C1** is a subclass of **C2**, **C2** is a subclass of **C3**, . . . , and **Cn-1** is a subclass of **Cn**,
- That is, **Cn** is the most general class, and **C1** is the most specific class.
- **Cn** is the **Object** class. If **o** invokes a method **p**, the JVM searches for the implementation of the method **p** in **C1**, **C2**, . . . , **Cn-1**, and **Cn**, in this order, until it is found.
- Once an implementation is found, the search stops and the first-found implementation is invoked.

Dynamic Binding (Contd…)

- Matching a method signature and binding a method implementation are two separate issues.
- *declared type* of the reference variable decides which method to match at compile time
- compiler finds a matching method according to the parameter type, number of parameters, and order of the parameters at compile time
- method may be implemented in several classes along the inheritance chain
- JVM dynamically binds the implementation of the method at runtime, decided by the actual type of the variable

Casting Objects and the **instanceof** Operator

- *One object reference can be typecast into another object reference. This is called casting object*
  - Object o = **new** Student(); // Implicit casting
  - m(o);
- statement **Object o = new Student()**, known as *implicit casting*, is legal because an instance of **Student** is an instance of **Object**
- Suppose you want to assign the object reference **o** to a variable of the **Student** type using the following statement:
  - Student b = o;
- In this case a compile error would occur
- **Student** object is always an instance of **Object**, but an **Object** is not necessarily an instance of **Student**

- To tell the compiler that **o** is a **Student** object, use *explicit casting*
  - Student b = (Student)o;
- It is possible to cast an instance of a subclass to a variable of a superclass (known as *upcasting*), because an instance of a subclass is *always* an instance of its superclass
- When casting an instance of a superclass to a variable of its subclass (known as *downcasting*), explicit casting must be used

The Object's equals Method

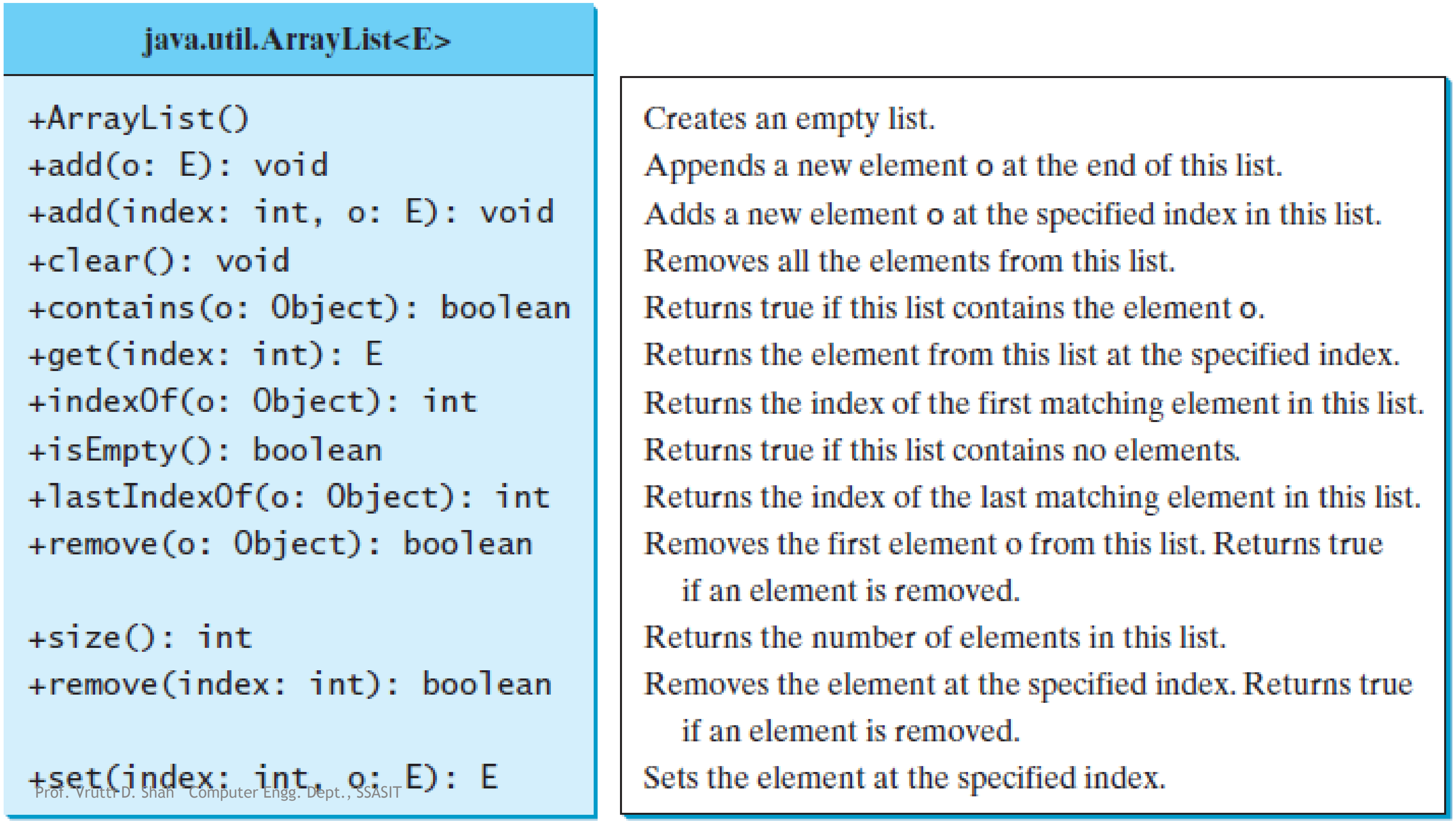
- signature is
  - **public boolean** equals(Object o)
- method tests whether two objects are equal
  - object1.equals(object2);
- **public boolean** equals(Object obj) {

  return (**this** == obj);

  }
- implementation checks whether two reference variables point to the same object using the == operator
- You should override this method in your custom class to test whether two distinct objects have the same content

The ArrayList Class

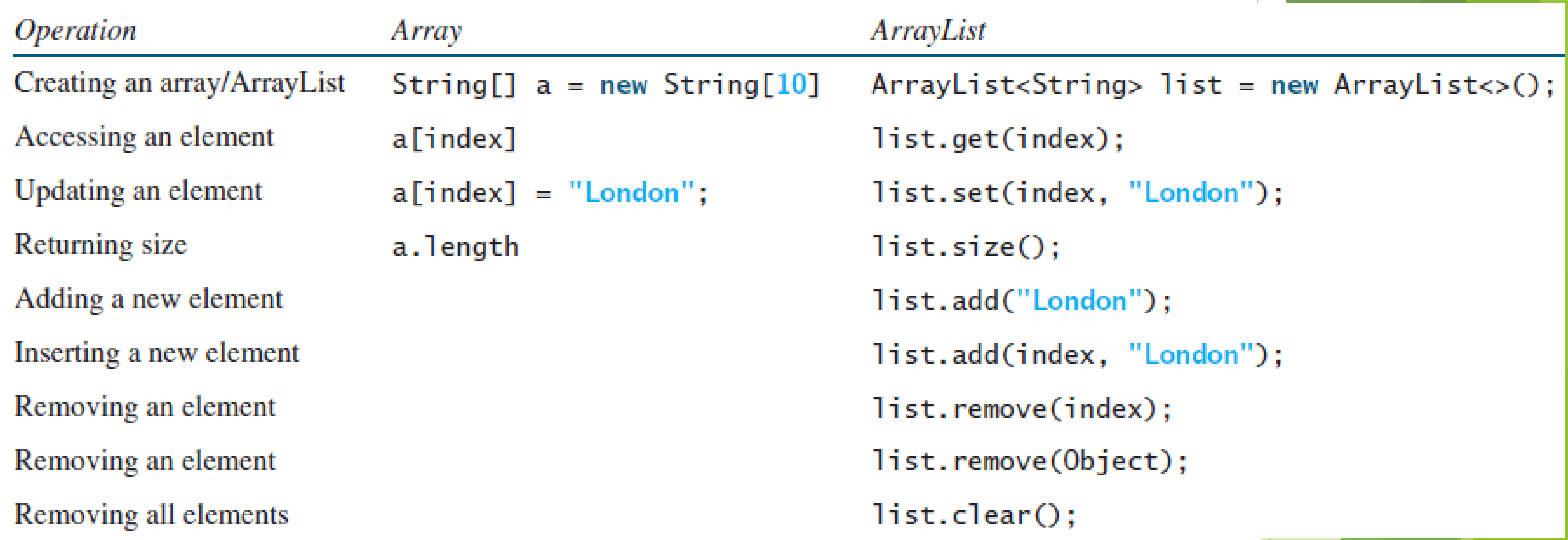
- ArrayList *object can be used to store a list of objects.*
- You can create an array to store objects.
- once the array is created, its size is fixed



java.util.ArrayList<E>	
+ArrayList()	Creates an empty list.
+add(o: E): void	Appends a new element o at the end of this list.
+add(index: int, o: E): void	Adds a new element o at the specified index in this list.
+clear(): void	Removes all the elements from this list.
+contains(o: Object): boolean	Returns true if this list contains the element o.
+get(index: int): E	Returns the element from this list at the specified index.
+indexOf(o: Object): int	Returns the index of the first matching element in this list.
+isEmpty(): boolean	Returns true if this list contains no elements.
+lastIndexOf(o: Object): int	Returns the index of the last matching element in this list.
+remove(o: Object): boolean	Removes the first element o from this list. Returns true if an element is removed.
+size(): int	Returns the number of elements in this list.
+remove(index: int): boolean	Removes the element at the specified index. Returns true if an element is removed.
+set(index: int, o: E): E	Sets the element at the specified index.

The ArrayList Class (Contd…)

- ArrayList is known as a generic class with a generic type E
- following statement creates an **ArrayList** and assigns its reference to variable **cities** which stores strings.
  - ArrayList<String> cities = **new** ArrayList<String>();
- following statement creates an **ArrayList** and assigns its reference to variable **dates**
  - ArrayList<java.util.Date> dates = **new** ArrayList<java.util.Date> ();



Operation	*Array*	*ArrayList*
Creating an array/ArrayList	`String[] a = new String[10]`	`ArrayList<String> list = new ArrayList<>();`
Accessing an element	`a[index]`	`list.get(index);`
Updating an element	`a[index] = "London";`	`list.set(index, "London");`
Returning size	`a.length`	`list.size();`
Adding a new element		`list.add("London");`
Inserting a new element		`list.add(index, "London");`
Removing an element		`list.remove(index);`
Removing an element		`list.remove(Object);`
Removing all elements		`list.clear();`

The ArrayList Class (Contd…)

- The following will not work because the elements stored in an **ArrayList** must be of an object type
  - ArrayList<**int**> list = **new** ArrayList<>();
- Instead of the above statement, the following will work.
  - ArrayList<Integer> list = **new** ArrayList<>();
- size of an **ArrayList** is flexible so you don't have to specify its size in advance. When creating an array, its size must be specified.
- **ArrayList** contains many useful methods, if you use an array, you have to write additional code to implement this method
- elements in an array list can also be traversed using a foreach loop using the following syntax:
- **for** (elementType element: arrayList) {

```
  // Process the element
  }
```

Useful Methods for Lists

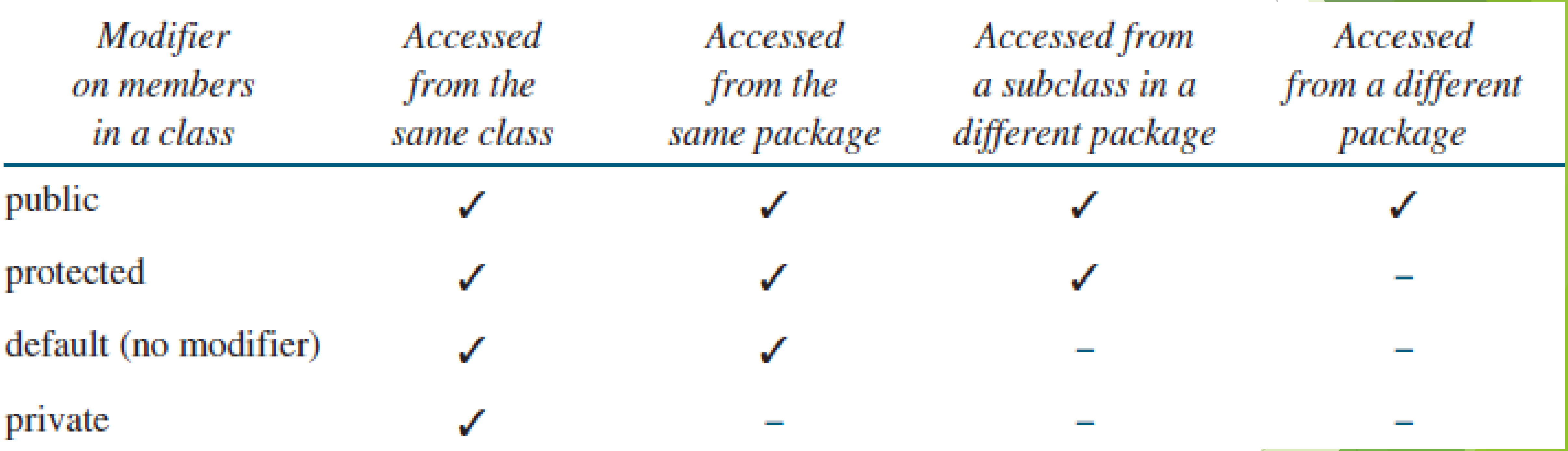
- *Java provides the methods for creating a list from an array, for sorting a list, and finding maximum and minimum element in a list, and for shuffling a list*
- to create an array list from an array:
- String[] array = {"**red**", "**green**", "**blue**"};

  ArrayList<String> list = **new** ArrayList<>(Arrays.asList(array));
- Conversely, following code to create an array of objects from an array list.
- String[] array1 = **new** String[list.size()];

  list.toArray(array1);

The **protected** Data and Methods

- *protected member of a class can be accessed from a subclass*
- Private members can be accessed only from inside of the class, and public members can be accessed from any other classes
- Using **protected** keyword, you can access protected data fields or methods in a superclass from its subclasses
- modifiers **private, protected**, and **public** are known as *visibility* or *accessibility modifiers*
- **private** modifier to hide the members of the class completely
- Use no modifiers (the default) in order to allow the members of the class to be accessed directly from any class within the same package but not from other packages.



Modifier on members in a class	*Accessed from the same class*	*Accessed from the same package*	*Accessed from a subclass in a different package*	*Accessed from a different package*
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default (no modifier)	✓	✓	–	–
private	✓	–	–	–

The **protected** Data and Methods (Contd…)

- class can be used in two ways: (1) for creating instances of the class and (2) for defining subclasses by extending the class
- Make the members private if they are not intended for use from outside the class.
- Make the members public if they are intended for the users of the class.
- Make the fields or methods protected if they are intended for the extenders of the class but not for the users of the class.
- **private** and **protected** modifiers can be used only for members of the class
- **public** modifier and the default modifier (i.e., no modifier) can be used on members of the class as well as on the class
- A class with no modifier (i.e., not a public class) is not accessible by classes from other packages

Preventing Extending and Overriding

- *Neither a final class nor a final method can be extended. A final data field is a constant.*
- You may want to prevent classes from being extended. In such cases, use the **final** modifier to indicate that a class is final and cannot be a parent class
- The **Math, String, StringBuilder**, and **StringBuffer** classes are final classes
- You can define a method to be final; a final method cannot be overridden by its subclasses.