Software Components

Generics

Objective

Allowing operations not be to tied to a specific data type

Outline

- There are programming solutions that are applicable to a wide range of different data types
  - The code is exactly the same other than the data type declarations
- In C, there is no easy way to exploit the similarity:
  - You need a separate implementation for each data type
- In Java, you can make use of **generic programming**:
  - A mechanism to specify solution without tying it down to a specific data type

1.2 Eg: The IntPair Class (non-generic)

- Let's define a class to:
  - Store a pair of integers, e.g. `(74, -123)`
  - Many usages, can represent 2D coordinates, range (min to max), height and weight, etc.

IntPair.java

```java
class IntPair {

  private int first;
  private int second;

  public IntPair(int a, int b) {
      first = a;
      second = b;
  }

  public int getFirst() {
      return first;
  }

  public int getSecond() {
      return second;
  }
}
```

1.2 Using the IntPair Class (non-generic)

TestIntPair.java

```java
// This program uses the IntPair class to create an object
// containing the lower and upper limits of a range.
// We then use it to check that the input data fall within that range.
import java.util.Scanner;

public class TestIntPair {
  public static void main(String[] args) {
      IntPair range = new IntPair(-5, 20);
      Scanner sc = new Scanner(System.in);

      int input;
      do {
          System.out.printf("Enter a number in (%d to %d): ",
                            range.getFirst(), range.getSecond());

          input = sc.nextInt();
      } while( (input < range.getFirst()) || (input > range.getSecond()) );
  }
}
```

```
Enter a number in (-5 to 20): -10
Enter a number in (-5 to 20): 21
Enter a number in (-5 to 20): 12
```

1.2 Observation

- The **`IntPair`** class idea can be easily extended to other data types:
  - **`double`**, **`String`**, etc.
- The resultant code would be almost the same!

IntPair.java

```java
class StringPair {

  private String first;
  private String second;

  public StringPair( String a, String b ) {
      first = a;
      second = b;
  }

  public String getFirst() {
      return first;
  }

  public String getSecond() {
      return second;
  }
}
```

Only differences are the data type declarations

1.3 The Generic Pair Class

Pair.java

```
class Pair<T> {

  private T first;
  private T second;

  public Pair(T a, T b) {
      first = a;
      second = b;
  }

  public T getFirst() {
      return first;
  }

  public T getSecond() {
      return second;
  }
}
```

- Important restriction:
  - The generic type can be substituted by reference data type only
  - Hence, primitive data types are NOT allowed
  - Need to use wrapper class for primitive data type

1.3 Using the Generic Pair Class

TestGenericPair.java

```java
public class TestGenericPair {
  public static void main(String[] args) {
      Pair<Integer> twoInt = new Pair<Integer>(-5, 20);
      Pair<String> twoStr = new Pair<String>("Turing", "Alan");

      // You can have pair of any reference data types!
      // Print out the integer pair
      System.out.println("Integer pair: (" + twoInt.getFirst()
                         + ", " + twoInt.getSecond() + ")");
      // Print out the String pair
      System.out.println("String pair: (" + twoStr.getFirst()
                          + ", " + twoStr.getSecond() + ")");
  }
}
```

- The formal generic type **`<T>`** is substituted with the actual data type supplied by the user:
  - The effect is similar to generating a new version of the **`Pair`** class, where **`T`** is substituted

1.4 Autoboxing/unboxing (1/2)

- The following statement invokes autoboxing

```
Pair<Integer> twoInt = new Pair<Integer>(-5, 20);
```

- **Integer** objects are expected for the constructor, but -5 and 20, of primitive type **int**, are accepted.
- ***Autoboxing*** is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes
  - The primitive values -5 and 20 are converted to objects of **Integer**
- The Java compiler applies autoboxing when a primitive value is:
  - Passed as a parameter to a method that expects an object of the corresponding wrapper class
  - Assigned to a variable of the correspond wrapper class

1.4 Autoboxing/unboxing (2/2)

- Converting an object of a wrapper type (e.g.: **Integer**) to its corresponding primitive (e.g: **int**) value is called unboxing.
- The Java compiler applies unboxing when an object of a wrapper class is:
  - Passed as a parameter to a method that expects a value of the corresponding primitive type
  - Assigned to a variable of the corresponding primitive type

```
int i = new Integer(5);   // unboxing
Integer intObj = 7;       // autoboxing
System.out.println("i = " + i);
System.out.println("intObj = " + intObj);
```

```
i = 5
intObj = 7
```

```
int a = 10;
Integer b = 10;      // autoboxing
System.out.println(a == b);
```

```
true
```

1.5 The Generic NewPair Class

- We can have more than one generic type in a generic class
- Let’s modify the generic pair class such that:
  - Each pair can have two values of **different data types**

NewPair.java

```java
class NewPair<S,T> {

  private S first;
  private T second;

  public NewPair(S a, T b) {
      first = a;
      second = b;
  }

  public S getFirst() {
      return first;
  }

  public T getSecond() {
      return second;
  }
}
```

You can have multiple generic data types.

Convention: Use single uppercase letters for generic data types.

1.5 Using the Generic NewPair Class

TestNewGenericPair.java

```java
public class TestNewGenericPair {

  public static void main(String[] args) {

      NewPair<String, Integer> someone =
          new NewPair<String, Integer>("James Gosling", 55);

      System.out.println("Name: " + someone.getFirst());
      System.out.println("Age: " + someone.getSecond());
  }
}
```

```
Name: James Gosling
Age: 55
```

- This NewPair class is now very flexible!
  - Can be used in many ways

2.1 Generic methods

- Generic methods are those methods that are written with a single method declaration and can be called with arguments of different types. The compiler will ensure the correctness of whichever type is used. These are some properties of generic methods:
  - Generic methods have a type parameter (the diamond operator enclosing the type) before the return type of the method declaration
  - Type parameters can be bounded (bounds are explained later in the article)
  - Generic methods can have different type parameters separated by commas in the method signature
  - Method body for a generic method is just like a normal method
- An example of defining a generic method to convert an array to a list:

```java
public <T> List<T> fromArrayToList(T[] a) {
    return Arrays.stream(a).collect(Collectors.toList());
}
```

The *<T>* in the method signature implies that the method will be dealing with generic type *T*. This is needed even if the method is returning void.

2.1 Generic methods

- the method can deal with more than one generic type, all generic types must be added to the method signature, for example, if we want to modify the above method to deal with type *T* and type *G*, it should be written like this:

```java
public static <T, G> List<G> fromArrayToList(T[] a, Function<T, G> mapperFunction) {
    return Arrays.stream(a)
      .map(mapperFunction)
      .collect(Collectors.toList());
}
```

- We're passing a function that converts an array with the elements of type *T* to list with elements of type *G.* An example would be to convert *Integer* to its *String* representation:

```java
@Test
public void givenArrayOfIntegersThanListOfStringReturnedOK() {
    Integer[] intArray = {1, 2, 3, 4, 5};
    List<String> stringList = Generics.fromArrayToList(intArray, Object::toString);
    assertThat(stringList, hasItems("1", "2", "3", "4", "5"));
}
```

- Oracle recommendation is to use an uppercase letter to represent a generic type and to choose a more descriptive letter to represent formal types, for example in Java Collections *T* is used for type, *K* for key, *V* for value.

2.2 Bounded Generics

- Type parameters can be bounded. Bounded means “*restricted*“, we can restrict types that can be accepted by a method.
- For example, we can specify that a method accepts a type and all its subclasses (upper bound) or a type all its superclasses (lower bound).

```
public <T extends Number> List<T> fromArrayToList(T[] a) {
    ...
}
```

- The keyword *extends* is used here to mean that the type *T* extends the upper bound in case of a class or implements an upper bound in case of an interface.

2.2 Multiple Bounds

- A type can also have multiple upper bounds as follows:

```
<T extends Number & Comparable>
```

- If one of the types that are extended by *T* is a class (i.e *Number*), it must be put first in the list of bounds. Otherwise, it will cause a compile-time error.

2.3 Using Wildcards With Generics

- Wildcards are represented by the question mark in Java "*?*" and they are used to refer to an unknown type. Wildcards are particularly useful when using generics and can be used as a parameter type but first, there is an *important* note to consider.

- **It is known that *Object* is the supertype of all Java classes, however, a collection of *Object* is not the supertype of any collection.**

- For example, a *List<Object>* is not the supertype of *List<String>* and assigning a variable of type *List<Object>* to a variable of type *List<String>* will cause a compiler error. This is to prevent possible conflicts that can happen if we add heterogeneous types to the same collection.

- The Same rule applies to any collection of a type and its subtypes. Consider this example:

```
public static void paintAllBuildings(List<Building> buildings) {
    buildings.forEach(Building::paint);
}
```

- if we imagine a subtype of *Building*, for example, a *House*, we can't use this method with a list of *House*, even though *House* is a subtype of *Building*.

- If we need to use this method with type Building and all its subtypes, then the bounded wildcard can do the magic:

```
public static void paintAllBuildings(List<? extends Building> buildings) {
    ...
}
```

- Now, this method will work with type *Building* and all its subtypes. This is called an upper bounded wildcard where type *Building* is the upper bound.
- Wildcards can also be specified with a lower bound, where the unknown type has to be a supertype of the specified type. Lower bounds can be specified using the *super* keyword followed by the specific type, for example, *<? super T>* means unknown type that is a superclass of *T* (= T and all its parents).

2.4 Type Erasure

- Generics were added to Java to ensure type safety and to ensure that generics wouldn't cause overhead at runtime, the compiler applies a process called *type erasure* on generics at compile time.
- Type erasure removes all type parameters and replaces it with their bounds or with *Object* if the type parameter is unbounded. Thus the bytecode after compilation contains only normal classes, interfaces and methods thus ensuring that no new types are produced. Proper casting is applied as well to the *Object* type at compile time.
- This is an example of type erasure:

```
public <T> List<T> genericMethod(List<T> list) {
    return list.stream().collect(Collectors.toList());
}
```

- With type erasure, the unbounded type *T* is replaced with *Object* as follows:

```
// for illustrationpublic
List<Object> withErasure(List<Object> list) {
    return list.stream().collect(Collectors.toList());
}
```

2.4 Type Erasure

- If the type is bounded, then the type will be replaced by the bound at compile time:

```java
public <T extends Building> void genericMethod(T t) {
    ...
}
```

- After compilation:

```java
public void genericMethod(Building t) {
    ...
}
```

2.5 Generics and Primitive Data Types

- **A restriction of generics in Java is that the type parameter cannot be a primitive type.**
- For example, the following doesn't compile:

```
List<int> list = new ArrayList<>();
list.add(17);
```

2.5 Generics and Primitive Data Types

- To understand why primitive data types don't work, let's remember that **generics are a compile-time feature**, meaning the type parameter is erased and all generic types are implemented as type *Object*.
- As an example, let's look at the *add* method of a list:

```
List<Integer> list = new ArrayList<>();
list.add(17);
```

- The signature of the *add* method is:

```
boolean add(E e);
```

- And will be compiled to:

```
boolean add(Object e);
```

- Therefore, type parameters must be convertible to *Object*. **Since primitive types don't extend *Object*, we can't use them as type parameters.**

2.5 Generics and Primitive Data Types

- **However, Java provides boxed types for primitives, along with autoboxing and unboxing to unwrap them:**

```
Integer a = 17;
int b = a;
```

- So, if we want to create a list which can hold integers, we can use the wrapper:

```
List<Integer> list = new ArrayList<>();
list.add(17);
int first = list.get(0);
```

- The compiled code will be the equivalent of:

```
List list = new ArrayList<>();
list.add(Integer.valueOf(17));
int first = ((Integer) list.get(0)).intValue();
```

- Caution:
  - Generics are useful when the code remains unchanged other than differences in data types
  - When you declare a generic class/method, make sure that the code is valid for all possible data types
- Additional Java Generics:
  - Generic methods
  - Bounded generic data types
  - Wildcard generic data types
  - Generics and Primitive Data Types

Objectives

Using the Vector and ArrayList classes

Outline

Drawback of Arrays

- Array, as discussed in previous weeks, has a major drawback:
  - Once initialized, the array size is fixed
  - Reconstruction is required if the array size changes
  - To overcome such limitation, we can use some classes related to array
- Java has an **Array** class
  - Check API documentation and explore it yourself
- However, we will not be using this **Array** class much; we will be using other classes such as **Vector** and **ArrayList**

Vector and ArrayList

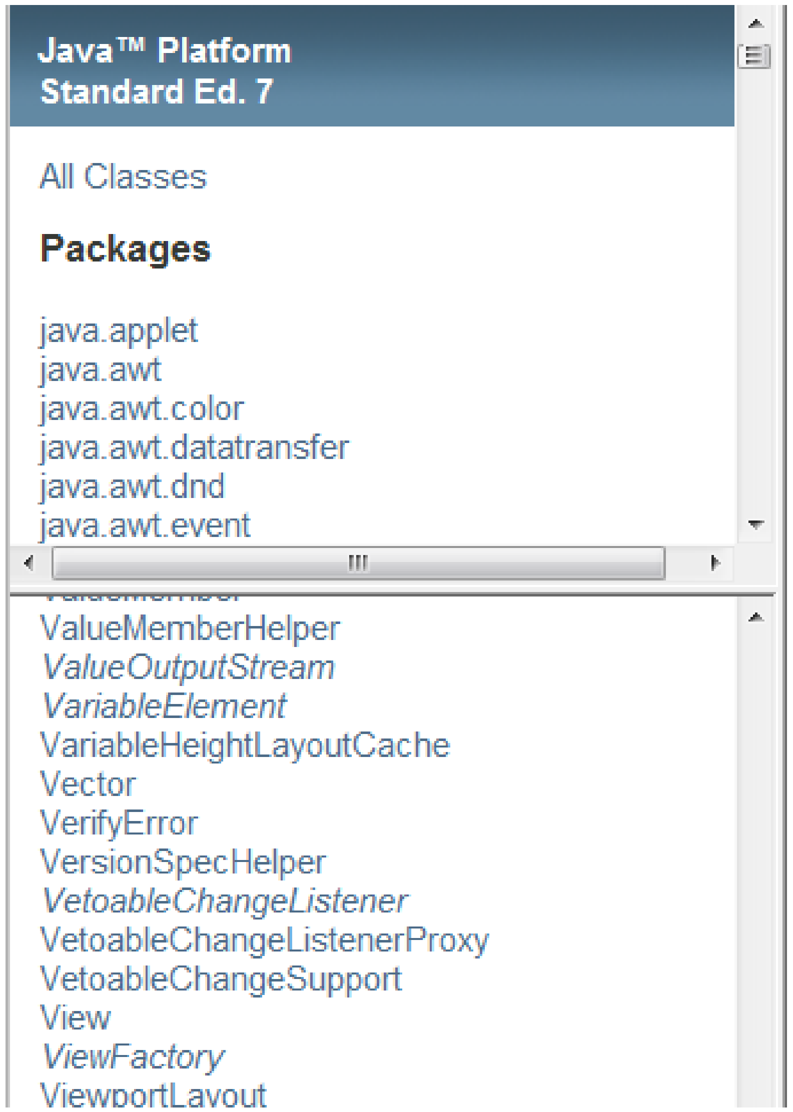
- Both provide re-sizable array, i.e. array that is growable
- Both are implementations of the **List** interface
  - We will cover interface later, under Abstract Data Types (ADTs)
- Differences between Vector and ArrayList

Class for dynamic-size arrays

Motivation

- Java offers a Vector class to provide:
  - Dynamic size
    - expands or shrinks automatically
  - Generic
    - allows any reference data types
  - Useful predefined methods
- Use array if the size is fixed; use Vector if the size may change.

API documentation (1/3)



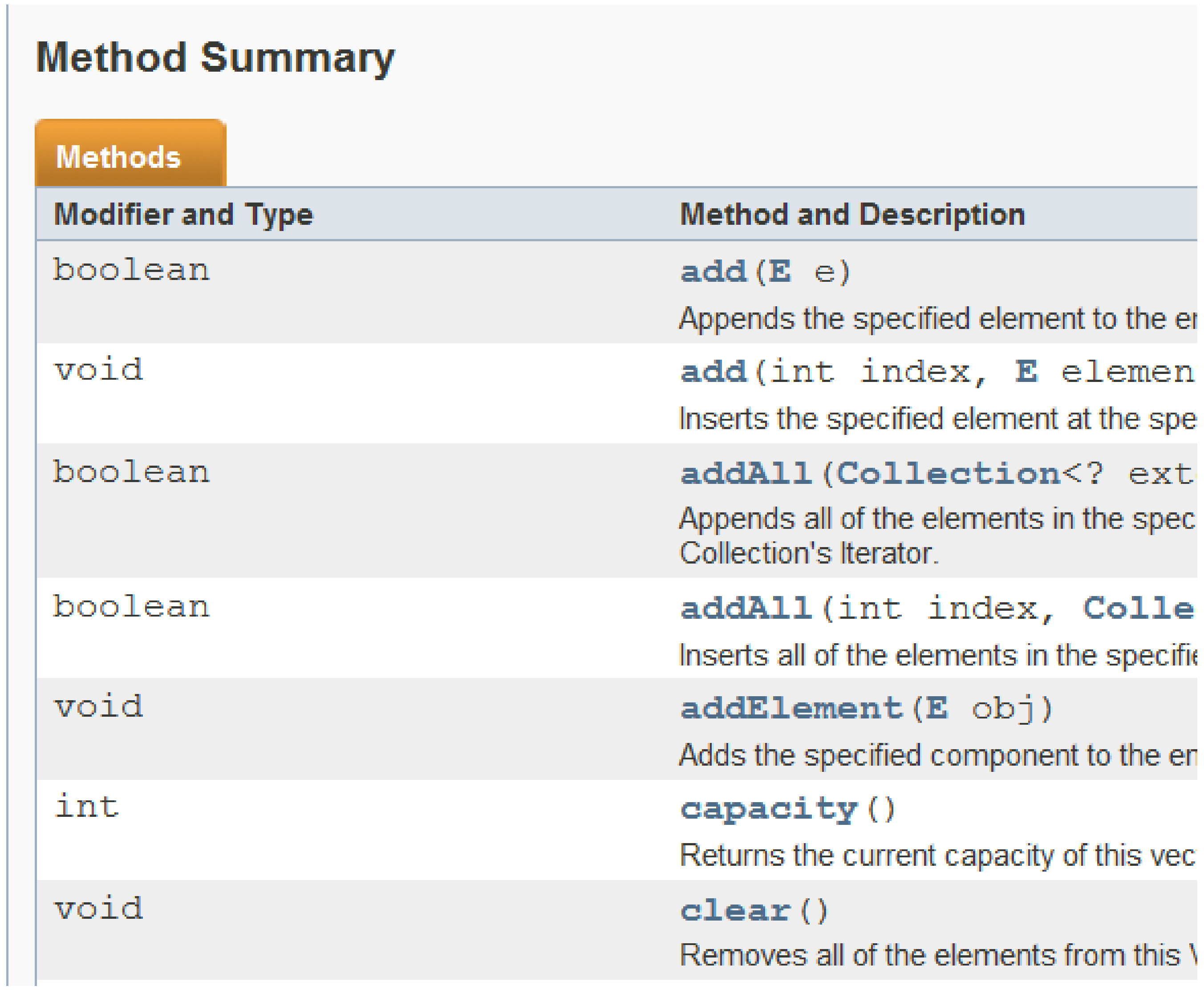
Java™ Platform
Standard Ed. 7

All Classes

Packages

java.applet
java.awt
java.awt.color
java.awt.datatransfer
java.awt.dnd
java.awt.event

ValueMemberHelper
ValueOutputStream
VariableElement
VariableHeightLayoutCache
Vector
VerifyError
VersionSpecHelper
VetoableChangeListener
VetoableChangeListenerProxy
VetoableChangeSupport
View
ViewFactory
ViewportLayout



Method Summary

Methods

Modifier and Type	Method and Description
`boolean`	**`add`**`(E e)` Appends the specified element to the e
`void`	**`add`**`(int index, E elemen` Inserts the specified element at the spe
`boolean`	**`addAll`**`(Collection<? ext` Appends all of the elements in the spec Collection's Iterator.
`boolean`	**`addAll`**`(int index, Colle` Inserts all of the elements in the specifi
`void`	**`addElement`**`(E obj)` Adds the specified component to the er
`int`	**`capacity`**`()` Returns the current capacity of this vec
`void`	**`clear`**`()` Removes all of the elements from this \

API documentation (2/3)

PACKAGE

```
import java.util.Vector;
```

SYNTAX

```
//Declaration of a Vector reference
Vector<E> myVector;

//Initialize a empty Vector object
myVector = new Vector<E>();
```

Commonly Used Method Summary	
boolean	*isEmpty()* Tests if this vector has no components.
int	*size()* Returns the number of components in this vector.

API documentation (3/3)

Commonly Used Method Summary (continued)	
boolean	*add*(E o) Appends the specified element to the end of this Vector.
void	*add*(int index, E element) Inserts the specified element at the specified position in this Vector.
E	*remove*(int index) Removes the element at the specified position in this Vector.
boolean	*remove*(Object o) Removes the first occurrence of the specified element in this Vector If the Vector does not contain the element, it is unchanged.
E	*get*(int index) Returns the element at the specified position in this Vector.
int	*indexOf*(Object elem) Searches for the first occurrence of the given argument, testing for equality using the equals method.
boolean	*contains*(Object elem) Tests if the specified object is a component in this vector.

API documentation (3/3)

TestVector.java

```java
import java.util.Vector;

public class TestVector {
  public static void main(String[] args) {
      Vector<String> courses = new Vector<String>();
      courses.add("CS1020");
      courses.add(0, "CS1010");
      courses.add("CS2010");

      System.out.println(courses);
      System.out.println("At index 0: " + courses.get(0));

      if (courses.contains("CS1020")) {
          System.out.println("CS1020 is in courses");
      }
      courses.remove("CS1020");
      for (String c: courses) {
          System.out.println(c);
      }
  }
}
```

Vector class has a nice **toString()** method that prints all elements

The enhanced for-loop is applicable to Vector objects too!

```
Output:
[CS1010, CS1020, CS2010]
At index 0: CS1010
CS1020 is in courses
CS1010
CS2010
```

Another class for dynamic-size arrays

Introduction (1/2)

- Java offers an ArrayList class to provide similar features as Vector:
  - Dynamic size
    - expands or shrinks automatically
  - Generic
    - allows any reference data types
  - Useful predefined methods
- Similarities:
  - Both are index-based and use an array internally
  - Both maintain insertion order of element
- So, what are the differences between Vector and ArrayList?
  - This is one of the most frequently asked questions, and at interviews!

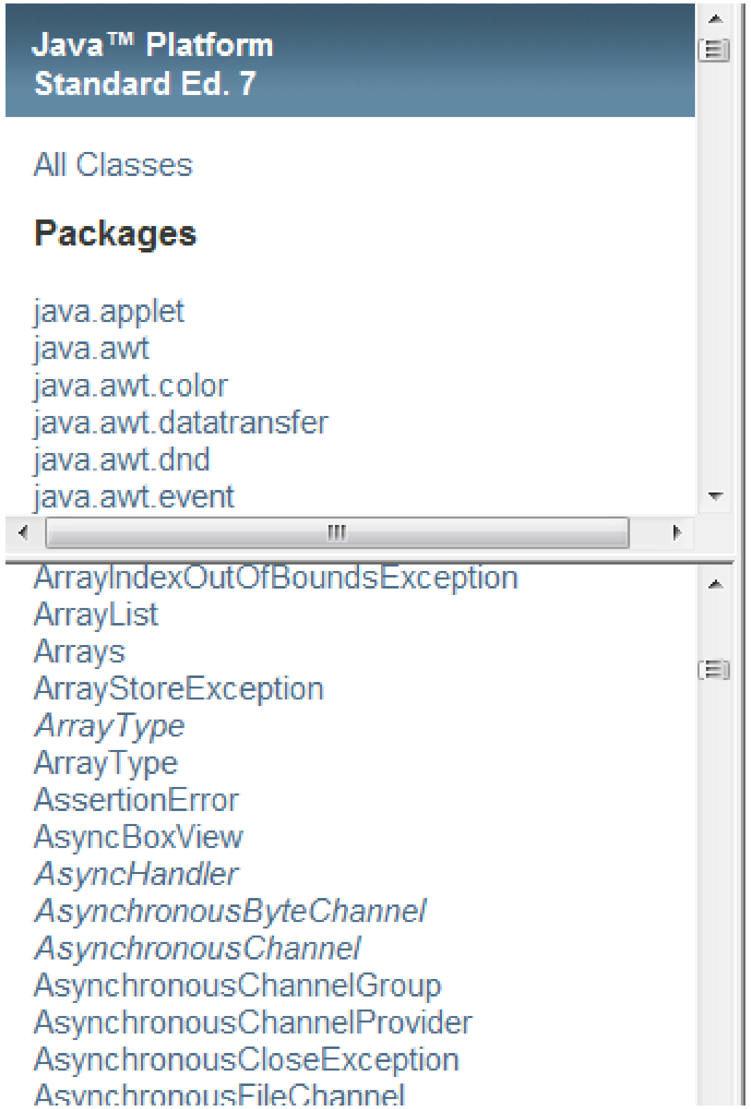
Introduction (2/2)

- Differences between Vector and ArrayList

Vector	ArrayList
Since JDK 1.0	Since JDK 1.2
Synchronised * (thread-safe)	Not synchronised
Slower (price of synchronisation)	Faster (≈20 – 30%)
Expansion: default to double the size of its array (can be set)	Expansion: increases its size by ≈50%

- ArrayList is preferred if you do not need synchronisation
  - Java supports multiple threads, and these threads may read from/write to the same variables, objects and resources. Synchronisation is a mechanism to ensure that Java thread can execute an object's synchronised methods one at a time.
- When using Vector /ArrayList, always try to initialise to the largest capacity that your program will need, since expanding the array is costly.
  - Array expansion: allocate a larger array and copy contents of old array to the new one

API documentation (1/3)



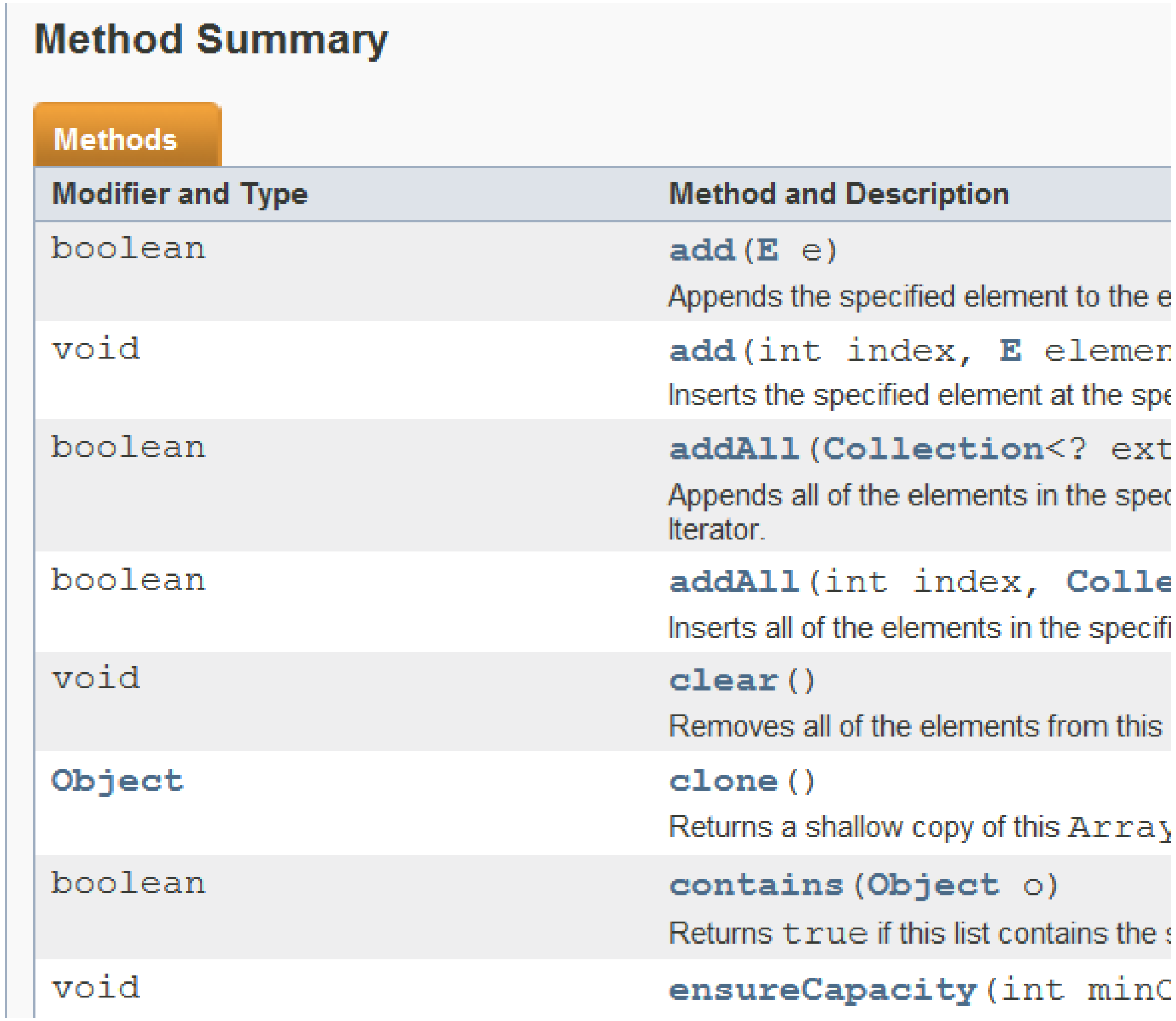
Java™ Platform
Standard Ed. 7

All Classes

Packages

java.applet
java.awt
java.awt.color
java.awt.datatransfer
java.awt.dnd
java.awt.event

ArrayIndexOutOfBoundsException
ArrayList
Arrays
ArrayStoreException
ArrayType
ArrayType
AssertionError
AsyncBoxView
AsyncHandler
AsynchronousByteChannel
AsynchronousChannel
AsynchronousChannelGroup
AsynchronousChannelProvider
AsynchronousCloseException
AsynchronousFileChannel



Method Summary

Methods

Modifier and Type	Method and Description
`boolean`	**`add`**`(E e)` Appends the specified element to the e
`void`	**`add`**`(int index, E elemen` Inserts the specified element at the spe
`boolean`	**`addAll`**`(Collection<? ext` Appends all of the elements in the spec Iterator.
`boolean`	**`addAll`**`(int index, Colle` Inserts all of the elements in the specifi
`void`	**`clear`**`()` Removes all of the elements from this
`Object`	**`clone`**`()` Returns a shallow copy of this `Array`
`boolean`	**`contains`**`(Object o)` Returns `true` if this list contains the s
`void`	**`ensureCapacity`**`(int minC`

API documentation (2/3)

PACKAGE

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

SYNTAX

```
 // Declaration of a ArrayList reference
ArrayList<E> myArrayList;

// Initialize a empty ArrayList object
myArrayList = new ArrayList<E>();
```

Commonly Used Method Summary	
boolean	*isEmpty*() Returns **true** if this list contains no element.
int	*size*() Returns the number of elements in this list.

API documentation (3/3)

Commonly Used Method Summary (continued)	
boolean	*add*(E e) Appends the specified element to the end of this list.
void	*add*(int index, E element) Inserts the specified element at the specified position in this list.
E	*remove*(int index) Removes the element at the specified position in this list.
boolean	*remove*(Object o) Removes the first occurrence of the specified element from this list, if it is present.
E	*get*(int index) Returns the element at the specified position in this list.
int	*indexOf*(Object o) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	*contains*(Object elem) Returns **true** if this list contains the specified element.

API documentation (3/3)

TestArrayList.java

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

public class TestArrayList {
  public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      ArrayList<Integer> list = new ArrayList<Integer>();
      System.out.println("Enter a list of integers, press ctrl-d to end.");
      while (sc.hasNext()) {
          list.add(sc.nextInt());
      }
      System.out.println(list); // using ArrayList's toString()

      // Move first value to last
      list.add(list.remove(0));
      System.out.println(list);
  }
}
```

```
Output:
Enter a list ... to end.
31
17
-5
26
50
(user pressed ctrl-d here)
[31, 17, -5, 26, 50]
[17, -5, 26, 50, 31]
```

Thank you!

HUS
VNU UNIVERSITY OF SCIENCE