OBJECT-ORIENTED LANGUAGE AND THEORY

5. MEMORY MANAGEMENT AND CLASS ORGANIZATION

Nguyen Thi Thu Trang
trangntt@soict.hust.edu.vn

1

Outline

1. Memory management in Java
2. Class organization
3. Utility classes in Java

2

3

1. Memory management in Java

- Java does not use pointer, hence memory addresses can not be overwritten accidentally or intentionally.
- The allocation or re-allocation of memory, management of memory that is controlled by JVM, are completely transparent with developers.
- Developers do not need to care about the allocated memory in heap in order to free it later.

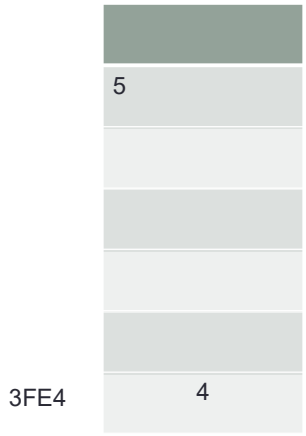
3

4

```
byte i;
i = 4;

byte *j;

j+2
```



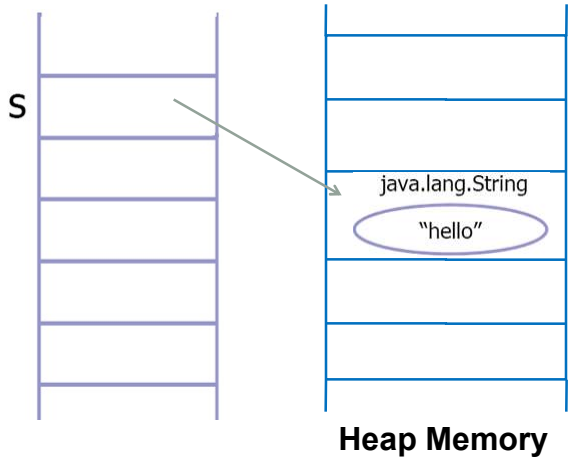
4

5

1.1. Heap memory

```
String s = new String("hello");
```

- Heap memory is used to write information created by **new** operator.



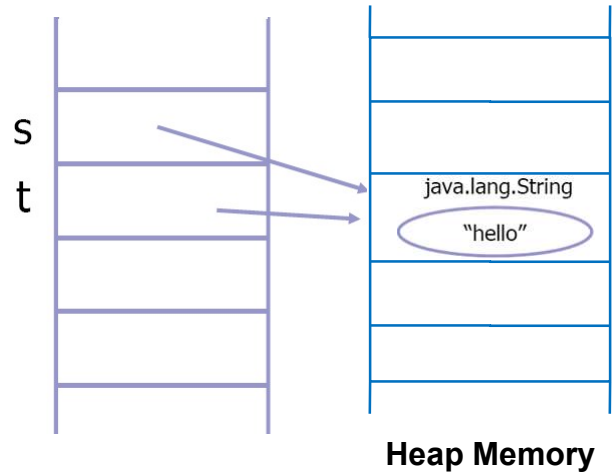
5

6

1.1. Heap memory (2)

```
String s = new String("hello");
String t = s;
```

- Heap memory is used to write information created by **new** operator.



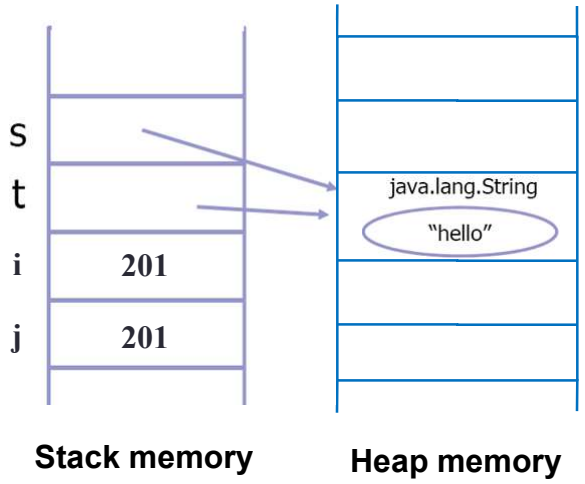
6

7

1.2. Stack memory

```
String s = new String("hello");
String t = s;
int i = 201;
int j = i;
```

- Local value in Stack memory is used as a reference pointer to Heap
- Value of primitive data is written directly in Stack



7

8

1.3. Garbage collector (gc)

- The garbage collector sweeps through the JVM's list of objects periodically and reclaims the resources held by unreferenced objects
- All objects that have no object references are eligible for garbage collection
  - References out of scope, objects to which you have assigned null, and so forth
- The JVM decides when the gc is run
  - Typically, the gc is run when memory is low
  - May not be run at all
  - Unpredictable timing

8

9

Working with the garbage collector

- You cannot prevent the garbage collector from running, but you can request it to run soon
  - `System.gc();`
  - This is only a request, not a guarantee
- The `finalize()` method of an object will be run immediately before garbage collection occurs
  - This method should only be used for special cases (e.g. cleaning up memory allocation from native calls) because of the unpredictability of the garbage collector
  - Things like open sockets, files, and so forth should be cleaned up during normal program flow before the object is dereferenced

9

10

Java destructors?

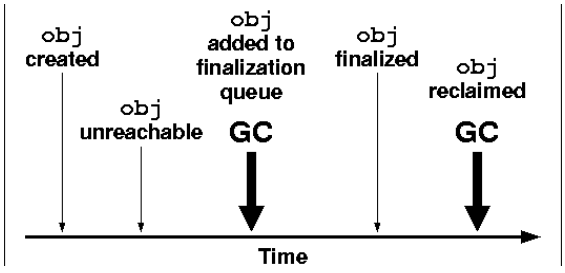
- Java does not have the concept of a destructor for objects that are no longer in use
- Deallocation of memory is done automatically by the JVM through the finallize() method
  - A background process called the garbage collector reclaims the memory of unreferenced objects
  - The association between an object and an object reference is severed by assigning another value to the object reference, for example:
    - objectReference = null;
  - An object with no references is a candidate for deallocation during garbage collection

10

11

`finalize()` method

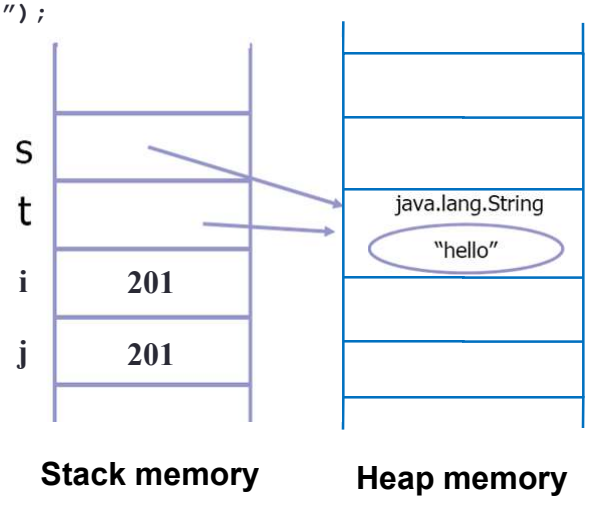
- Any class has method finalize() – that is executed right after the garbage collection process takes place (considered as destructor in Java despite not)
- Override this method in some special cases in order to "self-clean" used resources when objects are freed by `gc`
  - E.g. pack socket, file,... that should be handled in the main thread before the objects are disconnected from reference.



11

12

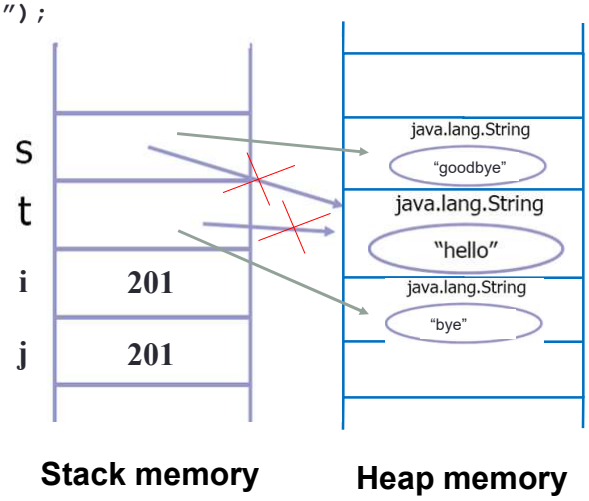
```
String s = new String("hello");
String t = s;
int i = 201;
int j = i;
s = new String("goodbye");
t = new String("bye");
```



12

13

```
String s = new String("hello");
String t = s;
int i = 201;
int j = i;
s = new String("goodbye");
t = new String("bye");
```



13

14

Memory Management in Java (Method and variables)

14

15

1.4. Object comparison

- **Primitive data types:** == checks whether their values are the equal

```
int a = 1;
int b = 1;
if (a==b)... // true
```

- **Objects**: == checks whether two objects are unique ~ whether they **refer to the same object**

```
Employee a = new Employee(1);
Employee b = a;
if (a==b)... // true
```

```
Employee a = new Employee(1);
Employee b = new Employee(1);
if (a==b)... // false
```

15

16

equals() method

- For primitive data types → does not exist.
- For objects: every object has this method
  - Compares values of objects

```
public class Equivalence {
 public static void main(String[] args) {
      Integer n1 = new Integer(47);
      Integer n2 = new Integer(47);
      System.out.println(n1 == n2);
      System.out.println(n1.equals(n2));
 }
}
```

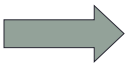
```
false
true
```

16

17

equals() method of your class

```
class Value {
 int i;
 public Value(int i) { this.i = i;}
}
public class EqualsMethod2 {
 public static void main(String[] args) {
   Value v1 = new Value(10);
   Value v2 = new Value(10);
   System.out.println(v1.equals(v2));
 }
}
```



17

18

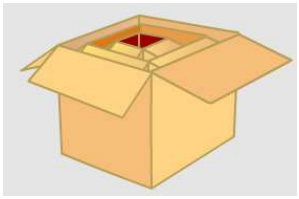
Outline

1. Memory management in Java
2. Class organization
3. Utility classes in Java

18

Class organization with Package

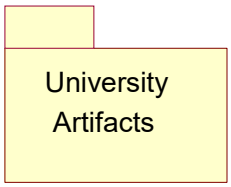
- Package is as a folder that helps:
  - Organize and locate easily the classes and use classes in a appropriate manner
  - Avoid conflict in naming classes
    - Different packages can contains classes with same name
  - Protect classes, data and methods in a larger area compared to relation between classes
- A package can also contain another package
  - "com" package contains "google" package
    - com.google



19

Package in UML

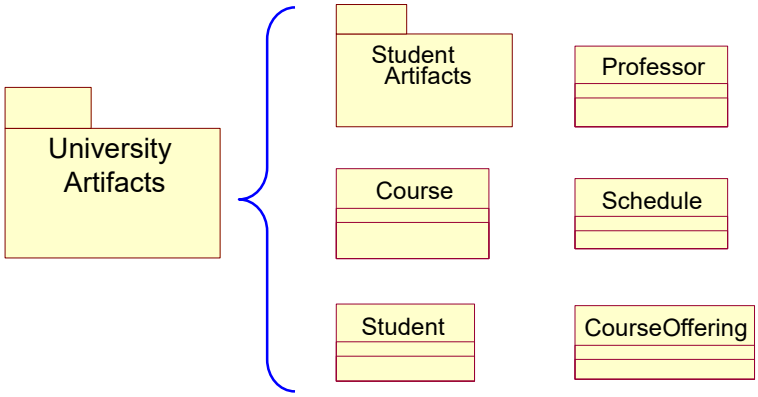
- A general purpose mechanism for organizing elements into groups.
- A model element that can contain other model elements.
- A package can be used:
  - To organize the model under development
  - As a unit of configuration management



20

A Package Can Contain Classes

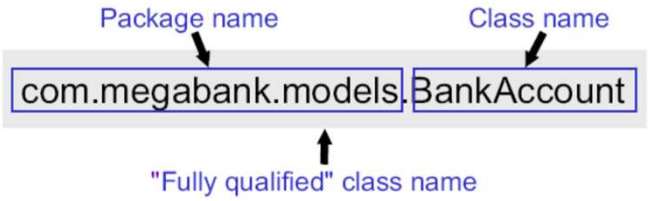
- The package, University Artifacts, contains one package and five classes.



21

Fully qualified class name

- A fullname of a class includes package name and class name:



22

- package oolt.hedspi;
- class AS1{
  - int as11;
  - void as1_method(){
    - IS1 as1 = new IS1();
    - is1.is1_method();
  - }
- }

- package oolt.hedspi;
- class IS1{
  - void is1_method(){}
- }

23

2.1. References between classes

- In the same package: use class name
- In different packages: must provide the full-name of class defined in other packages.
- Example:

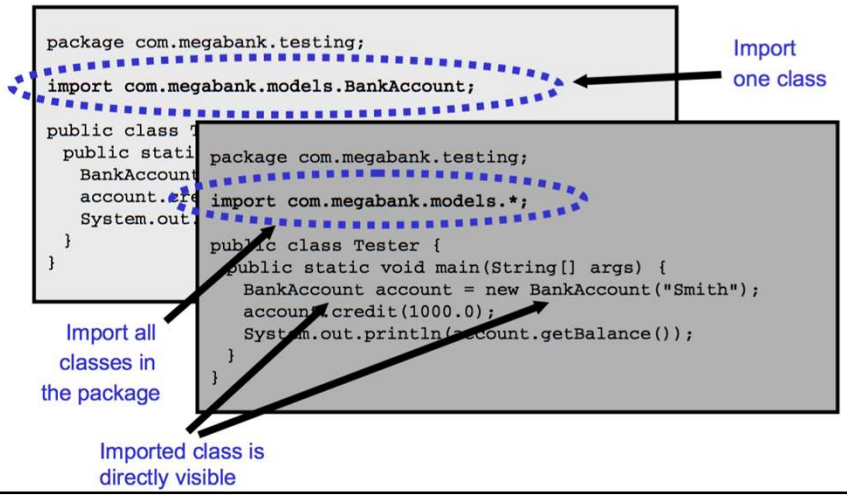
```
package oolt.hedspi;
public class HelloNameDialog{
  public static void main(String[] args){
    String result;
    result = javax.swing.JOptionPane.
          showInputDialog("Please enter your name:");
    javax.swing.JOptionPane.
          showMessageDialog(null, "Hi "+ result + "!");
  }
}
```

24

25

Using import command

To import packages or classes to make other classes directly visible to your class



25

26

More example

```
package oolt.hedspi;
public class HelloNameDialog{
 public static void main(String[] args){
   System.out.print("Hello world!");
 }
}
```

26

27

2.2. Packages in Java

- java.applet
- java.awt
- java.beans
- java.io
- java.lang
- java.math
- java.net
- java.nio
- java.rmi
- java.security
- java.sql
- java.text
- java.util
- javax.accessibility
- javax.crypto
- javax.imageio
- javax.naming
- javax.net
- javax.print
- javax.rmi
- javax.security
- javax.sound
- javax.sql
- javax.swing
- javax.transaction
- javax.xml
- org.apache.commons
- org.ietf.jgss
- org.omg.CORBA
- org.omg.IOP
- org.omg.Messaging
- org.omg.PortableInterceptor
- org.omg.PortableServer
- org.omg.SendingContext
- org.omg.stub.java.rmi
- org.w3c.dom
- org.xml

27

28

Basic packages in Java

- **java.lang**
  - Provides classes that are fundamental to the design of the Java programming language
  - Includes wrapper classes, String and StringBuffer, Object, and so on
  - Imported implicitly into all classes
- **java.util**
  - Contains the collections framework, event model, date and time facilities, internationalization, and miscellaneous utility classes
- **java.io**
  - Provides for system input and output through data streams, serialization and the file system

28

29

Basic packages in Java

- **java.math**
  - Provides classes for performing arbitrary-precision integer arithmetic and arbitrary-precision decimal arithmetic
- **java.sql**
  - Provides the API for accessing and processing data stored in a data source (usually a relational database)
- **java.text**
  - Provides classes and interfaces for handling text, dates, numbers, and messages in a manner independent of natural languages
- **javax.swing**
  - Provides classes and interfaces to create graphics

29

30

Sample package: java.lang

- **Basic Entities**
  - Class, Object, Package, System
- **Wrappers**
  - Number, Boolean, Byte, Character, Double, Float, Integer, Long, Short, Void
- **Character and String Manipulation**
  - Character.Subset, String, StringBuffer, Character.UnicodeBlock
- **Math Functions**
  - Math, StrictMath
- **Runtime Model**
  - Process, Runtime, Thread, ThreadGroup, ThreadLocal, InheritableThreadLocal, RuntimePermission
- **JVM**
  - ClassLoader, Compiler, SecurityManager
- **Exception Handling**
  - StackTraceElement, Throwable
- Also contains Interfaces, Exceptions and Errors

30

31

Outline

1. Memory management in Java
2. Class organization
3. Utility classes in Java

31

32

3.1. Wrapper class

- Primitives have no associated methods; there is no behavior associated with primitive data types
- Each primitive data type has a corresponding class, called a wrapper
  - Each wrapper object simply stores a single primitive variable and offers methods with which to process it
  - Wrapper classes are included as part of the base Java API

32

33

Wrapper classes

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

33

34

Converting data type

- Use **`toString()`** to convert number values to string.
- Use **`<type>Value()`** to convert an object of a wrapper class to the corresponding primitive value

```
Float objF = new Float("4.67");
float f = objF.floatValue(); // f=4.67F
int i = objF.intValue();  //i=4
```

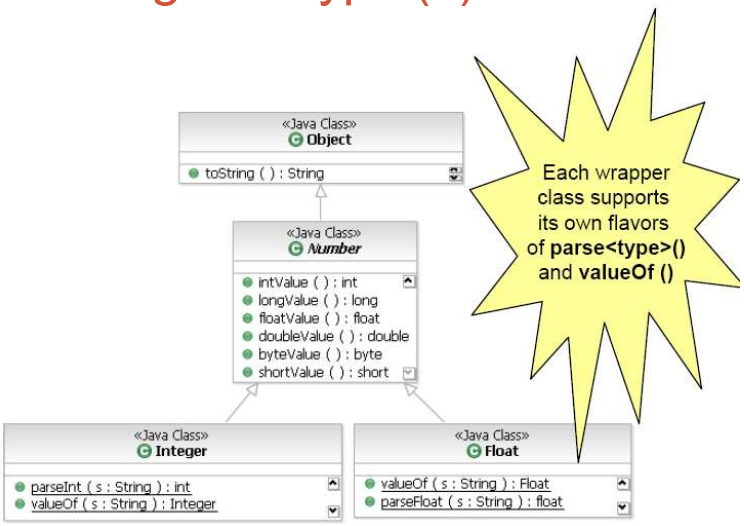
- Use **`parse<type>()`** and **`valueOf()`** to convert string to number values.

```
int i = Integer.parseInt("123");  //i=123
double  d = Double.parseDouble("1.5") ; // d=1.5
Double objF2 = Double.valueOf("-36.12");
long l = objF2.longValue(); // l=-36L
```

34

35

Converting data type (2)



35

36

Constants

- **Boolean**
  - Boolean FALSE
  - Boolean TRUE
- **Byte**
  - byte MIN_VALUE
  - byte MAX_VALUE
- **Character**
  - int MAX_RADIX
  - char MAX_VALUE
  - int MIN_RADIX
  - char MIN_VALUE
  - Unicode classification constants
- **Double**
  - double MAX_VALUE
  - double MIN_VALUE
  - double NaN
  - double NEGATIVE_INFINITY
  - double POSITIVE_INFINITY

- **Float**
  - float MAX_VALUE
  - float MIN_VALUE
  - float NaN
  - float NEGATIVE_INFINITY
  - float POSITIVE_INFINITY
- **Integer**
  - int MIN_VALUE
  - int MAX_VALUE
- **Long**
  - long MIN_VALUE
  - long MAX_VALUE
- **Short**
  - short MIN_VALUE
  - short MAX_VALUE

36

37

Example

```
double d = (new Integer(Integer.MAX_VALUE)).
                                        doubleValue();
System.out.println(d); // 2.147483647E9

String input = "test 1-2-3";
int output = 0;
for (int index = 0; index < input.length(); index++)
{
   char c = input.charAt(index);
   if (Character.isDigit(c))
     output = output * 10 + Character.digit(c, 10);
}
System.out.println(output);     // 123
```

37

38

3.2. String

- The String type is a class, not a primitive data type
- A String literal is made up of any number of characters between double quotes:

```
String a = "A String";
String b = "";
```

- A String object can be initialized in other ways:

```
String c = new String();
String d = new String("Another String");
String e = String.valueOf(1.23);//"1.23"
String f = null;
```

38

39

a. String concatenation

- The + operator concatenates Strings:

```
String a = "This" + " is a " + "String";
//a = "This is a String"
```

There are more efficient ways to concatenate Strings (this will be discussed later)

- Primitive data types used in in a call to println() are automatically converted to String

```
System.out.println("answer = " + 1 + 2 + 3);
System.out.println("answer = " + (1+2+3));
```

→ Do two above commands print out the same output?

39

40

b. Methods of String

Strings are objects; objects respond to messages

- ✓ Use the dot (.) operator to send a message
- ✓ String is a class, with methods

```
String name = "Joe Smith";
name.toLowerCase();          // "joe smith"
name.toUpperCase();          // "JOE SMITH"
"Joe Smith ".trim();         // "Joe Smith"
"Joe Smith".indexOf('e');    // 2
"Joe Smith".length();        // 9
"Joe Smith".charAt(5);       // 'm'
"Joe Smith".substring(5);    // "mith"
"Joe Smith".substring(2,5);  // "e S"
```

40

41

c. String comparison

- `oneString.equals(anotherString)`
  - Tests for equivalence
  - Return `true` or `false`

```
String name = "Joe";
if ("Joe".equals(name))
      name += " Smith";
```

- `oneString.equalsIgnoreCase(anotherString)`
  - Case insensitive test for equivalence

```
boolean same = "Joe".equalsIgnoreCase("joe");
```

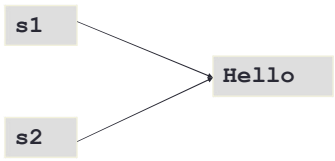
- `oneString == anotherString` is problematic
  - Compare two objects

41

42

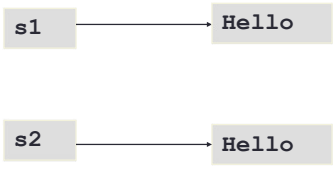
c. Comparing two Strings (2)

```
String s1 = new String("Hello");
String s2 = s1;
```
`(s1==s2)` returns true




```
String s1 = new String("Hello");
String s2 = new String("Hello");
```
`(s1==s2)` returns false

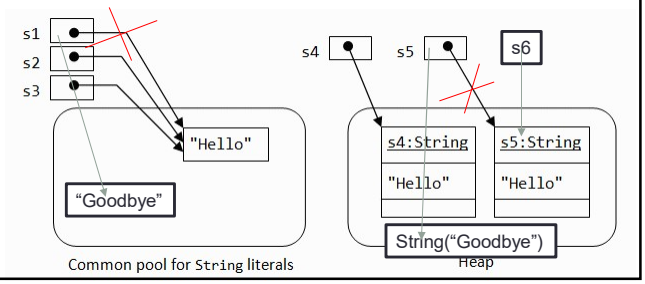
s1.equals(s2) return true



42

String Literal vs. String Object

```
String s1 = "Hello";                // String literal
String s2 = "Hello";                // String literal
String s3 = s1; // same reference
String s4 = new String("Hello");  // String object
String s5 = new String("Hello");  // String object
String s6 = s5;
s5 = new String("Goodbye");
s1 = "Goodbye";
```



43

44

- String str = "";
- for (int i=0; i<1.000.000; i++){
  - //read a line from a file
  - str += line;
- }

- StringBuffer str = "";
- for (int i=0; i<1.000.000; i++){
  - //read a line from a file
  - str.append(line);
- }

44

45

3.3. StringBuffer/StringBuilder

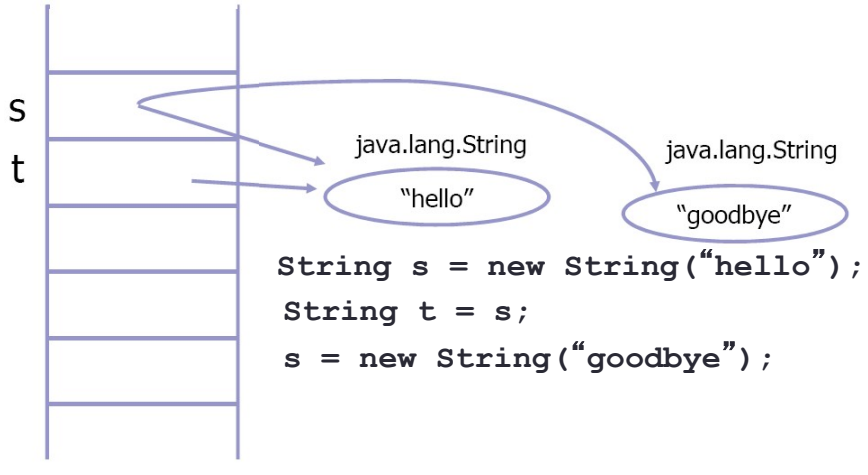
- String is an immutable type:
  - Object does not change the value after being created → Strings are designed for not changing their values.
  - Concatenating strings will create a new object to store the result → String concatenation is memory consuming.
- StringBuffer/StringBuilder is a mutable type:
  - Object can change the value after being created

=> String concatenation can get very expensive, only use in building a simple String

45

46

3.3. StringBuffer (2)



46

47

3.3. StringBuffer (3)

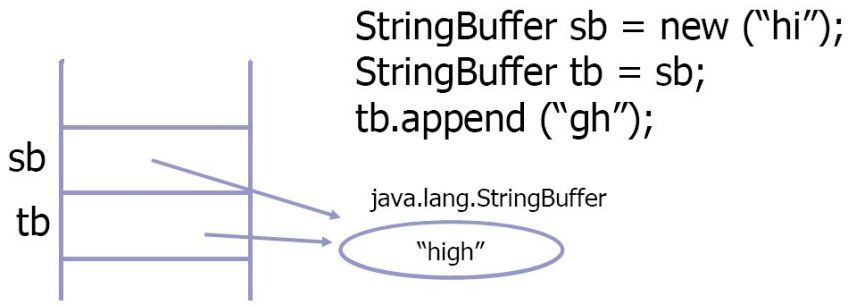
- **`StringBuffer`**:
  - Provides String object that can change the value → Use **`StringBuffer`** when:
    - Predict that characters in the String can be changed
    - When processing a string, e.g. reading text data from a text file or building a String through a loop
  - Provides a more efficient mechanism for building and concatenating strings:
    - String concatenation is often done by compiler in class **`StringBuffer`**

47

48

3.3. StringBuffer (4)

- Changing attribute: If an object is changed, all the relations with the object will receive the new value.



48

3.3. StringBuffer (5)

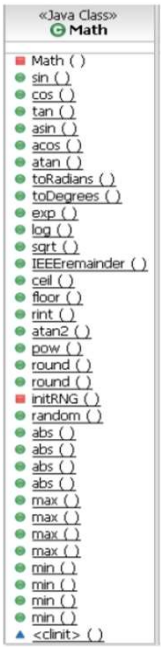
- If we create a String by a loop, we should use **`StringBuffer`**

```
StringBuffer buffer = new StringBuffer(15);
buffer.append("This is ") ;
buffer.append("String") ;
buffer.insert(7," a") ;
buffer.append('.');
System.out.println(buffer.length());      // 17
System.out.println(buffer.capacity());    // 32
String output = buffer.toString() ;
System.out.println(output); // "This is a String."
```

49

3.4. Math class

- java.lang.Math provides static data:
  - Math constants:
    - Math.E
    - Math.PI
  - Math functions:
    - max, min...
    - abs, floor, ceil…
    - sqrt, pow, log, exp…
    - cos, sin, tan, acos, asin, atan…
    - random



50

3.4. Math class (2)

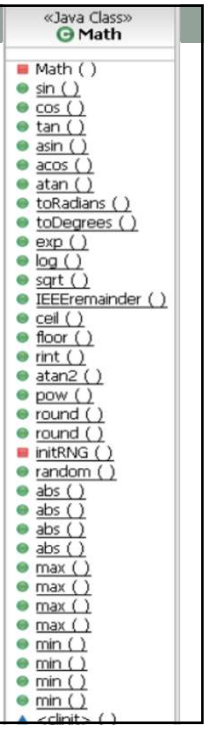
- Most of functions receive arguments with type **`double`** and also return values with tye **`double`**
  - Example:

$$e^{\sqrt{2\pi}}$$

```
Math.pow(Math.E,
     Math.sqrt(2.0*Math.PI))
```

Or:

```
Math.exp(Math.sqrt(2.0*Math.PI))
```



51