CS1020

Compiled By: Jin Zhe

NUS

Data Structures and Algorithms I

This set of notes is compiled from Mr. Aaron Tan’s lecture slides for AY2011/12 S2. All credits goes to him.

**Java**

**Produces source code**

1. **Writing / Editing Program**

* Tool: Any text editor
* Source code must have a .java extension

1. **Compiling Program**

* Tool: Use Java compiler e.g. javac
* e.g. "javac Hello.java"
* Compiled binary has .class extension:
* e.g. XXXX.class
* The binary is also known as Java Executable Bytecode

1. **Executing Binary**

Run on a Java Virtual Machine (JVM)

e.g. "java XXXX“ (leave out the .class extension)

**Produces Executable Bytecode**

Source Code

**import java.lang.\*; //optional and implicit**

**public class HelloWorld {**

**public static void main(String[] args) {**

**System.out.println("Hello World");**

**}//End of main**

**}//End of HelloWorld Class**

* Library in Java is known as package. Packages are organized into hierarchical grouping. (E.g., the "**System.out.println()**" is defined in the package "**java.lang.system**")
* The name of a public class is spelled exactly as the name of the file (*Case Sensitive*)
* All java programs begin execution with the method named main(). main method that gets executed has the following signature : **public static void main(String args[])**.
* Declaring this method as **public** means that it is accessible from outside the class so that the JVM can find it when it looks for the program to start it.
* It is necessary that the method is declared with return type **void** (*i.e. no arguments are returned from the method*).
* The main method contains a String argument array that can contain the command line arguments.
* The brackets **{** and **}** mark the beginning and ending of the class.
* The program contains a line ‘**System.out.println(“Hello World”);**’ that tells the computer to print out on one line of text namely ‘Hello World’.
* The semi-colon ‘**;**’ ends the line of code.
* The double slashes ‘**//**’ are used for comments that can be used to describe what a source code is doing. Everything to the right of the slashes on the same line does not get compiled, as they are simply the comments in a program.

Main method declarations

**class MainExample1 {**

**public static void main(String[] args) {}**

**}**

**class MainExample2 {**

**public static void main(String []args) {}**

**}**

**class MainExample3 {**

**public static void main(String args[]) {}**

**}**

* All the 3 valid main method’s shown above accepts a single String array argument.

Compiling and running

1. Save the program With .java Extension.
2. Compile the file from DOS prompt by typing javac <filename>.
3. Successful Compilation, results in creation of .class containing byte code
4. Execute the file by typing java <filename without extension>

Variable, Identifiers and Data Types

* **Variables** are used for data that change during program execution.
* All variables have an identifier, a data type, and a scope. The programmer assigns the names to variables, known as **identifiers**.
* An Identifier must be unique within a scope of the Java program.
* Variables have a **data type**, that indicates the kind of value they can store.
* Variables declared inside of a block or method are called local variables; They are not automatically initialized.
* The compiler will generate an error as a result of the attempt to access the local variables before a value has been assigned.

|  |  |  |
| --- | --- | --- |
| **Data Type** | **Default Value (for fields)** | **Range** |
| byte | 0 | -127 to +128 |
| short | 0 | -32768 to +32767 |
| int | 0 | -231 to 231-1 |
| long | 0L | -263 to 263-1 |
| float | 0.0f | Negative: -3.4028235E+38 to -1.4E-45  Positive: 1.4E-45 to 3.4028235E+38 |
| double | 0.0d | Negative: -1.7976931348623157E+308 to -4.9E-324  Positive: 4.9E-324 to 1.7976931348623157E+308 |
| char | ‘\u0000′ | 0 to 65535 |
| String (object) | null |  |
| boolean | false |  |

* ***Integer:*** byte, short, int, and long.
* ***Floating Point:*** float and double
* ***Character:*** char
* ***Boolean:*** variable with a value of true or false.

***Identifier Naming Rules***

* Can consist of upper and lower case letters, digits, dollar sign ($) and the underscore ( \_ ) character.
* Must begin with a letter, dollar sign, or an underscore
* Are case sensitive
* Keywords cannot be used as identifiers
* Within a given section of your program or scope, each user defined item must have a unique identifier
* Can be of any length.
* Use "**\_**" or **myXXXX** to denote attributes and differentiate them from methods

Classes, objects, interface, instance members, static members

A **class** is nothing but a blueprint for creating different objects which defines its properties and behaviors. An object exhibits the properties and behaviors defined by its class. A class can contain fields and methods to describe the behavior of an object. Methods are nothing but members of a class that provide a service for an object or perform some business logic.

An **object** is an instance of a class created using a new operator. The new operator returns a reference to a new instance of a class. This reference can be assigned to a reference variable of the class. The process of creating objects from a class is called instantiation. An object reference provides a handle to an object that is created and stored in memory. In Java, objects can only be manipulated via references, which can be stored in variables.

An **Interface** is a contract in the form of collection of method and constant declarations. When a class implements an interface, it promises to implement all of the methods declared in that interface.

Each object created will have its own copies of the fields defined in its class called instance variables which represent an object’s state. The methods of an object define its behaviour called instance methods. Instance variables and instance methods, which belong to objects, are collectively called **instance members**. The dot ‘.’ notation with a object reference is used to access Instance Members.

**Static members** are those that belong to a class as a whole and not to a particular instance (object). A static variable is initialized when the class is loaded. Similarly, a class can have static methods. Static variables and static methods are collectively known as static members, and are declared with a keyword static. Static members in the class can be accessed either by using the class name or by using the object reference, but instance members can only be accessed via object references.

**/\*\* Comment**

**\* Displays "Hello World!" to the standard output.**

**\*/**

**public class HelloWorld {**

**String output = "";**

**static HelloWorld helloObj;**

**public HelloWorld(){**

**output = "Hello World";**

**}**

**public String printMessage(){**

**return output;**

**}**

**public static void main (String args[]) {**

**helloObj = new HelloWorld();**

**System.out.println(helloObj.printMessage());**

**}**

**}**

**Class Name**

**Object Reference**

**Member Function**

**Instance Member**

**Object Created**

Object References

20

**x**

**int x = 20;**

* Primitive data types are stored in address

20

**y**

**Integer y = new Integer (20);**

* An object of a class comes into existence when applying the new operator.
* A reference variable only contains a reference or pointer to an object.
* Logical equality operator will test the address and not the value stored
* Assignment will change the address that reference pointer points to

c

b

A

**z**

**String z = new String (“A bc”);**

Operators

**Bitwise and Bit Shift Operators**

~ Unary bitwise complement

<< Signed left shift

>> Signed right shift

>>> Unsigned right shift

& Bitwise AND

^ Bitwise exclusive OR

| Bitwise inclusive OR

**Simple Assignment Operator**

= Simple assignment operator

**Arithmetic Operators**

+ Additive operator (also used

for String concatenation)

- Subtraction operator

\* Multiplication operator

/ Division operator

% Remainder operator

**Unary Operators**

+ Unary plus operator; indicates

positive value (numbers are

positive without this, however)

- Unary minus operator; negates

an expression

++ Increment operator; increments

a value by 1

-- Decrement operator; decrements

a value by 1

! Logical complement operator;

inverts the value of a boolean

**Equality and Relational Operators**

== Equal to

!= Not equal to

> Greater than

>= Greater than or equal to

< Less than

<= Less than or equal to

**Conditional Operators**

&& Conditional-AND

|| Conditional-OR

?: Ternary (shorthand for

if-then-else statement)

**Type Comparison Operator**

instanceof Compares an object to

a specified type

|  |  |  |
| --- | --- | --- |
| **()** | Parentheses Grouping | Left-to-right |
| **++, --** | Postfix incrementor/decrementor | Right-to-left |
| **++, --**  **+, -** | Prefix incrementor/decrementor  Unary +, - | Right-to-left |
| **%** | Remainder of integer division | Left-to-right |
| **\*, /** | Multiplication, Division | Left-to-right |
| **+, -** | Addition, Subtraction | Left-to-right |
| **=**  **+= -= \*= /= %=** | Assignment Operator  Shorthand Operators | Right-to-left |

**Higher Precedence**

Numeric Data Type Conversion

When *operands of an operation* have differing types:

* If one of the operands is **double**, convert the other to **double**
* Otherwise, if one of them is **float**, convert the other to **float**
* Otherwise, if one of them is **long**, convert the other to **long**
* Otherwise, convert both into **int**

When *value is assigned* to a variable of differing types:

* **Widening (Promotion):** 
  + Value has a smaller range compared to the variable (e.g. double = int)
  + Converted automatically
* **Narrowing (Demotion):**
  + Value has a **larger range** compared to the variable (e.g. int = double)
  + **Explicit type casting is needed. Else error**

**int i = 3141;**

**double pi;**

**pi = i/1000 //prints 3.000000**

**pi = i/1000.0 //prints 3.141000**

Strings

* A **String** object is immutable: Any method that modifies the **String** object actually constructs a new **String** object with the updated information.
* The **toString()** method is automatically invoked when an instance is printed

**//The following are equivalent**

**System.out.println(myBall);**

**System.out.println(myBall.toString());**

Control Statements

*If-else Switch*

**if (a > b) {**

**...**

**}**

**else {**

**...**

**}**

* Must be a *boolean* expression
* Unlike C, integer values are NOT valid
* Expression in **switch()** must evaluate to a value of **char**, **byte**, **short** or **int** type

**switch (a) {**

**case 1:**

**...  
 break;**

**case 2:**

**case 3:**

**...**

**default:**

**}**

*While loop Do-While loop*

**do {**

**... //body**

**} while (a > b);**

**while (a > b) {**

**... //body**

**}**

*For loop*

**for (initialization; condition; update) {**

**... //body**

**}**

Boolean operators

|  |  |  |
| --- | --- | --- |
|  | **Operators** | **Description** |
| **Comparison Operators** | **<** | lesser than |
| **>** | larger than |
| **<=** | lesser or equal |
| **>=** | larger or equal |
| **==** | equal |
| **!=** | not equal |
|  | | |
| **Logical Operators** | **&&** | AND |
| **||** | OR |
| **!** | NOT |
| **^** | EXCLUSIVE-OR |

Input

**import java.util.Scanner;**

**PACKAGE**

***//Declaration of Scanner "variable"***

**Scanner *scVar = \_initialization\_;***

***//Functionality provided***

***scVar.nextInt*();**

***scVar.nextDouble*();**

**......**

**SYNTAX**

Package

**import java.util.\*;**

**class TemperatureInteractive {**

**public static void main(String[] args) {**

**double fahrenheit, celcius;**

**Scanner myScanner = new Scanner(System.in);**

***System.out.print*("Enter temperature in Fahrenheit: ");**

**fahrenheit = myScanner.*nextDouble*();**

**celcius = (5.0 / 9) \* (fahrenheit – 32);**

***System.out.println*("Celcius: " + celcius);**

**}**

**}**

Initialization

Usage

**System.in** *is the standard input, i.e. keyboard*

Output

**SYNTAX**

***//Functionality provided***

***System.out.print*( *output\_string* );**

***System.out.println*( *output\_string* );**

***System.out.printf*( *format\_string, [items]* );**

**System.out.print("ABC");**

**System.out.println("DEF");**

**System.out.printf("Very C-like %3.2f\n", 3.14159);**

***%[-][W].[P]*type**

**-:** For left alignment

**W:** For width

**P:** For precision

**SYNTAX**

|  |  |
| --- | --- |
| ***%d*** | *for integer value* |
| ***%f*** | *for double floating point value* |
| ***%s*** | *for string* |
| ***%b*** | *for boolean value* |
| ***%c*** | *for character value* |

Methods

Parameters in Java are passed by value. A copy of the actual argument is created upon method invocation. The method parameter and its corresponding actual parameter are two independent variable.

Objects in Java are passed by reference. A method modifies the actual argument. An object reference data type is needed (similar to pointer in C)

A method can also be called

* An **accessor** if it accesses (retrieves) the value of an object’s attribute to perform some tasks but does not alter the objects attribute.
* A **mutator** if it mutates (modifies) the value of an object’s attribute

Instance Method

* Methods within a class
* You need an object reference of the right type to invoke these methods
* These methods have access to the attributes in the object automatically

Static Method

* Static methods have no access to object attributes (i.e. there is no additional data other than the parameter)
* Similar to function in C
* Distinguished by the modifier "**static**" in front of the method return type

Constructors

* Called when an object is created
* Useful for initializing the attributes of an object
* Constructor has NO return type.
* Constructor has the same name as the class

Driver/client class

**all classes** are descendant of a predefined class called "**Object**"

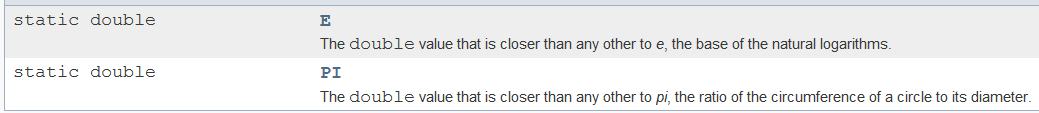
Driver/client class

Class containing the main() method

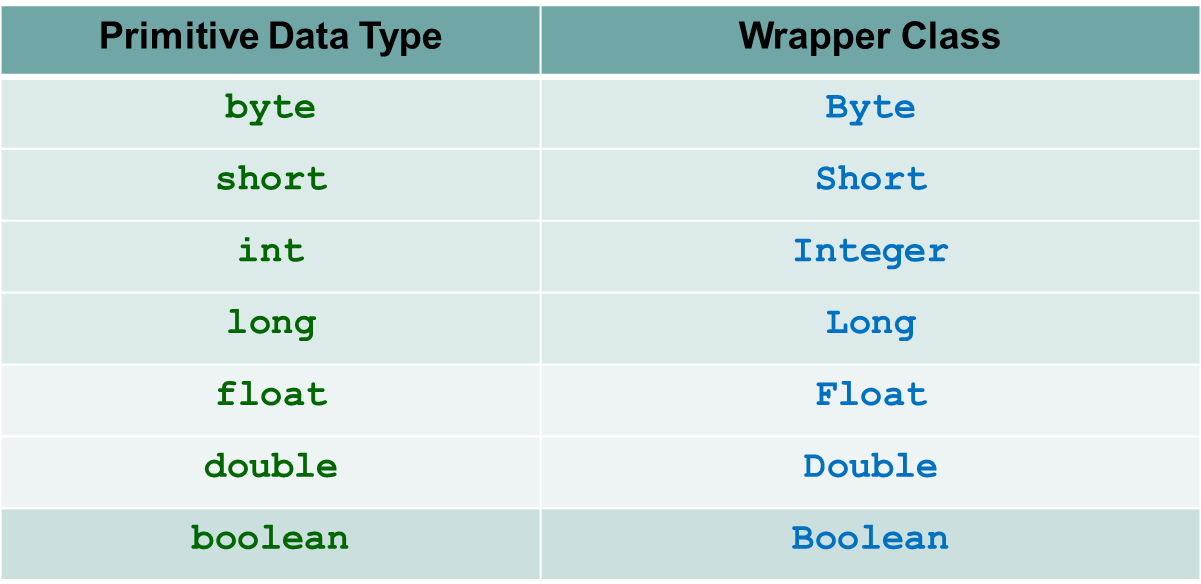
Service class

* Others non-driver classes within the same source code.
* They instantiate objects used in the driver class.

Math class

* Overloading methods – 2 or more methods within the same class with the same name but different parameters. (I.e. different parameters could be passed in)
* In the definition of every **Math** method, the keyword “**static**” appears. 🡺 This means that no object (instance) of the **Math** class is required to use the method.
* Any **Math** method is called by preceding its name with the name of the class: E.g. **Math.sqrt(area)**
* The **Math** class also has two class attributes
* Every instance of a class shares a class attribute.

Wrapper class



Wrapper class is used when we need an object representation of the primitive data types.

**Integer aInt = new Integer(4);**

**Integer bInt = 4; //auto boxing**

**sumInt = aInt.*intValue*();**

**sumInt = bInt; //auto unboxing**

Inheritance class

**class B extends A {…}**

* If **class B** is derived from **class A** (i.e. new class inherits all of the attributes and methods from the other class), then
* **class B** is called a **child (subclass or derived class)** of **class A**
* **class A** is called a **parent (superclass)** of **class B**
* **“Protected”** keyword allows subclass to access attributes directly
* Unlike normal methods, constructors are **not inherited** 🡺You need to define constructor(s) for the subclass
* The "**super**" keyword allows us to use the methods in the superclass directly including constructors.
* If you make use of superclass's constructor, make sure it is the **first statement** in the method body

**class SavingAcct extends BankAcct {**

**protected double \_rate; //interest rate**

**public SavingAcct(int aNum, double bal, double rate) {**

**super(aNum, bal); //Using constructor from parent**

**\_rate = rate; //Adding a new attribute**

**}**

**}**

* The "**super**" keyword can be used to invoke superclass's method as well

**class SavingAcct extends BankAcct {**

**protected double \_rate; //interest rate**

**public void *payInterest*() { …… }**

**public void *print*() {**

**super.*print*(); //print() is a method of BankAcct**

***System.out.printf*("Interest: %.2f%%\n", \_rate); //Addition**

**}**

**}**

* Whenever a super class object is expected, a sub class object **is acceptable as substitution!** (**Caution:** the **reverse is NOT true** as subclass may have additional modifications**)**
* Hence, all existing functions that works with the super class objects will work on sub class objects with **no modification**!

Generic Type

Important restrictions:

* The generic type can be substituted by **reference data type only**. I.e. Primitive data types are NOT allowed
* Need to use Wrapper class for the primitive data type
* Make sure the code is valid for all possible data types

**class Pair <S,T> {**

**private S \_first;**

**private T \_second;**

**public Pair(S a, T b) {**

**\_first = a;**

**\_second = b;**

**}**

**public S getFirst() { return \_first; }**

**public T getSecond() { return \_second; }**

**}**

**class Pair <T> {**

**private T \_first, \_second;**

**public Pair( T a, T b ) {**

**\_first = a;**

**\_second = b;**

**}**

**public T getFirst() { return \_first; }**

**public T getSecond() { return \_second; }**

**}**

You can have multiple generic data types

**Convention:** Use capital single letter for the generic data type

**//Old school generic implementation**

**class Pair {**

**private object \_first, \_second;**

**public Pair( object a, object b ) {**

**\_first = a;**

**\_second = b;**

**}**

**public object getFirst() { return \_first; }**

**public object getSecond() { return \_second; }**

**}**

Software Engineering Issues

* **Data abstraction** asks that you think in terms of what you can do to a collection of data independently of how you do it
* **Data structure** is a construct that can be defined within a programming language to store a collection of data. E.g. Arrays
* **Abstract data type (ADT)** is a collection of data & a specification on the set of operations on that data
  + Typical operations on data are: *add*, *remove*, and *query* (in general, management of data)
  + Specification indicates what ADT operations do, but not how to implement them
  + Data structures are part of an ADT’s implementation

Interface

* Java interfaces provide a way to specify and indicate common behaviors (methods) for a set of (perhaps unrelated) classes
* Explanations of methods used are described in the interface class. By default all are methods are public in the interface class.
* A class is said to implement the interface if it provides implementations for (contains) all of the methods as defined in the interface

// package in java.lang;

**public interface Comparable <T> {**

**int compareTo (T other);**

**}**

**class Shape implements Comparable <Shape> {**

**static final double PI = 3.14;**

**double area() {...};**

**double circumference() { ... };**

**int compareTo(Shape x) {**

**if (this.area() == x.area())**

**return 0;**

**else if (this.area() > x.area())**

**return 1;**

**else**

**return -1;**

**}**

**}**

Decimal format class

* Import java.text package
* Note that **df.format(x)** does not change the value in **x**. It merely displays the value in **x** in the specified format.

**public String toString() {**

**DecimalFormat df = new DecimalFormat("0.00"); // 2 dec. places**

**if (imagpart() == 0)**

**return (df.format(realpart()) + "");**

**else if (imagpart() < 0)**

**return (df.format(realpart()) + "" + df.format(imagpart()) + "i");**

**else**

**return (df.format(realpart()) + "+" + df.format(imagpart()) + "i");**

**}**

**//Alternative via Math**

**score = (double)Math.round(score\*100)/100; //2 decimal places**

**System.out.printf("Final score: %.2f", score);**

Array

**//Initialise array reference**

**datatype[] array\_reference;**

**//Instantiate array to datatype**

**arr\_reference = new datatype [VALUE];**

**//Initialise reference and instantiate datatype**

**datatype[] arr\_reference = new datatype [VALUE];**

**//Length of array (public attribute of array reference type)**

**Length=array\_reference.length;**

**//Enhanced for loop for all elements**

**for (dataype i: array\_reference){}**

**//Array as a parameter in method definition**

**public method(datatype[] array\_reference, …){…}**

**//Returning array from method**

**Public dataype[] method(…){…}**

* Note: As the reference to the array is passed into a method, any modification of the element in the method will affect the actual array

**//Swap function for array**

**public void *swap*(datatype[] array\_reference, int i, int j) {**

**datatype temp = array\_reference [i];**

**array\_reference [i] = array\_reference [j];**

**array\_reference [j] = temp;**

**}**

Note:

* To obtain length of a **String** object, we use the **length()** method (E.g. **str.length()**)
* To obtain length (size) of an array, we use the **length** attribute without bracket. (E.g. **arr.length**)
* When you have an array of objects, it’s very common to forget to instantiate the array’s objects. Results in **java.lang.NullPointerException**

**Point[] array = new Point[3];**

**for (int i=0; i<array.length; i++) {**

**array[i] = new Point(); //This is often forgotten!**

**array[i].setLocation(1,2);**

**}**

Vectors

**import java.util.Vector;**

**//Declaration of a Vector reference**

**Vector<datatype> myVector;**

**//Initialize an empty Vector object**

**myVector = new Vector<datatype>();**

**//Declare and initialize**

**Vector<datatype> myVector = new Vector<datatype>();**

**//Vector contains toString() and get() methods**

**System.out.println(myVector.get(i));**

**//Enhanced for-loop**

**For (datatype i: myVector){…}**

|  |  |
| --- | --- |
| **Commonly used vector methods** | |
| **boolean** | ***isEmpty*()** Tests if this vector has no components. |
| **int** | ***size*()** Returns the number of components in this vector**.** |
| **boolean** | ***add*(datatype o)**   Appends the specified element to the end of this Vector. |
| **void** | ***add*(int index, datatype element)**  Inserts the specified element at the specified position in this Vector. |
| **dataype** | ***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. |
| **datatype** | ***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.* |

Exception

* Instead of deciding how to deal with an error, Java provides the **exception** mechanism:

1. Indicate an error (exception event) has occurred
2. Let the user decide how to handle the problem in a separate section of code specific for that purpose
3. Crash the program if the error is not handled

* Exception mechanism consists of two components:

1. Exception indication
2. Exception handling

* a number of useful predefined exception classes:
  + ArithmeticException
  + NullPointerException
  + IndexOutOfBoundsException
  + IllegalArgumentException

**Exception indication**

**public datatype *method*(parameters) throws IllegalArgumentException {**

**if (condition) {**

**//Declare, initialise and instantiate exception object**

**IllegalArgumentException exObj**

**= new IllegalArgumentException(parameter + " is invalid!");**

**//Throws the exception**

**throw exObj;**

**}**

**/\*Method body\*/**

**}**

**//The above portion could be shortened as follows**

**If (condition){throw new IllegalArgumentException(parameter + " is invalid!");}**

**Exception handling**

**public static void main(String[] args){**

**public datatype *method*(parameters) throws IllegalArgumentException {**

**if (condition) {**

**throw new IllegalArgumentException(parameter + " is invalid!");**

**}**

**/\*Method Body\*/**

**}**

**try {**

**System.out.println("Before method()");**

**System.out.println("Ans = " + method(input));**

**System.out.println("After method()");**

**}**

**catch (IllegalArgumentException expObj){**

**System.out.println("In Catch Block");**

**System.out.*println*(expObj.getMessage());**

**}**

**finally {**

**System.out.println("Finally!");**

**}**

**}**

**Try**: error could occur there

**Catch**: if the error occurred do this

List node

**import java.util.\*;**

**class ListNode <E> {**

**protected E element;**

**protected ListNode <E> next;**

**/\* constructors \*/**

**public ListNode(E item) { element = item; next = null; }**

**public ListNode(E item, ListNode <E> n) { element = item; next=n;}**

**/\* get the next ListNode \*/**

**public ListNode <E> getNext() {**

**return this.next;**

**}**

**/\* get the element of the ListNode \*/**

**public E getElement() {**

**return this.element;**

**}**

**}**