Object oriented programming

(Static members, often used classes)

From practice

```
public class Main {
    public static void main(String[] args) {…}
}
```

what is `static`? What is it for?

```
public class PersonArray {
    private Person[] getPersons(int numberOfPersons) {…}
    public void sortArray() {…}
}
```

Class vs. Object – reminder

Object: a unit of a model identified by its place of existence, described by its interface (properties and behaviors), has internal states and internal structure

Class: template/type of objects with *same structure* (properties & methods)

Structure could change over levels of abstraction

Object state – reminder

Objects are identified by address, they have states and methods.

Object state

- described by a combination of hidden data members (values)
- initialized by special method (constructor)
- published via special methods (getter/setter)
- modified in a controlled way, by public methods (state changes)

Class properties

Class is a template of objects. This template can have properties:

- Help **describing the functionality** of the class, *values do not change* - constant
- Can store **business/task domain information**, *values could change*
- Logically related to **all instances**
- Variables declared in the *class* scope, **available for all instances**
- Only **one instance** of variables exist in the class

- Declared with **`static`** modifier
- Referenced by the **class itself**
- **Does not require any instances**

Class constants

- *Class properties* which values are **not subject of change**
- Written in capitals, words delimited by underscores ”_”
- Declared by `final` modifier
- Avoid hardwiring by using constants and good names
- Used in a logical unit – related to a class

Class constant

Declared by the `static` modifier:

```
class Document {
      public static final int MAX_NUMBER_OF_PAGES = 100;
      private int numberOfPages;
      public int getNumberOfPages() {…}
      public void setNumberOfPages(int newPageCount) {…}
      public void addPage(String newPageContent) {…}
}
```

Class constant – access from inside scope

```
public void addPage(String newPageContent) {
   if(numberOfPages < MAX_NUMBER_OF_PAGES) {
      numberOfPages++;
   }
   else {  <ERROR>  }
}
```

Class constant – access from outside scope

```
Document doc = new Document();
String pageContent;

if(doc.getNumberOfPages < Document.MAX_NUMBER_OF_PAGES){
      doc.AddPage(pageContent);
      doc.setNumberOfPages( doc.getNumberOfPages() );
}
else {  <ERROR>  }
```

Class properties – example

Functionality information

```
Integer.MAX_VALUE
Integer.MIN_VALUE
Integer.SIZE
```

Business information

```
class Ticket {
      public static int prize =
3500;
}
//Could change
Ticket.prize = 4000;
```

Breaks the rule of encapsulation with abstract interface

Class properties – getter/setter

```
class Ticket {
      public static final int DEFAULT_PRIZE = 3500;
      private static int prize = DEFAULT_PRIZE;
      public static int getPrize() {…}
      public static void setPrize(int newPrize) {…}
}

Ticket.setPrize( 4000 );
```

Component reference

- Objects have components (data members, methods)
- Components are declared in a block, that is their scope
- Components are referred by name

What if more components with same name exist in a scope?

Can more components with same name exist in a scope?

1. Variables with same name CAN NOT be declared in the same scope
2. Blocks can be nested → scopes can be nested
3. Scopes are extended by sub-blocks → variable is accessible in sub-blocks
4. Variables with same name CAN NOT be declared in nested scopes

Can more components with same name exist in a scope?

Class scope vs. Method scope

- Method scope is always **inside** a class scope
- Method scope **extends** the parent class scope
- **BUT redeclaration of name is allowed**

Components with same name

```
class Document {
    private String title;
    public String printTitle(Language languge) {
        String title =
language.translate(title);
    }
}
```

Component reference

- Components of objects or classes are **referenced by name**
- Compiler tries to resolve reference
  1. Inside current context – current scope
  2. Parent contexts – nearest scope
  3. If still ambiguous, requires **context descriptors**
- Context is the unit the component belongs to
- Context descriptors are delimited by dot ”.”

Components with same name

```
class Document {
    private String title;
    public String printTitle(Language languge) {
        String title =
language.translate(title);
    }
}
```

Variable is resolved to local title

Components with same name

```
class Document {
    private String title;
    public String printTitle(Language languge) {
        String title =
language.translate(this.title);
    }
}
```

Context reference is set to the object, refers to the data member

Component reference

```
number = 5;
```
// a local variable is set

// or an instance or class member

```
addPage(newPage);
```
// a method is called

// inside an object or a class

```
DocumentFactory.createDocument();
```
// a class method is called

```
document.addPage(newPage);
```
// an instance method is called

```
System.out.println("Hello world");
```

Context by declaration – Valid?

```
class Document {
    private static String title;  // class
context
    private String title;                // object
context
}
```

Both declarations are in the same (class) scope → **NOT VALID**

Stateless behaviors

- Class is a template of objects, representing a **logical units**
- Objects have states and **operations on these states**
- There could be **operations regardless of object states** – *stateless*
- Such operations are referred as **stateless behaviors** – class methods
- **Do not require instance but can create instance**
- Can call static methods and instance methods inside their scope

Class methods – examples

Integer

- compare
- getInteger
- max
- min
- parseInt

String

- Format
- Join
- vaueOf

Math

- abs
- ceil
- pow
- signum
- sqrt

Class methods - example

```
int oldest = Integer.max(myAge, yourAge);

String s1 = "The age of oldest is: ” + oldest;  //Bad practice
String s2 = String.format("The age of oldest is: %d", oldest);

double fourthPowerOf7 = Math.pow(7.0, 4.0);
```

Internal class state

Class is a set of objects, can be **stateful**:

- Usually represent business logic
- state described by a combination of hidden data members (values)
- initialized at declaration (no static constructor)
- published via special methods (getter/setter)
- modified in a controlled way, by public methods (state changes)

Internal class state

```
class Document {
   private static int documentsCreated = 0;
   private int id;
   {…}
   public Document() {
      id = ++documentsCreated;
   }
}
```

Breaks the single responsibility principal

Internal class state

```
class Document {
   private int id;

   public Document(int id) {
      this.id = id;
   }
}
```

No managed state for id creation

Stateful behaviors

- Like stateless class methods
- **Work on class state**, not on object state
- Require class state descriptors

- Syntax similar to stateless class methods

Stateful class method

```
class DocumentFactory {
    private static int documentsCreated = 0;

    public static Document createDocument() {
        documentsCreated += 1;
        return new Document(id: documentsCreated);
    }
}
```

Only responsible for appropriate document creation

Stateful class method

```
Document doc1 = DocumentFactory.createDocument();
Document doc2 = DocumentFactory.createDocument();

doc1.getId();        //returns 1
doc2.getId();        //returns 2

doc1 = DocumentFactory.createDocument();
doc1.getId();        //returns 3
```

Static classes

- Java have static classes, but **NOT LIKE C#** - not about having static components only
- In Java, class declarations can be **nested**
- Inner-classes access outer-class components
- `static` inner-classes has no reference to an instance of outer-class
  Can access only its static elements
- **Non-static** inner-classes has reference to an instance of outer-class
  Can access both static and non-static elements

Static nested classes

```
class OuterClass {
    private static String staticMsg = „Static msg";
    private String nonStaticMsg = „Non-static msg";

    public static class NestedStaticClass {
        public void printMessage() {
            System.out.println( staticMsg );
            System.out.println( nonStaticMsg ); // Invalid
        }
    }
}
```

Non-Static nested classes

```
class OuterClass {
    private static String staticMsg = „Static msg";
    private String nonStaticMsg = „Non-static msg";

    public class NestedClass {
        public void printMessage() {
            System.out.println( staticMsg );
            System.out.println( nonStaticMsg );
        }
    }
}
```

Usage of nested classes

```
OuterClass.NestedStaticClass printer =
            new OuterClass.NestedStaticClass();

printer.printMessage();

OuterClass outer = new OuterClass();
OuterClass.InnerClass innerPinter  =
            outer.new InnerClass();

inner.printMessage();
```

Object oriented programming

Often used classes

Array

- **Unmutable** collection of items of the same type
- Array is not a primitive type → object
- Instantiated by `new` operator → creating the array
- Items – data stored by the array object
  - Primitive types – value of items
  - Objects – references of items → items have to be instantiated separatelly

Array methods

- `int` **`length`** – number of items
- `<type of original array>` **`clone`** – copy contents of array (not deep)
- `boolean` **`equals`** – compares two arrays
- `String` **`toString`** – converts array to string
- `int` **`hashCode`** – gets a hash for the object (short content description)

Array usage - foreach

```
Document[] documents = new Document[NUMBER_OF_DOCUMENTS];
for(int i = 0; i < documents.length; i++) {
      documents[i] = new Document();
}
for(Document doc: documents) {                    //Copy of the array
items
      System.out.println(doc.getTitle());
}
```

`Arrays` class

- A class to **help** managing and maintaining arrays
- Does not contain data, has no data members → **stateless**
- Has only static methods, which require parameters to work on
- Can not be instantiated

`Arrays` class

- **`asList`** – convert to a mutable list – see them later
- **`binarySearch`** – quick search in a sorted array
- **`compare`** – compare arrays
- **`copyOf`** – creates a copy of content (not deep)
- **`deepEquals`**, **`deepHashCode`**, **`deepToString`** – multidimensional arrays
- **`sort`** – quick sort of content

`String` class

Class for representing character strings

- All literals are String instances
- UTF-16 representation
- Content is unmutable
- Contains methods to examine or modify content
- Special concatenation with ”+” operator

`String` class

- charAt
- compareTo
- compareToIgnoreCase
- Concat (+)
- endsWith
- getBytes
- indexOf
- isEmpty
- lastIndexOf
- length
- replace
- split
- startsWith
- substring
- toLowerCase
- toUpperCase
- trim
- valueOf

`Math` class

`Math` class

- A collection of methods to support basic mathematical operations
  - stateless
  - operands as parameters
  - monotonic
  - accurate (max. 1-2 ULP – Units in the Last Place)

Math class

- **abs**
- **ceil**
- **sin, cos**
- **exp**
- **floor**
- **log**
- **max**
- **min**
- **pow**
- **random** [0..1[
- **round**
- **sqrt**
- **toRadians**
- **toDegrees**

`Random` class

- An **instance** of this class is used to create stream of pseudo random numbers
- Pseudo random
  - instances use the same algorithm
  - output depends on seed
  - same seed results same output stream
- Seed is a parameter of constructor

Random class

```
Random rnd = new Random();
Random rnd2 = new Random(100);      // rnd1 and rnd2 will return
Random rnd1 = new Random(100);      // the same random sequence

      // (ith values will be equal)

int number = rnd.nextInt();         // get a random integer
int number = rnd.nextInt(100);      // get a random integer
[0..100[
```

Console

- Class to access the **character-based console** device associated with JVM
- Class declaration: `java.io.Console`
- Get instance: `System.console()`
- If no console associated with JVM, returns `null`
- Using without check could cause `NullPointerException`

Console

Modifier and Type	Method and Description
void	flush() Flushes the console and forces any buffered output to be written immediately .
Console	format(String fmt, Object... args) Writes a formatted string to this console's output stream using the specified format string and arguments.
Console	printf(String format, Object... args) A convenience method to write a formatted string to this console's output stream using the specified format string and arguments.
Reader	reader() Retrieves the unique Reader object associated with this console.
String	readLine() Reads a single line of text from the console.
String	readLine(String fmt, Object... args) Provides a formatted prompt, then reads a single line of text from the console.
char[]	readPassword() Reads a password or passphrase from the console with echoing disabled
char[]	readPassword(String fmt, Object... args) Provides a formatted prompt, then reads a password or passphrase from the console with echoing disabled.
PrintWriter	writer() Retrieves the unique PrintWriter object associated with this console.

System.out

- `System.console()` gets Console instance **if exists**
- When using IDE (like IntelliJ IDEA), **no console** for JVM

- `public static final PrintStream out`
- A *proxy* for streaming data out

System.out

- **`append`** – appends the specified character to output stream
- **`format`** – writes a formatted string to output stream print
- **`print`** – prints values
- **`printf`** – writes a formatted string to output stream print
- **`println`** – prints values and terminates the current line by separator
- **`write`** – writes byte(s) to the stream

System.in

- `System.console()` gets Console instance **if exists**
- When using IDE (like IntelliJ IDEA), **no console** for JVM

- `public static final InputStream in`
- A *proxy* for streaming data input

System.in

- **`available`** – returns estimated available bytes of stream
- **`mark`** – marks the current position in the stream
- **`markSupported`** – Tests if input stream supports marking
- **`read`** – reads byte(s) from the stream
- **`reset`** – repositions stream to the last mark (if supported)
- **`skip`** – skips and discards bytes from the stream

Scanner

- Class: **`java.util.Scanner`**
- A simple text scanner which can parse primitive types and strings using regular expressions
- Breaks its input into tokens using a delimiter pattern, which by default matches whitespace
- The resulting tokens may then be converted into values of different types using the various next methods.
- **`InputStream`** can be a source

Scanner

- **`delimiter`** – returns the delimiter pattern used to split input
- **`findInLine`** – Attempts to find a pattern (reg. exp.) in a line
- **`hasNext`** – returns true if scanner has token on input
- **`next`** – returns next complete token
- **`skip`** – skips input that matches specified pattern
- **`useDelimiter`** – sets the scanner's delimiter pattern

IDE console example

```
Scanner consoleScanner = new Scanner(System.in);
int nextNumber = consoleScanner.nextInt();
nextNumber = (nextNumber  * 2) + 14
System.out.println(”Computed: ” + nextNumber);
System.out.format(”Computed: %d”, nextNumber);
```