НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Object-Oriented Programming in the Java language

Part 1. Introduction to Objects

Yevhen Berkunskyi, NUoS
eugeny.berkunsky@gmail.com
http://www.berkut.mk.ua

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

The progress of abstraction

- Assembly language is a small abstraction of the underlying machine.
- Many so-called “imperative” languages that followed (such as FORTRAN, BASIC, and C) were abstractions of assembly language
- The object-oriented approach goes a step further by providing tools for the programmer to represent elements in the problem space

The progress of abstraction

- OOP allows you to describe the problem in terms of the problem, rather than in terms of the computer where the solution will run.
- There's still a connection back to the computer:

  Each object looks quite a bit like a little computer — it has a state, and it has operations that you can ask it to perform

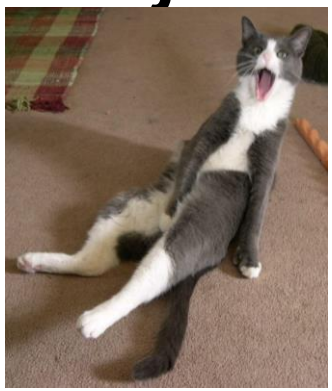
Characteristics of OOP

1. Everything is an object
2. A program is a bunch of objects telling each other what to do by sending messages
3. Each object has its own memory made up of other objects
4. Every object has a type
5. All objects of a particular type can receive the same messages

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

What object is?

An object has state, behavior and identity



This means that an object can have internal data (which gives it state), methods (to produce behavior), and each object can be uniquely distinguished from every other object — to put this in a concrete sense, each object has a unique address in memory

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

An object has an interface

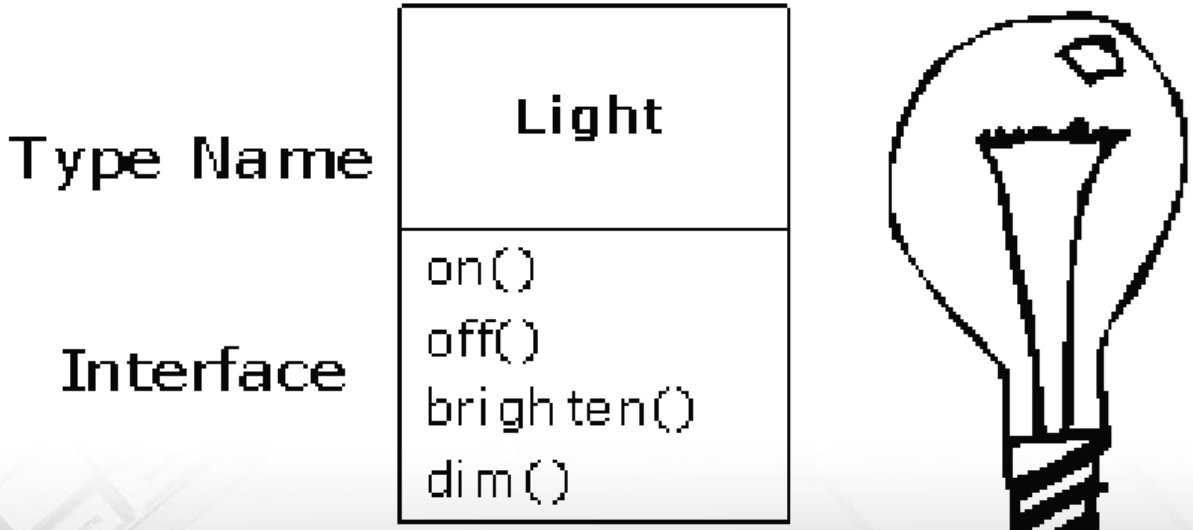
In object-oriented programming we create new data types, but all object-oriented programming languages use the “class” keyword.

When you see the word “type” think “class” and vice versa

Once a class is established, you can make as many objects of that class as you like, and then manipulate those objects as if they are the elements that exist in the problem you are trying to solve.

An object has an interface

Each object can satisfy only certain requests. The requests you can make of an object are defined by its *interface*, and the type is what determines the interface




```
Light lt = new Light();
lt.on();
```

Reusing the implementation

Composition (Aggregation)

"has-a"

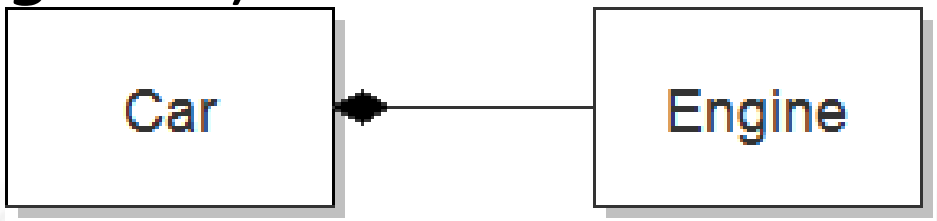
Inheritance

"is-a"

Composition / Aggregation

The simplest way to reuse a class is to just use an object of that class directly, but you can also place an object of that class inside a new class

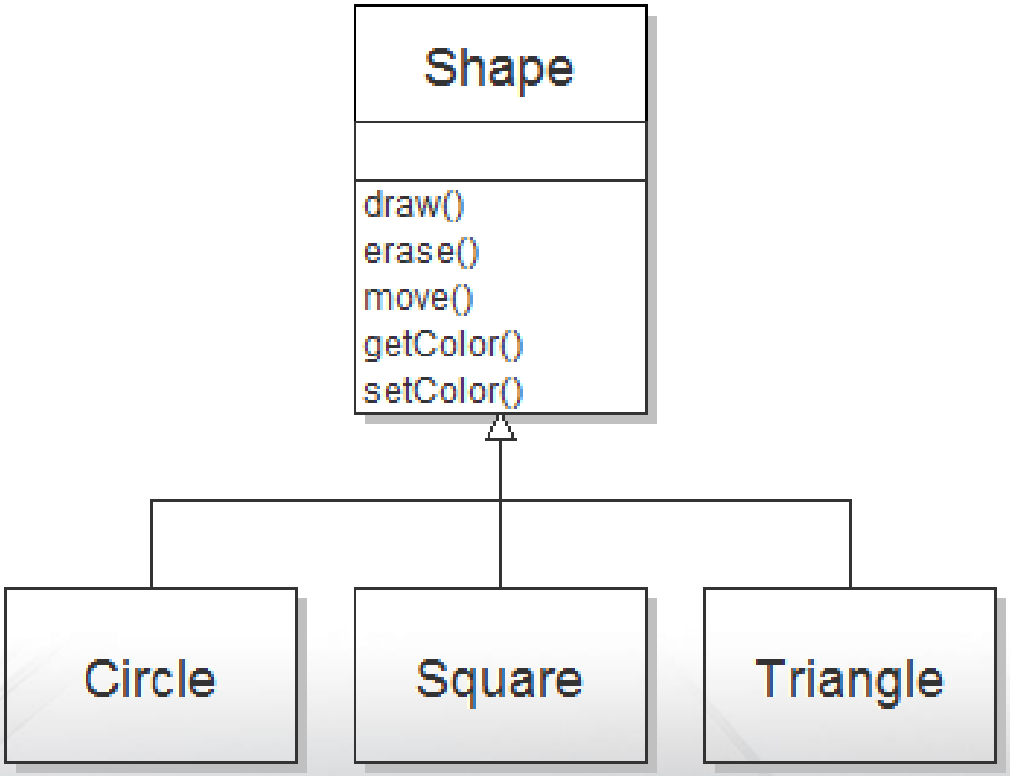
Because you are composing a new class from existing classes, this concept is called ***composition*** (if the composition happens dynamically, it's usually called ***aggregation***).



Composition is often referred to as a "**has-a**" relationship, as in "**A car has an engine**"

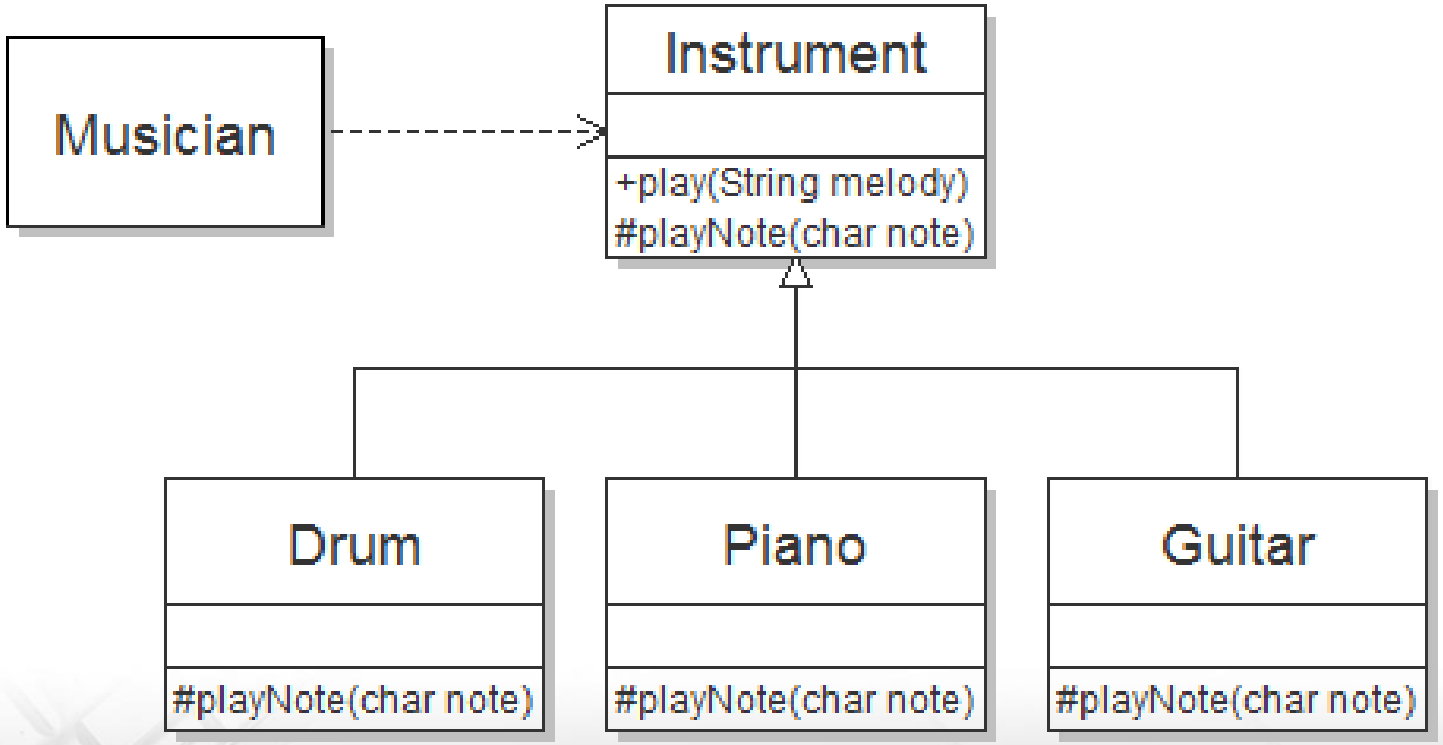
Inheritance

We can take the existing class, clone it, and then make additions and modifications to the clone



Polymorphism

Let’s consider a musician, that uses musical instrument for play



НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Example

Java Basics

You treat everything as an object, using a single consistent syntax. Although you treat everything as an object, the identifier you manipulate is actually a “reference” to an object

You must create all the objects

When you create a reference, you want to connect it with a new object. You do so, in general, with the **new** operator:

```
String s = new String("asdf");
Scanner in = new Scanner(System.in);
```

Java Basics

Special case: primitive types

Java determines the size of each primitive type.

These sizes don't change from one machine architecture to another as they do in most languages.

This size invariance is one reason Java programs are more portable than programs in most other languages.

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Primitive types

Primitive type	Size	Minimum	Maximum	Wrapper type
boolean	—	—	—	**Boolean**
char	16 bits	Unicode 0	Unicode 2^{16}- 1	**Character**
byte	8 bits	-128	+127	**Byte**
short	16 bits	-2^{15}	$+2^{15}$-1	**Short**
int	32 bits	-2^{31}	$+2^{31}$-1	**Integer**
long	64 bits	-2^{63}	$+2^{63}$-1	**Long**
float	32 bits	IEEE754	IEEE754	**Float**
double	64 bits	IEEE754	IEEE754	**Double**
void	—	—	—	**Void**

Arrays in Java

- Java array is guaranteed to be initialized and cannot be accessed outside of its range.
- The range checking comes at the price of having a small amount of memory overhead on each array as well as verifying the index at run time, but the assumption is that the safety and increased productivity are worth the expense

Arrays in Java

- When you create an array of objects, you are really creating an array of references, and each of those references is automatically initialized to a special value with its own keyword: **null**
- You can also create an array of primitives. Again, the compiler guarantees initialization because it zeroes the memory for that array

```
Instrument[] ensemble = new Instrument[5];
int[] nums = new int[10];
double[] x = {0.1, -0.4, 0.6, 0.2};
```

Access modifiers

- Java provides access specifiers to allow the library creator to say what is available to the client programmer and what is not.
- The levels of access control from “most access” to “least access” are **public**, **protected**, package access (which has no keyword), and **private**.

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Access modifiers

For members (fields and methods)

Modifiers	Members
`public`	Accessible everywhere.
`protected`	Accessible by any class in the same package as its class, and accessible only by subclasses of its class in other packages.
default (no modifier)	Only accessible by classes, including subclasses, in the same package as its class (package accessibility).
`private`	Only accessible in its own class and not anywhere else.

For top-level types (Classes, Interfaces, Enums…)

Modifiers	Top-Level Types
default (no modifier)	Accessible in its own package (*package accessibility*)
`public`	Accessible anywhere

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Other access modifiers

For top-level types (Classes, Interfaces, Enums…)

Modifiers	Classes	Interfaces	Enum types
`abstract`	A non-`final` class can be declared `abstract`. A class with an `abstract` method must be declared `abstract`. An `abstract` class cannot be instantiated.	Permitted, but redundant.	Not permitted.
`final`	A non-`abstract` class can be declared `final`. A class with a `final` method need not be declared `final`. A `final` class cannot be extended.	Not permitted.	Not permitted.

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Other access modifiers

For members (fields and methods)

Modifiers	Fields	Methods
`static`	Defines a class variable.	Defines a class method.
`final`	Defines a constant.	The method cannot be overridden.
`abstract`	Not applicable.	No method body is defined. Its class must also be designated `abstract`.
`synchronized`	Not applicable.	Only one thread at a time can execute the method.
`native`	Not applicable.	Declares that the method is implemented in another language.
`transient`	The value in the field will not be included when the object is serialized.	Not applicable.
`volatile`	The compiler will not attempt to optimize access to the value in the field.	Not applicable.

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Example

Initialization & Cleanup

- C++ introduced the concept of a ***constructor***, a special method automatically called when an object is created.
- Java also adopted the ***constructor***, and in addition has a ***garbage collector*** that automatically releases memory resources when they're no longer being used.

Guaranteed initialization with the constructor

- In Java, creation and initialization are unified concepts – you can't have one without the other.
- The constructor is an unusual type of method because it has no return value.
- This is distinctly different from a **void** return value, in which the method returns nothing but you still have the option to make it return something else.
- Constructors return nothing and you don't have an option (the **new** expression does return a reference to the newly created object, but the constructor itself has no return value).

Example

Method overloading

- Constructor's name is predetermined by the name of the class, there can be only one constructor name.
- Suppose you build a class that can initialize itself in a standard way or by reading information from a file. You need two constructors, the default constructor and one that takes a String as an argument, which is the name of the file from which to initialize the object.
  - Both are constructors, so they must have the same name – the name of the class.
  - Method overloading is a must for constructors, it's a general convenience and can be used with any method.

Distinguishing overloaded methods

- There's a simple rule: ***Each overloaded method must take a unique list of argument types***

- ***You cannot use return value types to distinguish overloaded methods***

```
void f()  {      }

int f()  { return 1; }
```

Default constructors

When (if) you create a class that has no constructors, the compiler will automatically create a default constructor for you.

```
class Bird {}
```

```
Bird b = new Bird();
```

Default constructors

However, if you define any constructors (with or without arguments), the compiler will *not* synthesize one for you

```
class Bird2 {
    Bird2(int i) {}
    Bird2(double d) {}
}
```

```
// Bird2 b = new Bird2(); // <-- Wrong!!!
   Bird2 b2 = new Bird2( i: 1);
   Bird2 b3 = new Bird2( d: 1.5);
```

this keyword

- Suppose you're inside a method and you'd like to get the reference to the current object. Since that reference is passed *secretly* by the compiler, there's no identifier for it.
- However, for this purpose there's a keyword: **this**.
- The **this** keyword – which can be used only inside a non-**static** method – produces the reference to the object that the method has been called for.

```
public class Apricot {
    void pick() { /* ... */ }
    void pit() { pick(); /* ... */ }
}
```

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Calling constructors from constructors

- When you write several constructors for a class, there are times when you'd like to call one constructor from another to avoid duplicating code.
- You can make such a call by using the **this** keyword

```
class Bird2 {
    Bird2(int i) { this(d:1.0*i); }
    Bird2(double d) {}
}
```

The meaning of static

- With the **this** keyword in mind, you can more fully understand what it means to make a method **static**. It means that there is no **this** for that particular method.
- You cannot call non-**static** methods from inside **static** methods (although the reverse is possible), and you can call a **static** method for the class itself, without any object.

Cleanup: finalization and garbage collection

It is important to distinguish between C++ and Java:

- in C++, objects always get destroyed,
- in Java, objects do not always get garbage collected.

1. ***Your objects might not get garbage collected.***
2. ***Garbage collection is not destruction.***
3. ***Garbage collection is only about memory.***

- Java doesn't allow you to create local objects – you must always use **new**.
- But in Java, there's no "delete" for releasing the object, because the garbage collector releases the storage for you.
- So, one could say that because of garbage collection, Java has no destructor.

Member initialization

Java goes out of its way to guarantee that variables are properly initialized before they are used.

```
void f() {
    int i;
    i++;  // Error - not initialized
}
```

If a primitive is a field in a class, however, things are a bit different. As you saw in previous lectures, each primitive field of a class is guaranteed to get an initial value.

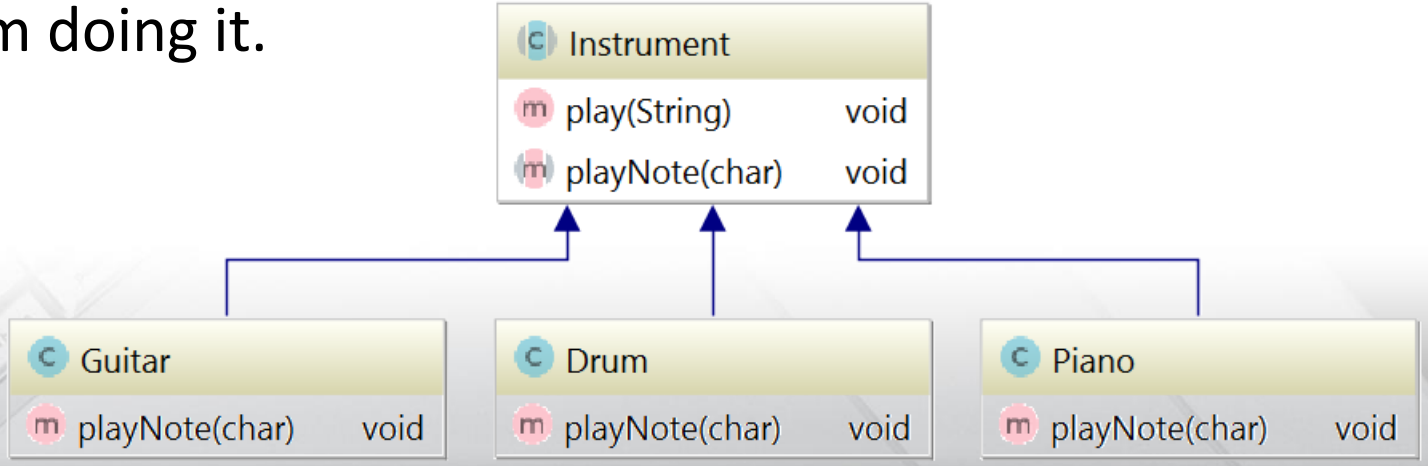
Specifying initialization

- What happens if you want to give a variable an initial value?
- One direct way to do this is simply to assign the value at the point you define the variable in the class. (Notice you cannot do this in C++)

```
public class Primitives {
    int x = 6;
    double t = 5.5;
```

Interfaces

- Interfaces and abstract classes provide more structured way to separate interface from implementation
- If you have an abstract class like **Instrument**, objects of that specific class almost always have no meaning. You create an abstract class when you want to manipulate a set of classes through its common interface.
- **Instrument** is meant to express only the interface, and not a particular implementation, so creating an Instrument object makes no sense, and you'll probably want to prevent the user from doing it.



Interfaces

- The **interface** keyword produces a completely abstract class, one that provides no implementation at all*.
- It allows the creator to determine method names, argument lists, and return types, but no method bodies*.
- An interface provides only a form, but no implementation.

* Java 8 interface changes include static methods and default methods in interfaces. Prior to Java 8, we could have only method declarations in the interfaces. But from Java 8, we can have **default methods** and **static methods** in the interfaces.

Default methods

1. Java interface default methods will help us in extending interfaces without having the fear of breaking implementation classes.
2. Java interface default methods has bridge down the differences between interfaces and abstract classes.
3. Java 8 interface default methods will help us in avoiding utility classes, such as all the Collections class method can be provided in the interfaces itself.
4. Java interface default methods will help us in removing base implementation classes, we can provide default implementation and the implementation classes can chose which one to override.

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Default methods

5. One of the major reason for introducing default methods in interfaces is to enhance the Collections API in Java 8 to support lambda expressions.
6. If any class in the hierarchy has a method with same signature, then default methods become irrelevant. A default method cannot override a method from java.lang.Object.
7. Java interface default methods are also referred to as Defender Methods or Virtual extension methods.

Static methods

1. Java interface static method is part of interface, we can't use it for implementation class objects.
2. Java interface static methods are good for providing utility methods, for example null check, collection sorting etc.
3. Java interface static method helps us in providing security by not allowing implementation classes to override them.
4. We can't define interface static method for Object class methods.
5. We can use java interface static methods to remove utility classes such as Collections and move all of it's static methods to the corresponding interface, that would be easy to find and use..

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Example

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Questions?

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Object-Oriented Programming in the Java language

Part 1. Introduction to Objects

Yevhen Berkunskyi, NUoS
eugeny.berkunsky@gmail.com
http://www.berkut.mk.ua