Chapter 5.

ArrayList and Wrapper Classes

2020-2021
COMP2396 Object-Oriented Programming and Java
Dr. T.W. Chim (E-mail: twchim@cs.hku.hk)
Department of Computer Science, The University of Hong Kong

Arrays

- In Java, arrays are objects and they live on the heap
- Unlike many other objects, arrays in Java
  - Do not have any method (save for those inherited from the Object class)
  - Have one and only one instance variable (i.e., length)
  - Use special array syntax (i.e., the subscript operator []) that is not used anywhere else in Java
- Limitations
  - The size of an array must be determined at the time of creation, and cannot be changed afterwards
  - Data (primitives or references) can be put into and read from an array using array syntax, but cannot be actually removed from the array

Arrays

Example

```
Dog[] myDogs = new Dog[4];

for (int i = 0; i < myDogs.length; i++) {
   myDogs[i] = new Dog();
}

for (int i = 0; i < myDogs.length; i++) {
   Dog dog = myDogs[i];
   dog.makeNoise();
}

myDogs[2] = null;

for (Dog dog : myDogs) {
   if (dog != null) {
      dog.makeNoise();
   }
}
```

Creating a Dog array with 4 elements

Putting a Dog reference into the array

Getting a Dog reference from the array

"Removing" a Dog reference from the array

The size of the array does not change

Arrays

- Wouldn't it be fantastic if an array
  - Could grow when you add something to it?
  - Could shrink when you remove something from it?
  - Could tell you if it contains what you're looking for without having you to loop through and check each element?
  - Could let you get things out of it without having you to know exactly which slots the things are in?

ArrayList is the answer!

ArrayList

- ArrayList is a class in the core Java library (the API)
- Can be used in your code as if you wrote it yourself

ArrayList
add(Object elem) Adds the object parameter to the list remove(int index) Removes the object at the index parameter remove(Object elem) Removes this object (if it is in the ArrayList) contains(Object elem) Returns true if there is a match for the object parameter isEmpty() Returns true if the list has no element indexOf(Object elem) Returns the index of the object parameter or –1 size() Returns the number of elements currently in the list get(int index) Returns the object currently at the index parameter …

ArrayList

Example

```
// create an ArrayList
ArrayList<Egg> myList = new ArrayList<Egg>();

// put something into it
Egg a = new Egg();
myList.add(a);

// put another thing into it
Egg b = new Egg();
myList.add(b);

// find out how many things are in it
int theSize = myList.size();

// find out if it contains something
boolean isIn = myList.contains(a);
```

The type parameter in the angle-brackets specifies the type of objects that can be stored in the ArrayList

The ArrayList grows when an item is added to it

The ArrayList grows when an item is added to it

The size() method returns the number of elements currently in the list (i.e., 2)

The contains() method returns true as the ArrayList does contain (a reference to) the object referenced by a

ArrayList

Example

```
// find out where something is (i.e., its index)
int idx = myList.indexOf(b);

// find out if it is empty
boolean empty = myList.isEmpty();

// Remove something from it
myList.remove(a);

// loop through it
for (Egg egg : myList) {
   // egg.xxxx
}
```

The indexOf() method returns the (zero-based) index of the object referenced by b (i.e., 1)

The isEmpty() method returns false as the ArrayList is not empty

The ArrayList shrinks when an item is removed from it

The enhanced for loop can be applied to loop through the ArrayList

ArrayList vs Regular Array

ArrayList

```
ArrayList<String> myList = new
ArrayList<String>();
```

An ArrayList does not need to know its size at the time of creation. It grows and shrinks as objects are added or removed from it. String s = myList.get(1);

```
int theSize = myList.size();

String s = myList.get(1);

myList.remove(1);

boolean isIn = myList.contains(b);
```

Regular Array

```
String[] myList = new String[2];

String a = new String("whoohoo");
String b = new String("Frog");
myList[1] = b;

int theSize = myList.length;

String s = myList[1];

myList[1] = null;

boolean isIn = false;
for (String item : myList) {
  if (b.equals(item)) {
    isIn = true;
    break;
  }
}
```

A regular array has to know its size at the time of creation

ArrayList vs Regular Array

ArrayList

```
ArrayList<String> myList = new
ArrayList<String>();

String a = new String("whoohoo");
myList.add(a);

String b = new String("Frog");
myList.add(b);
```

An object can be added without specifying a location (i.e. an index) in an ArrayList. The ArrayList will keep growing to make room for the new thing.

Regular Array

```
String[] myList = new String[2];

String a = new String("whoohoo");
myList[0] = a;

String b = new String("Frog");
myList[1] = b;
```

An object must be assigned to a specific location (zero-based index) in a regular array. If the index is outside the boundaries of the array, it blows up at runtime.

```
for (String item : myList) {
  if (b.equals(item)) {
    isIn = true;
    break;
  }
}
```

ArrayList vs Regular Array

ArrayList

```
ArrayList<String> myList = new
ArrayList<String>();

String a = new String("whoohoo");
myList.add(a);

String b = new String("Frog");
myList.add(b);

int theSize = myList.size();

String s = myList.get(1);
boolean isIn = myList.contains(b);
```

The (dynamic) size of an ArrayList can be retrieved by calling its size() method

Regular Array

```
String[] myList = new String[2];

String a = new String("whoohoo");
myList[0] = a;

String b = new String("Frog");
myList[1] = b;

int theSize = myList.length;

String s = myList[1];

for (String item : myList) {
  if (b.equals(item)) {
    isIn = true;
    break;
  }
}
```

The size of a regular array is stored in its (final) instance variable length

ArrayList vs Regular Array

ArrayList

```
ArrayList<String> myList = new
ArrayList<String>();

String a = new String("whoohoo");
myList.add(a);
```

An ArrayList is just a plain Java object, and uses no special syntax

```
int theSize = myList.size();

String s = myList.get(1);

myList.remove(1);

boolean isIn = myList.contains(b);
```

Regular Array

```
String[] myList = new String[2];

String a = new String("whoohoo");
myList[0] = a;
```

A regular array uses array syntax that is not used anywhere else in Java

```
String s = myList[1];

myList[1] = null;

boolean isIn = false;
for (String item : myList) {
  if (b.equals(item)) {
    isIn = true;
    break;
  }
}
```

ArrayList vs Regular Array

ArrayList

```
ArrayList<String> myList = new
ArrayList<String>();

String a = new String("whoohoo");
myList.add(a);
```

An object can be removed from an ArrayList and the ArrayList shrinks accordingly

```
String s = myList.get(1);

myList.remove(1);

boolean isIn = myList.contains(b);
```

Regular Array

```
String[] myList = new String[2];

String a = new String("whoohoo");
myList[0] = a;
```

An object cannot be actually removed from a regular array (assigning null to an array element does not change the array size)

```
String s = myList[1];

myList[1] = null;

boolean isIn = false;
for (String item : myList) {
  if (b.equals(item)) {
    isIn = true;
    break;
  }
}
```

ArrayList vs Regular Array

ArrayList

```
ArrayList<String> myList = new
ArrayList<String>();

String a = new String("whoohoo");
myList.add(a);

String b = new String("Frog");
myList.add(b);
```

An ArrayList can tell if it contains an object by calling its contains() method

```
myList.remove(1);

boolean isIn = myList.contains(b);
```

Regular Array

```
String[] myList = new String[2];

String a = new String("whoohoo");
myList[0] = a;

String b = new String("Frog");
myList[1] = b;
```

There is no simple way to check if an object is in a regular array without having to loop through and check each element

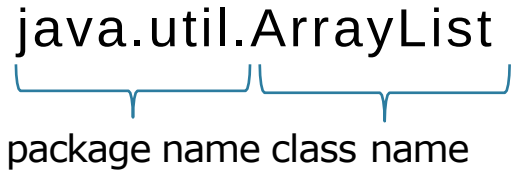
```
myList[1] = null;

boolean isIn = false;
for (String item : myList) {
  if (b.equals(item)) {
    isIn = true;
    break;
  }
}
```

Packages

- In the Java API, classes are grouped into packages (e.g., ArrayList is in the package `java.util` which holds a pile of utility classes)
- Packages are important for 3 main reasons
  - Help the overall organization (classes are grouped into packages for specific kinds of functionality, e.g., GUI, data structures, etc.)
  - Provide a name-scoping that helps to prevent collisions of names
  - Provide a level of security (allowing placing restrictions on code such that only other classes in the same package can access it)

Packages

- A class has a full name which is a combination of the package name and the class name
  e.g.,



- To use a class in a package other than java.lang, the full name of the class must be specified
  e.g.,

```
java.util.ArrayList<Dog> list = new java.util.ArrayList<Dog>();
```

```
public void go(java.util.ArrayList<Dog> list) { … }
```

parameter type

```
public java.util.ArrayList<Dog> foo() { … }
```

return type

Packages

- Alternatively, include an import statement at the top of the source code to avoid typing the full name everywhere, e.g.,

```
import java.util.ArrayList;

public MyClass {
   ArrayList<Dog> list;
   // ...
}
```

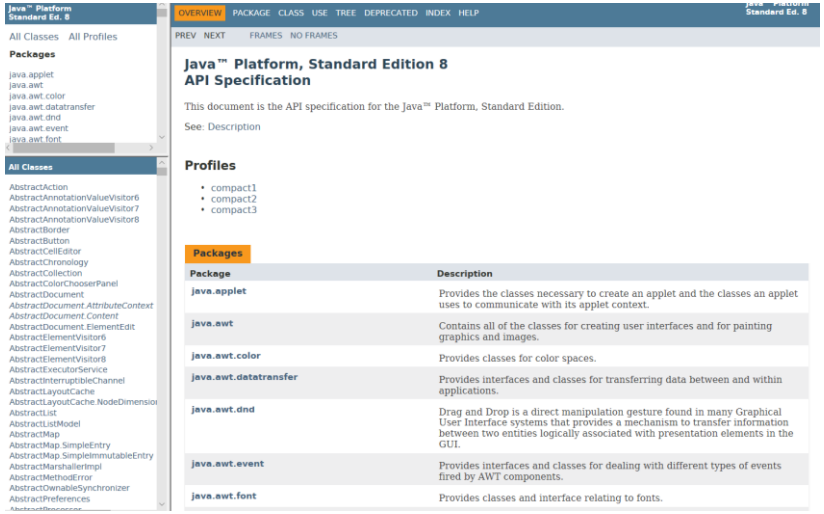
- It is also possible to import all classes in a package using a wildcard character *, e.g.,

```
import java.util.*;
```

- Note that an import statement simply saves you from typing the full name of a class, it will not make your code bloated or slower

How to Play with the API

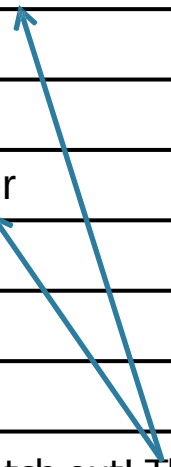
- Use the HTML API docs
  - Java comes with a fabulous set of online docs
    http://docs.oracle.com/javase/8/docs/api/index.html
  - The API docs are the best reference for
    - Finding out what are in the Java library
    - Getting details about a class and its methods



Wrapper Classes

- The type parameter of an ArrayList supports classes only (i.e., it is not possible to create ArrayLists of primitive types)
- There is a wrapper class for every primitive type such that a primitive can be treated like an object
- Each wrapper class is named after the primitive type (except char and int), but with the first letter capitalized
- The wrapper classes are in the java.lang package (i.e., no import statement is needed)

Primitive Type	Wrapper Class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double



Watch out! The names are not mapped exactly to the primitive types

Wrapper Classes

- Examples: wrapping a value

```
boolean b = true;
Boolean bWrap = new Boolean(b);
char c = 'K';
Character cWrap = new Character(c);
int i = 288;
Integer iWrap = new Integer(i);
double d = 1.234567;
Double dWrap = new Double(d);
```

Simply give the primitive to the constructor of the wrapper class

- Examples: unwrapping a value

```
boolean bUnWrap = bWrap.booleanValue();
char cUnWrap = cWrap.charValue();
int iUnWrap = iWrap.intValue();
double dUnWrap = dWrap.doubleValue();
```

All the wrapper classes work like this. E.g., Byte has a byteValue() method, Short has a shortValue() method, etc.

An ArrayList of a Primitive Type

- Example

```
import java.util.*;

public class WithoutAutoBoxing {
   public static void main(String[] args) {
      ArrayList<Integer> listOfNumbers = new ArrayList<Integer>();
      listOfNumbers.add(new Integer(3));
      Integer num = listOfNumbers.get(0);
      int intNum = num.intValue();

}
```

Wrap an int value 3 into an Integer object and add it to the ArrayList

Unwrap an int value from an Integer object

- The wrapping and unwrapping of primitives sound rather tedious

Autoboxing

- The autoboxing feature in Java blurs the line between primitives and wrapper objects
- Autoboxing performs the conversion from primitives to wrapper objects, and vice versa, automatically
- Example

```
import java.util.*;

public class WithAutoBoxing {
    public static void main(String[] args) {
        ArrayList<Integer> listOfNumbers = new ArrayList<Integer>();
        listOfNumbers.add(3);
        int intNum = listOfNumbers.get(0);
    }
}
```

The compiler does all the wrapping (boxing) for you

The compiler unwraps (unboxes) the Integer object automatically

Autoboxing

- Autoboxing works almost everywhere
- Assignments
  - One can assign either a wrapper or primitive to a variable declared as a matching wrapper or primitive type, e.g.,

```
int p = new Integer(42);
Integer q = p;
```

Autoboxing

- Method arguments
  - If a method takes a primitive, one can pass in either a compatible primitive or a reference to a wrapper of that primitive type

```
public void takePrimitive(int i) { … }
```

  - If a method takes a wrapper type, one can pass in either a reference to a wrapper or a primitive of the matching type

```
public void takeWrapper(Integer i) { … }
```

Autoboxing

- Return values
  - If a method declares a primitive return type, one can return either a compatible primitive or a reference to the wrapper of that primitive type

```
public int returnPrimitive() { … }
```

  - If a method declares a wrapper return type, one can return either a reference to a wrapper or a primitive of the matching type

```
public Integer returnWrapper() { … }
```

Autoboxing

- Boolean expressions
  - Any place a boolean value is expected, one can use either an expression that evaluates to a boolean, a primitive boolean, or a reference to a Boolean wrapper

```
if (bool) { … }
```

Autoboxing

- Operations on numbers
  - One can use a wrapper type as an operand in operations where the primitive type is expected e.g.,

```
Integer i = new Integer(42);
i++;

Integer j = new Integer(5);
Integer k = j + 3;
```

Autoboxing

Example

```
public class TestBox {
        Integer i;
        int j;

        public static void main(String[] args) {
                TestBox t = new TestBox();
                t.go();
        }

        public void go() {
                j = i;
                System.out.println(j);
                System.out.println(i);
        }
}
```

Will this code compile?
Will it run?
If it runs, what will it do?

Autoboxing

Example

```
public class TestBox {
        Integer i;
        int j;

        public static void main(String[] args) {
                TestBox t = new TestBox();
                t.go();
        }

        public void go() {
                j = i;
                System.out.println(j);
                System.out.println(i);
        }
}
```

Will this code compile? Yes
Will it run? Yes, but with error
If it runs, what will it do?

Autoboxing

- Example

```
public class TestBox {
        Integer i;
        int j;

        public static void main(String[] args) {
                TestBox t = new TestBox();
                t.go();
        }

        public void go() {
                j = i;
                System.out.println(j);
                System.out.println(i);
        }
}
```

This instance variable will get a default value of null

Autoboxing fails as i is not referencing any valid Integer object. This will result in a NullPointerException

String to Primitive

- The wrapper classes have static parse methods that take a string and return a primitive value
- Examples

```
String s = "2";
int x = Integer.parseInt(s);
double d = Double.parseDouble("420.24");

boolean b = Boolean.parseBoolean("True");
```

```
String t ="two";
int y = Integer.parseInt(t);
```

This compiles just fine, but at runtime it blows up. Anything that cannot be parsed as a number will cause a NumberFormatException

Primitive to String

- The easiest way to turn a number into a string is by simply concatenating the number to an existing string, e.g.,

```
double d = 42.5;
String doubleString = "" + d;
```

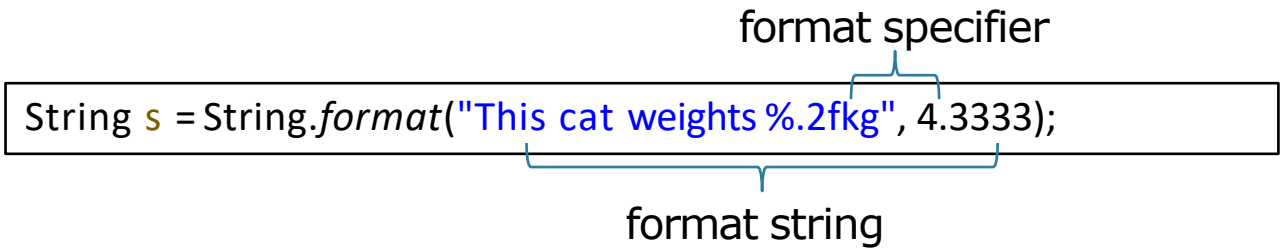
- Alternatively, this can be done by calling the static `toString()` method of a wrapper class, e.g.,

```
double d = 42.5;
String doubleString = Double.toString(d);
```

Number Formatting

- In Java, formatting numbers is a simple matter of calling the static format() method of the String class
- The first argument to the format() method is called the format string, and it can include characters that are printed as-is, together with one or more format specifiers that begin with a percentage sign (%)

e.g.,



- The rest of the arguments to the format() method are the numbers to be formatted by the format specifiers

Number Formatting

- Some common format specifiers
  - "%,d" means "inserts commas and format the number as a decimal integer"
  - "%.2f" means "format the number as a floating point with a precision of 2 decimal places"
  - "%,.2f" means "inserts commas and format the number as a floating point with a precision of 2 decimal places"
  - "%,5.2f" means "insert commas and format the number as a floating point with a precision of 2 decimal places and with a minimum of 5 characters, padding spaces and zeros as appropriate"
  - "%h" means "format the number as a hexadecimal"
  - "%c" means "format the number as a character"

Number Formatting

Example

```
System.out.println(String.format("Balance = %,d", 10000));
System.out.println(String.format("10000 / 3 = %.2f", 10000.0/3));
System.out.println(String.format("10000 / 3 = %,.2f", 10000.0/3));
System.out.println(String.format("10000 / 3 = %,10.2f", 10000.0/3));
System.out.println(String.format("255 = %h in hexadecimal", 255));
System.out.println(String.format("ASCII code 65 = %c", 65));
```

Sample output

```
Balance = 10,000
10000 / 3 = 3333.33
10000 / 3 = 3,333.33
10000 / 3 =     3,333.33
255 = ff in hexadecimal
ASCII code 65 = A
```

Chapter 5.

End

2020-2021

COMP2396 Object-Oriented Programming and Java

Dr. T.W. Chim (E-mail: twchim@cs.hku.hk)

Department of Computer Science, The University of Hong Kong