Introduction to Java

Vũ Thị Hồng Nhạn

(vthnhan@vnu.edu.vn)

Dept. of Software Engineering, UET

Vietnam National Univ., Hanoi

Contents

- Primitive data types & objects
- Reference parameter
- Garbage collection

Data types

- Java has 2 categories of data
  - **Primitive** data (e.g., number, character)
  - **Object** data (programmer created types)

Primitive data types

Number	• byte, short, int, long, float, double • unsigned doesn't exists in Java
logic	boolean
char	char

- Primitive data is not **an object**
  - int a=5;
  - if(a==b)…
- The corresponding class of an Integer object: **Integer**
  - Integer count = **new** Integer(0);
- Refer to the link for more details:
  http://docs.oracle.com/javase/7/docs/api/java/lang/Integer.html

Primitive data types...

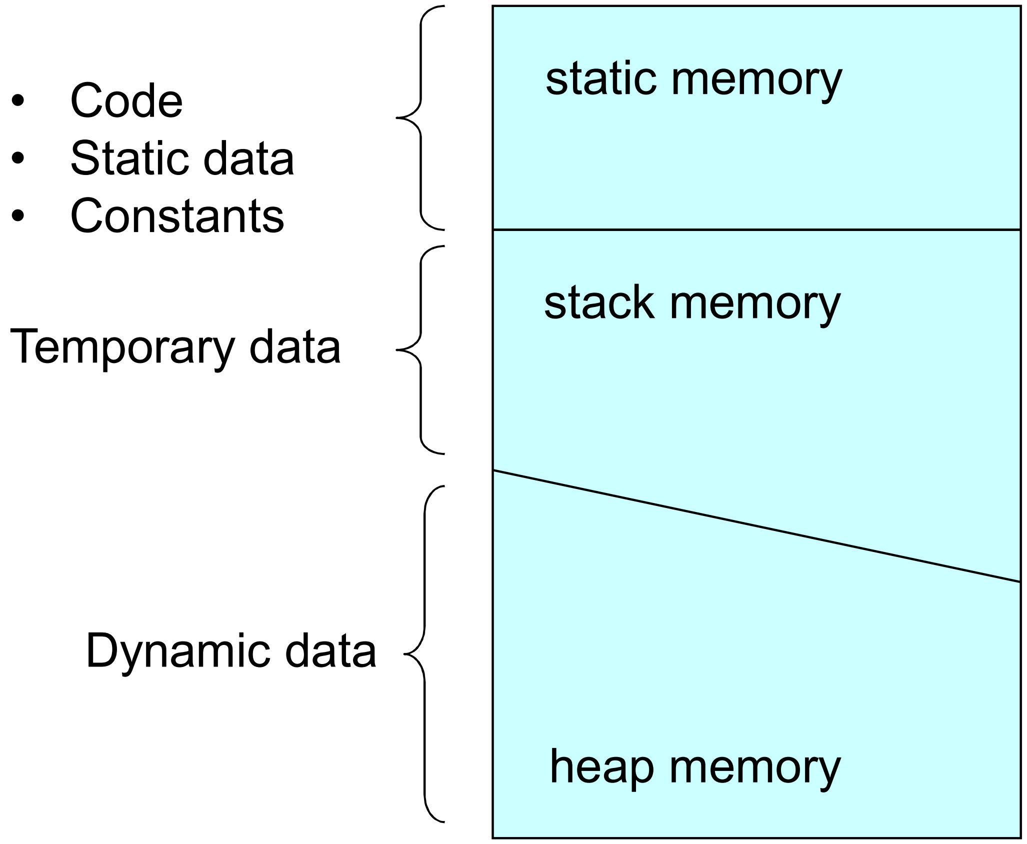
data type	Size (bits)	min value	max value
char	16	0x0	0xffff
byte	8	-128 or (-2^{7})	+127 or $(2^{7}-1)$
short	16	-32768 or (-2^{15})	32767 $(2^{15}-1)$
int	32	$- 2^{31}$, 0x80000000	$+ 2^{31} - 1$, 0x7fffffff
long	64	$- 2^{63}$	$+ 2^{63} - 1$
float	32	1.40129846432481707e-45	3.40282346638528860e+38
double	64	4.94065645841246544e-324	1.79769313486231570e+308
boolean	1	true; false	

Where are data stored?

- Primitive data
  - Works via ***variables***
- Attributes of objects are responsible for storing data
  - **Objects** work via ***reference variables***

- Where are *primitive variables, reference* & *objects* stored?

Memory


• Code
• Static data
• Constants
Temporary data
Dynamic data
static memory
stack memory
heap memory

Java objects stored in heap

- In java, **all objects** are *dynamically* allocated on **Heap**
  - This is different from C++ where objects can be allocated memory either on Stack or Heap
- Java doesn’t have pointers, java has **references**

Java reference

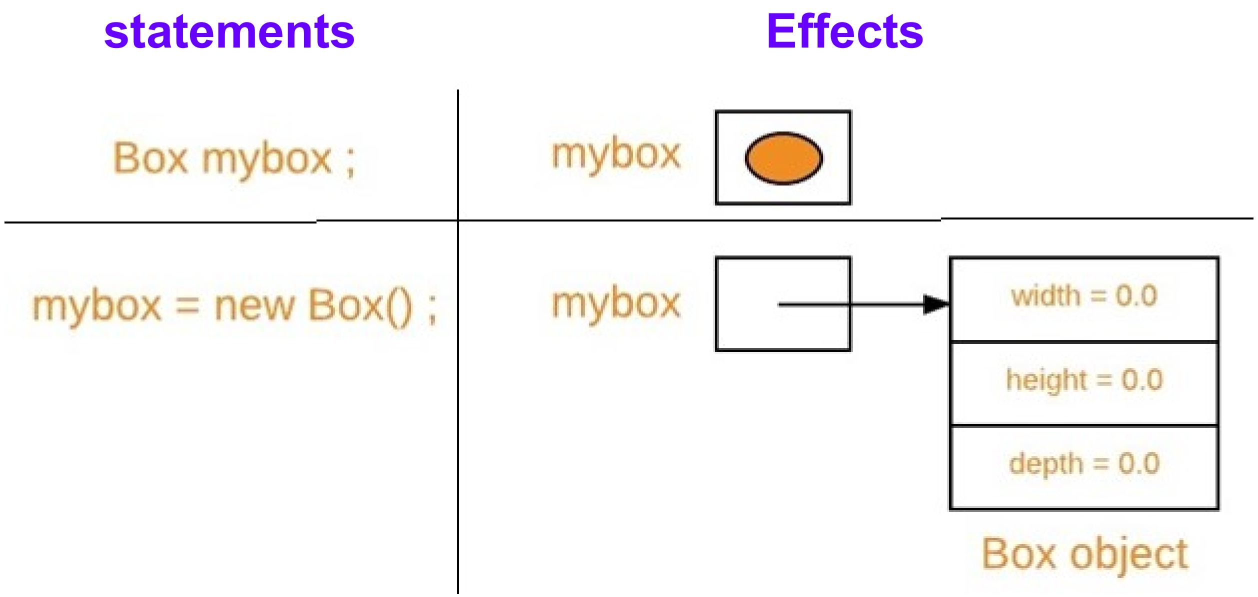
- A reference is *a variable* that refers to something else
  - Pointer is a variable that **stores** *a memory address* (i.e., pointer is a reference, but reference is not a pointer)

- When **a variable** of a class type is **declared**
  - Only **a reference** is created
  - Memory is ***not*** *created for the object*
  - To allocate memory to an object, we must use **new()**

new operator in Java

- The new operator
  - *dynamically* allocates memory for a **new** object
  - and returns a reference to that memory
  - This reference is then stored in the variable we declared for the object
- Following the **new** operator is a class constructor, which initializes the new object

```
class Box {
        double width;
        double height;
        double depth;
        …
}
```



statements

Effects

Box mybox ;

mybox

mybox = new Box() ;

mybox

Box object

Assignment operator “=“

- For primitive data, assign a value for a primitive variable
- For an object, **two references** refer to the same object

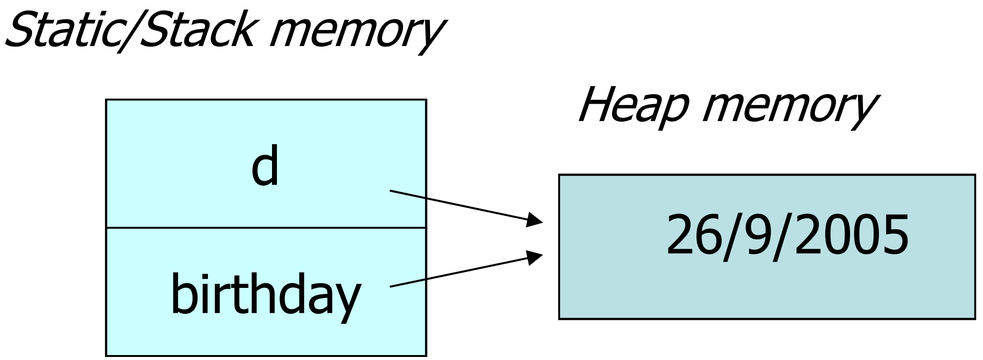
```
int x=10, y=20;

x=y;
x=50;

System.out.println(y);
```

```
Box x=new Box(1,1,1);
Box  y=new Box(2,2,2);
x=y;
x.setWidth(50);
System.out.println(y.getWidth());
```

“new” vs. “=“

```
class MyDate{
        int d; int m; int y
        MyDate(int d,int m, int y ){this.d=d; this.m=d; this.y=y;}
}
```

- MyDate d;
- MyDate birthday;
- d= **new** MyDate(26,9,2005);
- birthday = d;



== operator

- Can apply for every primitive types (e.g., int, char, double, boolean…)
- Can use for **reference comparison**
  - i.e., check if both objects point to *the **same** memory location*

```
String s1= new String(“Hello”);
String s2= new String(“Hello”);
String s3 = s2;
System.out.println(s1==s2); // false
System.out.println(s2==s3);//  true
```

Compare two objects

```
class MyDate{
        int d; int m; int y
        ……………
        boolean equalTo(MyDate date){return d==date.d &&  m==date.m &&
            y==date.y;  }
}
 …
MyDate d1= new MyDate(10,10,1954);
MyDate d2= new MyDate(d1);
MyDate d3= new MyDate(1,1,1954);

System.out.println(d1.equalTo(d2));

System.out.println(d1.equalTo(d3));
```

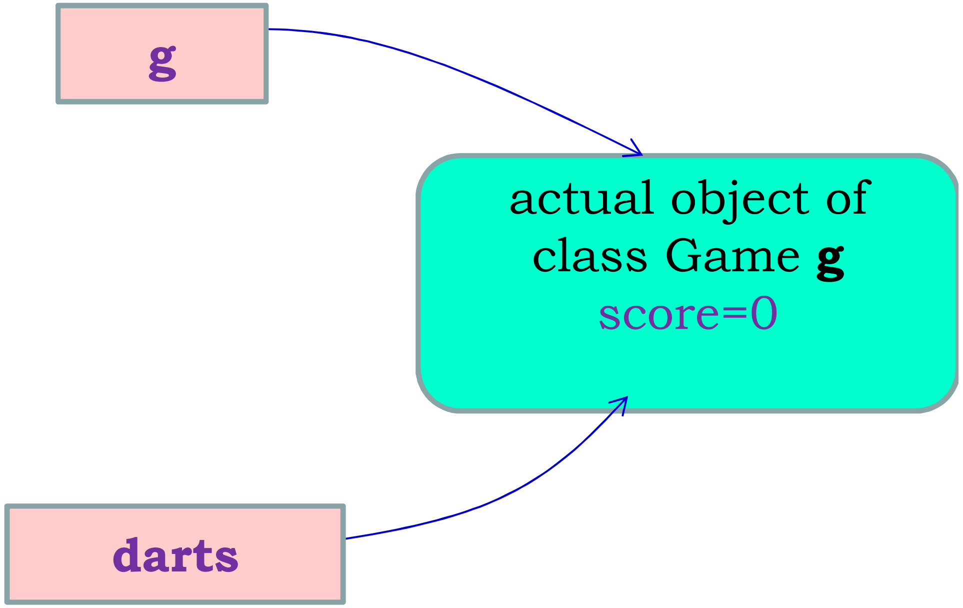
Remarks

Game class defined as follows

```
class Game {
      private int score;
     public Game(){score=0;}
     public void setScore(int sore) {
             this.score = score
        }
     public int getScore() {
             return year;
        }
   …
}
```

Object vs. reference variable

- An object of a certain class is created by using the keyword **new** followed by a constructor
  - A reference variable points to **an actual object**
  - Game **g**= **new** Game();
  - Game **darts** = g;



g

actual object of
class Game **g**
score=0

darts

Accessing error

```
class Game {
        private int score;
        public Game(){score=0;}
        public void setScore(int sore) {
                this.score = score
            }
        public int getScore() {
                return score;
            }
}
```

```
Game g = new Game();
…
g.score = 100;    // compile error


g.setScore(2000);
System.out.println(g.getScore());
```

Method overloading

- **A class** can have **more than one** method having **the same name,** however their **parameters** must be *different*
  - It's similar to constructor overloading in Java mentioned before!

```
class Game{
        …
        public void setScore(int sore){…}
        public void setScore(String s){…}
}
…
g.setScore(10);
g.setScore("Ten");
```

Example

```
public class Game {
    private int score;
    public Game() {score=0;}

    public Game(Game g) {
        score = g.score;
    }
}
```

Example …

```
Game g = new Game();
g.setSore(10);


Game startGame = new Game(g);


Game secondGame = g;
secondGame = new Game();
```

Example …

```
Game g= new Game();
g.setScore(100);
```

```
Game firstGame = g;
firstGame = new Game();
firstGame = new Game(g);
```

How different?

Primitive data type

Passing value to function

```
public class Main
{
    public static void main(String[]
     args)
    {
        int x = 5;
        change(x);
        System.out.println(x);
    }
    public static void change(int x)
    {
        x = 10;
    }
}
```

- **Output: 5**

- Like C/C++, Java creates *a* ***copy*** *of the variable being passed* in the method and placed in **Stack** and then do the manipulation

- Hence, the change is not reflected in the **main** method

Passing objects/references

- All non-primitives are always **references**
- When passing object references to methods
  - Java creates **a copy of references** and pass it to method
  - but they **still point to** the **same** memory reference

Passing objects/references...

```
class Test {
    int x;
    Test(int i) { x = i; }
}
class Main {
    public static void main(String[] args)
     {
       Test t = new Test(5);
        change(t);
        System.out.println(t.x);
    }
    public static void change(Test t)
    {       t.x = 10;     }
}
```

Output: 10

```
class Main {
    public static void main(String[] args)
    {
        Test t = new Test(5);
        change(t);
         System.out.println(t.x);
    }
    public static void change(Test t)  {
        t = new Test();
        t.x = 10;     }
}
```

Output: 5

Passing objects/references…

```
class MyDate{
    int year, month, day;
    MyDate(){year=0, month=0, day=0;}
    public MyDate(int year, int m, int d){
        this.year=year; month=m; day=d;
    }
    public void copyTo(MyDate d){
        d.year=year; d.month=month; d.day=day;
    }
    public MyDate copy(){
        return new MyDate(day, month, year);
    }
}
```

Passing objects/references...

```
MyDate d1= new MyDate(2005, 9, 26);
MyDate d2= new MyDate(2000,1,1);
d1.copyTo(d2);
d2.getYear(); //???
MyDate d3 = new MyDate();
d3= d1.copy();
```

How many were **MyDate objects** created?

Garbage collector

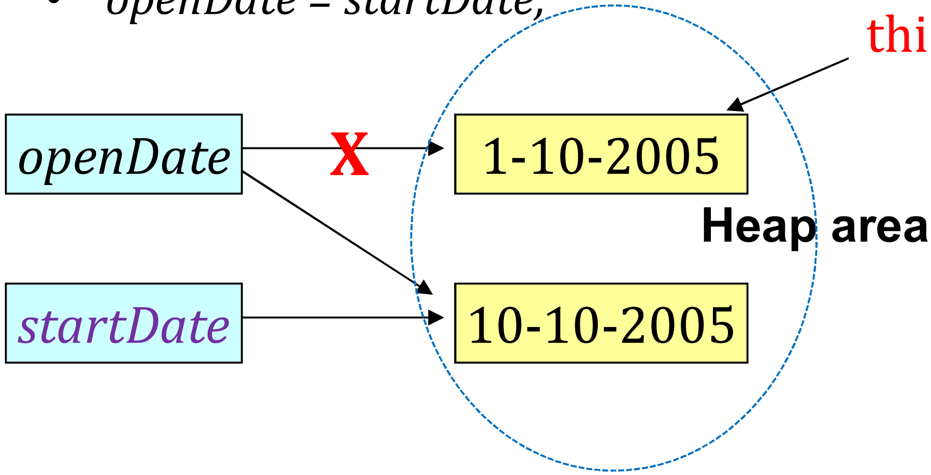
Garbage collection (GC)

- in C/C++ programmer *is responsible for both* creation & destruction of objects
  - Usually programmer neglects destruction of useless objects
- In Java, the programmer need not to care for *all those objects* which are no longer in use
  - **Garbage collector** destroys these objects
  - Main objective of Garbage collector is to free heap memory by destroying **unreachable objects**

TAKIPI

Unreachable objects

- An object is said to be **unreachable** iff *it **doesn't** contain any reference to it*
  - MyDate *openDate* = **new** MyDate(1,10,2005); *//The new MyDate object is reachable via the reference in openDate*
  - MyDate *startDate* = **new** MyDate(10,10,2005);
  - *openDate = startDate;*



this object is no longer reachable

- *openDate =null; startDate =null; //the 2nd MyDate object is no longer reachable*

Ways to make an object eligible for GC

- **Even though** programmer is ***not*** *responsible* for destroying useless objects but it is *highly recommended* to make an object **unreachable** if it is no longer required
- 4 different ways to make an object eligible for GC
  - Nullifying the reference variable
  - Re-assigning the reference variable
  - Object created inside method
  - Island of isolation
    - is a group of objects that reference each other but they are not referenced by any active object in the application

Ways for requesting JVM to run Garbage collector

- There are 2 ways to request JVM to run Garbage Collector
- Using **System.gc()**
  - **System** class contain static method **gc()** for requesting JVM to run Garbage Collector
- Using **Runtime.getRuntime().gc()**
  - **Runtime** class allows **the application** *to interface with* the JVM in which the **application** is running

Example 1

```
public class Test{
        public static void main(String[] args) throws InterrupedException{
                Test t1= new Test()
                Test t2= new Test();

                //till here, no object's eligible for GC

                t1=null; //1 object eligible for GC
                t2= null; //now 2 object eligible for GC
                System.gc();//Calling garbage collector
        }

        //override finalize method which is called on object once before garbage
             collecting it
        protected void finalize() throws Throwable {
                System.out.println(" Finalize method called!");
        }
}
```

Example 2: Island of isolation

```
public class Test{
      Test test;
        public static void main(String[] args) throws InterrupedException{
                Test t1= new Test();
                Test t2= new Test();
                t1.test =t2, t2.test1;

                t1=null; //this object eligible for GC
                System.gc();//requesting JVM for running Garbage Collector

                t2= null; //this object eligible for GC
                Runtime.getRuntime().gc(); //requesting JVM for running Garbage Collector
        }
        //override finalize method which is called on object once before garbage collecting it
        protected void finalize() throws Throwable {
                System.out.println(“ Garbage collector called”);
                System.out.println(“Object garbage collected” + this);
        }
}
```

Finalization

- *Before destroying* an object, Garbage Collector calls **finalize()** method on the object to perform cleanup activities
  - Once finalize() method *completes*, Garbage Collector *destroys* that object
- **finalize()** method is present in Object class with following prototype
  - protected void finalize() **throws** Throwable

Notes

- In the previous example of GC
  - There's no guarantee that *any of the methods* will definitely run Garbage Collector
  - Because the **finalize()** method is called by Garbage Collector not **JVM**
- **finalize()** method of **Object** class has **empty** implementation
  - so it is recommended to override finalize() method to dispose of the system resources

this reference

“this” reference

- “this” is **a reference variable** that refers to the *current object*

1. Using “**this**” keyword to refer *current class instance variables*

```
class Test {
        int a;
        int b;
        Test(int a, int b)
        {
            this.a = a;
            this.b = b;
        }
    }
```

“this” reference…

2. Using **this** keyword to invoke *current class method*

```
//Định nghĩa lớp
class Test {
    void display(){
       this.show();
       System.out.println("outside show()");
    }
    void show(){System.out.println("inside show()");}
}
Test object = new Test();
object.display(); //???
```

Output:

inside the show function

inside the display function

“this” reference…

3. Using **this()** to invoke current class constructor

```
class Test
{
    int a=111; int b=111;
    //Default constructor
    Test()
    {
        this(222, 222);// constructor call must be the first
                       //statement in the constructor
        System.out.println("Inside  default constructor");
    }
    Test(int a; int b){ this.a=a; this.b=b; }
}
Test test= new Test();
```

“this” reference...

4. Using **this** keyword to return *the* ***current*** *class instance*

```
class Test {
    int a; int b;
    //Default constructor
    Test()    {  a = 10;  b = 20;  }
    //Method that returns current class instance
    Test get() {  return this;     }
    void display(){
        System.out.println( a + “, ” b);
    }
}
….
Test object = new Test();
object.get().display(); //???
```

Output: 10, 20

"this" reference…

5. Using **this** keyword as a method parameter

```
class Test {
    int a; int b;
    Test() //Default constructor
    {   a = 100;  b = 200;  }
    void display(Test obj){
        System.out.println( obj.a + “, ” obj.b);
    }
   void get() {   display(this);     }
}
….
Test object = new Test();
object.get(); //???
```

Output: 100, 200

“this” reference…

6. Using **this** keyword as an argument in *the constructor call*

```
class A {
    B obj;
    A(B obj)    {
        this.obj = obj;
        obj.display();
    }
}
class B {  // an inner class
    int x =0;
       B(){ x=10;  A obj = new A(this);   }
    void display() {
        System.out.println("Value of x in Class B : " + x);
    }

    public static void main(String[] args) {
       A o1= new A();
        A.B o2= o1.new B(); //must access B via an object of A
    }
}
```

Output: *Value of x in Class B : 10*