Software Engineering Department
School of Information & Communication Technology
Hanoi University of Science and Technology

Objected Oriented Programming

Chapter 04. Techniques for Class Construction and Object Usage

Cao Tuan Dung
dungct@soict.hut.edu.vn

Goals

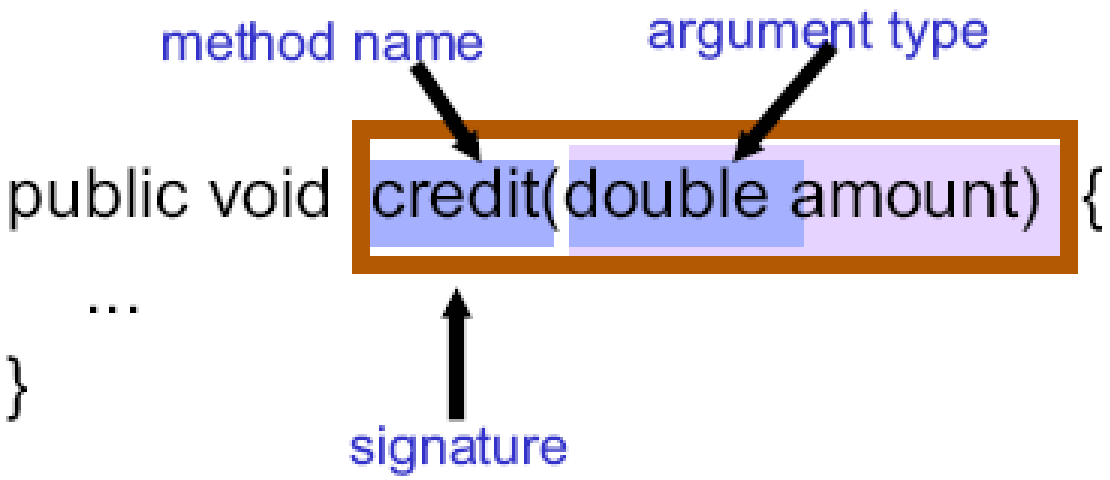
- Understand notions, roles and techniques for overloading methods and overloading constructors
- Object member, class member
- How to manage memory and objects in Java
- How to pass arguments of functions
- How to use package, some utility classes in Java: Wrapper class, Math, System, String vs. StringBuffer

Outline

1. Method overloading
2. Object member and class member
3. Memory management in Java
4. Passing arguments to methods
5. Some utility classes in Java

Method recalls

- Each method has it own signature
- A method signature is composed of:
  - Method's name
  - Number of arguments and their types



1.1. Method overloading

- Method Overloading: Methods in a class might have the same name but their signature must be different:
  - Numbers of arguments are different
  - If the numbers of arguments are the same, types of arguments must be different
- Advantages:
  - The same name describes the same task
  - Is easier for developers because they don’t have to remember too many method names. They remember only one with the appropriate arguments.

1.1. Method overloading (2)

- Example 1:
  - Method println() in System.out.println() has 10 declarations with different arguments: boolean, char[], char, double, float, int, long, Object, String, and one without argument.
  - Do not need to use different names (for example "printString“ or "printDouble“) for each data type to be displayed.

1.1. Method overloading (3)

- Example 2:

```
class MyDate {
  int year, month, day;
  public boolean setMonth(int m) { …}
  public boolean setMonth(String s) { …}
}
public class Test{
   public static void main(String args[]){
    MyDate d = new MyDate();
    d.setMonth(9);
    d.setMonth(”September”);
  }
}
```

Some notes on method overloading

- Methods are considered as overloading only if they belong to the same class
- Only apply this technique on functions describing the same kind of task; do not abuse
- When compiling, compilers rely on number or types of arguments to decide which appropriate method to call.

  → If there is no method or more than one method to call, an error will be reported.

Discussion

- Given a following method:
  `public double test(String a, int b)`
- Let select overloading functions of the given method from the list below::
  1. **`void test(String b, int a)`**
  2. **`public double test(String a)`**
  3. **`private int test(int b, String a)`**
  4. **`private int test(String a, int b)`**
  5. **`double test(double a, int b)`**
  6. **`double test(int b)`**
  7. **`public double test(String a, long b)`**

Discussion

```
void prt(String s) { System.out.println(s); }
void f1(char x) { prt("f1(char)"); }
void f1(byte x) { prt("f1(byte)"); }
void f1(short x) { prt("f1(short)"); }
void f1(int x) { prt("f1(int)"); }
void f1(long x) { prt("f1(long)"); }
void f1(float x) { prt("f1(float)"); }
void f1(double x) { prt("f1(double)"); }
```

- What will happens if we do as follows:
  - `f1(5);`
  - `char x='a'; f1(x);`
  - `byte y=0; f1(y);`
  - `float z = 0; f1(z);`…

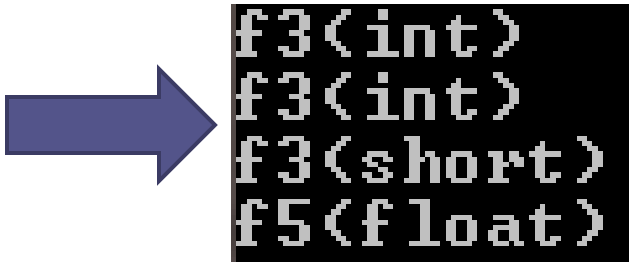
5 → int

```
f1(int)
f1(char)
f1(byte)
f1(float)
```

Discussion

```
void prt(String s) { System.out.println(s); }
void f2(short x) { prt("f3(short)"); }
void f2(int x) { prt("f3(int)"); }
void f2(long x) { prt("f5(long)"); }
void f2(float x) { prt("f5(float)"); }
```

- What will happen if we do as follows:
  - `f2(5);`
  - `char x='a'; f2(x);`
  - `byte y=0; f2(y);`
  - `float z = 0; f2(z);`



- What will happen if we call f2(5.5)?

Error: cannot find symbol: method f2(double)

1.2. Constructor overloading

- In different contexts, we need to create objects in different ways.
- → Need to build different constructors by using constructor overloading principles.

Example

```
public class BankAccount{
  private String owner;
  private double balance;
  public BankAccount(){owner = “noname”;}
  public BankAccount(String o, double b){
    owner = o; balance = b;
  }
}
public class Test{
  public static void main(String args[]){
    BankAccount acc1 = new BankAccount();
    BankAccount acc2 =
                      new BankAccount(“Thuy”, 100);
  }
}
```

1.3. this keyword

- Recall: “this” refers to the current object, it is used inside the class of the object that it refers to.
- It uses attributes or methods of object through “.” operator, for example:

```
public class BankAccount{
  private String owner;
  public void setOwner(String owner){
    this.owner = owner;
  }
  public BankAccount() { this.setOwner(“noname”); }
  …
  }
```

- Call another constructor of the class:
  - `this(danh_sach_tham_so); //neu co tham so`

- Example

```
public class Ship {
  private double x=0.0, y=0.0
  private double speed=1.0, direction=0.0;
  public String name;

  public Ship(String name) {
    this.name = name;
  }
  public Ship(String name, double x, double y) {
    this(name); this.x = x; this.y = y;
  }
  public Ship(String name, double x, double y,
    double speed, double direction) {
    this(name, x, y);
    this.speed = speed;
    this.direction = direction;
  }
  //continue…
```

```
//(cont.)
private double degreeToRadian(double degrees) {
  return(degrees * Math.PI / 180.0);
}
public void move() {
  move(1);
}
public void move(int steps) {
  double angle = degreesToRadians(direction);
  x = x + (double)steps*speed*Math.cos(angle);
   y = y + (double)steps*speed*Math.sin(angle);
}
public void printLocation() {
  System.out.println(name + " is at ("
                        + x + "," + y + ").");
}
} //end of Ship class
```

Outline

1. Method overloading
2. Object member and class member
3. Memory management in Java
4. Passing arguments to methods
5. Some utility classes in Java

Static parts

- Local variables declared in a function:
  - A local variable that is not declared as static will be created as a new instance everytime the function is called.
  - If a local variable is declared as statis, it is created only once (at the first function call), then from the second calls, it will be referred to by a stack pointer without having to create a new instance.

- Creating once/referring many times/saving value of the last reference

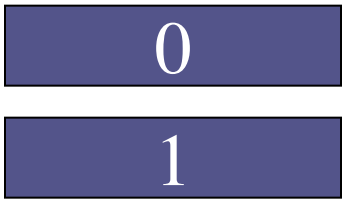
Static local variable

Static local variable:

```
void f()
{ static int x=0;
  x++;
}
```

First call: f()

Second call: f()

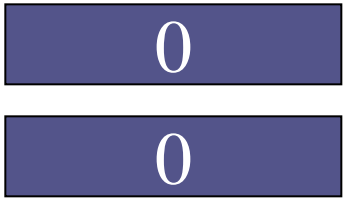


Non-static local variable

```
void f()
{ int x=0;
  x++;
}
```

First call: f()

Second call: f()

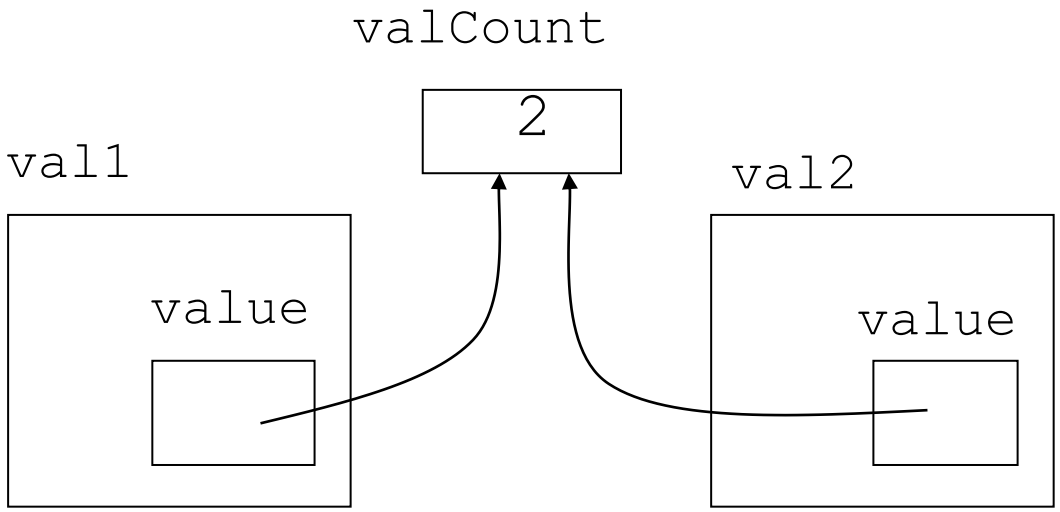


Instance member vs. Class member

- Attributes/methods can only be accessed via objects
- Each object has it own copy of an object's attribute
- **Values** of an attribute of different objects are different.

- Attributes/methods can be accessed through class
- All objects have the same copy of class attributes
- **Values** of a class attribute of different objects are the same.

Static parts: are shared between all objects



2.1. Static member

- In Java
  - Regular members are members of objects
  - Class members are declared as **static**
- Syntax for declaring static member:

```
chi_dinh_truy_cap static kieu_du_lieu tenBien;
```

- Example:

```
public class MyDate {
    public static long getMillisSinceEpoch() {
        …
    }
}
…
long millis = MyDate.getMillisSinceEpoch();
```

Example: Class JOptionPane in javax.swing

- Attributes

Field Summary	
static int	CANCEL_OPTION Return value from class method if CANCEL is choser
static int	CLOSED_OPTION Return value from class method if user closes window CANCEL_OPTION or NO_OPTION.
static int	DEFAULT_OPTION Type used for showConfirmDialog.
static int	ERROR_MESSAGE Used for error messages.

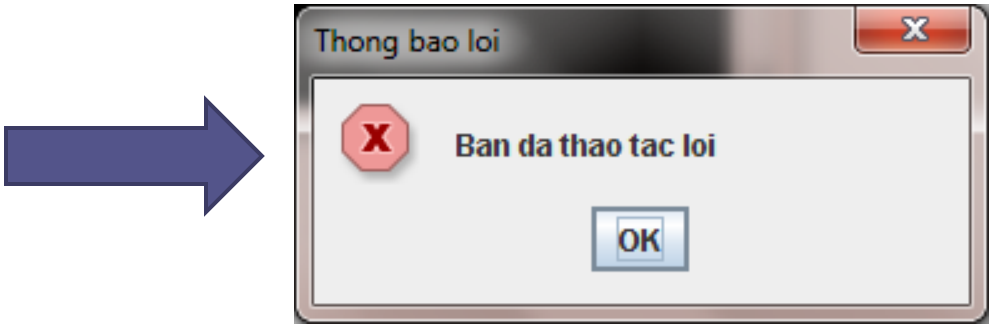
static int	WARNING_MESSAGE Used for warning messages.
static int	YES_NO_CANCEL_OPTION Type used for showConfirmDialog.
static int	YES_NO_OPTION Type used for showConfirmDialog.
static int	YES_OPTION Return value from class method if YES is chosen.

- Methods:

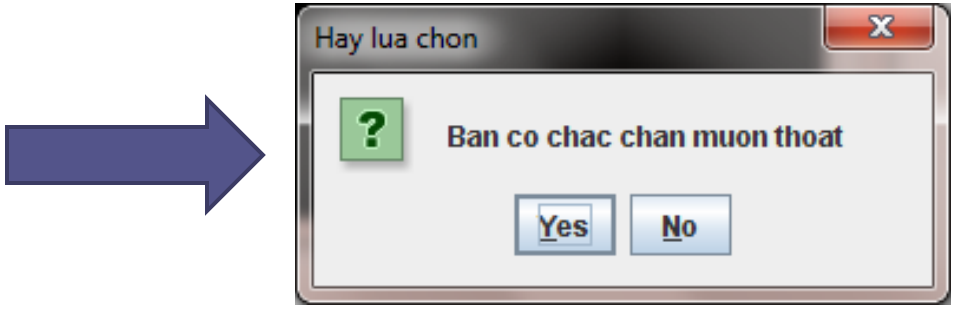
static void	showMessageDialog(Component parentComponent, Object message) Brings up an information-message dialog titled "Message".
static void	showMessageDialog(Component parentComponent, Object message, String title, int messageType) Brings up a dialog that displays a message using a default icon determined by the messageType parameter.
static void	showMessageDialog(Component parentComponent, Object message, String title, int messageType, Brings up a dialog displaying a message, specifying all parameters

Example – using static attributes and methods in class JOptionPane

```
JOptionPane.showMessageDialog(null,"Ban da thao tac
  loi", "Thong bao loi", JOptionPane.ERROR_MESSAGE);
```




```
JOptionPane.showConfirmDialog(null,"Ban co chac chan
  muon thoat?", "Hay lua chon",
  JOptionPane.YES_NO_OPTION);
```



Example – using static attributes and methods in class JOptionPane (2)

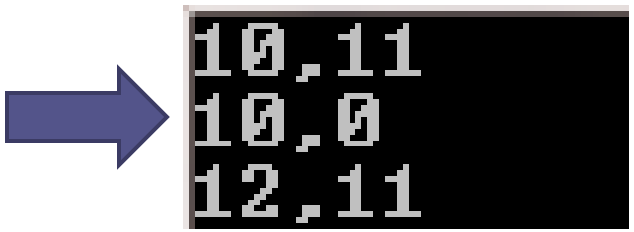
```
Object[] options = { "OK", "CANCEL" };
JOptionPane.showOptionDialog(null,"Nhan OK de tiep tuc",
  "Canh bao", JOptionPane.DEFAULT_OPTION,
  JOptionPane.WARNING_MESSAGE,null,options,options[0]);
```

2.1. Static member (2)

- Modifying value of a **`static`** member in an object will modify the value of this member in all other objects of the class.
- **`Static`** methods can access only **`static`** attributes and can call **`static`** methods in the same class.

Example 1

```
class TestStatic{
  public static int iStatic;
  public int iNonStatic;
}
public class TestS {
 public static void main(String[] args) {
  TestStatic obj1 = new TestStatic();
  obj1.iStatic = 10; obj1.iNonStatic = 11;
  System.out.println(obj1.iStatic+","+obj1.iNonStatic);
  TestStatic obj2 = new TestStatic();
  System.out.println(obj2.iStatic+","+obj2.iNonStatic);
  obj2.iStatic = 12;
  System.out.println(obj1.iStatic+","+obj1.iNonStatic);
 }
}
```



Example 2

```
public class Demo {
 int i = 0;
 void tang(){ i++; }
 public static void main(String[] args) {
    tang();
    System.out.println("Gia tri cua i la" + i);
 }
}
```

non-static method tang() cannot be referenced from a static contex
non-static variable i cannot be referenced from a static context

Java static methods

```
class MyUtils {
     . . .
     //====================== mean
     public static double mean(int[] p) {
               int sum = 0;
       for (int i=0; i<p.length; i++) {
                  sum += p[i];
               }
               return ((double)sum) / p.length;
     }
     . . .
}
…..
```

// Calling a static method from inside of a class
class double avgAtt = mean(attendance);
// Calling a static method from outside of a class
double avgAtt = MyUtils.mean(attendance);

Why static methods?

- Methods that do not interact with class instance should be declared as static
- The method "mean" in the previous example might not be declared as static, however in this case it must be called though an object

Counting number of objects of a class (C++)

```
class MyClass {
public:
MyClass(); // Constructor
~MyClass(); // Destructor
void printCount(); // Output current value of count
private:
static int count; // static member to store
// number of instances of MyClass
};
```

Static data parts
Definition and declaration (C++)

- Static members are stored independently of class representations, hence, static members must be defined as follows:

  int MyClass::count;

- We often define static members in a file containing method definitions
- Construtor:

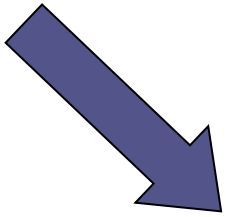
  int MyClass::count = **0**;

Static part: My Class

```
int MyClass::count = 0;
MyClass::MyClass() {
this->count++; // Increment the static count
}
MyClass::~MyClass() {
this->count--; // Decrement the static count
}
void MyClass::printCount() {
cout << "There are currently " << this->count
<< " instance(s) of MyClass.\n";
}
```

Using class MyClass

```
int main()
{
   MyClass* x = new MyClass;
   x->PrintCount();
   MyClass* y = new MyClass;
   x->PrintCount();
   y->PrintCount();
   delete x;
   y->PrintCount();
}
```



```
There are currently 1 instance(s) of MyClass.
There are currently 2 instance(s) of MyClass.
There are currently 2 instance(s) of MyClass.
There are currently 1 instance(s) of MyClass.
```

2.2. Constant members

- An attribute/method that can not change its values/content during the usage.
- Declaration syntax:

```
chi_dinh_truy_cap final kieu_du_lieu
                  TEN_HANG = gia_tri;
```

- For example:

```
final double PI = 3.141592653589793;
public final int VAL_THREE = 39;
private final int[] A = { 1, 2, 3, 4, 5, 6 };
```

2.2. Constant members (2)

- Normally, constants related to the class are declared as **`static final`** in order to be easily accessed

```
public class MyDate {
    public static final long SECONDS_PER_YEAR =
        31536000;
    …
}
…
long years = MyDate.getMillisSinceEpoch() /
    (1000*MyDate.SECONDS_PER_YEAR);
```

javax.swing

Class JOptionPane

ERROR_MESSAGE

```
public static final int ERROR_MESSAGE
```

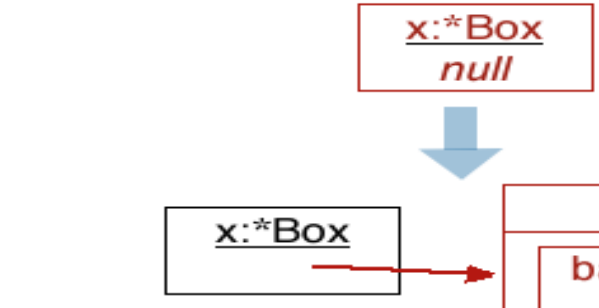
Objects in C++ and Java

- C++: objects in a class are created at the declaration:
  - Point p1;
- Java: Declaration of an object creates only a reference that will refer to the real object when **new** operation is used:
  - Box x;
  - x = new Box();
  - Objects are dynamically allocated in heap memory

Object in Java

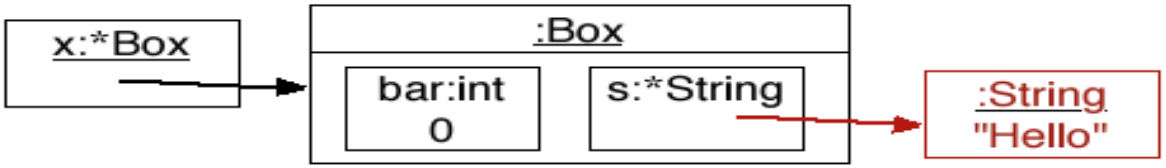
```
class Box
{
    int bar;
    String s;
}
```

```
Box x;
```




```
x = new Box ();
```


```
x.s = "Hello";
```

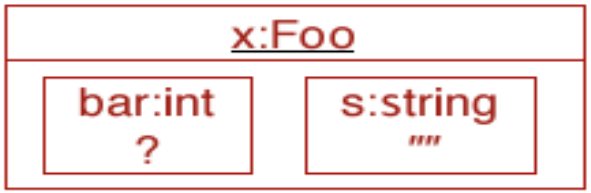


Object in C++

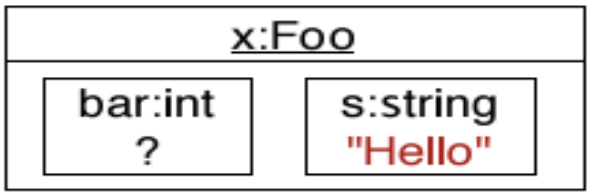
```
class Foo
{
        int bar;
        string s;
};
```

gotta have that semi

```
Foo x;
```




```
x.s = "Hello";
```



Outline

1. Method overloading
2. Object member and class member
3. Memory management in Java
4. Passing arguments to methods
5. Some utility classes in Java

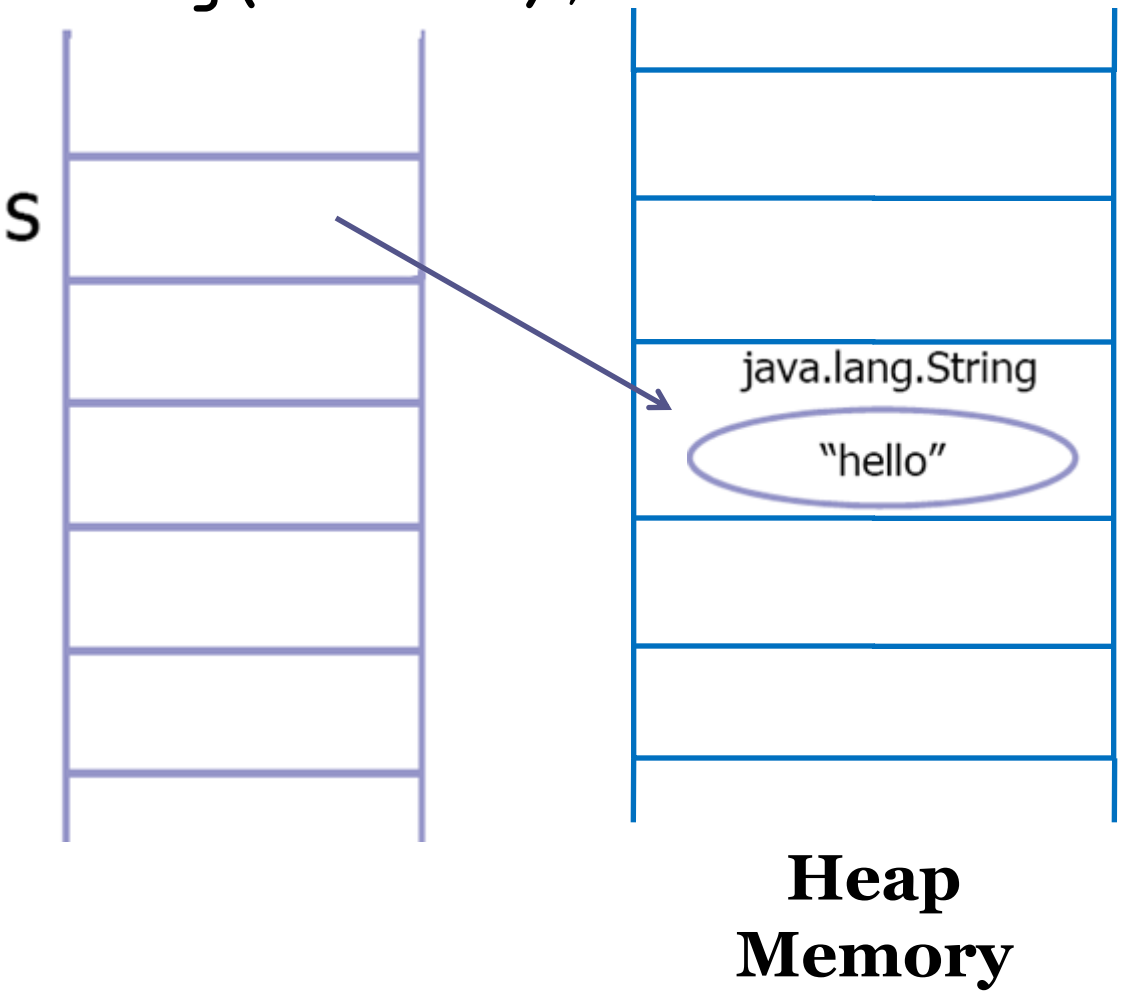
3. Memory management in Java

- 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.

3.1. Heap memory

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

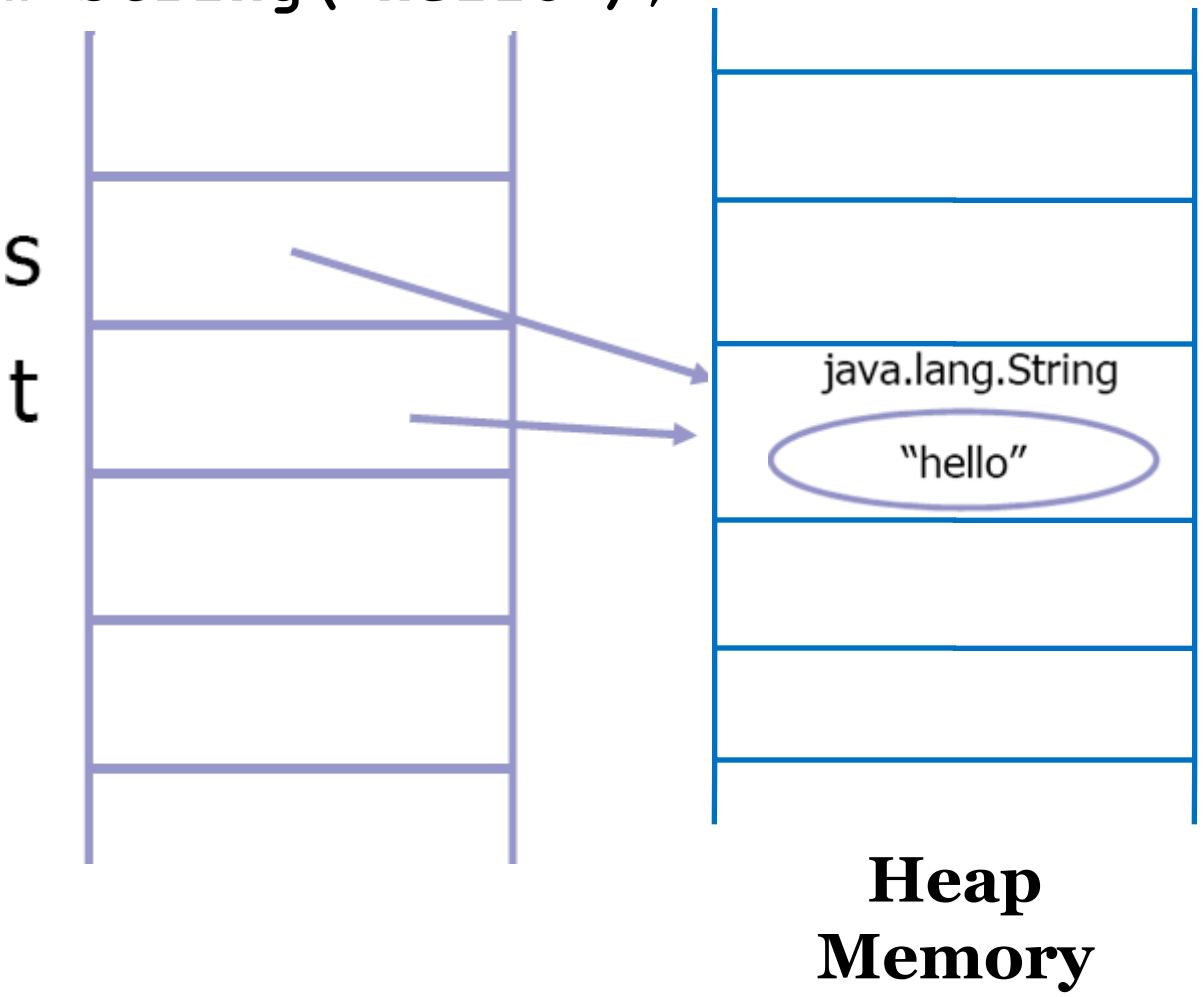
- Heap memory is used to write information created by **new** operator.



3.1. Heap memory

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

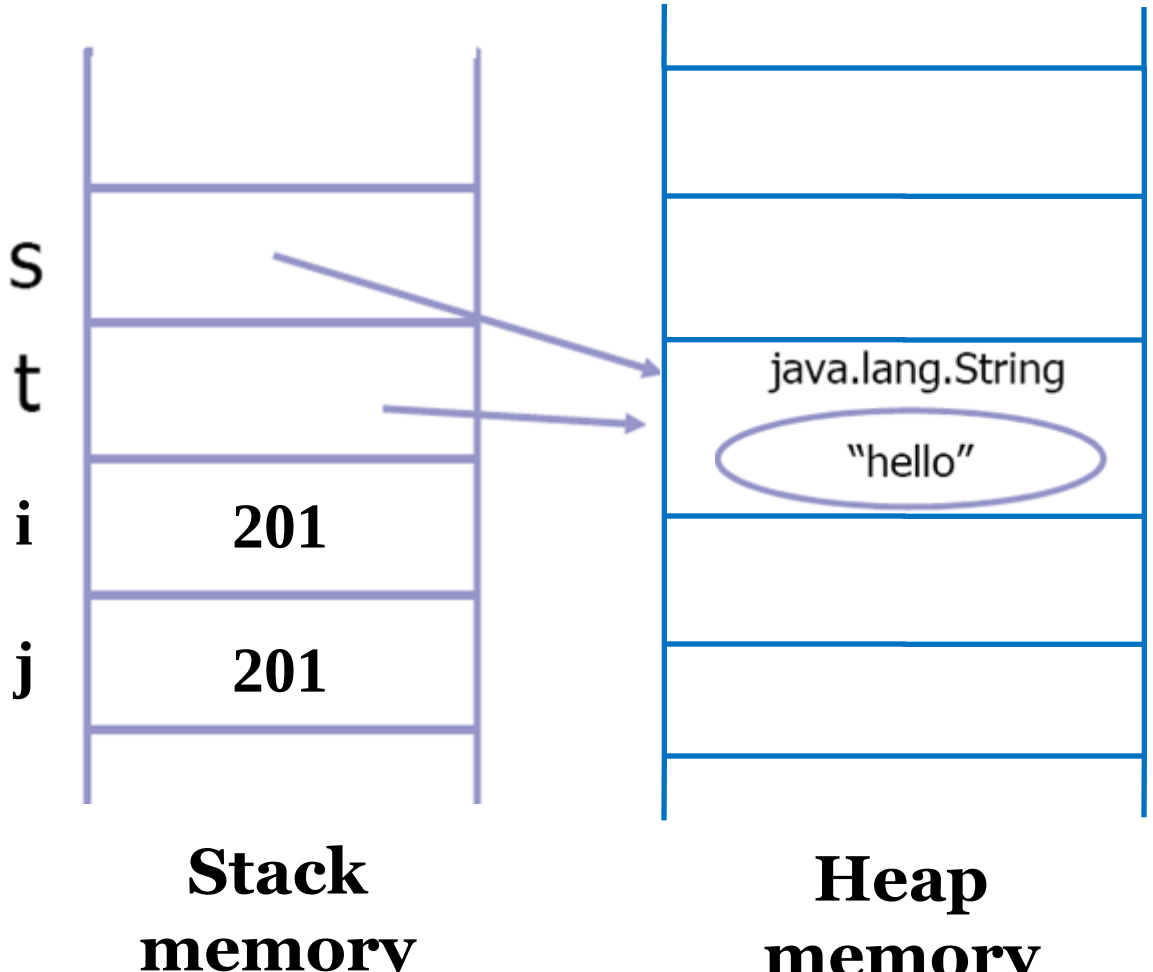
- Heap memory is used to write information created by **new** operator.



3.2. 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



3.2. Garbage Collector

- A back-ground process calls “Garbage collector” to recover memory that are not pointed to by objects
- Objects that have not reference to are assigned to null.
- Garbage collector checks the object list of JVM and recovers resources of objects that do not have any reference.

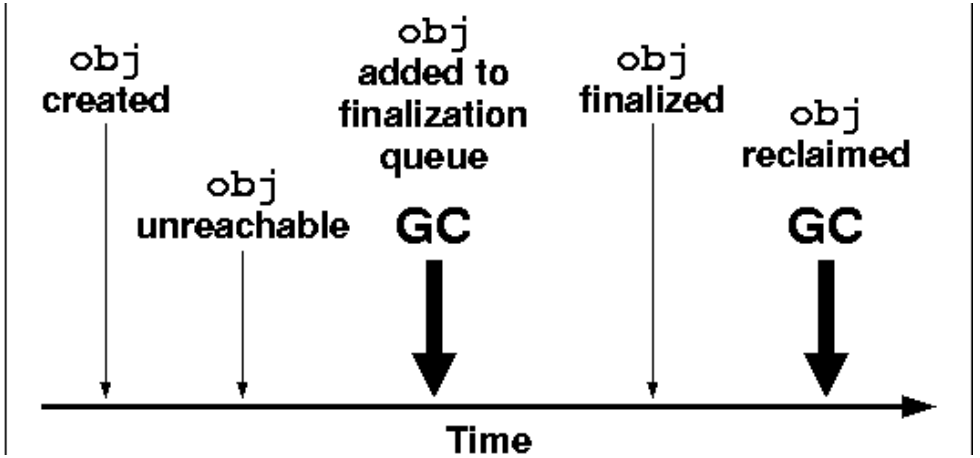
3.2. Garbage Collector (2)

- JVM decides when to collect garbage:
  - When there is not enough memory
  - At some un-predictable points of time
- We can not prevent the garbage collection process from running but we can force it to run earlier:

```
System.gc(); hoặc Runtime.gc();
```

Method `void finalize()`

- Any class also has method finalize() – that is executed right after the garbage collection process takes place
- Is only used in some special cases in order to "self-clean" used resources when objects are freed by **`gc`**
  - For example: need to pack socket, file,... that should be handled in the main thread before the objects are disconnected from reference.
- Can be considered as class destructor although Java does not have this notion.



3.3. Object comparison

- For primitive data types, operator == checks whether their values are the equal
- Example:

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

3.3. Object comparison (2)

- For objects, operator == checks whether two objects are unique, whether they have the same reference to an object.
- For example:

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

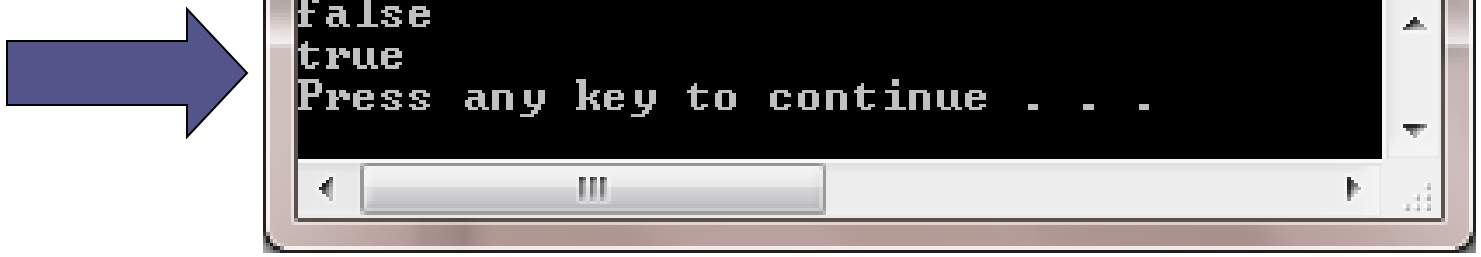
```
Employee a = new Employee(1);
Employee b = a;
if (a==b)... // true
```

3.3. Object comparison (3)

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

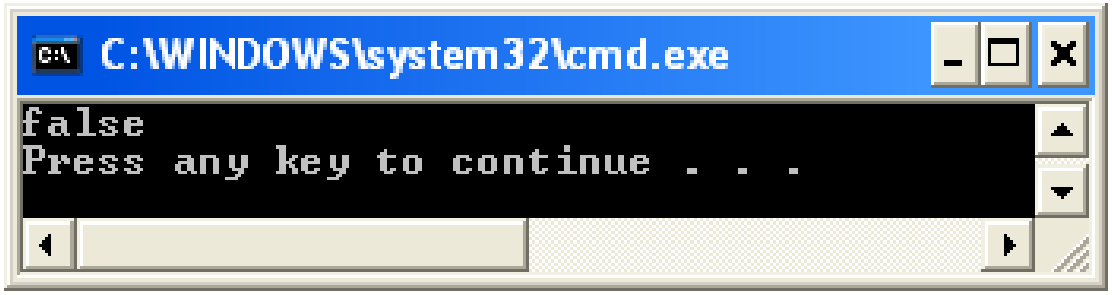
Example of == and equals -Integer class

```
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));
  }
}
```



Example 3 – equals of a class

```
class Value {
  int i;
}
public class EqualsMethod2 {
  public static void main(String[] args) {
    Value v1 = new Value();
    Value v2 = new Value();
    v1.i = v2.i = 100;
    System.out.println(v1.equals(v2));
  }
}
```



Outline

1. Method overloading
2. Object member and class member
3. Memory management in Java
4. Passing arguments to methods
5. Some utility classes in Java

Argument passing

- C++
  - Passing values, references, pointers
- Java
  - Passing references

4. Passing arguments to a method

- Java passes all arguments to a method in form of pass-by-value: Passing value/copy of the real argument
  - For arguments of value-based data types (primitive data types): passing value/copy of primitive data type argument
  - For argument of reference-based data types (array and object): passing value/copy of original reference.

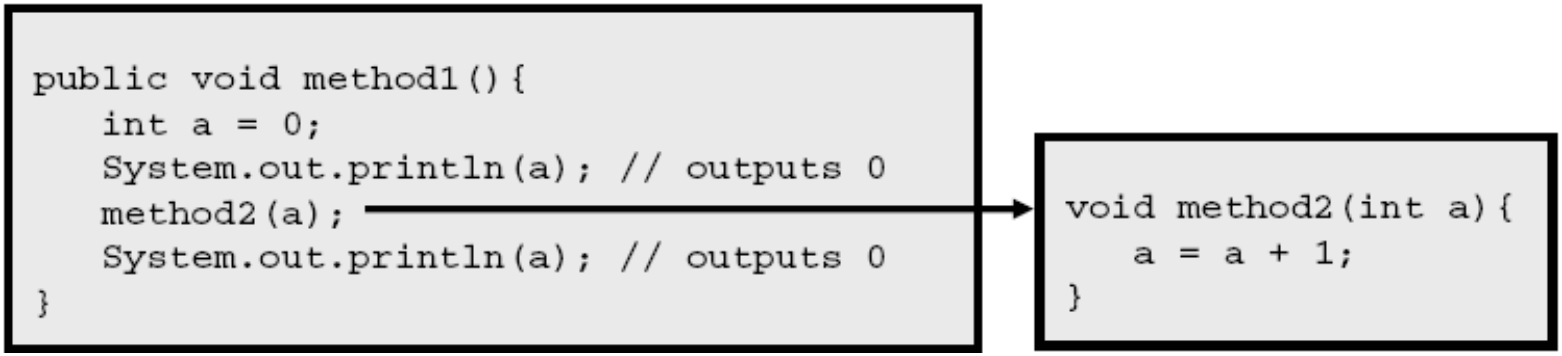
→ Modifying formal arguments does not effect the real arguments

Question:

- What will happen if:
  - We modify the internal state of object parameters inside a method?
  - We modify the reference to an object?

4.1. With value-based date type

- Primitive values can not be changed when being passed as a parameter

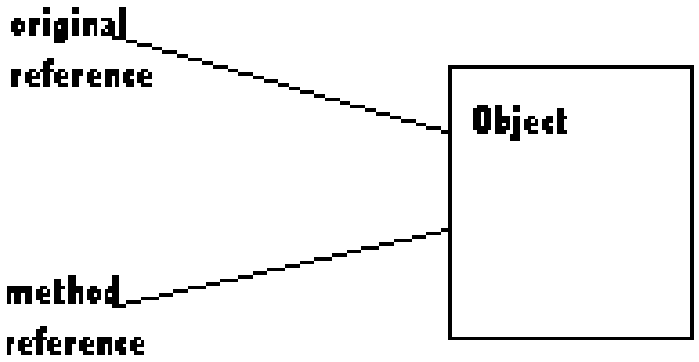


- Is this swap method correct?

```
public void swap(int var1, int var2) {
  int temp = var1;
  var1 = var2;
  var2 = temp;
}
```

4.2. With reference-based data type

- Pass the references by value, not the original reference or the object



- After being passed to a method, a object has at least two references

Passing parameters

```
public class ParameterModifier
{
  public void changeValues (int f1, Num f2, Num f3)
  {
    System.out.println ("Before changing the values:");
    System.out.println ("f1\tf2\tf3");
    System.out.println (f1 + "\t" + f2 + "\t" + f3 + "\n");

    f1 = 999;
    f2.setValue(888);
    f3 = new Num (777);

    System.out.println ("After changing the values:");
    System.out.println ("f1\tf2\tf3");
    System.out.println (f1 + "\t" + f2 + "\t" + f3 + "\n");
  }
}
```

```
public class Num
{
   private int i;
   public Num(int i) {
            this.i = i;
            }
  public void
setValue(int i){this.i =
i;}
   public String
toString() {
            return “”  + i;
 }

}
```

Passing parameters

```
public class ParameterTester
{
  public static void main (String[] args)
  {
    ParameterModifier modifier = new ParameterModifie

    int a1 = 111;
    Num a2 = new Num (222);
    Num a3 = new Num (333);

    System.out.println ("Before calling changeValues:");
    System.out.println ("a1\ta2\ta3");
    System.out.println (a1 + "\t" + a2 + "\t" + a3 + "\n");

    modifier.changeValues (a1, a2, a3);

    System.out.println ("After calling changeValues:");
    System.out.println ("a1\ta2\ta3");
    System.out.println (a1 + "\t" + a2 + "\t" + a3 + "\n");
  }
}
```

Before calling changeValues:
a1 a2 a3
111 222 333

Before changing the values:
f1 f2 f3
111 222 333

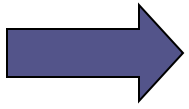
After changing the values:
f1 f2 f3
999 888 777

After calling changeValues:
a1 a2 a3
111 888 333

For example

```
public class Point {
  private double x;
  private double y;
  public Point() { }
  public Point(double x, double y) {
     this.x = x; this.y = y;
  }
  public void setX(double x) { this.x = x; }
  public void setY(double y) { this.y = y; }
  public void printPoint() {
     System.out.println("X: " + x + " Y: " + y);
  }
}
```

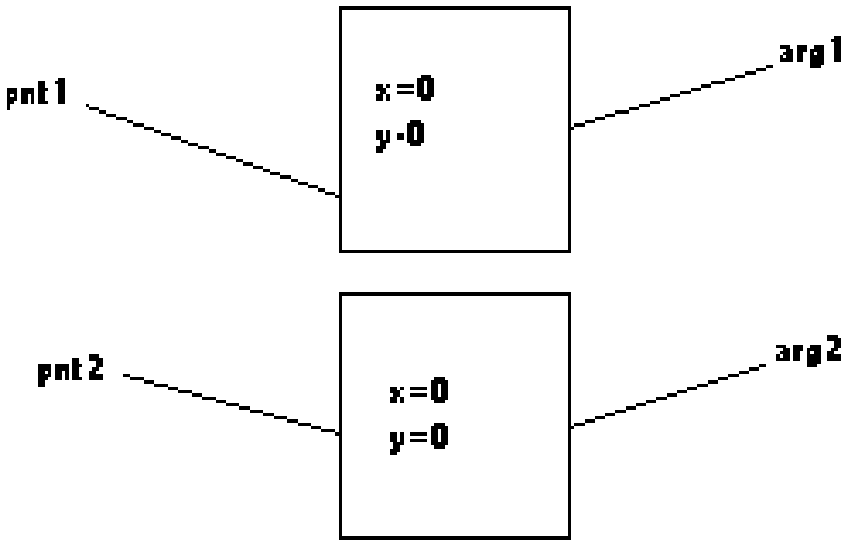
```
public class Test {
  public static void tricky(Point arg1, Point arg2) {
   arg1.setX(100); arg1.setY(100);
   Point temp = arg1;
   arg1 = arg2; arg2 = temp;
  }
  public static void main(String [] args) {
   Point pnt1 = new Point(0,0);
   Point pnt2 = new Point(0,0);
   pnt1.printPoint(); pnt2.printPoint();
   System.out.println(); tricky(pnt1, pnt2);
   pnt1.printPoint(); pnt2.printPoint();
  }
}
```

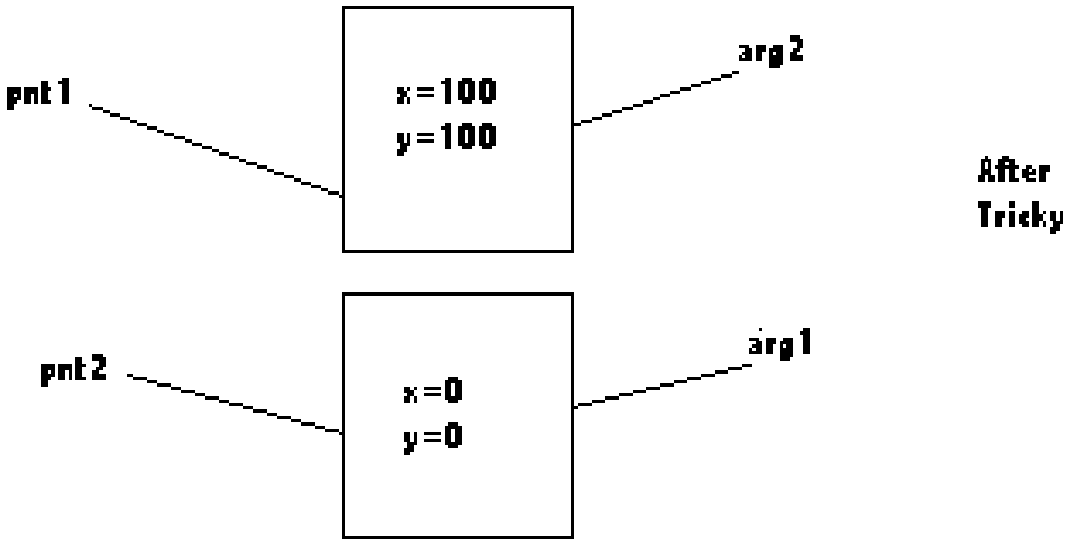


```
X: 0.0 Y: 0.0
X: 0.0 Y: 0.0

X: 100.0 Y: 100.0
X: 0.0 Y: 0.0
Press any key to continue . . .
```

- Only the method references are swap, not the original references





Function with default arguments (C++)

- Function declaration: void hamf (int x, float y=1.0)
  - Explanation: x is an argument without default value, y is an argument with a default value
  - There are two ways to call the function:
    - func(10); x receives a value x=10 and y receives a value y=1.0 (default value)
    - func(10, 5.0); x receives value x=10 and y receives value y=5.0 (passed value)
- Functions with default arguments allow changing the format when passing values to argments.

Reference (C++)

- A reference can be a variable, a formal argument of a function or used to be return value of a function.
- When using reference, we have to make sure the following:
  - A reference must be initialized with a value at the declaration.
  - After the initialization, a reference is assigned to a variable and we can not change the reference to another variable.
  - We can not have a reference with Null value.
- This is the reason we still use pointers even we already have reference.

```
#include <iostream>
using namespace std;

int y;
int& r = y;
const int& q = 12;  // (1)
int x = 0;          // (2)
int& a = x;         // (3)
int main() {
  cout << "x = " << x << ", a = " << a << endl;
  a++;
  cout << "x = " << x << ", a = " << a << endl;
}
```

Using reference in functions

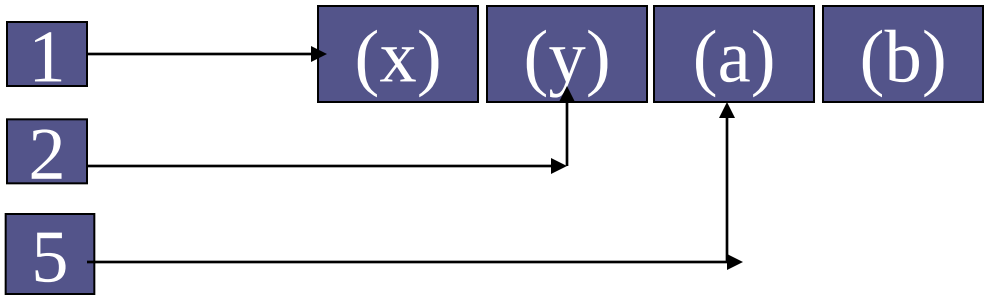
- Passing arguments to a function by reference.
  - Using reference in formal argument declaration will require compiler to pass variable addresses to the function and the function will manipulate directly on these variables.
  - The initialization of formal arguments that are references is done automatically in every function call.
  - Arguments that are passed to a function by references must be variable (except if its declaration is preceded by Const keyword).

Swapping values of two variables (C++)

```
void  swap (int &x,int &y)
{   int  temp=x ;
    x=y;
    y=temp;
}
void main()
{
  int  a=1; b=2;
  swap (a,b);
}
```

Functions with default arguments (C++)

- Principles for declaring functions with default arguments:
  - To ensure that the compiler specifies correctly values/variables passed to arguments, we have to respect the following:
    - Arguments without default values are located at the head of the argument list
    - Arguments with default values are located at the tail of the argument list
- void f (int x, int y, int a=0, float b=1.0)
- Function call:
- f(1, 2, 5)



Function with default arguments (C++)

```
int getSum(int, int=0, int=0);// OK
int getSum(int, int=0, int); //wrong!
```

- When an argument is skipped in a function called, all the arguments after it will be skipped too.

```
sum = getSum(num1, num2);  // OK
sum = getSum(num1, , num3);// wrong!
```

Java language does not support this feature

When to use ?

- Using functions with default-values arguments:
  - The task performed by the function does not change the nature or algorithm to be done. Passing default values or other values to the arguments will change only the output but will not affect the task.
  - The second case to use functions with default-values arguments: function's task can be extended in case of passing values to arguments.

When to use?

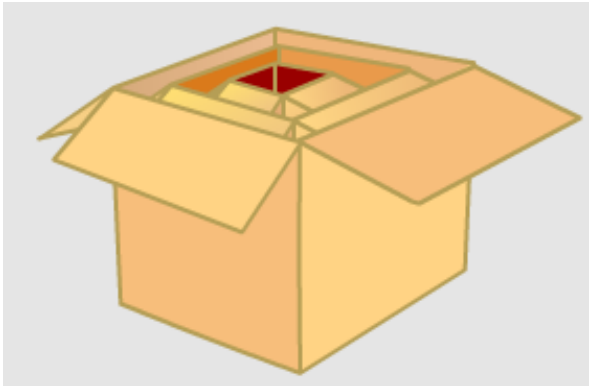
- Example (1): void f (int x, int y, int a=0, float b=1.0)
  - The task performed in f depends on 4 arguments x, y, a, b and often a=0 và b=1 but in some cases, a and b can receive other values

- Example (2): void f (int x, int y)
  - Normally f depends on two arguments x,y. Now we need to extend f to 3 arguments f(int x, int y, int a).
  - How to re-define f so that the existing function calls of f with two arguments will not be affected? The solution is to declare the argument a with a default value:
  - f (int x, int y, int a=0);

Outline

1. Method overloading
2. Object member and class member
3. Memory management in Java
4. Passing arguments to methods
5. Some utility classes in Java

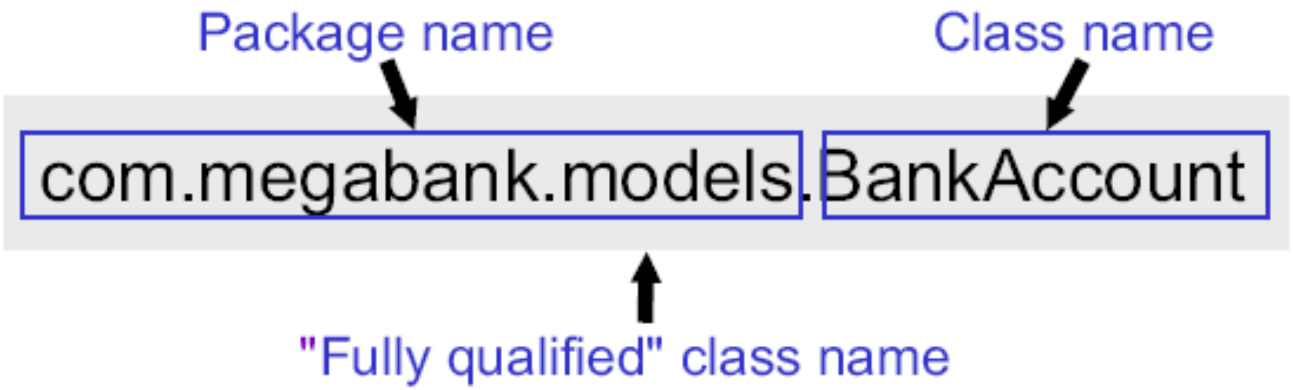
5.1. Package in Java



- 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

5.1. Package in Java (2)

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



a. 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:

```
public class HelloNameDialog{
  public static void main(String[] args){
    String result;
    result = javax.swing.JOptionPane.showInputDialog
                                    (“Hay nhap ten ban:”);
    javax.swing.JOptionPane.showMessageDialog(null,
                              ”Xin chao “+ result + “!”);
  }
}
```

a. References between classes(2)

- **Import** command:
  - Use **import** command to declare packages or classes in order to avoid providing the full-name.
  - Example:

```
import javax.swing.JOptionPane;
public class HelloNameDialog{
  public static void main(String[] args){
    String result;
    result = JOptionPane.showInputDialog
                                ("Hay nhap ten ban:");
     JOptionPane.showMessageDialog(null,
                            "Xin chao "+ result + "!");
     System.exit(0);
  }
}
```

b. 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.ietf.jgss
- org.omg.CORBA
- org.omg.CosNaming
- org.omg.Dynamic
- 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

b. Packages in Java (2)

- The basic packages in trong Java
  - **java.lang**
    - Provides basic classes to Java programming language
    - Includes wrapper classes, String and StringBuffer, Object, ...
    - Import by default all the classes
  - **java.util**
    - Includes a set of frameworks, models, events, and other utilities.
  - **java.io**
    - Provides I/O capacity of system with data flow and a file system..

b. Packages in Java (3)

- The basic classes in Java
  - **java.math**
    - Provides classes for performing math operations with integer and real number
  - **java.sql**
    - Provides API to access and manipulate data stored in data source (often the relational database)
  - **javax.swing**
    - Provides classes and interfaces to create graphics.
  - …

5.2. Wrapper class

- Primitive data types do not have involving methods.
- A primitive data type has a class that is called wrapper class:
  - Wrapper classes will “wrap” primitive data and provide appropriate methods for the data.
  - An object of a wrapper class simple store a single variable and provides methods to handle it.
  - Wrapper classes take part of Java API

5.2. Wrapper classes (2)

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

a. Converting data type

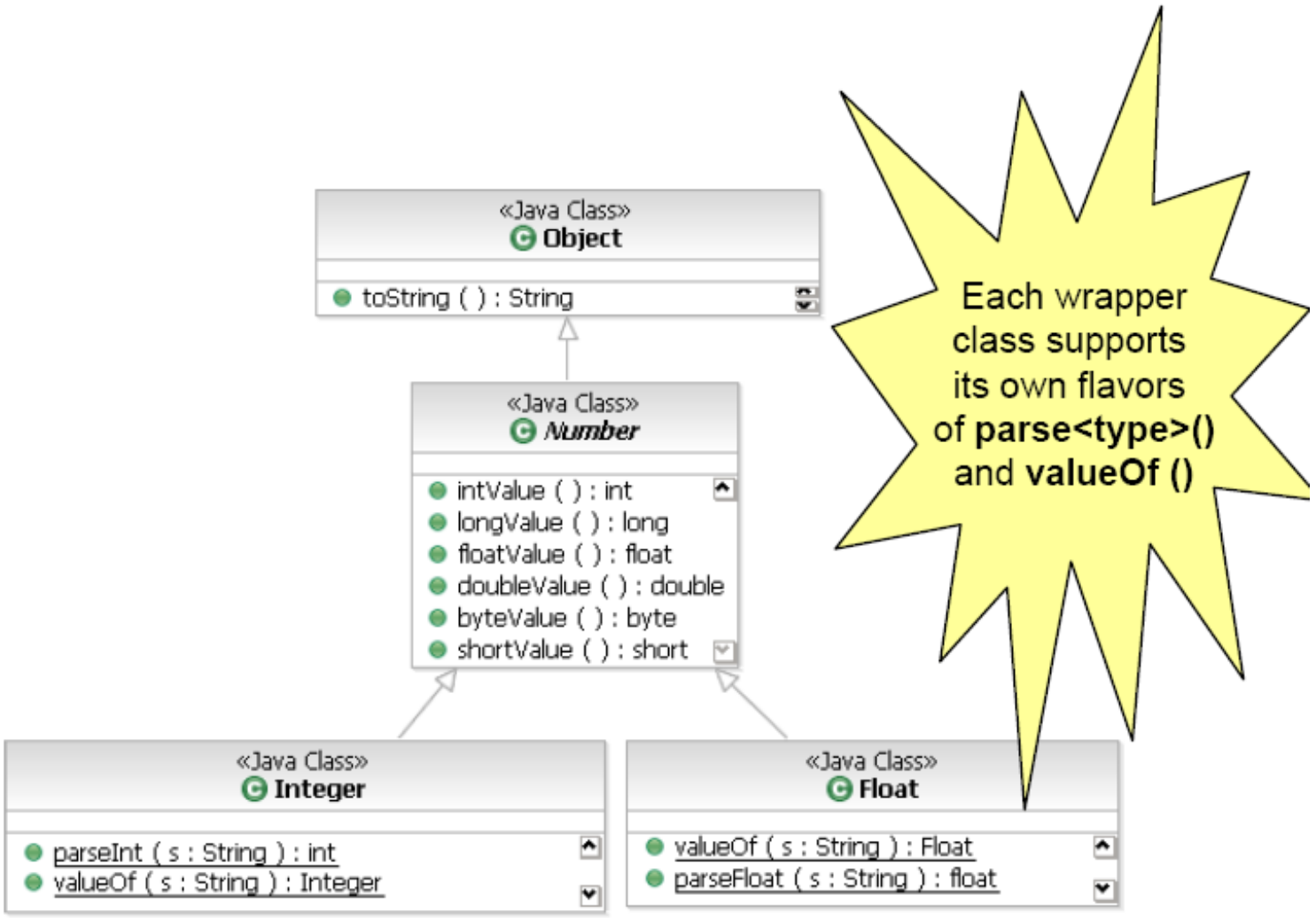
- Use **`toString()`** to convert number values to string.
- Use **`<type>Value()`** to convert an object of a wrapper class to the corresponding primitive value

```
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
```

a. Converting data type (2)



b. 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

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
```

5.3. String

- String is a class and is not a primitive data type
- A String is created from a set of characters in the double quotes:

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

- A String object can be initialized in several ways:

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

a. String concatenation

- Operation + can be used to concatenate Strings:

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

- Basic data types used in function println() will be 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?

b. Methods of String

```
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"
```

c. String comparison

- **`oneString.equals(anotherString)`**
  - Comparing the equivalence
  - Return **`true`** or **`false`**

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

- **`oneString.equalsIgnoreCase(anotherString)`**
  - Comparison ignores the lowercase and uppercase

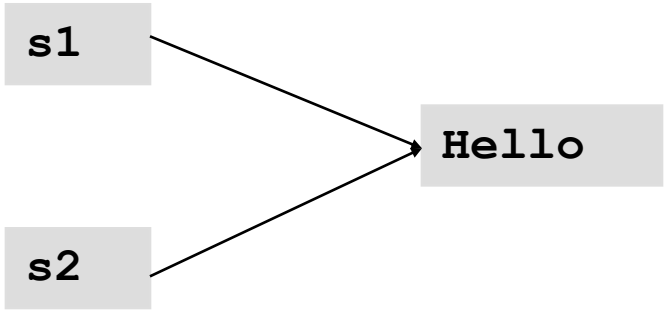
```
boolean same = "Joe".equalsIgnoreCase("joe");
```

- Compare **`oneString == anotherString`** will make confusion
  - Compare two objects

c. Comparing two Strings (2)

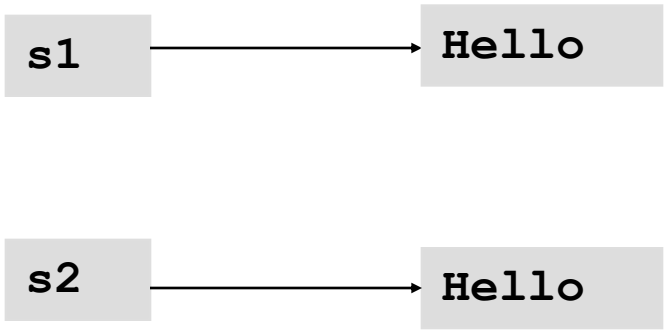
```
String s1 = new String("Hello");
String s2 = s1;
```

(s1==s2) returns true




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

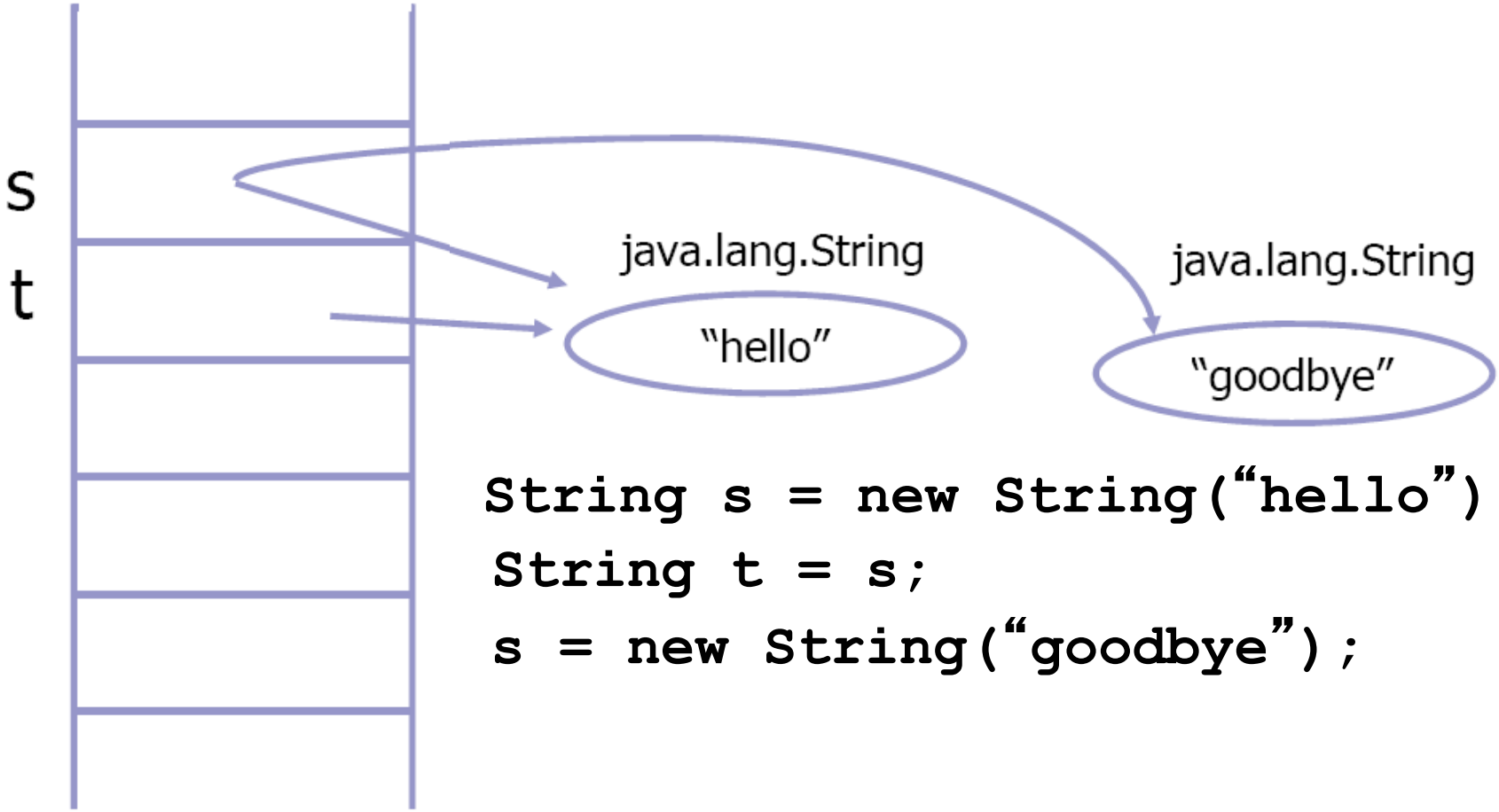
(s1==s2) returns false



5.4. StringBuffer

- String is an invariant 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 is a variant type:
  - Object can change the value after being created

5.4. StringBuffer (2)



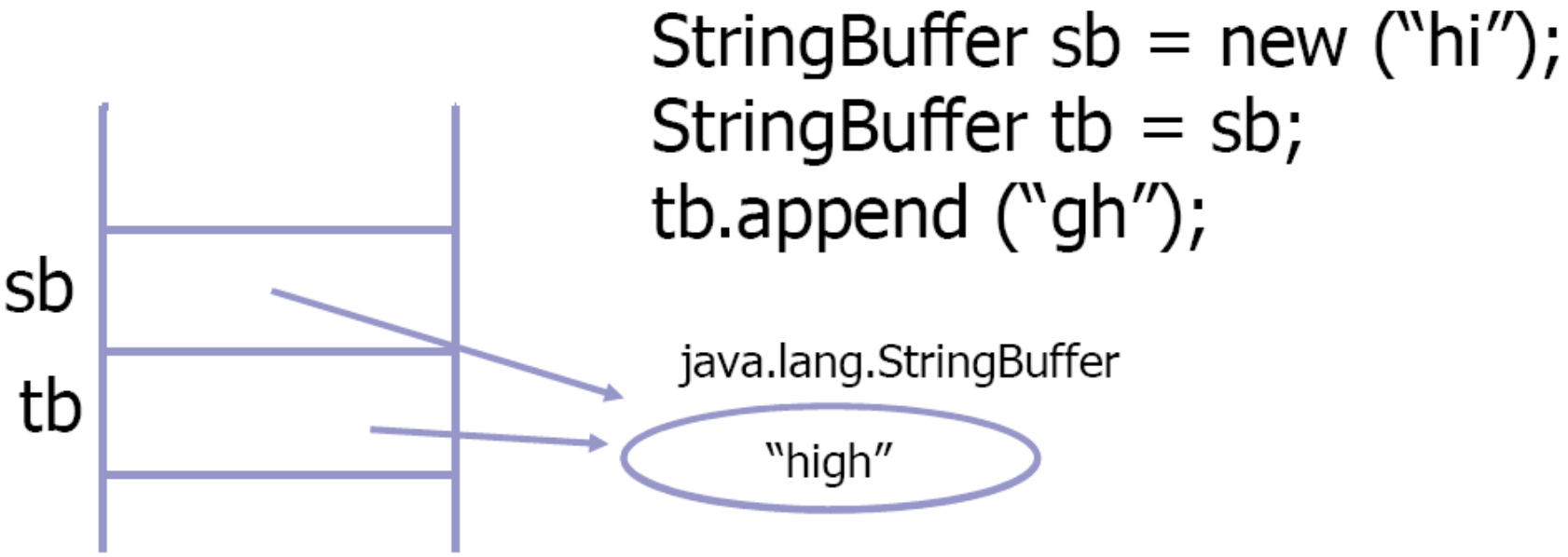

```
String s = new String("hello");
String t = s;
s = new String("goodbye");
```

5.4. 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.
  - Provides a more effective mechanism for building and concatenating strings:
    - String concatenation is often done by compiler in class StringBuffer

5.4. StringBuffer (4)

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



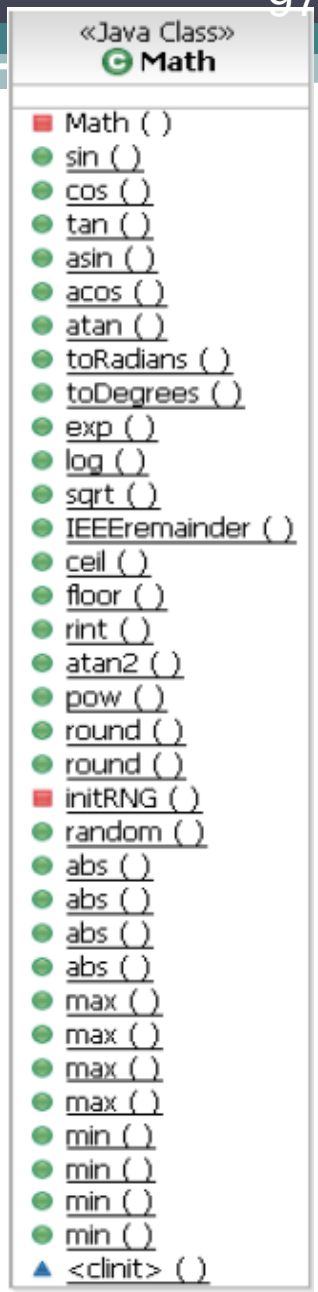
5.4. StringBuffer (5)

- 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."
```

5.5. 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



5.5. 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)
  )
```

«Java Class»
Math

- Math ( )
- sin ( )
- cos ( )
- tan ( )
- asin ( )
- acos ( )
- atan ( )
- toRadians ( )
- toDegrees ( )
- exp ( )
- log ( )
- sqrt ( )
- IEEEremainder ( )
- ceil ( )
- floor ( )
- rint ( )
- atan2 ( )
- pow ( )
- round ( )
- round ( )
- initRNG ( )
- random ( )
- abs ( )
- abs ( )
- abs ( )
- abs ( )
- max ( )
- max ( )
- max ( )
- max ( )
- min ( )
- min ( )
- min ( )
- min ( )
- <clinit> ( )