НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Object-Oriented Programming in the Java language

Part 4. Using Objects in Java

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

Initialization & Cleanup

- Many C bugs occur when the programmer forgets to initialize a variable.
  - This is especially true with libraries when users don't know how to initialize a library component, or even that they must.
- Cleanup is a special problem because it's easy to forget about an element when you're done with it, since it no longer concerns you.
  - Thus, the resources used by that element are retained and you can easily end up running out of resources (most notably, memory).

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.

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

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).

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

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.

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

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();
```

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

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(); /* ... */ }
}
```

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) {}
}
```

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

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;
```

Order of initialization

- Within a class, the order of initialization is determined by the order that the variables are defined within the class.
- The variable definitions may be scattered throughout and in between method definitions, but the variables are initialized before any methods can be called – even the constructor.

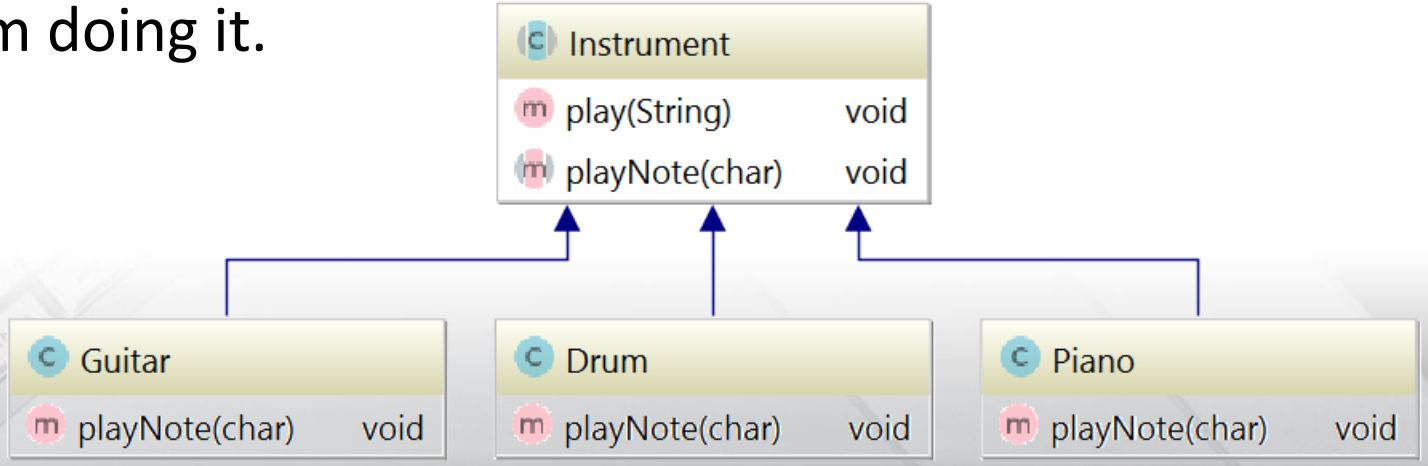
Order of initialization

```
class Window {
    Window(int marker) {
        System.out.println("Window(" + marker + ")");
    }
}
class House {
    Window w1 = new Window(1); // Before constructor
    House() {
        // Show that we're in the constructor:
        System.out.println("House()");
        w3 = new Window(33); // Reinitialize w3
    }
    Window w2 = new Window(2); // After constructor
    void f() {
        System.out.println("f()");
    }
    Window w3 = new Window(3); // At end
}
```

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

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.

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

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.

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.

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

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..

Example

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Questions?

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Object-Oriented Programming in the Java language

Part 4. Using Objects in Java

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