Unit 2: Basic elements of Object Orientation

Software Design (614G01015)

David Alonso Ríos
Eduardo Mosqueira Rey (Coordinator)

Department of Computer Science, Faculty of Computer Science

UNIVERSIDADE DA CORUÑA

Table of Contents

Table of Contents

Objects and classes

Object-Oriented Programs

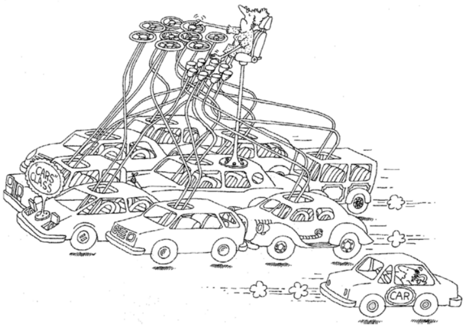
Consist of objects that communicate with each other through messages.

- **Where are the classes?**
  - They are not strictly necessary, although almost all OOPLs have them.
  - Prototype-based programming is OO and does not use classes (inheritance is achieved by cloning preexisting objects, i.e., prototypes, and extending their functionalities).
  - Example: JavaScript (Although in version 6 from 2015 classes were definitely added).

Definition of class

Class

A template that describes the structure and behavior of a particular kind of object and allows creating objects of that type.



A class represents a set of objects that share a common structure and a common behavior.

The definition of a class includes...

- **Structure**
  - Defines the state (i.e., attributes) and the behavior (i.e., methods) of the objects in the class.
- **Relations**
  - Defines the dependencies (e.g., inheritance) between classes.
- **Creation of new objects**
  - Defines the mechanisms for instantiating new objects of that class (constructor methods).

Defining classes in Java

Class declaration

```
[Modifiers] class Name [extends superclass]
[implements interface1, ...] { // Attributes // Methods }
```

- **Modifiers** (similar but not identical to the modifiers for attributes):
  - **`public`**: Makes the class accessible from another package.
  - **`abstract`**: Defines classes that cannot have instances.
  - **`final`**: Defines classes that cannot have "children" classes.
- **`extends` clause**:
  - Defines the "parent" class of the current class (by default, `Object`).
- **`implements` clause**:
  - Defines the interfaces that are implemented by the class.
- `abstract`, `final`, `extends` and `implements` will be explained in Unit 3.

Defining classes in Java

- In this example we define the class `Box`, along with its state, behavior and constructor methods.
- These elements can be defined in any order.

Declaring a class

```
public class Box {
    // Attributes
    private int value;

    // Methods
    public void setValue (int v) {
        value = v;
    }
    public int getValue () {
        return value;
    }

    // Constructor methods
    public Box() {
        value = 0;
    }
    public Box(int v) {
        value = v;
    }
}
```

Defining classes in Java

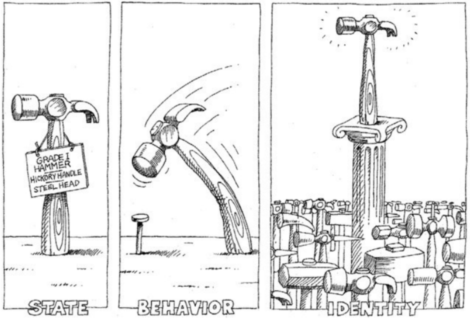
Exercise: Card game

Using the class `Box` as an inspiration...
Define the class `Card`.

Definition of object

Object

An **identifiable** element that contains **declarative characteristics** (that determine its state) and **procedural characteristics** (that model its behavior).



An object has state, exhibits some well-defined behavior, and has a unique identity.

Creating objects in Java

- Objects are instantiated using the `new` operator and a constructor method of the corresponding class.

Creating an instance

```
Box x = new Box();
Box y = new Box(5);
```

Table of Contents

Object identity

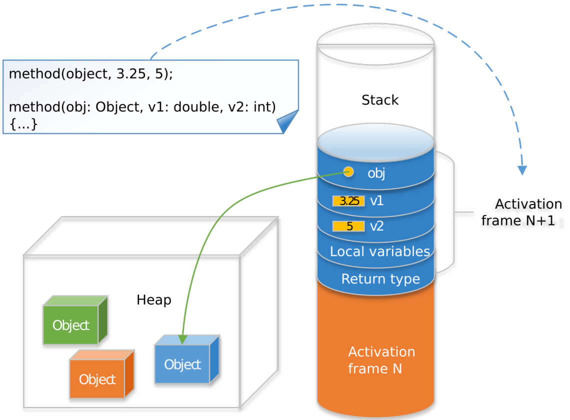
Identity

The property that distinguishes an object from other objects independently of their attributes.

- Uniqueness is achieved with an **Object Identifier (OID)**:
  - OIDs are independent from the attributes that determine the object's state.
  - OIDs are generated by the system and cannot be modified by users.
  - In Java, OIDs correspond to the address in which the object is allocated in memory.

OID implementation

- **An object type in Java is really a pointer to an object that is in the heap.**

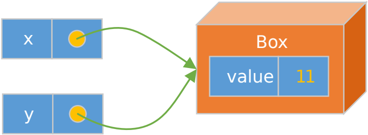


Comparisons between objects

- **Identity**
  - Two objects are identical if and only if they are in fact the same object (i.e., they have the same OID).

Code

```
Box x = new Box();
x.setValue(7);

Box y = x;
y.setValue(11);

System.out.println("x=" + x.getValue());
System.out.println("y=" + y.getValue());
```

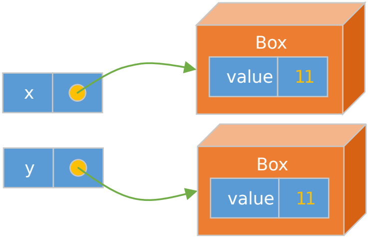


Comparisons between objects

- **Equality**
  - Two non-identical objects (i.e., with different OIDs) are typically considered equal if they have the same attributes.

Code

```
Box x = new Box();
x.setValue(11);

Box y = new Box();
y.setValue(11);

System.out.println("x=" + x.getValue());
System.out.println("y=" + y.getValue());
```



Object comparison in Java

- **Identity**: use the == operator.
- **Equality**: use the `equals()` method.

What is the result of these comparisons?

```
Box x = new Box();
x.setValue(7);
Box y = new Box();
y.setValue(7);

// Identity
if (x==y) System.out.println("x and y are identical");
else System.out.println("x and y are NOT identical");
// Equality
if (x.equals(y)) System.out.println("x and y are equal");
else System.out.println("x and y are NOT equal");
```

Object comparison in Java

- Comparing **identity** is easy – if the OIDs are different, then the objects are not identical.
- Comparing **equality** depends on the context.
  - Example: Two "Seat Ibiza" cars are considered equal if they are the same model and color, regardless of the differences in license plate.
  - The responsibility of implementing the `equals` method belongs to the class itself. You choose which attributes are relevant ("logical equality").
- **Why was the example from the previous slide "NOT equal"?**
  - Java has a default implementation for `equals`. By default, it simply calls the == operator (i.e., it compares identity).
  - It's up to you to redefine the `equals` method suitably.

Object comparison in Java

Important!!

When a class has a notion of "logical equality" that differs from the mere *identity of objects*, then that class must redefine the `equals` method.

Object comparison in Java

`equals` contract

- **Reflexivity**:
  - `x.equals(x)` must return `true`.
- **Symmetry**:
  - `x.equals(y)` must return the same value as `y.equals(x)`.
- **Transitivity**:
  - If `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` must return `true`.
- **Consistency**:
  - Unless the object's state is expressly modified, `x.equals(y)` must always return the same value every time it is invoked.
- **Use of null values**:
  - `x.equals(null)` must return `false` for each non-null value of `x`.

Object comparison in Java

- The `equals` *signature* expects an `Object` as an argument ⇒ see **U3 - Hierarchy : Inheritance**.
- If objects are identical, it returns true.
- If `obj` is null, it returns false.
- If `obj`'s class is not equal to the current class (i.e., `Box`), it returns false. (Some use less restrictive approaches with `instanceof`, but they break the `equals` "contract".)

Redefining equals in Java

```
@Override
public boolean equals(Object obj) {

    if (this == obj) { return true; }

    if (obj == null) { return false;}

    if (getClass()!=obj.getClass()) {
        return false;
    }

    .
    .
 }
```

Object comparison in Java

- We cast `obj` to `Box` to gain access to the `value` attribute in order to compare objects. This is necessary due to Java's static typing ⇒ See **U3 - Typing**
- **The `this` keyword is a pointer to the current object**. It is usually omitted, but here it is used to distinguish the current object's `value` from the other object's `value`.

Redefining equals in Java

```
@Override
public boolean equals(Object obj) {

    if (this == obj) { return true; }

    if (obj == null) { return false;}

    if (getClass()!=obj.getClass()) {
        return false;
    }

    Box other = (Box) obj;
    return this.value == other.value;
}
```

Object comparison in Java

Exercise: Card game

Write the `equals` method for the class `Card`.

Be careful when redefining `equals`

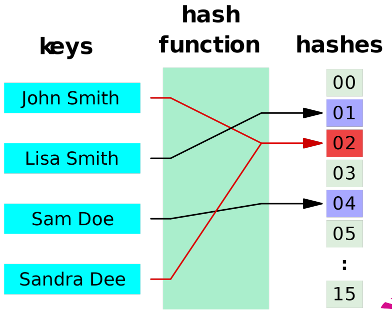
- **Do not change the parameter's type**
  - The argument *must* be an `Object`.
  - Incorrect example:
    - `public boolean equals (Box o) { ... }`
    - Compilation succeeds, but **we are not actually redefining equals**, we are ADDING a new equals alongside the old one.
    - It is erroneous and confusing.
  - If we use the `@Override` annotation (optional but highly recommended), the compiler will notify us about this mistake.
  - This is the difference between **overriding** and **overloading**, which will be discussed in **U3 - Polymorphism**.

Hash functions

Hash function

A hash function is any function that can be used to map data of arbitrary size onto data of a fixed size.

- Hash functions have multiple applications (hash tables, cryptography, checksums, etc.).
- The letter in the DNI (the Spanish ID) is an example of a hash function used to implement a checksum.



The `hashCode()` method

General contract of the `hashCode` method of `Object`

- The hashCode method must consistently return the same integer, provided no information **used in equals comparisons** on the object is modified. This integer need not remain consistent between different executions of the same application.
- **If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result**.
- Unequal objects according to the equals(Object) method, can have the same `hashCode` value. However, producing distinct integer results for unequal objects may improve the performance of hash applications.

`equals(Object)` and `hashCode()`

- The default `equals` method makes identity comparisons.
- The default `hashCode` method returns an integer representation of the object's memory address (each object has an unique *hash* value).
- **`equals` and `hashCode` should work in a coordinated way**, if we change the `equals` we should change the `hashCode` accordingly.

The hashCode() method

- A simple method for obtaining a reasonably efficient `hashCode` is to obtain an integer representation of each field involved in the `equals` and add it and multiply it successively by two arbitrary prime numbers.
- IDEs automatically suggest a default `hashCode` method, and with uniformly-distributed values. But this default `hashCode` is not necessarily valid. Depending on how `equals` is defined, it may be necessary to change the `hashCode`.

`hashCode` method for `Box` generated by NetBeans

```
@Override
public int hashCode() {
    int hash = 3;
    hash = 79 * hash + this.value;
    return hash;
}
```

The `hashCode()` method

Exercise: Card game

What would be the default `hashCode` for the class `Card`?

Be careful when redefining `equals`

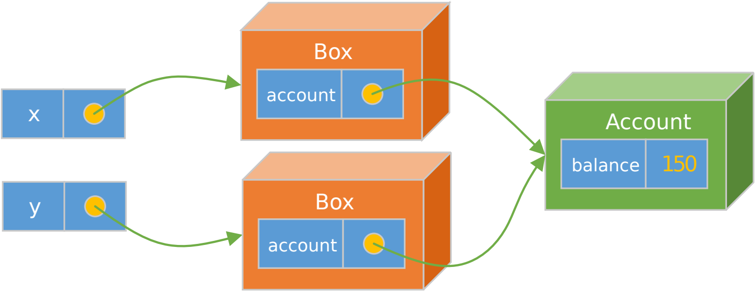
Can `equals` be based on `hashCode`?

```
public boolean equals (Object o)
{ return this.hashCode() == o.hashCode() }
```

Is it valid to return a constant hash value (e.g., 42)? Is it useful?

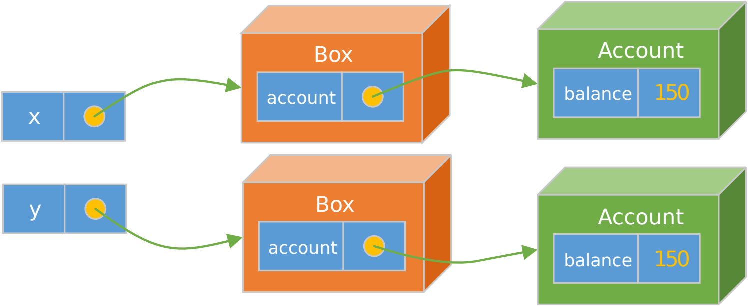
Equality in composite objects

- **Shallow equality**: Two objects are *shallowly* equal if their attributes are identical.



Equality in composite objects

- **Deep equality**: Two objects are *deeply* equal if their attributes are recursively equal.



Copying composite objects

- **Shallow copies**:
  - Easy to make (Java offers the `clone` method).
  - Dangerous (we are sharing pointers to internal objects).
  - For this reason, Java limits the use of `clone` ⇒ We'll skip this, as it is confusing and not very well implemented.
- **Deep copies**:
  - Safer.
  - Time- and memory-consuming, depending on the complexity of the objects.
  - It is recommended to define a *copy constructor* in which we return a new object that is equal to the one passed as an argument ⇒ see the subsequent chapter on Constructors.

Table of Contents

Declaring attributes

```
[Access_Specifier] [Modifiers] type attributeName
[= initial_value];
```

- **Access specifier**:
  - Where the object can be accessed from.
- **Modifiers**:
  - They allow defining "static attributes" and constants.
- **Initial value**:
  - All attributes are initialized to zero (if number), `false` (if boolean) or `null` (if object).
  - In the definition it is possible to explicitly specify an initial value, if considered necessary.

Access specifiers in Java

- **`public`**
  - The attribute can be accessed by any class.
- **Package (no specifier)**
  - This is the default specifier.
  - The attribute can be accessed by any class that belongs to the same package.
- **`protected`**
  - The attribute can be accessed by the subclasses.
  - Incidentally, it grants access to all the classes that belong to the same package (in Java – not necessarily in other PLs).
- **`private`**
  - The attribute can only be accessed by the same class (any object of that class).

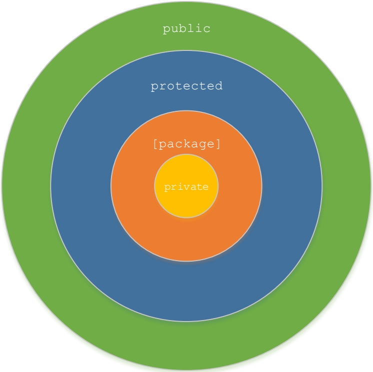
Access specifiers in Java

		Access level				
		Same class	Subclass (same package)	Subclass (other package)	Other class (same package)	Other class (other package)
Modifier	public	YES	YES	YES	YES	YES
	protected	YES	YES	YES[1]	YES	NO
	[package]	YES	YES	NO	YES	NO
	private	YES	NO	NO	NO	NO

[1] Only from objects that belong to the subclass

Access specifiers in Java

- Why do access specifiers work like this in Java? Probably to make them contained inside each other.
- "Protected" does not actually protect that much...
  it's the most permissive level after "public"!



Access specifiers in Java

Which accesses are valid?

```
package package_a;
public class Class {
    private int privateAtt;
    protected int protectedAtt;
    int packageAtt;
    public int publicAtt;

    public void accessMethod_1() {
        Class c = new Class();
        c.privateAtt   = 1;
        c.protectedAtt = 2;
        c.packageAtt   = 3;
        c.publicAtt    = 4;
    }
}
```

Which accesses are valid?

```
package package_a;
class ClassSamePackage {
    public void accessMethod_2() {
        Class c = new Class();
        c.privateAtt   = 1;
        c.protectedAtt = 2;
        c.packageAtt   = 3;
        c.publicAtt    = 4;
    }
}

package package_a;
class SubClassSamePackage
extends Class {
    public void accessMethod_3() {
        Class c = new Class();
        c.privateAtt   = 1;
        c.protectedAtt = 2;
        c.packageAtt   = 3;
        c.publicAtt    = 4;
    }
}
```

Access specifiers in Java

Which accesses are valid?

```
package package_b;
import package_a.Class;
class ClassOtherPackage {
    public void accessMethod_4() {
        Class c = new Class();
        c.privateAtt   = 1;
        c.protectedAtt = 2;
        c.packageAtt   = 3;
        c.publicAtt    = 4;
    }
}
```

Which accesses are valid?

```
package package_b;
import package_a.Class;
class SubClassOtherPackage
extends Class {
    public void accessMethod_5() {
        Class c = new Class();
        c.privateAtt   = 1;
        c.protectedAtt = 2;
        c.packageAtt   = 3;
        c.publicAtt    = 4;

        SubClassOtherPackage scop =
        new SubClassOtherPackage();
        scop.privateAtt   = 1;
        scop.protectedAtt = 2;
        scop.packageAtt   = 3;
        scop.publicAtt    = 4;
    }
}
```

Access specifiers in Java

Conventions in Object Orientation

Attributes must be declared private and must be accessed only through public read/write methods.

Read/write methods

```
class Class {
    private int value;

    public int getValue() {
        return value;
    }
    public void setValue(int value) {
        this.value=value;
    }
}
```

Rationale for private attributes

- **Controlling access**
  - Prevents from assigning invalid values (e.g., wrong DNIs).
- **Abstracting from implementation**
  - We can hide that a property is composed of several attributes (e.g., first name and last name).
- **Limiting the propagation of changes**
  - For the same reason, if the actual implementation is hidden, internal changes do not affect external classes.

`Get`/`Set` nomenclature

- **This `get/set` notation is just a convention. It is not mandatory** (e.g., Java's Collections API does not obey it).
- **It is only necessary in specific scenarios**: serializing objects, persistence frameworks, Java Beans, etc.
- In other scenarios although it is not mandatory people usually follow it because it adds clarity.

Accessing private mutable objects

- `Account` has a private `int` attribute called `balance` (NB: an `int` is a primitive type, not an object).
- A constructor method and read/write methods are provided for `balance`, but the attribute cannot be accessed outside of `Account`.

Example: Class `Account`

```java
class Account {
    // Attributes
    private int balance = 0;

    // Constructor methods
    public Account (int amount) {
        balance = amount;
    }

   // Read and write methods
    public int getBalance() {
        return balance;
    }
    public void withdraw (int amount) {
        balance = balance - amount;
    }
    public void deposit (int amount) {
        balance = balance + amount;
    }
}
```

Accessing private mutable objects

- `Client` has two private attributes. One of them is an `Account`.
- We define a constructor method and read/write methods. The `Account` attribute is only accessible through those methods. Direct access is expressly forbidden.

Be careful!

It's not that simple...

Example: Class `Client`

```
class Client {
    private String name;
    private Account account;

    public Client(String n, int a) {
        name = n;
        account = new Account(a);
    }

    public String getName() {
        return name;
    }
    public Account getAccount() {
        return account;
    }
}
```

Accessing private mutable objects

Problem

We are returning references (i.e., pointers) to private mutable objects (i.e., those that can be modified after instantiation). This means that these objects could be now modified from outside.

Example

```
...
Client cJohn = new Client("John", 1000); // One thousand euros.
Account a = cJohn.getAccount(); // We return a pointer to a private object.
a.withdraw(1000);                 // Object is mutable, so it can be modified.

// John is now 1000 euros poorer.
System.out.println("John's balance = " + cJohn.getAccount().getBalance());
...
```

Accessing private mutable objects

- **Problem:**
  - Private objects can be modified from outside.
- **Causes:**
  - Sharing references.
  - Mutable objects.
- **Possible solutions:**
  - Do not return references $\Rightarrow$ **Clone** instead.
  - Make objects **immutable** (so they cannot be modified after instantiation).

Accessing private mutable objects: Cloning

- We add a **copy constructor**, which receives an account as an argument and copies its values into a new account.
- `getAccount` in `Client` does not return the actual account but creates a "clone".
- Modifying the "clone" does not affect the original account.

Example: Cloning

```
class Account {
    ...
    // Copy constructor method
    public Account (Account a) {
        this.balance = a.balance;
    }
    ...
}

class Client {
    ...
    public Account getAccount() {
        return new Account (account);
    }
    ...
}
```

Accessing private mutable objects: Immutable objects

- We remove the write methods in `Account` (i.e., `deposit` and `withdraw`) .
- This way, an object cannot be modified after instantiation.
- We can now share references to immutable objects freely (e.g., `String`s in Java are immutable).

Example: Immutable Objects

```
class Account {
    // Attributes
    private int balance;

    // Constructor methods
    public Account (int amount) {
        balance = amount;
    }

    // Access methods
    public int getBalance() {
        return balance;
    }
}
```

Joshua Bloch. *"Effective Java". Addison-Wesley, 2001*

Favor immutability over mutability.

- Immutable objects are simple, can be shared freely and are *thread-safe* (can be accessed concurrently without synchronization).
- In Java:
  - The objects that are closest to primitive types are immutable (e.g., `String`, `BigDecimal`, etc.)
  - Exception: The class `Date` is mutable (it has methods such as `setMonth` or `setYear`), which means that dates can be modified after sharing them ⇒ New date API in Java 8.
- All this is a design pattern in its own right – the *Immutable* pattern (which will be discussed in **U6**).

Accessing private mutable objects: Immutable objects

Exercise: Card game

Should the Card class be immutable (i.e, without setters)?

Attribute modifiers

- **`static`**
  - Attributes belong to the whole class, not to a particular instance.
  - They can be modified even when no instances exist.
- **`final`**
  - Constants.
  - Once a value has been assigned, it cannot be changed.
- **Other attributes**:
  - `transient` or `volatile` are Java-specific and less used (not part of this course).

Static attributes

Static attributes

Also known as "class attributes". They belong to the whole class, not a particular instance. Therefore, they are shared by all the instances.

Static attributes

- **Example. Minimum voting age:**
  - It is an attribute shared by all people.
  - It makes sense to store it in the class and not in every instance. When it changes it will change for all instances.
  - They are accessed by putting the class name before the attribute name (although Java allows access to them using the name of an instance).

Example of a static attribute

```
public class Person {
    public int age;
    public static int votingAge = 18;

    public static void main (String[] args) {
        Person p1 = new Person();
        System.out.println("Voting age = " + Person.votingAge);
        System.out.println("Voting age = " + p1.votingAge);
    }
}
```

Static attributes

What does the following code print?

```
public class StaticAccess {
    public static int i;

    public static void main(String[] args) {
        StaticAccess c1 = new StaticAccess();
        StaticAccess c2 = new StaticAccess();

        c1.i = 5;
        c2.i = 10;

        System.out.println("c1.i = " + c1.i);
        System.out.println("c2.i = " + c2.i);
    }
}
```

Static attributes

Important!!

Even though it is allowed, avoid calling static methods through instance names. Use class names instead.

Cleaner code using class variables

```
...
StaticAccess.i = 5;
StaticAccess.i = 10;

System.out.println("StaticAccess.i = " + MyClass.i);
System.out.println("StaticAccess.i = " + MyClass.i);
...
```

Static attributes

- Modern IDEs like NetBeans warn the user about this fact and suggest replacing the instance name with the class name.

```java
public class Clase
{
    public static int i;

    public static void main(String[] args)
    {
        Clase c1 = new Clase();
        Clase c2 = new Clase();

        Clase.i = 5;
        c2.i = 10;

        System.out.println("c1.i = " + Clase.i);
        System.out.println("c2.i = " + c2.i);
    }
}
```

`final` attributes

`final` attributes

Constant attributes. Once a value has been assigned, it cannot be changed.

`final` attributes

- **Example. Class constants (PI)**:
  - Constants are often defined as class attributes, as it makes no sense to duplicate values unnecessarily.
  - Constants make the code cleaner and help to avoid "magic numbers".
  - Represented in capital letters for reasons of style.

Example. Class constants (PI number)

```
public class Circle {
    public int radius;
    public static final double PI=3.1416;

    public Circle (int r) {
        radius = r;
    }

    public double circleArea () {
        return PI*radius*radius;
    }
}
```

`final` attributes

- ***blank finals* (instance constants)**:
  - Constants are declared with no assigned value. Instead, value is assigned in the constructor method.
  - This allows having different constant values for each instance.
  - **Useful for immutable objects**: Add "final" to `number` and `suit` in `Card`

Example of *blank final*

```
public class Person {
    public final String ID; // no value!

    // Value is assigned here
    public Person (String id) {
        ID = id;
    }
}
```

Table of Contents

Messages and methods

Messages

They tell an object to perform an action accompanied by additional information (i.e., arguments) needed to perform it.

Methods

If an object accepts a message, it accepts the responsibility of carrying out the action by executing a method.

- At first, messages and methods can be considered equivalent.
- Later we will see that the same message might cause the execution of different methods depending on different circumstances ⇒ see **overloading** and **overriding** in **Unit 3**.

Defining methods

Method declaration

[Access_Specifier] [Modifiers] returnType methodName ([parameter1, parameter2, ...]) [throws ListOfExceptions] { methodBody }

- **Access specifiers**:
  - Same as the attributes.
- **Modifiers**:
  - Allow defining static methods (`static`), methods to be overridden (`abstract`), methods that cannot be overridden (`final`), etc.
- **`throws` clause**:
  - *Checked* exceptions that can be thrown by the method.

Argument passing

Definitions

- **Formal parameters**:
  - Those declared in the method's definition.
- **Actual parameters**:
  - The actual variables used when invoking the method.

Argument passing

- **By value**:
  - The values of the actual parameters are copied into the formal parameters (changes are not externally propagated).
- **By reference**:
  - The formal parameter points to the same address as the actual parameter (changes are externally propagated).

Argument passing

Argument passing in Java

All arguments are passed by value.

- **Primitive types**
  - Actual parameters (e.g., `int`, `float`, etc.) are copied into formal parameters.
- **Objects**
  - **Objects are implemented via pointers**.
  - Passing an object consists, in fact, in passing its pointer.
  - In practice we are passing the real object (stored in the heap) by reference.

Argument passing

What is the result of executing this? 500, 1000 or 1500?

```
public class Parameters {
    public static void manipulateAccount(Account c1) {
        c1.deposit(500);
        Account c2 = new Account(500);
        c1 = c2;
    }

    public static void main(String[] args) {
        Account c = new Account(1000);
        manipulateAccount(c);
        System.out.println("Balance = " + c.getBalance());
    }
}
```

`final` parameters

Would this be fixed by declaring the parameter as final?

```
public class Parameters {
    public static void manipulateAccount(final Account c1) {
        c1.deposit(1000);
        Account c2 = new Account(500);
        c1=c2; // COMPILATION ERROR
    }
}
```

- Not really. It would allow modifications in the instance, but it would throw a compilation error if you tried to change the pointer.
- `final` is more appropriate for preventing accidental modifications in references.
- Nevertheless, `final` parameters are necessary in some scenarios (that are outside the scope of this course), such as internal classes.

Parameters and shadowing

Careful!

- If an attribute and a parameter have the same name, the parameter *shadows* the attribute.
- When in doubt, Java gives priority to the element that is closest (i.e., the paramether).
- This ambiguity can be resolved by using the `this` pointer, which refers to the current object.

Attribute shadowing and the `this` pointer

```
public class Parameters {
    private int value;

    public void setValue(int value) {
        this.value = value;
    }
}
```

Return types

- In Java, return types can be either an object, a primitive type or a `void` value (which means that the method returns nothing).
- Besides returning a value, the `return` statement stops the execution of the current method (even inside a loop).
- In Java, methods can only return one element at a time. Parameters cannot be used to avoid this limitation, either. In order to return multiple results, you must group them into an object.

Method modifiers

- **`static`**:
  - Static methods belong to the class itself, not an instance.
  - Also known as "class methods".
- **`abstract`**:
  - Typically, methods with no implementation. Meant to be implemented by subclasses.
  - Will be described in detail when discussing inheritance.
- **`final`**:
  - Prevents a method from being overwritten.
  - If a class is final, all its methods are final.
  - Will be described in detail when discussing polymorphism.
- **`native`**:
  - Written in another PL (currently C and C++).
- **`synchronized`**:
  - Used in multi-threading. A synchronized method cannot be called by two threads at the same time.

Static methods

Static methods

Methods that are not executed on a particular instance of a class but on the class itself.

- They are declared using the `static` modifier.
- They can be executed even if no instances exist.
- They are invoked by typing the name of the class before the method.
- ...Or, alternatively, the name of an instance (not recommended).

Static methods

```
public class Person {
    private static int votingAge = 18;
    public static int getVotingAge() { return votingAge; }
    public static void main (String[] args) {
        System.out.println("Voting age is: "+ Person.getVotingAge());
    }
}
```

Static methods

- **Static methods and the `this` pointer.**
  - A static method can only refer to static elements (i.e., attributes and methods), because the `this` pointer does not exist in this context. (This can only be circumvented by instantiating an object).

Static methods

```
public class Static {
    private int value;

    public static void staticMethod() {
        //this.value;
        Static s = new Static());
        s.value = 5; // Access granted
    }
}
```

Compilation error

Static methods

- **When to use them:**
  - Static methods do not fit neatly into the OO approach, where a program consists in a set of objects that communicate via messages.
  - Invoking a static method is basically passing a message to the entire class.
  - Static methods are useful when the object's state is not relevant (all the information needed is in the arguments).

Static methods

```
// Java API
public class Math {
    public static int max(int a, int b) { /* ... */ }
    // ...
    }

// Our code
System.out.println("The maximum of " + a + " and " + b + ": "
                                                        + Math.max(a,b));
```

Constructor methods

Constructor methods

Methods for creating and initializing instances.

Constructor methods

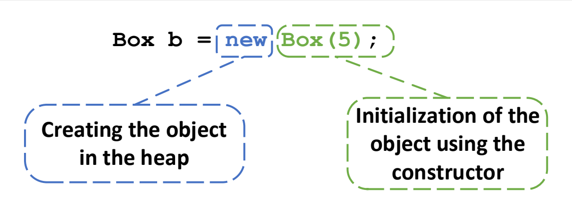
- Their name must be identical to the class name.
- They don't return any values, not even `void`.
- They are implicitly invoked when using the `new` operator.

Constructor methods

```
public class Box {
    private int value;

    public void setValue(int v) {value = v;}
    public int getValue() { return value; }

    // Constructor method
    public Box(int value) {
        this.value = value;
    }

    public static void main(String[] args) {
        Box b = new Box(5);
        System.out.println("Value = " +
                                    c.getValue());
    }
}
```

Constructors methods

- **We must distinguish two phases in the creation of an object:**
  - Creation itself through the operator `new`.
  - Initialization that is done through the constructor method once the object has been created.



- For this reason constructors can access the `this` pointer that represents the current object.

Default constructor methods

- If a class does not explicitly define a constructor method, Java assumes a "default constructor method" (without parameters) that initializes all attributes to zero, `false`, `null`, etc.
- As soon as we explicitly define a constructor method, this **default** constructor method disappears. (If you want a constructor without parameters, you must define it explicitly.)

Default constructor method

```
public class Box {
    private int value;
    public void setValue(int v) {value = v;}
    public int getValue() { return value; }

    // No constructor method

    public static void main(String[] args) {
        Box b = new Box(); // Initializes attributes to zero
        System.out.println("Value = " + b.getValue());
    }
}
```

Constructor chaining

- A class can have several constructor methods at the same time, as long as the number of parameters and/or their types differ ⇒ See **Overloading** (Unit 3).
- Constructor chaining consists in calling a constructor method from another constructor method (using `this`).

Constructor chaining, with `this`

```
public class Box {
    private int side;
    private int value;

    // Constructor methods
    public Box(int value, int side) {
        this.value = value;
        this.side = side;
    }

    public Box(int value) { this(value, 10); }

    public Box() { this(0, 10); }
}
```

Constructor methods

Is there a point to private constructors?

- **Dilemma**:
  - **NO**, because you wouldn't be able to create objects.
  - **YES**, because you can always create objects inside that particular class (but, is this useful?)

Enumerated types

Enumerated type

An enumerated type is a data type consisting of a set of named values called elements that behave as constants in the language.

Enumerated types in other languages

- **Pascal:**
  - `TColor = (RED, GREEN, BLUE)`
  - In Pascal, an enumeration is a new type that can be used, for instance, as an array index:
    `TArray = array [TColor] of integer`
- **C:**
  - `typedef enum {DESSERT, JUICE} oranges`
  - C enumerations are basically "syntactic sugar" that just gives a new name to an integer: `DESSERT=0`, `JUICE=1`
  - They allow constructions like these:
    ```
    typedef enum {FUJI, GOLDEN} apples
    oranges miOrange = (GOLDEN - FUJI) / JUICE;
    //fruit salad
    ```

Enumerated types in Java

- Java didn't used to have enumerated types, so they used to be defined as integer constants.

Enumerated types in Java: integer constants

```
class Grade {
    public static final int A        =  4;
    public static final int B        =  3;
    public static final int C        =  2;
    public static final int D        =  1;
    public static final int E_F      =  0;
    public static final int NO_SHOW  = -1;
}
/* ... */
    // The parameter is just an int
    public void insertGrade(int grade) { }

/* ... */
    p.insertGrade(Grade.C); // OK
    p.insertGrade(287); // Uncaught error
```

Enumerated types

- Later, the **TypeSafe Enum** allowed defining safer enumerations using private constructor methods.

Enumerated types in Java: Simple TypeSafe Enum

```
class Grade  {
    private Grade() {}; // Private constructor
    public static final Grade A = new Grade();
    public static final Grade B = new Grade();
    public static final Grade C = new Grade();
    public static final Grade D = new Grade();
    public static final Grade E_F = new Grade();
    public static final Grade NO_SHOW = new Grade();
}

/* ... */
    // The parameter helps to document the code
    public void insertGrade (Grade g) {   }

/* ... */
    p.insertGrade(Grade.C); // OK
    p.insertGrade(287); // Compilation error
```

Enumerated types

- **TypeSafe Enums** could get very complicated if we added things like conversion to strings, order relationships between values, etc. All very desirable to work with.

Enumerated types in Java: Complex TypeSafe Enum

```
class Grade implements Comparable {
    private int value;
    private String name;

    public int getValue() { return value; }
    public String toString() { return name; }
    private Grade(String n, int v) { name = n; value = v; }

    public static final Grade A = new Grade("A", 10);
    public static final Grade B = new Grade("B", 9);
    public static final Grade C = new Grade("C", 7);
    public static final Grade D = new Grade("D", 5);
    public static final Grade E_F = new Grade("E/F", 0);
    public static final Grade NO_SHOW = new Grade("NS", 0);

    private static int nextOrdinal = 0;
    private final int ordinal = nextOrdinal++;
    public int compareTo(Object o) { return ordinal-((Grade)o).ordinal; }

    public static final Grade[] VALUES = { A, B, C, D, E_F, NO_SHOW};
}
```

Enumerated types

Simple enums in Java (since version 5)

```
enum Grade {A, B, C, D, E_F, NO_SHOW};
```

- Included from language version 5 onwards.
- Actually, this is also “syntactic sugar” (to make things easier). The compiler basically recreates the TypeSafe Enum described before.
- Ultimately, `enum` is used here to define a class (`Grade`), with a private constructor and six predefined constants (`A`, `B`, etc.).

Enumerated types

- As enumerations are ultimately classes, it is possible to define attributes and methods for them.
- We can also create constructor methods.
- Enumeration values will be created by means of these constructors.

Complex enums in Java (since version 5)

```
enum Grade {
  A(10),
  B(9),
  C(7),
  D(5),
  E_F(0),
  NO_SHOW(0);

  private int value;
  public int getValue() {
      return value;
  }

  Grade(int value) {
      this.value = value;
  }
}
```

Enumerated types

Exercise: Card game

Define the class `Card` with enumerations.

Enumerated types

Exercise: Card game

Write the complex enum version of Number. That is, associate each number with a "literal" (e.g., ACE("A"), TWO("2"), JACK("J"), etc.).

Other benefits of enumerated types

- **Equality**:
  - Equality comparisons can be made using `==` and `equals`.
- **Order**:
  - Enumerations implement the `Comparable` interface, which means that they can make use of the `CompareTo` methods.
  - The `ordinal()` method returns the ordinal of a given enumeration value.
  - The `values()` method returns an array of enumeration values that can be used in a `for` loop:
    `for (Grade g : Grade.values()) ...`
- **Enumerations and Strings**:
  - Enumerations override the `toString()` method. For example, `Grade.C.toString()` returns "C"
  - Enumerations have a `valueOf()` method that does the opposite: `Grade.valueOf(``C'')` returns the enumeration value `Grade.C`

Destructor methods and garbage collection

Destructor methods

Destructor methods free the memory that has been allocated for objects in the heap. This memory is now available for allocating other objects. **The Java language does not have destructor methods, and frees memory using garbage collection**.

Garbage collection

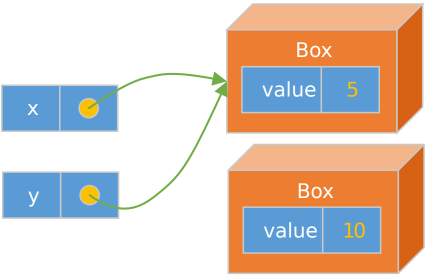
Automatic and asynchronous process that identifies those objects in the heap that are considered "garbage" and frees their memory.

- An object is considered garbage when it is not referenced anymore, i.e. when no stack element points to them so it is not accessible from the main program.
- Garbage collections relieves the programmer from the responsibility of managing memory.

Destructor methods and garbage collection

- Creating "orphan" objects is easier than it seems.

```
Box x = new Box(5);
Box y = new Box(10);

y = x;
```



toString methods

- toString() returns a String that represents an object.
- The default implementation returns "className + @ + hashCode", which is not very useful. It is meant to be overridden.
- Java implicitly calls this method when a String representation of an object is needed.

toString method

```
public class Box {
    private int value;
    public void setValue(int v) {value = v;}
    public int getValue() { return value; }
    public Box(int v) {value = v; }

    @Override
    public String toString() {
        return "[Value: " + value + "]";
    }

    public static void main(String[] args) {
        Box b = new Box(5);
        System.out.println("Box = " + b);
    }  // Prints "Box = [Value: 5]"
}
```

`main` method

- Any class can have a `main` method.
- Running a Java program means calling the `main` method of a particular class.
- The method's signature must be followed to the letter. Otherwise, Java will think it's a regular method.

`main` method

```java
public class Box {
    // ...
    public static void main(String[] args) {
        Box b = new Box(5);
        System.out.println("Box = " + b);
    }
}
```

"Running" a class

```
// Compiling
D:\>javac Box.java

// Running
D:\>java Box
Box = [Value: 5]
```

Complete version of Box

Box: part 1

```
public class Box {
    private int side;  // Side of the box
    private int value; // Internal value
    private static int numBoxes = 0; // Static value that stores the number of boxes created
    public static final double PI = 3.1416;

    // Constructors
    public Box() { this(0, 10); }
    public Box(int value) { this(value, 10); }

    public Box(int value, int side) {
        this.value = value;
        this.side = side;
        numBoxes++;
    }

    // Getters and setters
    public void setValue(int value) { this.value = value; }
    public int getValue() { return value; }
    public int getSide() { return side; }
    public void setSide(int side) { this.side = side; }
    public static int getNumBoxes() { return numBoxes;  }

    // Other methods
    public int sideArea() { return side * side; } // Area of a side
    public double perimeterCircle() { return PI * side; } // Perimeter circle side
    @Override
    public String toString() { return "[Content: " + value + "]"; }
```

Complete version of Box

Box: part 2

```
    // Equals and hashCode
    @Override
    public boolean equals(Object obj) {
        if (obj == null) { return false; }
        if (getClass() != obj.getClass()) { return false; }
        final Box other = (Box) obj;
        if (this.side != other.side) { return false; }
        if (this.value != other.value) { return false; }
            return true;
    }
    @Override
    public int hashCode() {
        int hash = 7;
        hash = 43 * hash + this.side;
        hash = 43 * hash + this.value;
        return hash;
    }

    // Test main
    public static void main(String[] args) {
        Box b1 = new Box(5, 20);
        Box b2 = new Box();

        System.out.println("area box b1 : " + b1.sideArea());
        System.out.println("Perimeter circle b2 : " + b2.perimeterCircle());
        System.out.println("Number of boxes : " + Box.getNumBoxes());
        System.out.println("b1 = " + b1);
    }
}
```

Exercise

Exercise: Card game

Finish the code for the definitive versions of the classes `Card`, `Number` and `Suit`.

- **Remember**:
  - `Card` is immutable.
  - `Number` and `Suit` are enumerations.

Unit 2: Basic elements of Object Orientation

Software Design (614G01015)

David Alonso Ríos
Eduardo Mosqueira Rey (Coordinator)

Department of Computer Science, Faculty of Computer Science

UNIVERSIDADE DA CORUÑA