STATIC, FINAL AND CODING CONVENTION

Static Methods

Math Class

Static Variables

Initializer Blocks

Final Variables

Constant Variables

Coding Conventions

СТАТИЧНИ МЕТОДИ

Методите са два вида по тип:

- Инстантни методи – изискват да се създаде обект от неговия клас преди да бъде извикан метода.
- Статични методи – не е необходимо създаването на обект. За да достъпиш до статичен метод на друг клас е необходимо преди името на метода да напишете и името на класа, в който е този статичен метод. В Java всички utility methods са дефинирани статични – като например Math методите.

Static Methods

- Keyword **static** in declaration
- *main* method is static
- Class-level methods
  - No access to *state*, i.e., can't access instance variables/methods
- Can access *static variables*
- Can access *static methods*
- **Invocation**: *className.methodName()*, e.g., Math.*min()*
  - Saves *heap space*

PRIVATE CONSTRUCTOR

Item 4: Enforce *noninstantiability* with a *private* constructor

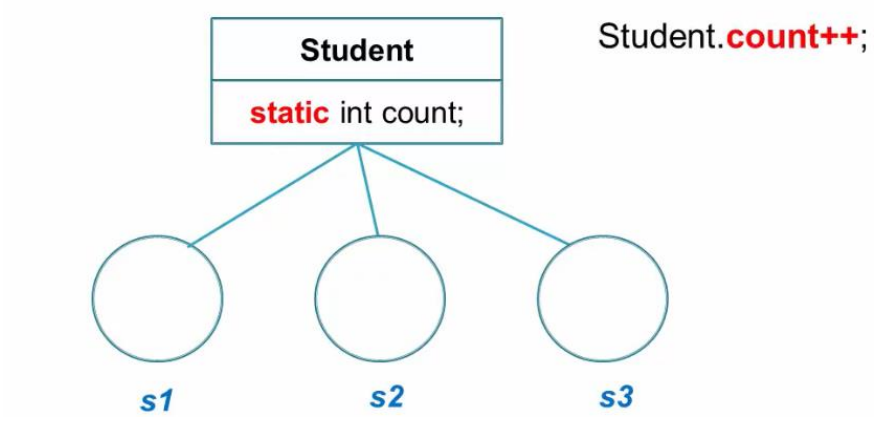
MATH CLASS

java.lang.Math

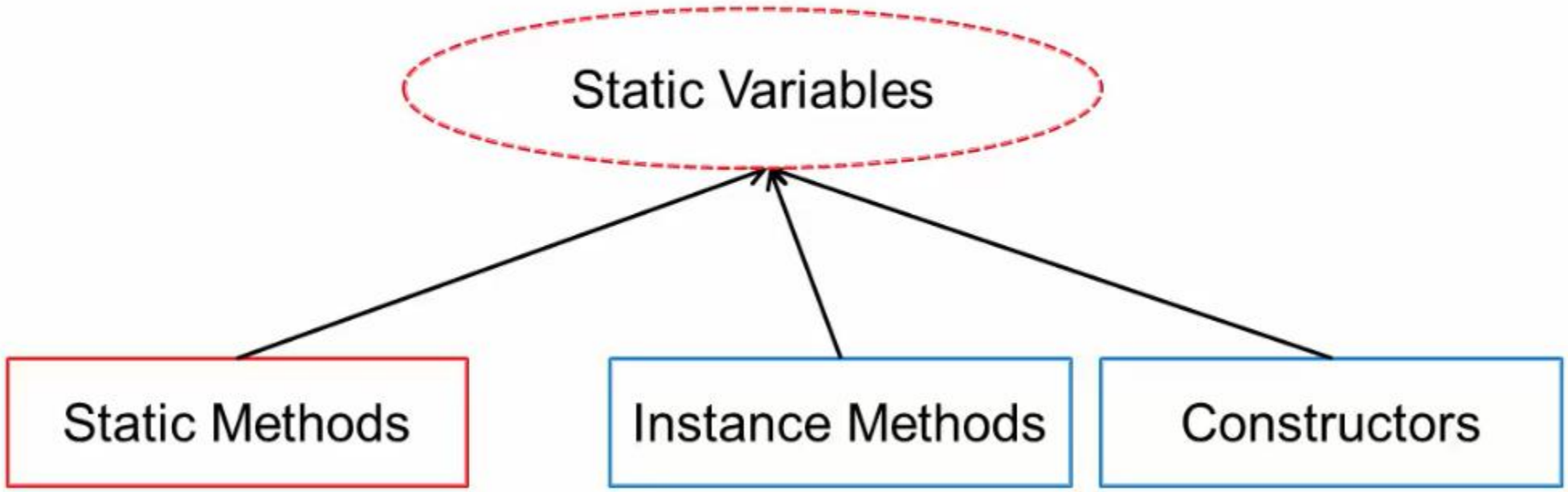
- Math.*random*()
  - *double* between 0.0 & 1.0 (exclusive)
  - 0 <= **(int) (Math.random() * 5)** < 5
- Math.*abs*()
  - Absolute value
  - Math.abs(-240) → 240
- Math.*min*()/*max*()
- Math.*sqrt*()
  - Positive square root of a double
  - **NaN** if argument is NaN or -ve
  - NaN ➔ Not-a-Number (*undefined*), *e.g.*, 0.0/0.0
- cbrt, logarithmic & trignometric functions
- Math.*round*()
  - **Nearest** *long* or *int* based on argument
  - Math.round(24.8) → 25L, Math.round(24.25f) → 24 (*int*)
- Math.*ceil*()
  - Smallest double **>=** argument & equal to integer
  - Math.ceil(20.1) returns 21.0
- Math.*floor*()
  - Largest double **<=** argument & equal to integer
  - Math.floor(24.8) returns 24.0

СТАТИЧНИ ПРОМЕНЛИВИ

- Декларитат се със служебната дума static
- Трябва да са декларирани в класа
- Трябва да им се зададе стойност по подразбиране
- Достъпност : <име на класа>.променлива
- Променлива на клас
  - създава се за всеки клас копие



Accessibility within Class



```
class StaticExample {
    int instanceVar;
    static int staticVar;

    void instanceMethod() { // can access static & instance members
        instanceVar++;
        staticVar++;
        staticMethod();
    }
    static void staticMethod() { // can access only static members
        staticVar++;
        instanceVar++;      // compiler error
        instanceMethod(); // compiler error
        (new StaticExample()).instanceMethod(); // ok
    }
}
```

```
class StaticExample {
    int instanceVar;
    static int staticVar;

    void instanceMethod() { // can access static & instance members
        instanceVar++;
        staticVar++;
        staticMethod();
    }
    static void staticMethod() { // can access only static members
        staticVar++;
        instanceVar++;      // compiler error
        instanceMethod(); // compiler error
        (new StaticExample()).instanceMethod(); // ok
    }
}
```

```
class StaticExample {
    int instanceVar;
    static int staticVar;

    void instanceMethod() { // can access static & instance members
        instanceVar++;
        staticVar++;
        staticMethod();
    }
    static void staticMethod() { // can access only static members
        staticVar++;
        instanceVar++;      // compiler error
        instanceMethod();  // compiler error
        (new StaticExample()).instanceMethod();  // ok
    }
}
```

```java
public class Student {
    private static int studentCount;

    private int id;
    private String name;
    private String gender = "male";

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

    public static int getStudentCount() { return studentCount; }

    // Constructors
    public Student(int id, String name, String gender) {
        this.id = id;
        this.name = name;
        this.gender = gender;

        studentCount++;
    }

    // method definitions
    public boolean updateProfile(String name) {
        this.name = name;
        return true;
    }
}
```

Student > src > StudentTest

StudentTest

Student.java | StudentTest.java

Student
- .idea
- out
- src
- Stude
- External L

```java
*/
class StudentTest {

    public static void main(String[] args) {
        int[] studentIds = new int[] {1001, 1002, 1003};

        // Creating first student object and setting its state
        Student student1 = new Student(studentIds[0], name: "joan", gender: "male");

        // Creating second student object and setting its state
        Student student2 = new Student(studentIds[1], name: "raj", gender: "male");

        // Creating third student object and setting its state
        Student student3 = new Student(studentIds[2], name: "anita", gender: "female");

        // Print each students name
        System.out.println("Name of student1: " + student1.getName());
        System.out.println("Name of student2: " + student2.getName());
        System.out.println("Name of student3: " + student3.getName());

        student1.setName("john");
        System.out.println("Updated name of student1: " + student1.getName());

        System.out.println("# students created so far: " + Student.getStudentCount());
    }
}
```

Run StudentTest

```
"C:\Program Files\Java\jdk1.8.0_221\bin\java" ...
Name of student1: joan
Name of student2: raj
Name of student3: anita
Updated name of student1: john
# students created so far: 3
```

Static Initializer

- Initialization needs *multiple* lines
  - Populating a data structure
  - Initialization with error handling

Static Initializer Example 1

```
static HashMap map = new HashMap();

static {
   map.put("John", "111-222-3333");
   map.put("Anita", "222-333-4444");
}
```

Static Initializer Example 2

```
static Stuff stuff;

static {
    try {
      stuff = getStuff();
    } catch(Exception e) { … }
}
```

Example 2 – Private Static Method

```
static Stuff stuff = initializeStuff();

private static Stuff initializeStuff () {
    try {
      return getStuff();
    } catch(Exception e) { … }
    return null;
}
```

Static Initializer

- Multiple initializers ~ executed in order
- Cannot reference instance members

Instance Initializer

Initializes instance variables

```
{
    ...
}
```

Constructors initialize state. *Why* instance initializer?

<u>Share code</u> between multiple constructors

Initializer *copied* into *beginning of every constructor*

ПРОМЕНЛИВА FINAL

```
public static final double PI = 3.14159265358979323846;
```

- Implies **constant**
  - Primitive ~ value is constant
  - Reference variable ~ reference is constant, *not object content*
- Don’t get *default* value
- Used with *instance*, *local*, or *static* variables

final Instance Variable

- Constant for ***life of the object***
- MUST be initialized in
  - Declaration
  - Constructor
  - Instance initializer

```
final int b = getB();

int function(){
    ...
}

int calculate(){
   for(int i = 0; i<1000; i++{
       val += b*Function();
}
.....
```

final Local Variable

- Constant for ***life of the block***

```
public void register (final int courseId) {
    courseId++; // illegal
}
```

final Static Variable

- Constant irrespective of # *instances*

```
public static final int MAX_VALUE = 0x7fffffff;
```

- MUST be initialized in
  - Declaration
  - Static initializer
- Naming convention
  - All CAPS with underscore separating words
  - private static final int COPY_THRESHOLD = 10;

Constant Variables

- Compile-time constants

  public static **final** double *PI* = 3.14159265358979323846;

- Compiler optimization

  int x = Math.PI → int x = 3.14159265358979323846;

  Stored in ***.class*** file

Constant Variables

- ✓ final
- ✓ primitive or String
- ✓ Initialized in declaration statement
- ✓ Initialized with compile-time constant expression

Constant Variables – Valid Examples

- final int x = 23;
- final String x = “hello”;
- final int x = 23 + 5;
- final String x = “hello ” + “world!”;
- final int z = 5;

  final int x = 23 + z; // z is *hard-wired*

Constant Variables – Invalid Example

```
int z = 5;
final int x = 23 + z;
```

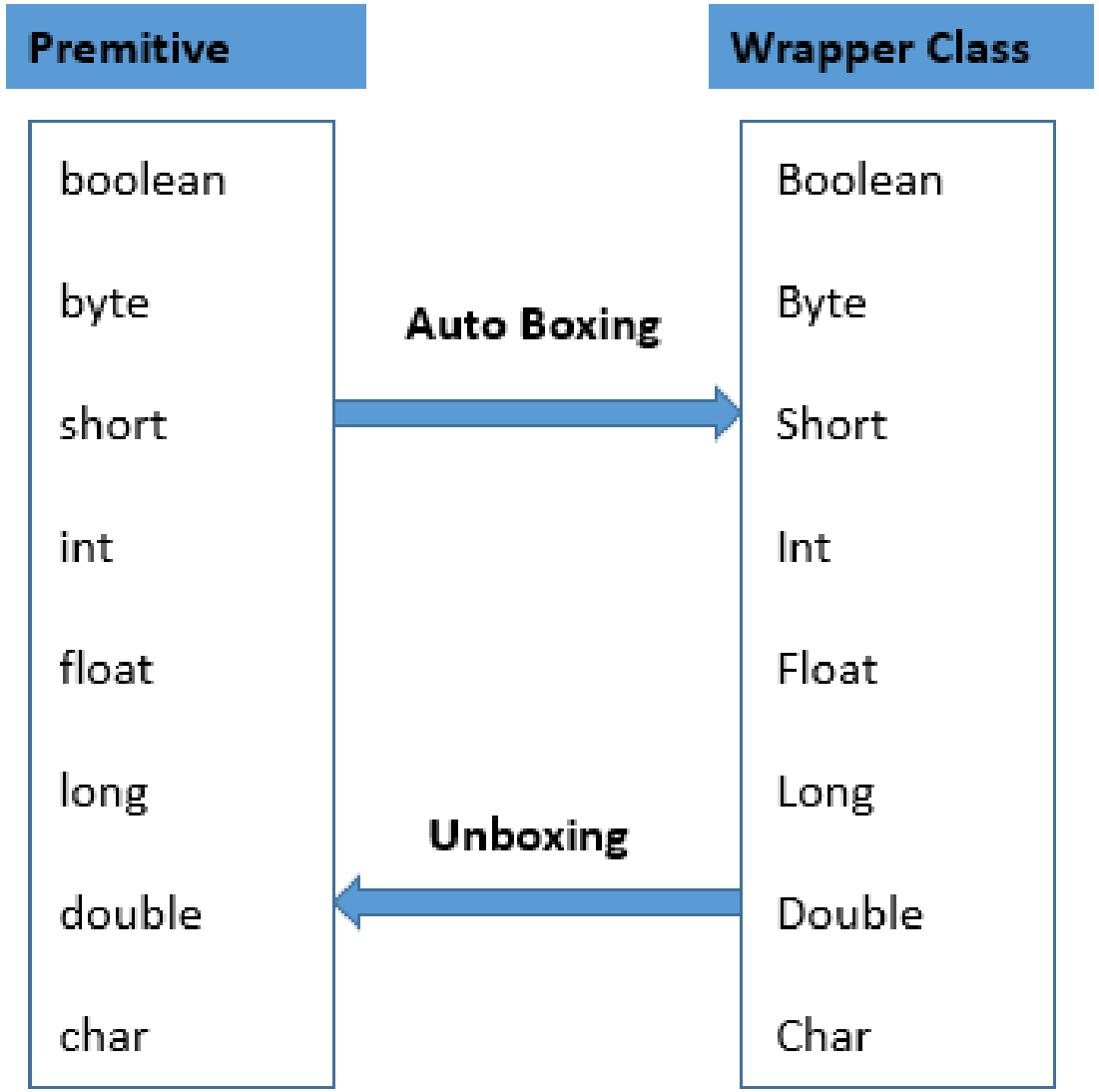
```java
static void switchExample() {
    System.out.println("\nInside switchExample ...");
    final byte month2 = 2;
    byte month = 3;
    switch (month) {
        case 1: System.out.println("January");
                break;
        case month2: System.out.println("February");
                break;
        case 3: System.out.println("March");
                break;
        default: System.out.println("April");
    }
```

```java
static final byte month2 = 2;
static void switchExample() {
    System.out.println("\nInside switchExample ...");

    byte month = 3;
    switch (month) {
        case 1: System.out.println("January");
                break;
        case month2: System.out.println("February");
                break;
        case 3: System.out.println("March");
                break;
        default: System.out.println("April");
    }
```

```java
static final byte month2;
static {
    month2 = 2;
}
static void switchExample() {
    System.out.println("\nInside switchExample ...");

    byte month = 3;
    switch (month) {
        case 1: System.out.println("January");
                break;
        case month2: System.out.println("February");
                break;
        case 3: System.out.println("March");
                break;
        default: System.out.println("April");
```

Boxed Primitives

int ~ **Integer**
long ~ **Long**
byte ~ **Byte**
short ~ **Short**
float ~ **Float**
double ~ **Double**
boolean ~ **Boolean**
char ~ **Character**



Boxed Primitive Examples

```
Integer data = new Integer(25);
Boolean data = new Boolean(true);
Character data = new Character('c');
Double data = new Double(25.5);

Integer data = new Integer("25");
Integer data = new Integer("one"); // error at runtime
```

Uses of Boxed Primitives

- *String* to *primitive* conversions
  - int i = Integer.parseInt("25");
- Useful *public static* fields
  - MAX_VALUE, MIN_VALUE
- Utility methods
  - **Character**: *isLetter*, *isDigit*, *isLetterOrDigit*, *isLowerCase*, *isUpperCase*, *isWhitespace*
  - Integer.*toBinaryString*(int)
  - Double.*isNaN*(double)

Uses of Boxed Primitives

- Populating data structures
  - Can't add primitives

```
ArrayList list = new ArrayList();
list.add(25); // illegal before Java 5
list.add(new Integer(25));
```

- Generics
  - Parameterized types, e.g., ArrayList<Integer>

Common Methods

- Unwrap (Boxed to Primitive)
  - int i = (new Integer(25)).**intValue**();
- Parsing Strings
  - <orderId>25</orderId>
  - To *primitive*: int i = Integer.**parseInt**("25");
  - To *boxed*: Integer i = Integer.**valueOf**("25"); // *simple factory*
- To String
  - String s = Integer.**toString**(25);
- Wrap: Integer.***valueOf***(int) ~ *better performance!!*

Autoboxing

```
Integer boxed = 25;
Integer boxed = new Integer(25);
```

Auto-unboxing:

```
int j = boxed;
int j = boxed.intValue();
```

Method Invocation

```
ArrayList list = new ArrayList();
list.add(25);
```

↓ *autoboxing*

```
list.add(new Integer(25));
```

Reduces Verbosity

Method Invocation

Autoboxing:

```
void go(Integer boxed) {}
go(25);
```

Auto-unboxing:

```
void go(int i) {}
go(new Integer(25));
```

Operations

```
Integer boxed = new Integer(25);
boxed++;
int i = 3 * boxed;
```

No autoboxing for ***arrays***

```
Integer[] items = new int[] {1,2}; // compiler error
```

Time & Space Efficiency

```
void veryExpensive() {
    Long sum = 0L;
    for (long i = 0; i < Integer.MAX_VALUE; i++) {
        sum = sum + i;
    }
}
```

- auto-unboxes sum
- addition
- autoboxes again

Boxed Primitives are Classes!!

- Primitives have only *values*
- Boxed primitives have *identities* too
  - == & != → identity comparison
  - <, <=, >, >= → auto-unboxing

Boxed Primitives are Classes!!

- Primitives have only *values*
- Boxed primitives have *identities* too
  - == & != → identity comparison
  - <, <=, >, >= → auto-unboxing
  - Mixed-type computations lead to confusing results

```
Integer i;
void unbelievable() {
    if (i == 0)
        System.out.println("weird!");
}
```

Boxed Primitives are Classes!!

- Primitives have only *values*
- Boxed primitives have *identities* too
  - == & != → identity comparison
  - <, <=, >, >= → auto-unboxing
  - Mixed-type computations lead to confusing results

```
Integer i;
void unbelievable() {
    if (i == 0)
        System.out.println("weird!");
}
```

```java
// Hideously slow program! Can you spot the object creation?
public static void main(String[] args) {
    Long sum = 0L;
    for (long i = 0; i < Integer.MAX_VALUE; i++) {
         sum += i;
    }
    System.out.println(sum);
}
```