LESSON V.
Encapsulation, Overloading and Aggregation

Trinh Thanh TRUNG (MSc)
trungtt@soict.hust.edu.vn
094.666.8608

Content

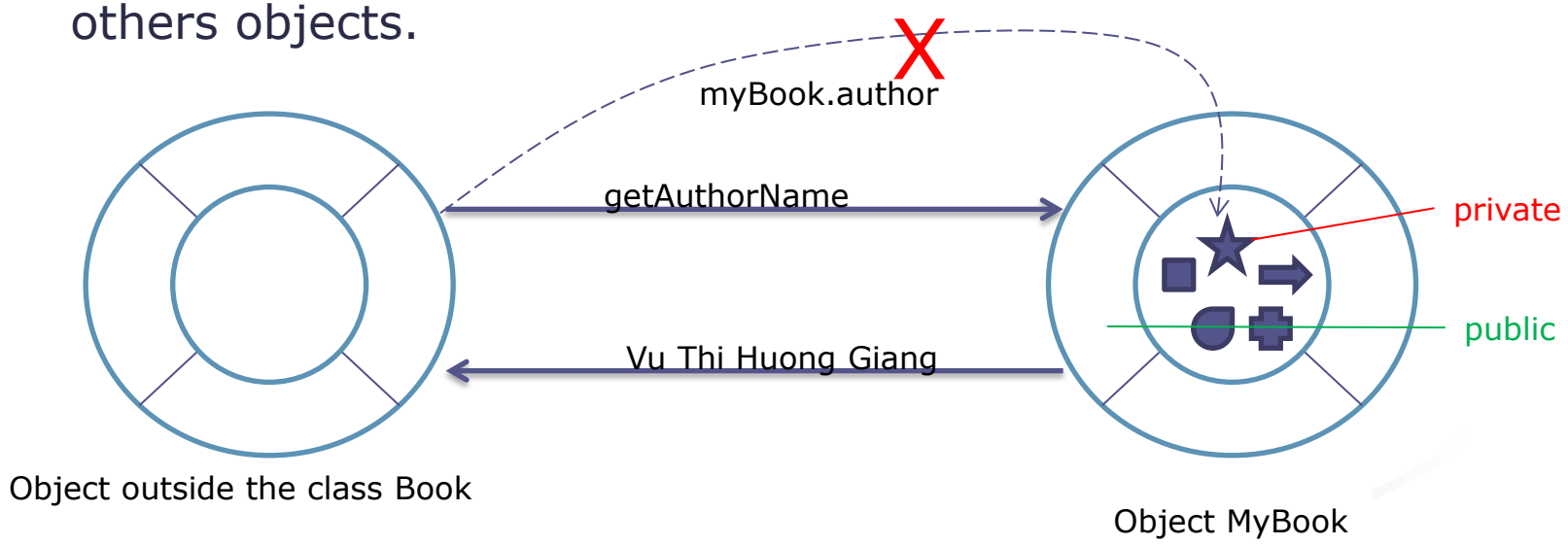
- Encapsulation
  - Visibility scope
  - Data hiding
- Overloading
  - Principles
  - Constructor overloading
- Aggregation
  - Principles
  - Order of initialization
- Class usage
  - Any classes
  - Java classes

I. Encapsulation

- Encapsulation: Prevents the code and data being randomly accessed by other code defined outside the class.
  - Group data and operations performed on these data together in a class
  - Hide details about the class implementation from the user.

1. Visibility scope (revisited)

- Scope determines the visibility of program elements with respect to other program elements.
- Given a class:
  - A private attribute or operation of an object is accessible by all others objects of the same class
  - A public attribute or operation of an object is accessible by all others objects.

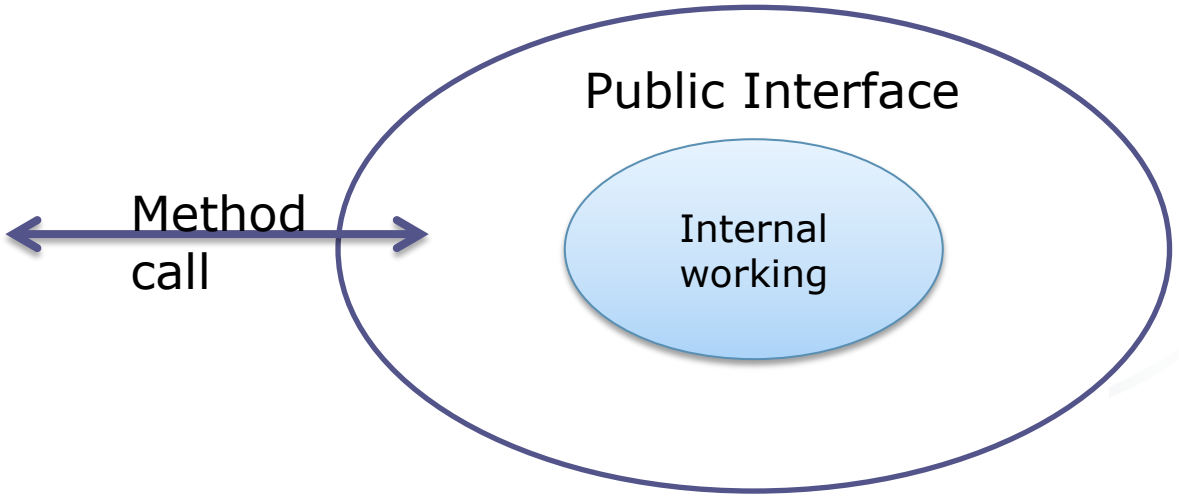


If an attribute or operation is declared private, it cannot be accessed by anyone outside the class

2. Data hiding

- A class provides data hiding
  - Data is in a unique scope; access controlled with public, private, protected keywords
  - Data is accessed through public methods; a collection of the signatures of public methods of a class is called interface.

  → Avoid illegal modification of attributes



2. Data hiding

- To get or set the value of attributes:
  - Accessor (getter): Return the current value of an attribute (value)
    - usually *getX*, which X is the name of the attribute
    - e.g. *getUsername( )*
  - Mutator (setter): Change the value of an attribute
    - usuall *setX,* which X is the name of the attribute
    - e.g. *setUsername( )*

Getter

- Getter is the query to get the value of data member of an object
- Several types of query
  - Simple query (e.g. "What is the value of x?")
  - Conditional query (e.g. "Is x greater than 10?")
  - Derived query (e.g. "What is the summary of x and y?")
- The query function should not change current state of object

Setter

- Setter is the query to change the value of data member of an object
- Control the input before changing the data
- Avoid illegal change to data members

restricted access: *private* members are *not externally accessible*; but we need to know and modify their values

set methods: *public* methods that allow clients to *modify* *private* data; also known as *mutators*

```
public class Time {
    private int hour;
    private int minute;
    private int second;

    public Time () {
        setTime(0, 0, 0);
    }

    public void setHour (int h) { hour = ( ( h >= 0 && h < 24 ) ? h : 0 ); }

    public void setMinute (int m) { minute = ( ( m >= 0 && m < 60 ) ? m : 0 ); }

    public void setSecond (int s) { second = ( ( s >= 0 && s < 60 ) ? s : 0 ); }

    public void setTime (int h, int m, int s) {
        setHour(h);
        setMinute(m);
        setSecond(s);
    }

    public int getHour () { return hour; }

    public int getMinute () { return minute; }

    public int getSecond () { return second; }
}
```

get methods: *public* methods that allow clients to *read* *private* data; also known as *accessors*

Quiz 1 – variable and method declaration

- Consider Media class in our previous lesson
  - Set a title, price, category for a media product

Quiz 1 – solution (previous)

```
public class Media {
    String title;
    String category;
    float cost;

    void displayInfo(){
       System.out.println("Title: " + title + "\nCategory: " + category
       + "\nPrice: " + cost);
    }

}
```

Quiz 1 – solution

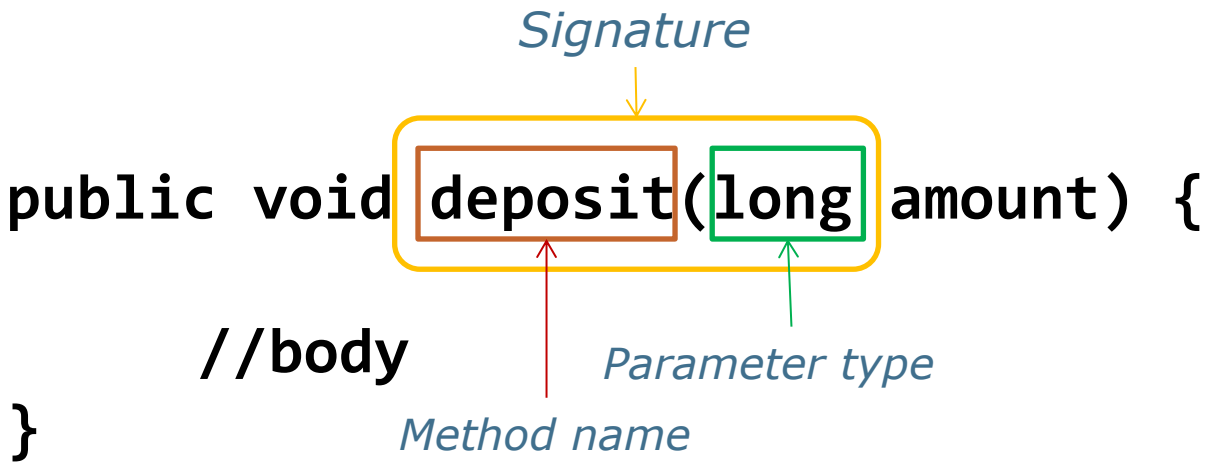
```
public class Media {
    private String title;
    private String category;
    private float cost;
    public String getTitle() {
       return title;
    }
    public String getCategory() {
       return category;
    }
    public float getCost() {
       return cost;
    }
    public void setCategory
       (String category) {
       this.category = category;
    }
    public void setCost
       (float cost) {
       this.cost = cost;
    }
    public void setTitle
       (String title) {
       this.title = title;
    }
}
```

II. OVERLOADING

1. Method signature
2. Principles
3. Contructor overloading

1. Method signature (revisited)

- The signature of a method consists of
  - The method's name
  - A list of parameter types, in which the number and the order of parameters are respected.
- Example: the signature of the following method is **`deposit(long)`**



2. Principles

- Method overloading: Methods in a class which have the same name but different *signatures*
  - Different number of parameters, or
  - Equal number of parameters, but different types
- Objectives
  - For describe the same purpose of message
  - Convenient in programming because the programmers don't have to remember too many methods name but only remember one and choose appropriate parameters

2. Principles

- Example:
  - println() in System.out.println() has 10 declaration with different types of parameter: boolean, char[], char, double, float, int, long, Object, String, and one with no parameter
  - Don't have to use different method names (e.g. "printString‟ or "printDouble‟) for each data types

Example

```
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”);
  }
}
```

Notes

- The method is only considered overloaded when they are in the same class
- Avoid abuse, only use overload methods for ones having the same purpose, function.
- When compiled, the compiler looks for the number or type of parameters to determine the appropriate method
  - If no method or more than one methods found, the compiler will give an error

Example 1

```
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)"); }
```

- `f1(5);`
- `char x='a'; f1(x);`
- `byte y=0; f1(y);`
- `float z = 0; f1(z);`

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

Example 2

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

- `f3(5);`
- `char x='a'; f3(x);`
- `byte y=0; f3(y);`
- `float z = 0; f3(z);`
- `f3(5.5);`

```
f3(int)
f3(int)
f3(short)
f5(float)
```

2. Constructor overloading

- In certain situations we need to initialize objects in many different ways
  - Create different constructors using overloading
- Example

```
public class BankAccount{
   private String owner;
   private double balance;
   public BankAccount(){owner = “noname”;}
   public BankAccount(String o, double b){
     owner = o; balance = b;
   }
}
```

2. Constructor overloading

```
public class Test{
   public static void main(String args[]){
      BankAccount acc1 = new BankAccount();
      BankAccount acc2 =
            new BankAccount(“Thuy”, 100);
   }
}
```

Example

```
public class Ship {
    private double x=0.0, y=0.0
    private double speed=1.0, angle=0.0; //in radian
    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 angle) {
       this(name, x, y);
       this.speed = speed;
       this.angle = angle;
    }
```

Example

```
    public void move() {
      move(1);
    }
    public void move(int steps) {
        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 + ").");
    }
}
```

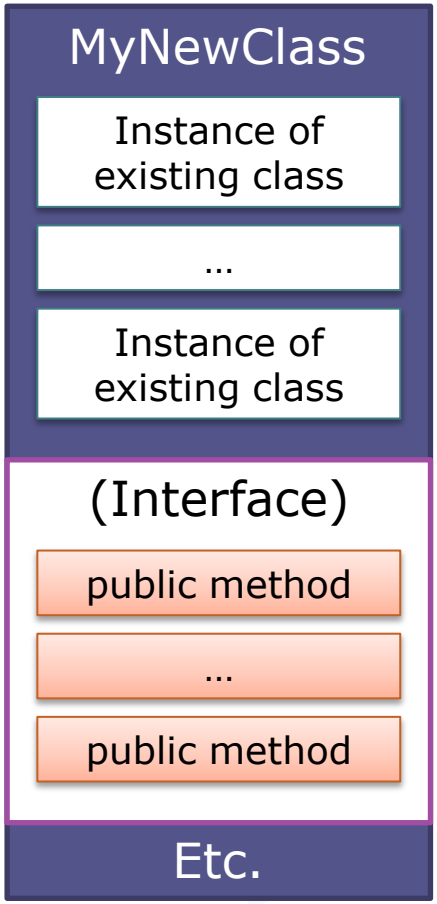
III. AGGREGATION

1. Principles
2. Order of initialization

1. Principle

- Reusing through object:
  - Create new functionality: taking existing classes and combining them into a new whole class
  - Create an interface comprising of public methods for this new class for interoperability with other code
- Relation:
  - Existing class **is a part of** new whole class
  - New whole class **has** an existing class

  → Reuse attributes and operations of existing classes through the instances of these classes.

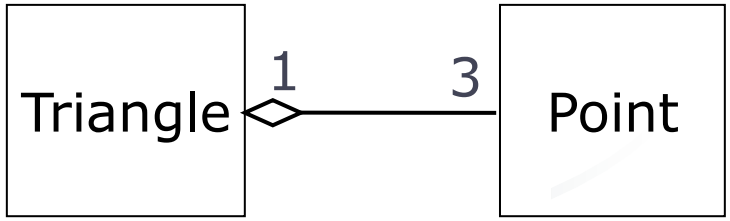


Example: Point – an existing class

```
public class Point {
    private int x; // x-coordinate
    private int y; // y-coordinate
    public Point(){}
    public Point(int x, int y) {
       this.x = x;
       this.y = y;
    }
    public void setX(int x){ this.x = x; }
    public int getX(){ return x; }
    public void setY(int y){ this.y = y; }
    public int getY(){ return y; }
    public void displayPoint(){
       System.out.print("(" + x + "," + y + ")");
    }
}
```

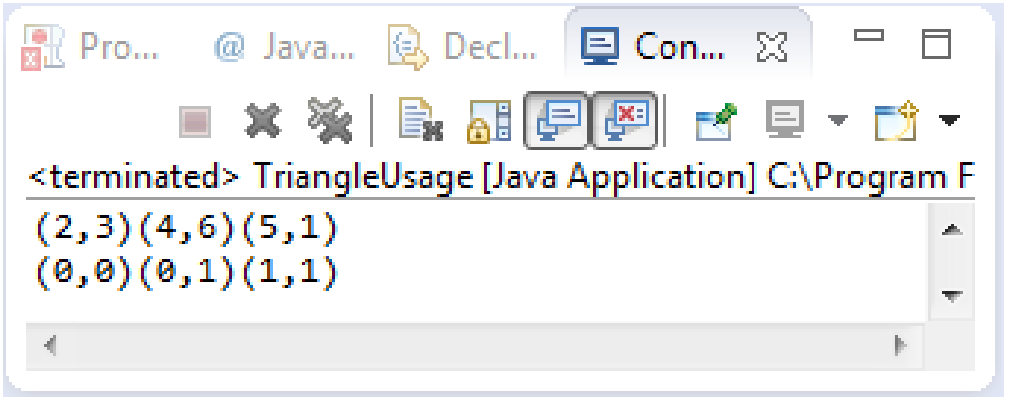
Triangle – whole class

```
public class Triangle {
    private Point d1, d2, d3;
    public Triangle(Point p1, Point p2, Point p3){
       d1 = p1; d2 = p2; d3 = p3;
    }
    public Triangle(){
        d1 = new Point();
        d2 = new Point(0,1);
        d3 = new Point (1,1);
    }
    public void displayTriangle(){
       d1.displayPoint();
       d2.displayPoint();
       d3.displayPoint();
       System.out.println();
    }
}
```



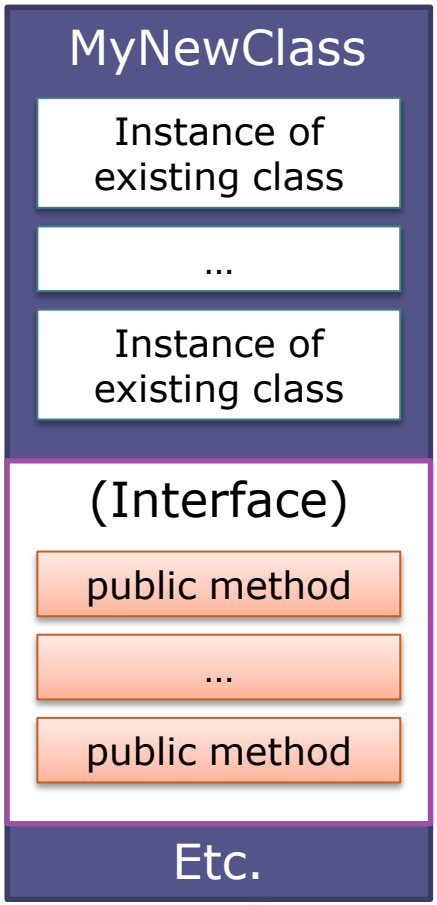
Triangle - usage

```
public class TriangleUsage {
    public static void main(String[] args) {
        Point d1 = new Point(2,3);
        Point d2 = new Point(4,6);
        Point d3 = new Point (5,1);
        Triangle triangle1 = new Triangle(d1, d2, d3);
        Triangle triangle2 = new Triangle();
        triangle1.displayTriangle();
        triangle2.displayTriangle();
    }
}
```



2. Order of initialization

- Initialize all instances of reused existing classes
  - Call their constructors
- Initialize an instance of the whole class
  - Call its constructor

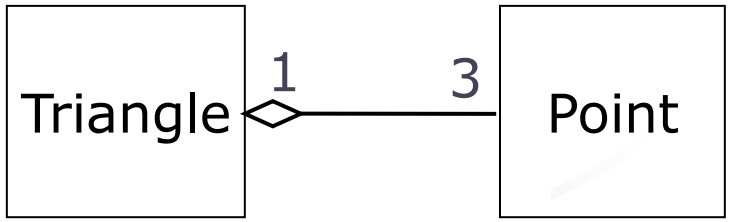


Quiz 1: Aggregation

- Implement the class Triangle in a different way
  - Hint : using the array data type.

Quiz: Aggregation
Another implementation of Triangle

```
public class AnotherTriangle {
    private Point[] Point = new Point[3];
    public AnotherTriangle(Point p1, Point p2, Point p3){
        Point[0] = p1;
        Point[1] = p2;
        Point[2] = p3;
    }
    public void displayTriangle(){
        Point[0].displayPoint();
        Point[1].displayPoint();
        Point[2].displayPoint();
        System.out.println();
    }
}
```



IV. CLASS USAGE

1. Syntax
2. Wrapper class
3. String
4. StringBuffer
5. Math

1. Syntax

- To reference to an existing class
  - Same package: Use class name as usual
  - Different package: MUST provide package name AND class name
    - **E.g. *oop.sie.ltu9.Student***
- Example

```
public class HelloNameDialog{
  public static void main(String[] args){
    String result;
    result = javax.swing.JOptionPane.showInputDialog
                                          (“Name: ”);
    javax.swing.JOptionPane.showMessageDialog
                          (null, "Hi " + result + "!");
  }
}
```

1. Syntax

- To avoid using the same package reference everytime
  - Use **import**
  - Syntax

    import <fullclassname>;
- Example

```
import javax.swing.JOptionPane;
public class HelloNameDialog{
  public static void main(String[] args){
    String result;
    result = JOptionPane.showInputDialog("Name: ");
    JOptionPane.showMessageDialog
                        (null, "Hi "+ result + "!");
    }
}
```

1. Syntax

- To import all the classes in a package
  - Use wildcard *

    import <packagename>.*;
  - Example **import java.io.*;**

Existing Java packages

- 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

2. Wrapper class

- Most of the objects collection store objects and not primitive types.
  - Primitive types can be used as object when required.
- Wrapper classes are classes that allow primitive types to be accessed as objects.
  - Wrapper class is wrapper around a primitive data type because they "wrap" the primitive data type into an object of that class.
  - Wrapper classes make the primitive type data to act as objects.

2. Wrapper class

- Two primary purposes of wrapper classes:
  - provide a mechanism to “wrap” primitive values in an object so that the primitives can be included in activities reserved for objects
  - provide an assortment of utility functions for primitives. Most of these functions are related to various conversions: converting primitives to and from String objects, etc.

2. Wrapper class

Primitive	Wrapper
boolean	java.lang.Boolean
byte	java.lang.Byte
char	java.lang.Character
double	java.lang.Double
float	java.lang.Float
int	java.lang.Integer
long	java.lang.Long
short	java.lang.Short
void	java.lang.Void

Wrapper class features

- The wrapper classes are **immutable**
- The wrapper classes are **final**
- All the methods of the wrapper classes are static.
- All the wrapper classes except Boolean and Character are subclasses of an abstract class called Number
  - Boolean and Character are derived directly from the Object class

Creating Wrapper Objects with the new Operator

- All of the wrapper classes provide two constructors (except Character):
  - One takes a primitive of the type being constructed,
  - One takes a String representation of the type being constructed

```
Boolean wbool = new Boolean("false");
Boolean ybool =new Boolean(false);
Byte wbyte = new Byte("2");
Byte ybyte =new Byte(2);
Float wfloat = new Float("12.34f");
Float yfloat = new Float(12.34f);
Double wdouble = new Double("12.56d");
Double vdouble = new Double(12.56d);
Character c1 = new Character('c');
```

(Similar to Short, Integer, Long)

Creating wrapper objects by wrapping primitives using a static method

- `valueOf(String s)`
- `valueOf(String s, int radix)`
- Return the object that is wrapping what we passed in as an argument.

```
Integer i2 = Integer.valueOf("101011", 2);
// converts 101011 to 43 and assigns the value 43 to the
//Integer object i2

Float f2 = Float.valueOf("3.14f");
// assigns 3.14 to the Float object f2
```

Using Wrapper Conversion Utilities

- *primitive-type*Value(): convert the value of a wrapped numeric to a primitive
  - No-argument methods
  - 6 conversions for each six numeric wrapper class

```
Integer i2 = new Integer(42); // make a new wrapper object

byte b = i2.byteValue(); // convert i2's value to a byte primitive

short s = i2.shortValue(); // another of Integer's xxxValue methods
double d = i2.doubleValue(); // yet another of Integer's xxxValue
                             //methods
```

Converting Strings to Primitive Types

- Convert a string value to a primitive value (except Character wrapper class):
  `static <type> parse<Type>(String s)`
  - s: String representation of the value to be converted
  - <type>: primitive type to convert to (byte, short, int, long, float, double, boolean, except char)
  - <Type> : the same as <type> with the first letter uppercased

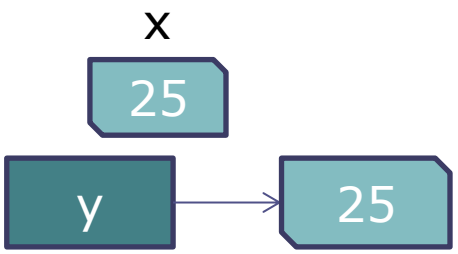
```
String s = "123";
int i = Integer.parseInt(s); //assign an int value of 123 to the int variable i
short j= Short.parShort(s) //assign an short value of 123 to the short variable j
```

Difference between wrapper class and primitive data types

- E.g

```
int x = 25;
Integer y = new Integer(25);

int z = x + y; // ERROR
int z = x + y.intValue(); // OK!
```



3. String

- String IS a class, NOT a primitive data type
- String consist of characters inside the " "

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

- Constructing a String object:

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

Concatenate String

- Use '+' operator

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

- Primitive data types used in println() are automatically converted to String
  - E.g.

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

String methods

- Use '+' operator

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

- Primitive data types used in println() are automatically converted to String
  - E.g.

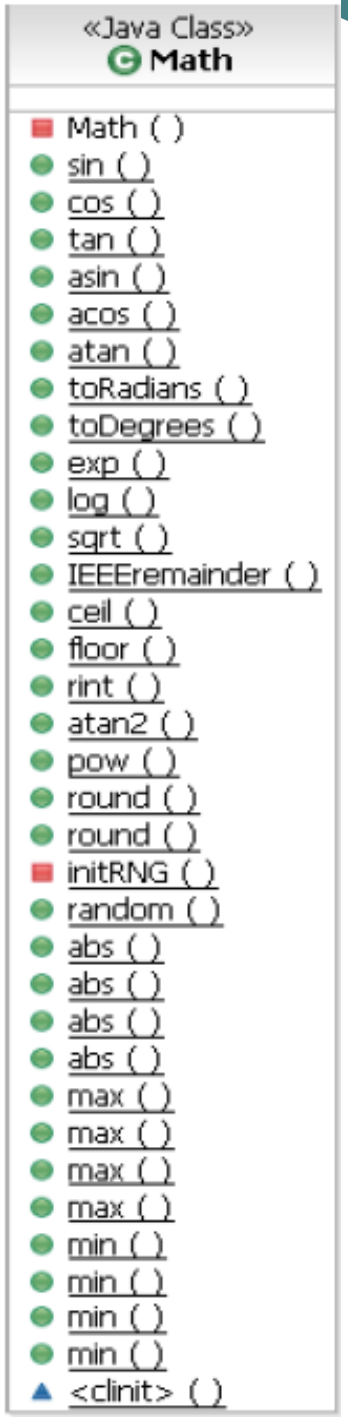
```
System.out.println("answer = " + 1 + 2 +
3);
System.out.println("answer = " + (1+2+3));
```

String 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"
```

4. Math

- The *java.lang.Math* class provides useful mathematical functions and constants.
- All of the methods and fields are static, and there is no public constructor.
- Most of methods get double parameters and return double value.
- It’s unnecessary to import any package for the *Math* class.



4. Math Class-Constants

- double E: the base of natural logarithm, 2.718...
- double PI: The ratio of the circumference of a circle to its diameter, 3.14159...

4. Math Class-Methods (1/3)

- static double abs (double x):Returns the absolute value of the argument
- static float abs (float x): Returns the absolute value of the argument
- static int abs (int x): Returns the absolute value of the argumentstatic
- static long abs (long x): Returns the absolute value of the argument

4. Math Class-Methods (2/3)

- static double acos (double): Returns the arccos of the argument (in radians), in the range (0, pi)
- static double asin (double): Returns the arcsin of the argument (in radians), in the range (-pi/2, pi/2)
- static double atan (double): Returns the arctan of the argument (in radians), in the range (-pi/2, pi/2)
- static double atan2 (double y, double x): Returns the arctan of y/x (in radians), in the range [0, 2*pi)
- static double cos (double): Returns the cosine of the argument (in radians)
- static double sin (double): Returns the sine of the argument (in radians)

4. Math Class-Methods (3/3)

- static double exp (double): Returns *e* raised to the power of the argument
- static double log (double): Returns the natural logarithm of the argumentstatic
- double pow (double base, double exp): Returns the *base* raised to the *exp* power
- static double sqrt (double): Returns the square root of the argument
- static long round (double): Returns the nearest integer to the argument
- static int round (float): Returns the nearest integer to the argument
- static double random (): Returns a pseudorandom number in the range [0, 1)