

Object Oriented Programming

LEEC/MEEC - 2021/22

Lecturers

- Lectures and labs (responsible)

Alexandra Carvalho

homepage: <http://www.lx.it.pt/~asmc/>

- Labs

Gonçalo Pais, André Amaral, Francisco Ferreira

No emails, please!

Why? Because you are 100+ students.

Let's use Slack... MANDATORY!

Evaluation method

- **Final exam** [10 points out of 20]
- **UML/Java project** [10 points out of 20]
 - Team project in groups of 3 students.
 - With oral discussion.
 - UML – 2.5 points; Java – 7.5 points.

There are no minimum mark requirements!

Important dates

May 25–27 (3rd week)

Jun 20, 12:00 (7th week)

Jun 22–24 (7th week)

Project assignment

Project submission

Project oral discussion

Tuesday, July 5, 10:30

Tuesday, July 19, 18:00

1st Exam

2nd Exam

Program

1. OO history and background.
2. UML modeling: classes, objects, methods, inheritance, associations, interfaces, packages, and exceptions.
3. Java programming: classes, objects, methods, inheritance, associations, interfaces, packages, exceptions, containers, comparators, iterators, input/output, and graphical programming with Swing.

Bibliography

- **Recommended Bibliography**

- Grady Booch, James Rumbaugh, Ivar Jacobson
The Unified Modeling Language User Guide
- Ken Arnold, James Gosling
The Java Programming Language

- **Optional Bibliography**

- Martin Fowler, Kendall Scott
UML Distilled: A Brief Guide to the Standard Object Modeling Language
- Bruce Eckel
Thinking in Java

Tools

- **[Java] Java SE JDK (last version)**
<http://www.oracle.com/technetwork/java/javase/downloads/index.html>
- **[Java IDE] Eclipse, for Java developers (last version)**
<http://www.eclipse.org/downloads/eclipse-packages>
- **[UML modelling] Visual Paradigm (with IST license)**
<https://si.tecnico.ulisboa.pt/software/visual-paradigm/>

Object Oriented Programming Introduction

Motivation (1)

- Before OOP languages, the typical distribution of human resources in a software house was:
 - **80%** were devoted to **maintenance** of existing systems.
 - 20% were free to design and develop new applications.

Motivation (2)

- In the context of maintenance:
 - Most of the cost is due to the growing need of doing updates over updates.
 - Successive updates are commonly performed over the source code, splitting up from the initial specifications that are not updated accordingly.
 - Subsequent updates require to work exclusively with the source code, without the support of the high level descriptions created in the analysis and design phase.

Motivation (3)

- The increase of software production can be accomplished in two different ways:
 - Increasing the number of programmers.
 - **Increasing their productivity and/or promoting the reuse of software components already existing.**

Motivation (4)

- The decrease of maintenance points out for a growing automation in transforming high level specifications into final source code.
- In this context, **CASE tools** (*Computer-Aided Software Engineering*) appeared, allowing to integrate tasks that comprise:
 - design,
 - analysis
 - programming, and
 - tests.

Basic concepts in OO programming (1)

- **Object:**
 - Something that exists (in the real world or in an information system), that is created and eventually destroyed.
 - Meanwhile, during its existence, it might suffer updates in its **state** by interacting with other entities.
 - Its state is reflected in the value of its **attributes/fields**.
 - It changes its state whenever a **method** is called.
- For instance:
 - Object: bank account.
 - Attributes/fields: balance.
 - Methods: withdraw, deposit, interest_rate.
- **Specifying a system consists in defining a set of objects.**

Basic concepts in OO programming (2)

- **Class:**
 - In general several similar objects, of the same type, coexist and are grouped in classes.
 - In fact, we have to specify the classes and not the objects, the objects appear as **instances** of the classes.
- For instance:
 - We specify the bank account class.
 - We admit that there might be several instances of that bank account (bank account objects).
- **The system built in this manner consists in a community of objects (instances of classes) that interact with each other (through method calls).**
- **The objects of the community evolve independently of each other, except when interactions take place.**

OO ingredients (1)

- **Encapsulation:**

- Bundling of related ideas in an unit so that it can be referenced by a name.
- In OO, it refers to pack together attributes with methods.
 - It concerns combining the state jointly with the mechanism to access and modify that state.

OO ingredients (2)

- **Specialization/inheritance:**
 - Mechanism that promotes the reuse of code.
 - Inheritance allows one object to be simultaneously an instance of more than one class.
 - The inheritance mechanism of **subclass** B over a **superclass** A allows class B to reuse some or all the methods from class A. Subclasses are also called **child classes**, whereas, superclasses are also called **parent classes** or **base classes**.
 - **Simple inheritance**, where one subclass has only a direct superclass, versus **multiple inheritance**, where one subclass has more than one direct superclass.

OO ingredients (3)

- **Specialization/inheritance** (cont):
 - For instance:
 - In the specification phase, or later in maintenance, it might be necessary to introduce a time deposit which is a bank account with some particularities (more attributes, more methods, and constrained behaviour).
 - In a time deposit, the money cannot be withdrawn for a certain period of time, unless a penalty is paid.
 - Specialization allows to specify a time deposit subclass reusing all the code needed from the bank account superclass (for instance, balance and deposit).
 - Other objects that interact with bank account objects may also interact with time deposit objects, seeing them as bank accounts (without noticing that they are in fact time deposit objects).

OO ingredients (4)

- **Polymorphism:**
 - **Redefinition** of a method with the same identifier in different classes within the same hierarchy, being possible to have distinct implementations in each of the classes.
 - In OO, polymorphism is usually implemented through **dynamic binding**, i.e., the method being executed is determined only in runtime (and not in compile time).

OO ingredients (5)

- **Polymorphism** (cont):
 - For instance:
 - The time deposit redefines the method withdraw, as it differs from the withdraw method of the bank account.
 - In compile time, the system offers a way to withdraw money of all account types (without distinguishing bank account from time deposit).
 - In runtime, the withdraw method being executed is determined by the type of the account from where it was called.

Advantages/disadvantages of OO programming

- Advantages:
 - Approximation to the real world.
 - Encapsulation of information.
 - Extensible, being easier to update and/or accommodate new requisites.
 - Reuse, by inheritance of more general classes.
- Disadvantages:
 - New approach, with more complex concepts.
 - Lower performance.

Applications of OO programming

- Bank and insurances
- Robotic
- Telecommunications
- VLSI design
- Simulation
- Databases
- Mathematic modelling
- Air traffic control
- Graphical user interfaces
- ...

Object Oriented Programming

UML

UML (1)

- A **modelling language** describes the models of the application to develop.
 - Increases readability (less information than code, enabling a global view of the application).
 - Shows the structure of the application, without implementation details.
 - Graphic representation increases semantic clearness.
 - Due to complexity, OO programs are described through diverse models, where each model describes a particular aspect of the application.

UML (2)

- **UML** (*Unified Modeling Language*) combines techniques from different analyses systems:
 - Booch (G. Booch)
 - Focused on the design, that is, in the passage from the problem space to the solution space.
 - *Objected Oriented Software Engineering*, OOSE (I. Jacobson)
 - First methodology that included steps from specification to implementation.
 - *Object-modeling technique*, OMT (J. Rumbaugh)
 - Focused on the analysis for the identification, specification and description of the functional requirements of the system.
- 1st proposal made available in 1997
 - 1999 - v1.1, 2000 - v1.3, 2001 - v1.4, 2003 - v1.5, Abril 2004 - **v2.0**
- Tools: Rational Rose, Visual Paradigm, ArgoUML, Dia, etc

UML (3)

- UML v2 makes available 13 diagrams, grouped in:
 - Structural diagrams (**package**, **class**, **object**, composite structure, component, deployment and profile).
 - Behaviour models (use case, activity, state machine, communication, sequence, timing, interaction overview).
- In OOP we cover only UML **central diagrams**, in the sequence:
concept \Rightarrow UML representation \Rightarrow Java implementation

Classes – definition

- A **class** is a template for objects defined by:
 - **Identifier**
 - **Attributes/fields** (that define the state of the objects) with possible values:
 - primitive types (integers,...)
 - references to other objects (identifying relations between objects)
 - **Methods** (operations that might update the state of the objects)
- Attributes and methods are designated as **members of the class**.

Classes – example (1)

“A bank account contains always a balance and belongs to a person. A person has a name, a phone number and an ID number. It is possible to deposit and withdraw money to and from the bank account, respectively. The owner of the account may consult its balance”.

Classes

- Account
- Person

Primitive attributes

- Account: balance (float)
- Person: name (string), phoneNb (long), idNb (long)

Classes – example(2)

Reference attributes

- Account: owner (instance of a Person)

Methods

- Account:
 - withdraw (parameter: amount to withdraw)
 - deposit (parameter: amount to deposit)
 - balance (return: account balance)
- Person:
 - phoneNb (return: phone number)
 - idNb (return: ID number)

Methods – definition (1)

- A **method** is a sequence of actions, executed by an object, that may read and/or write the state of the object (value of the attributes).
- The value of the attributes reside in the object, whereas the methods reside in the class.
- **Signature** of a method:
 - method identifier;
 - identifier and type of the parameters;
 - return value.

Methods – definition (2)

- Methods are categorized as:
 - **Constructor**: executed when building the object.
 - Usually it has the same identifier as the class.
 - Never return a type.
 - Cannot be called explicitly.
 - Commonly used to initialize the attributes of the object.
 - **Destructor**: executed when destroying the object.
 - **Setter**: updates the value of the attributes.
 - **Getter**: returns the value of the attributes, without updating them.

Classes – UML

- Represented by a rectangle, divided in 3 regions:

- Identifier of the class
- Attributes (**primitive only!**)
- Methods

Account
balance: float = 0
+ deposit(amount: float) + withdraw(amount: float) + balance(): float

Note: Only primitive type attributes are represented,
reference attributes are represented as associations.

- The attribute and method regions are optional.
- A method and an attribute might have the same identifier.

Attributes – UML

Syntax

Visib Ident: Type [=Init][{Prop}]

- **Visib**: visibility
- **Ident**: attribute identifier
- **Type**: usual data types include primitive types (boolean, int, char, float,...)
- **Init**: initialization
- **Prop**: additional property

Account
balance: float = 0
+ deposit(amount: float) + withdraw(amount: float) + balance(): float

Methods – UML

Sintaxe

**Visib Ident([id:ParamType [, id:ParamType]*)
[:RetType] [{Prop}]**

- **Visib**: visibility
- **Ident**: method identifier
- **id**: parameter identifier
- **ParamType**: parameter type
- **RetType**: return type
- **Prop**: additional property

Account
balance: float = 0
+ deposit(amount: float) + withdraw(amount: float) + balance(): float

Visibility of attributes and methods – UML

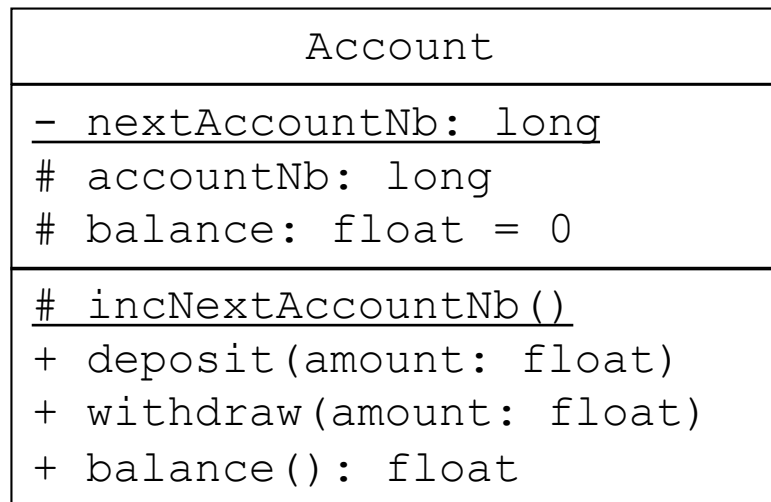
- Visibility of attributes and methods represented by a character before the identifier, and it determines the access permissions:
 - **public**: visible outside of the class (+)
 - **private**: visible only inside the class (–)
 - **protected**: visible in class and subclasses (#)
 - **package**: visible in all classes of the same package (~)

Class attributes and class methods – definition

- The attributes and methods can be:
 - instance: one for each instance of the class
 - class: one for all instances of the class

Class attributes and class methods – UML

- Represented with an underline in the class diagram.

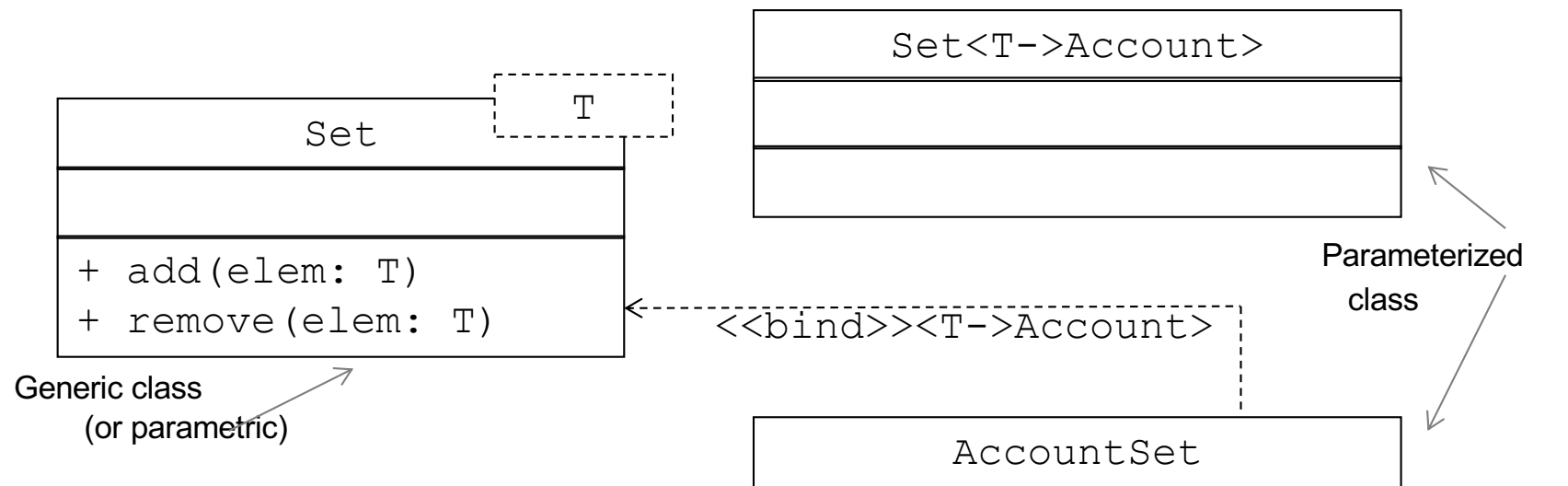


Generic classes – definition

- A **generic class** (or **parametric class**) is a class that receives as argument other classes. Instances of **generic classes** are denominated **parameterized classes**.
- Generic classes are commonly used to define collections (sets, lists, queues, trees, etc).

Generic classes – UML

- Represented in UML with a dashed box in the superior right corner of an UML representation of a class. The dashed box contains a list with the parameter types to pass to the generic class.



Relationships – definition

- Objects do not live isolated and they establish cooperative relationships within applications.
- A **relationship** is a connexion between elements.
There are different kinds of relationships:
 - **Association**: relates objects between themselves.
 - **Composition/Aggregation**: relationship that denotes that the *whole* is constituted by *parts*.
 - **Inheritance**: mechanism of generalization-specialization of classes.
 - **Realization**: a class implements a functionality offered by an interface.

Association – UML (1)

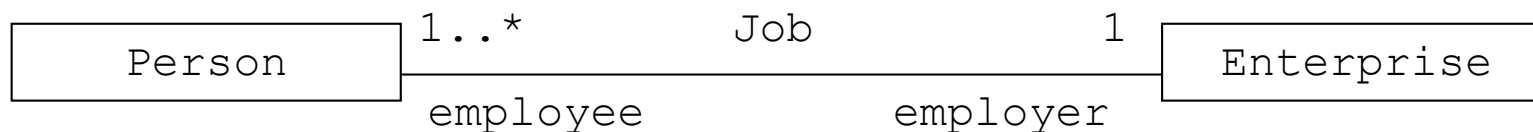
- An **association** represents a reference between objects.
- In an association is defined:
 - **Identifier** – term that describes the association.
 - **Role** – roles represented by the association in each of the related classes.
 - **Multiplicity** – number of objects associated in each of the related classes.

Association – UML (2)

- The identifier and the roles are optional.
- Associations might have diverse multiplicities:
 - exactly one (1).
 - zero or one (0..1).
 - zero or more (0..*).
 - one or more (1..*).
 - more complex multiplicities are also accepted, for instance, 0..1, 3..4, 6..*, to say any number greater than 0 except 2 and 5.

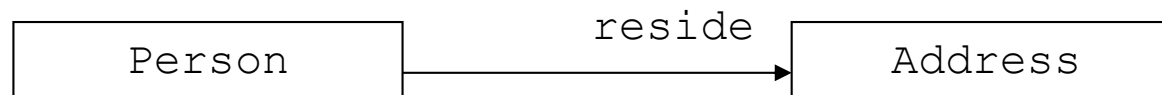
Association – UML (3)

- The association is represented by a solid line between the associated classes.
- The association identifier appears on top of the association line.
- The role of each class in the association appears in the respective endpoints of the association.
- The multiplicities also appear in the endpoints.



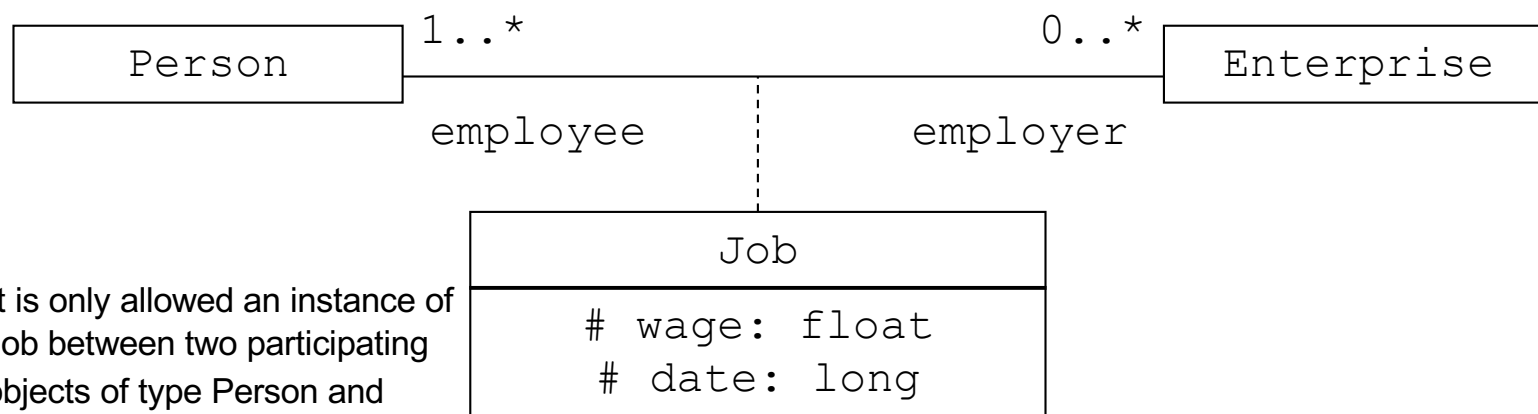
Association – UML (4)

- Associations might be **directed**, and in that case they are represented by an arrow.
- The direction of an association is related with implementation issues.



Association – UML (5)

- The associations might, by themselves, have extra information, being in that case an **association class** linked with a dashed line to the association.
- The association identifier is in that case given by the identifier of the associated class.



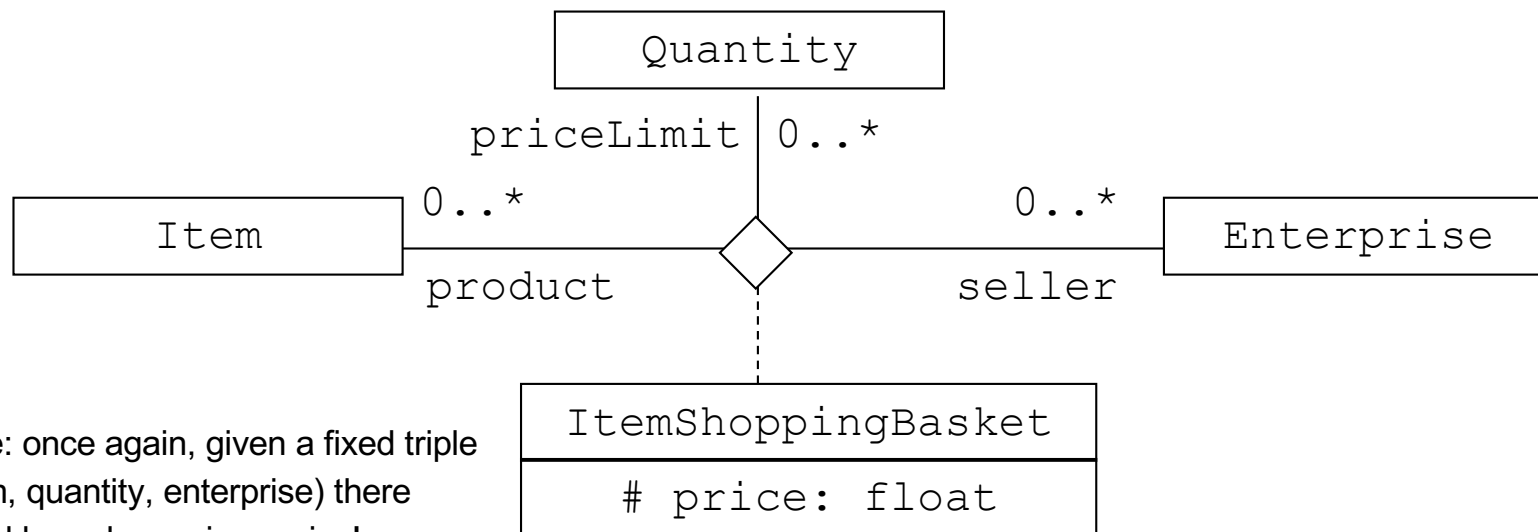
Note: it is only allowed an instance of Job between two participating objects of type Person and Enterprise!

Association – UML (6)

- By default, an association is:
 - bi-directional;
 - from one to one;
 - does not have extra information.

Association – UML (7)

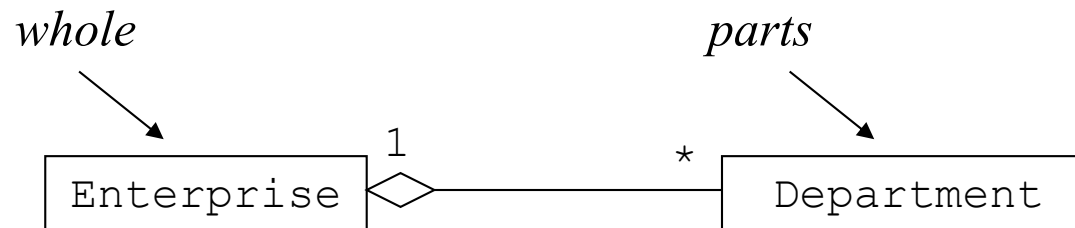
- **Ternary associations** represented by an unfilled diamond that links different solid lines of the associated classes.



Note: once again, given a fixed triple (item, quantity, enterprise) there could be only a unique price!

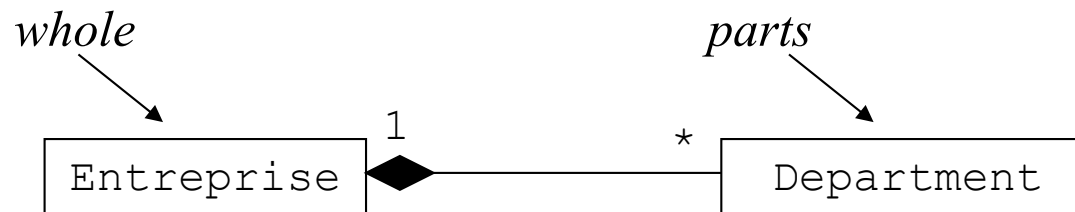
Aggregation/Composition – UML (1)

- An **aggregation** is an association which denotes that the *whole* is formed by *parts* – **does not imply ownership**.
- The aggregation is said to be as a relationship of “**has-a**”.
- Represented as an association with a solid line, but with an unfilled diamond in the endpoint related to the *whole*.



Aggregation/Composition – UML (2)

- In **composition**, when the owning object is destroyed, so are the contained objects – **no sharing**.
- Represented as an aggregation but the diamond is filled.

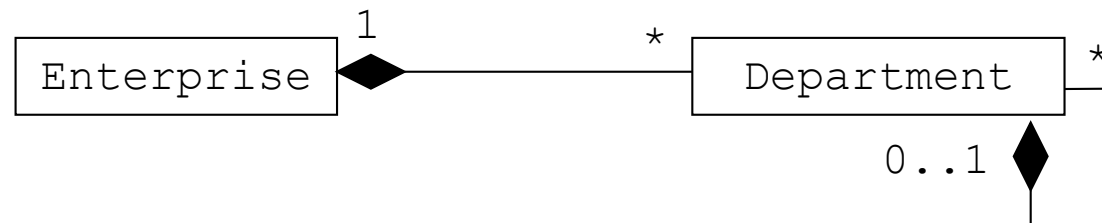


Aggregation/Composition – UML (3)

- Generally, both aggregation and composition do not have identifier, as the meaning of these relationships are implicit in the whole-part relation.
- The multiplicity should appear in both endpoints of the related classes. By default, multiplicity of exactly one (1) is considered.

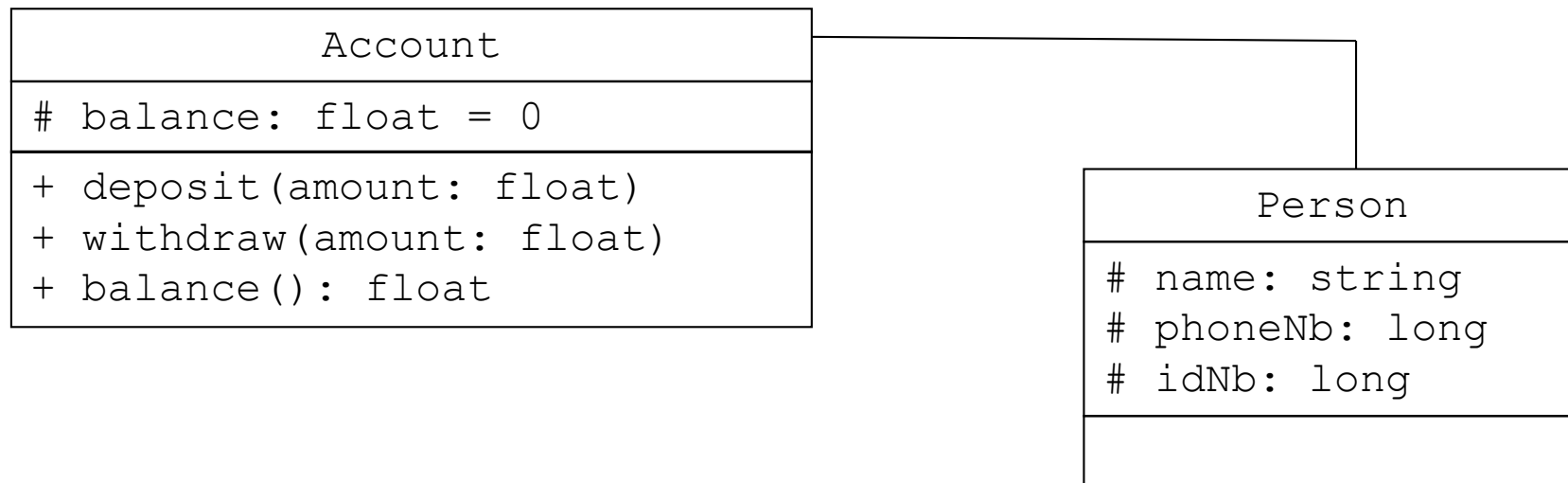
Reflexivity in relations – UML

- Relationships of association, aggregation and composition might be reflexive, with an element composed of several similar elements.



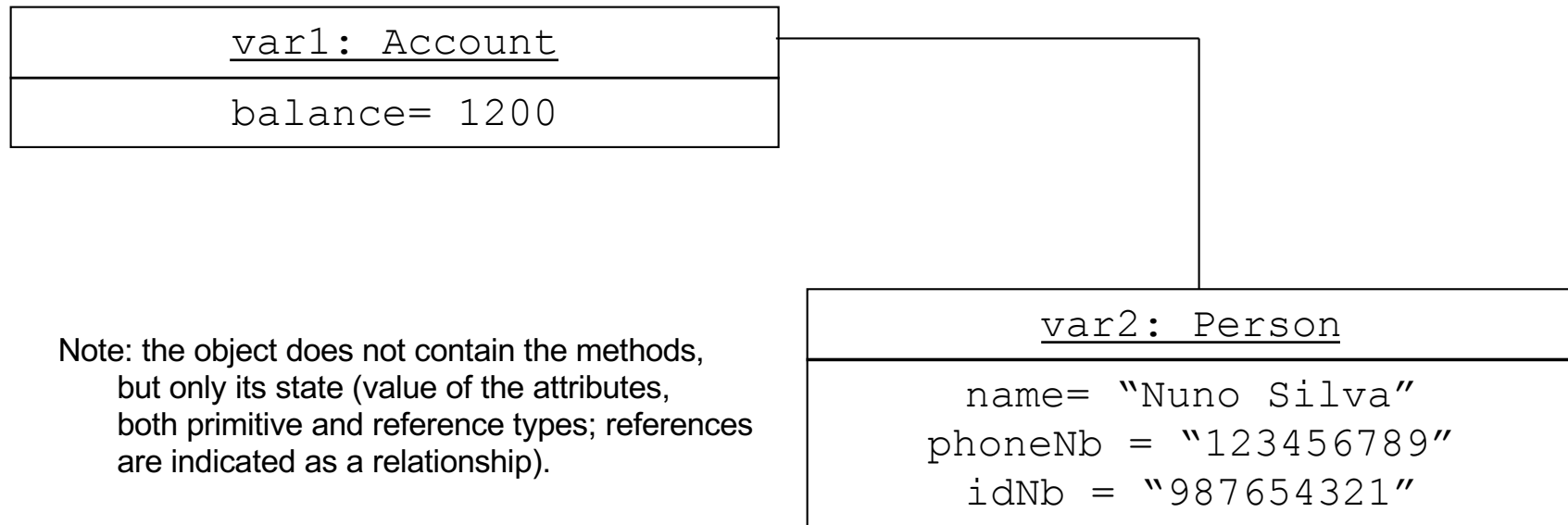
Relations – example (1)

“A bank account contains always a balance and belongs to a person. A person has a name, a phone number and an ID number. It is possible to deposit and withdraw money to and from the bank account, respectively. The owner of the account may consult its balance”.



Objects – UML

- Represented by a rectangle, divided in 2 regions:
 - idObject:IdClass (or solely :IdClass)
 - Attributes initialization, one per line, as Id=Init



Inheritance – definition (1)

- **Open-closed principle**
 - **Software entities should be open for extension, but closed for modification.**
 - When designing a class, all attributes and methods should be offered and the class should be closed to future updates.
 - Classes should be open to extension, so that new requisites are incorporated with minimal impact in the system.

Inheritance – definition (2)

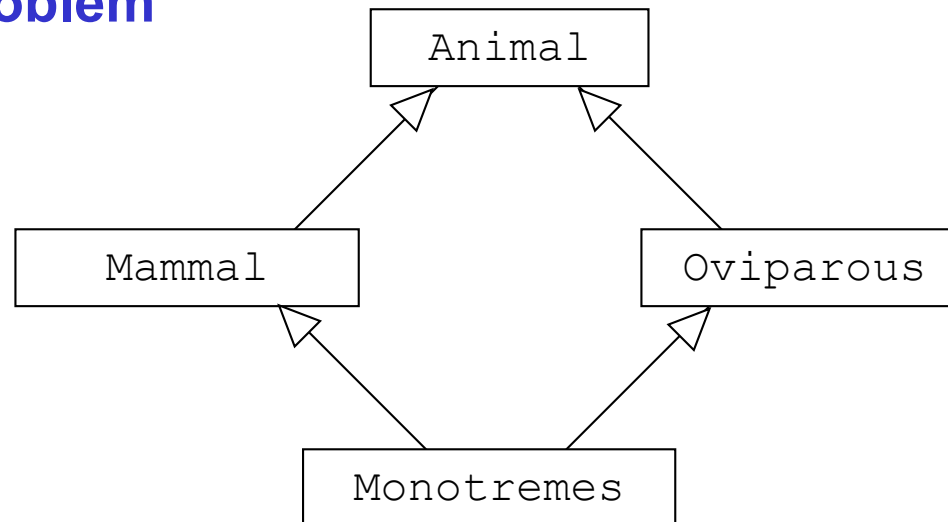
- Inheritance is a mechanism where the **subclass** constitute a specialization of a **superclass**. The superclass might be seen as a **generalization** of its subclasses.
- Inheritance is said as a relationship of “**is-a**”.
- Subclasses inherit the attributes and methods of superclasses. The inherited methods might be modified. New attributes and methods might be added to the subclasses.

Inheritance – definition (3)

- **Polymorphism** occurs when there is **redefinition** of methods (methods with the same signature) of the superclass in the subclass.
- In OO, polymorphism is usually implemented through **dynamic binding**, i.e., the method being executed is determined only in runtime (and not in compile time).

Inheritance – definition (4)

- In **simple inheritance** each subclass has only a direct superclass.
- In **multiple inheritance** a subclass might have more than one direct superclass.
 - **Diamond problem**



Inheritance – definition (5)

- Advantages:
 - More readability, as superclasses describe common aspects.
 - Easier to incorporate new requisites, as usually they coincide in particular aspects.
 - Promote reuse of the code of the superclasses.
- Disadvantages:
 - Need to detect for common aspects.

Inheritance – example (1)

“A bank account does not receive interest rates. A time deposit receives interest rates in the end of a time period, except if a withdraw transaction is made before the end of that period. In that case, the immobilization period is restarted”.

Subclasses

- TimeDeposit (superclass: Account)

Inheritance – example (2)

Added attributes

- TimeDeposit: interest_rate (type float), begin (type long), period (type int)

Updated/redefined methods

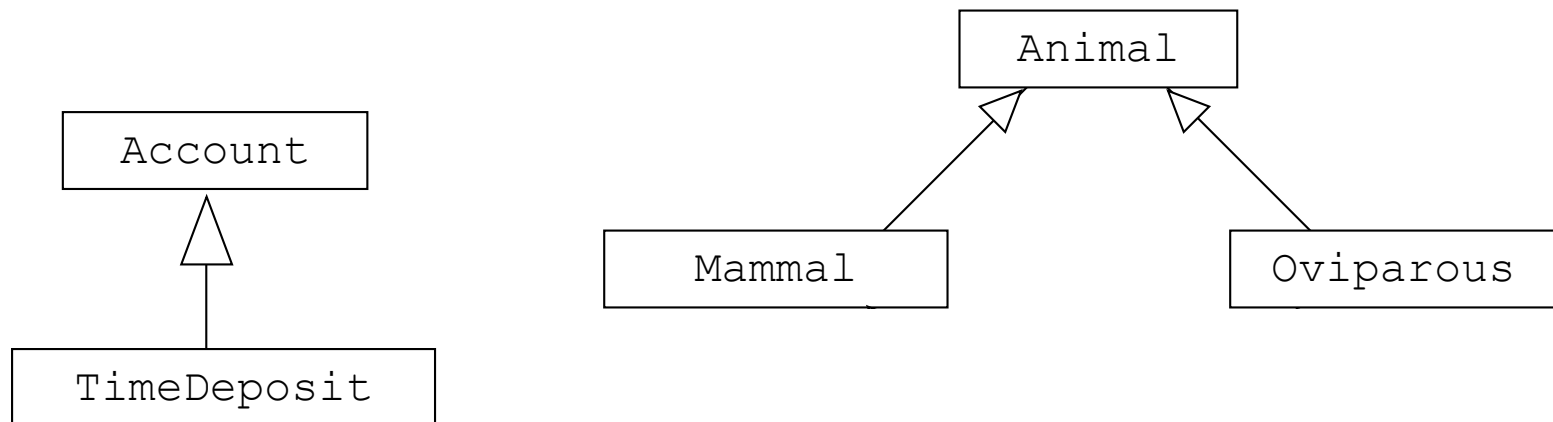
- TimeDeposit: withdraw

Added methods

- TimeDeposit: interest_rate

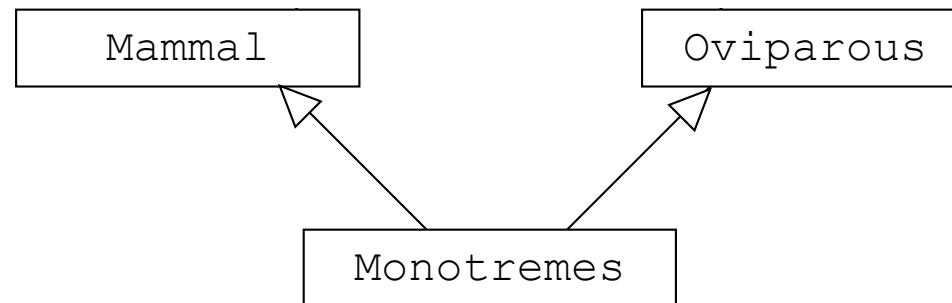
Inheritance – UML (1)

- Simple inheritance is represented with an unfilled arrowhead from subclasses towards superclass.



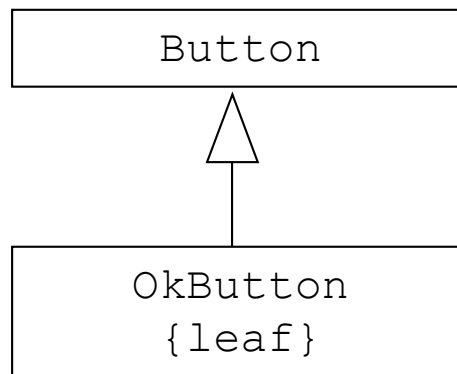
Inheritance – UML (2)

- Multiple inheritance is represented as an unfilled arrowhead from subclasses towards superclasses.



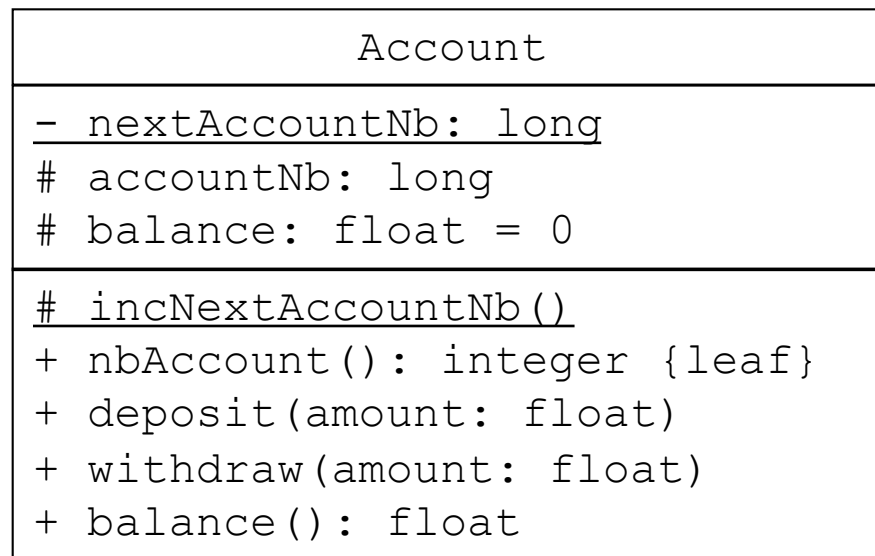
Inheritance – UML (3)

- **Not extensible classes:**
 - **Class without superclasses:** depicted with the property **{root}** written under the class identifier.
 - **Class without subclasses:** depicted with the property **{leaf}** written under the class identifier.



Inheritance – UML (4)

- UML also allows to specify that a certain method cannot be redefined in subclasses. Such methods are represented with the property **{leaf}** written after the signature of the method.



Inheritance – UML (5)

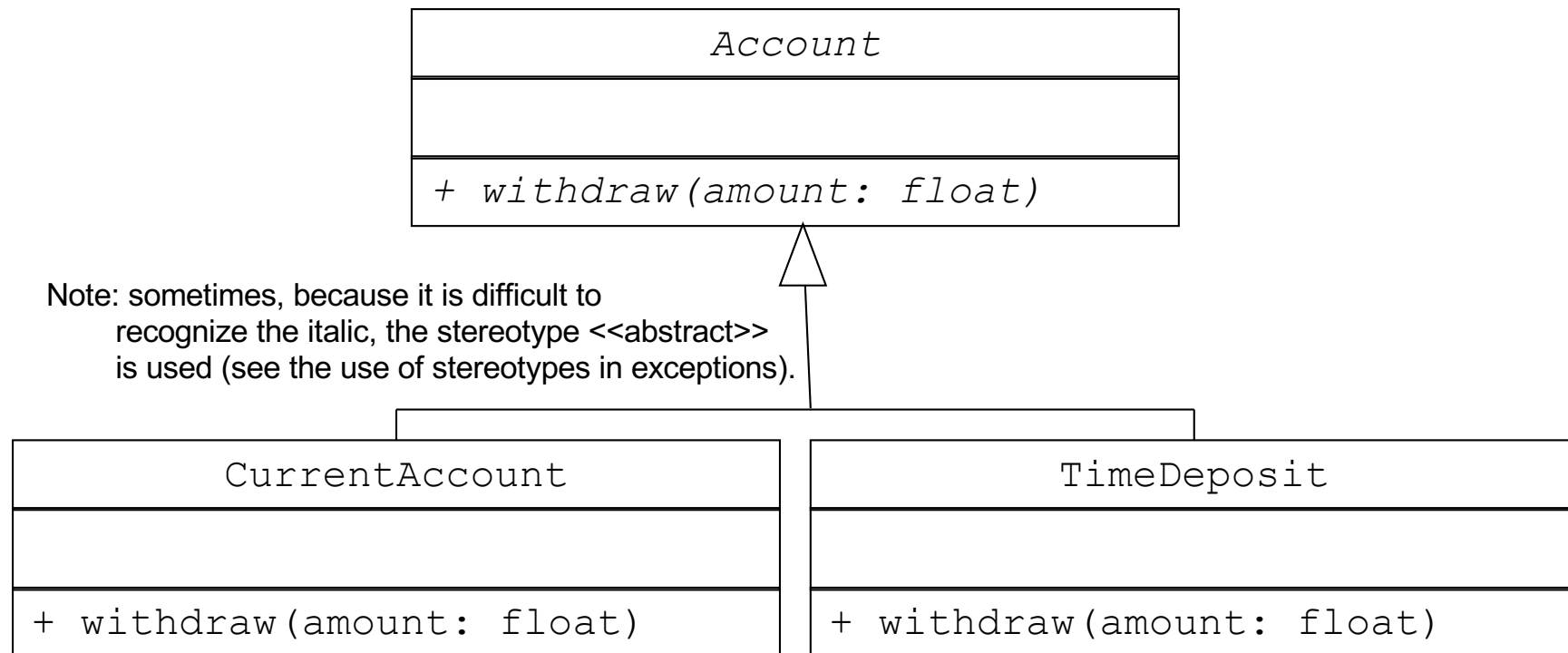
- Visibility of attributes and methods represented by a character before the identifier, and it determines the access permissions:
 - **public**: visible outside of the class (+)
 - **private**: visible only inside the class (–)
 - **protected**: visible in class and subclasses (#)
 - **package**: visible in all classes of the same package (~)

Abstract methods and classes – definition

- An **abstract method** is a method without implementation (it is only a prototype).
- An **abstract class** is a class that cannot be instantiated.
 - A class that has at least one abstract method (defined within the class or inherited from a superclass, direct or indirect), is an abstract class.

Abstract methods and classes – UML

- The abstract methods/classes are represented with their signature/identifier in italic.



Exceptions – definition (1)

- Frequently, applications are subject to many kind of errors:
 - Mathematic errors (for instance, divide by 0 arithmetic).
 - Invalid data format (for instance, integer with invalid characters).
 - Attempt to access a null reference.
 - Open an unexisting file.
 - ...

Exceptions – definition (2)

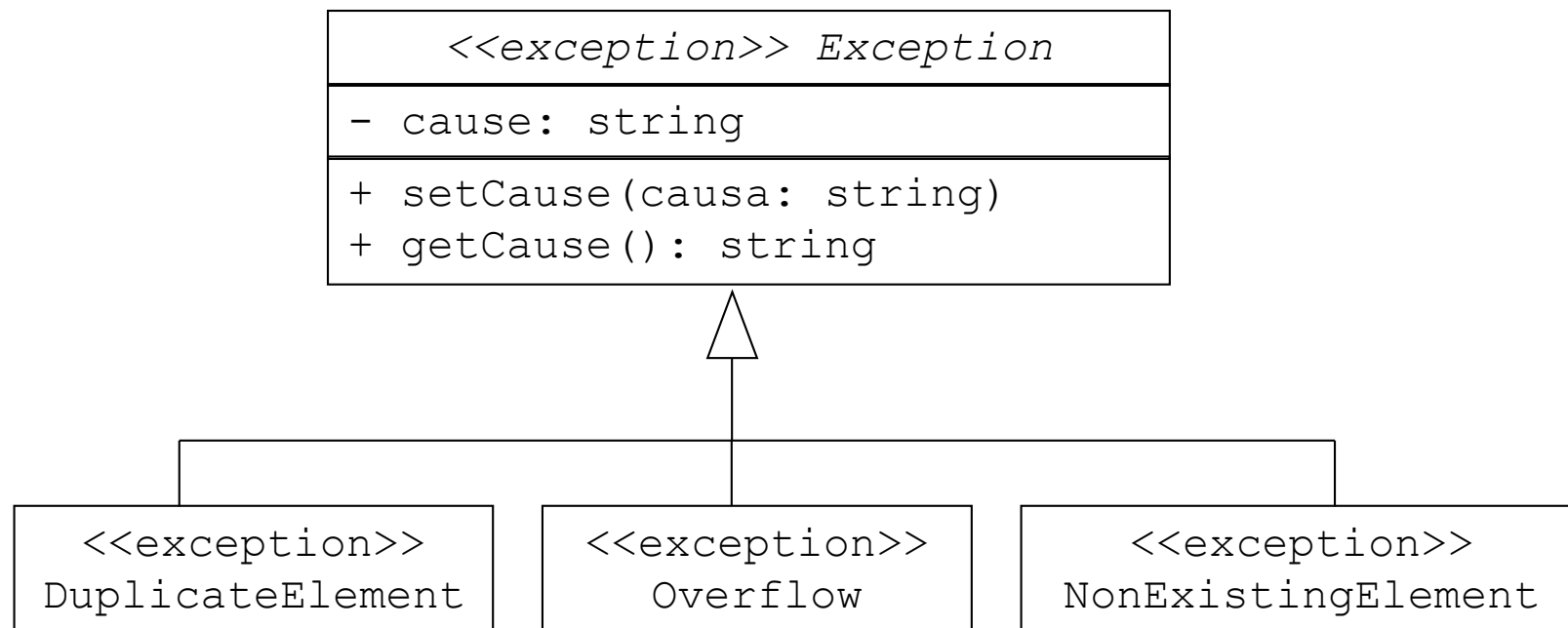
- An **exception** is a signal that is thrown when an unexpected error is encountered.
 - The signal is synchronous if it occurs directly as a consequence of a particular user instruction.
 - Otherwise, it is asynchronous.
- Exceptions can be handled in different ways:
 - Terminating abruptly execution, with warning messages and printing useful information (unacceptable in critical systems).
 - Being managed in specific places, denominated **handlers**.

Exceptions – definition (3)

- Advantages:
 - Provide a clean way to check for errors without cluttering code.
 - Provide a mechanism to signal errors directly, rather than indirectly with flags or side effects such as fields that must be checked.
 - Make the error conditions that a method can signal an explicit part of the method signature.

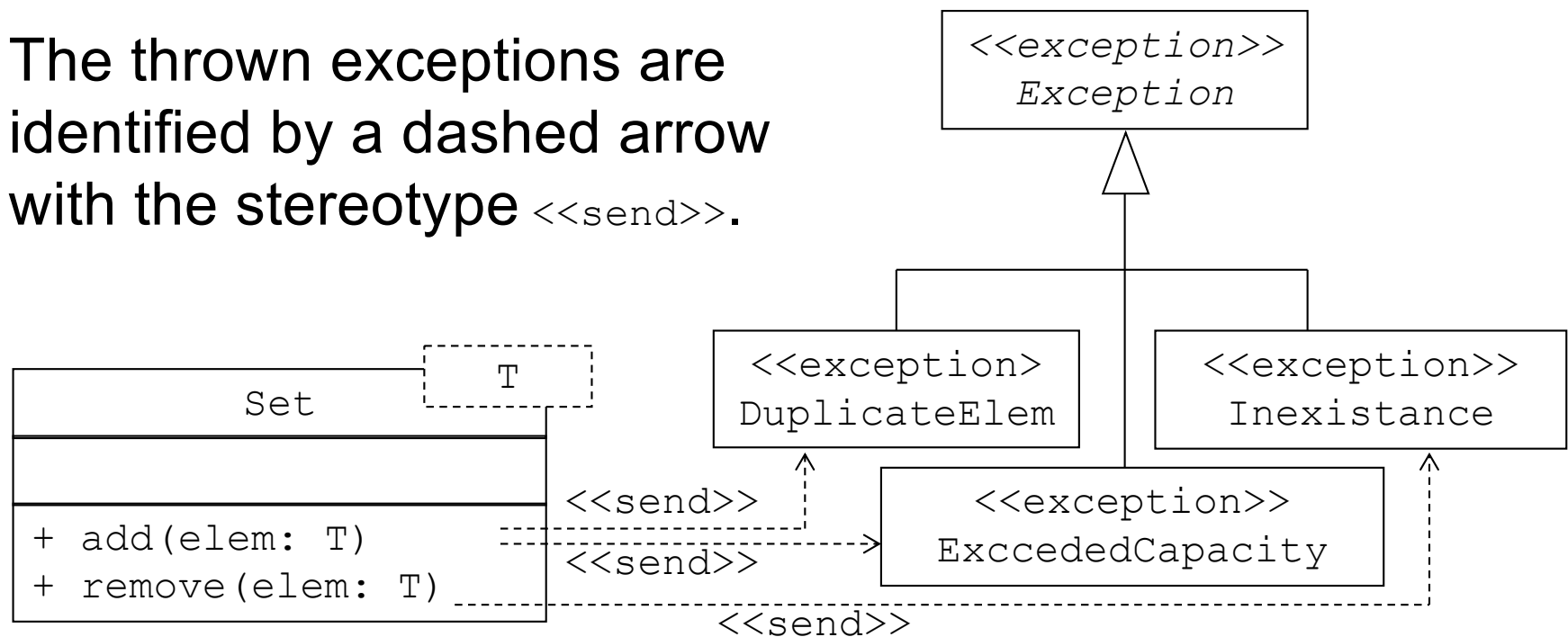
Exceptions – UML (1)

- Exceptions are represented as classes with the stereotype `<<exception>>`.



Exceptions – UML (2)

- The exception classes model the exceptions that an object can throw in the execution of their methods.
- The thrown exceptions are identified by a dashed arrow with the stereotype <<send>>.



Interfaces and Packages – definition

- Interfaces and packages are useful mechanisms to develop high dimensional systems:
 - The **interfaces** decouple the specification from the implementation.
 - The **packages** group distinct elements in a same unit.

Interfaces – definition (1)

- An **interface** is a collection of operations (without implementation) that are used to specify a service of a class:
 - Interfaces do not contain attributes, except for constants.
 - An interface may be realized (or implemented) by a **concrete class**. In this **implementation** or **realization**:
 - The attributes needed to the correct implementation of the interface are determined.
 - The code of the methods made specified by the interface is provided.

Interfaces – definition (2)

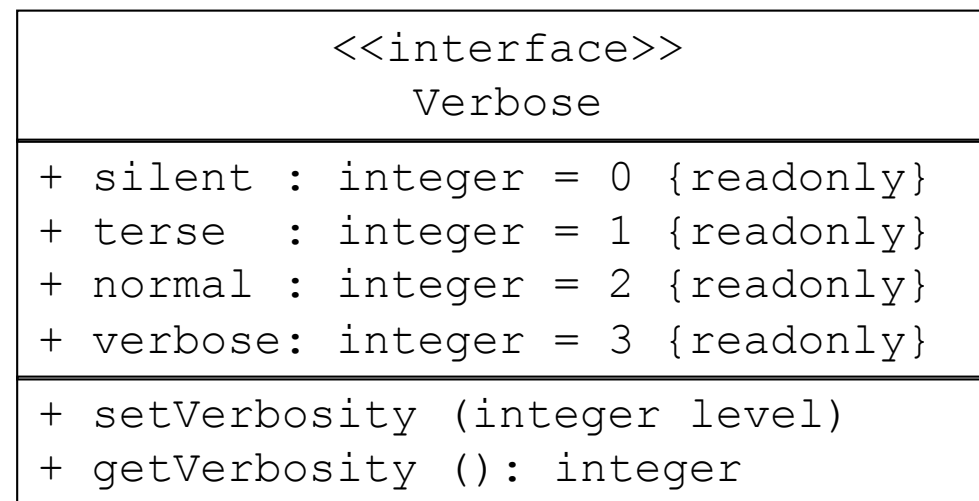
- An interface may inherit the definitions of another interface.
 - Interfaces may use polymorphism.
 - If a class realizes more than one interface, and different interfaces have methods with the same signature, the class should provide only one implementation of those methods.
- An interface cannot be instantiated.

Interface vs abstract class

- Similarities:
 - Both interfaces and abstract classes cannot be instantiated directly.
- Differences:
 - An abstract class may have attributes (constants or not), whereas an interface can only have constant attributes.
 - An abstract class may have methods with implementation.

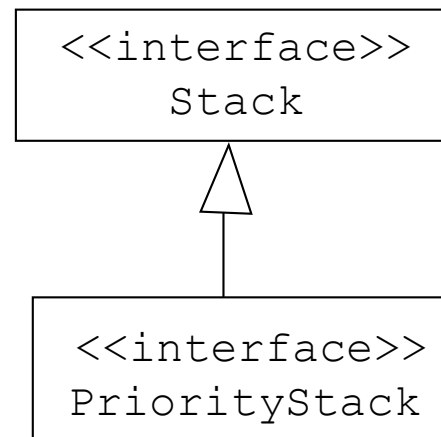
Interfaces – UML (1)

- Interfaces are represented as classes with the keyword `<<interface>>` above the name.
- An interface, besides the method prototypes, can only have **constant attributes**, represented with the property `{readonly}`.



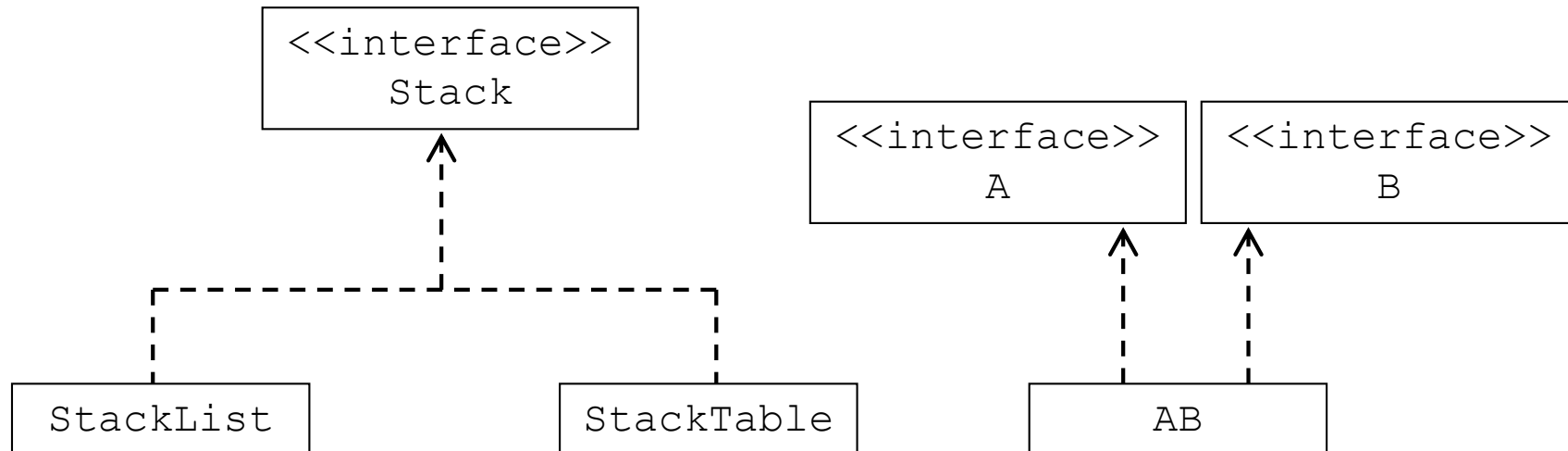
Interfaces – UML (2)

- Interfaces may participate in inheritance relationships (simple or multiple).



Interfaces – UML (3)

- An implementation class is linked to the interface by a relation of **realization**, graphically represented with a dashed arrow.
- One class may realize one or more interfaces.



Packages – definition (1)

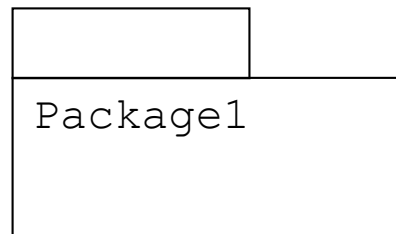
- Packages are a mechanism to group information:
 - Packages may contain other packages, classes, interfaces and objects.
 - The package defines a **namespace**, so its members need to have unique identifiers (for instance, in a package two classes with the same name cannot exist).
 - The identifier of a package may consist in a **simple name** or in a **qualified name**. The qualified name corresponds to the simple name prefixed with the name of the package where it resides, if it exists. It is common to use `::` to split the simple names.

Packages – definition (2)

- An **import** adds the content of the imported package to the namespace of the importing package, so that the members of the imported package need not to be used by their qualified name.
- Importing is not transitive.
 - If package B imports package A, and package C imports package B, in package C the members of A are not imported.
 - If package C also wants to import the members of A, two imports are needed, one to import package A and other to import package B.
- The set of methods of a package is referred as **API** (*Application Programmer Interface*).

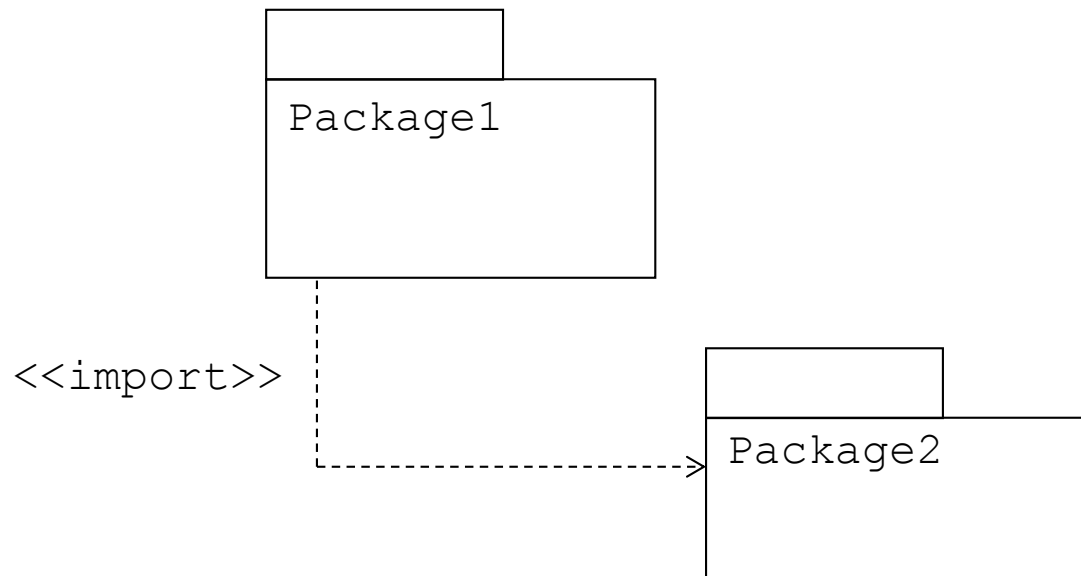
Packages – UML (1)

- Packages are represented as folders, identified by the respective name.



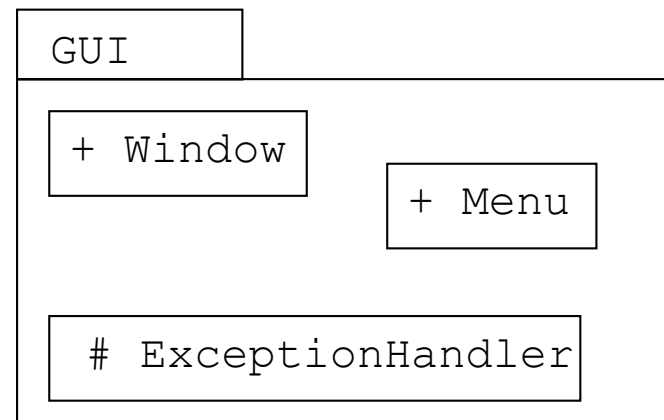
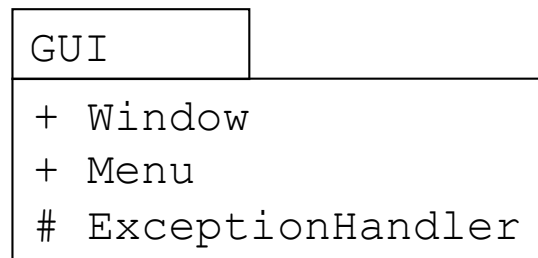
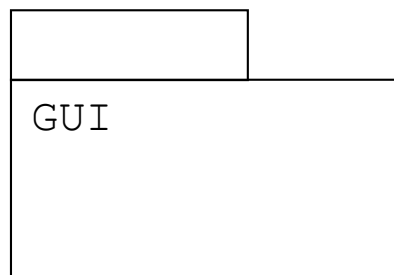
Packages – UML (2)

- An import is represented as a dashed arrow with the stereotype `<<import>>`.



Packages – UML (3)

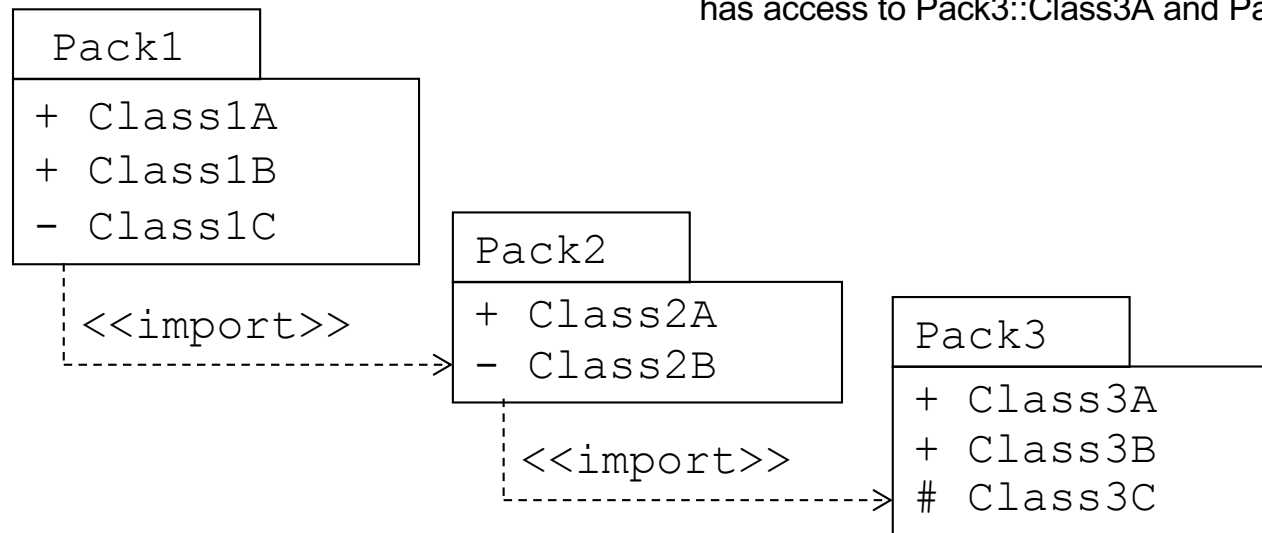
- The elements of a package may have diverse visibility:
 - **public**: elements visible to the package and to the importing packages (+)
 - **private**: elements not visible outside the package (-)
 - **protected**: elements visible in the package and subpackages (#)
 - **package**: elements visible only inside the package (~)
- The public members of a package constitute the **package interface**.



Packages – UML (4)

- A package **exports** only the public members.
- The exported members of a package are only visible by packages that explicitly import them.

Nota: Package Pack1 has access to Pack2::Class2A, package Pack2 has access to Pack3::Class3A and Pack3::Classe3B.



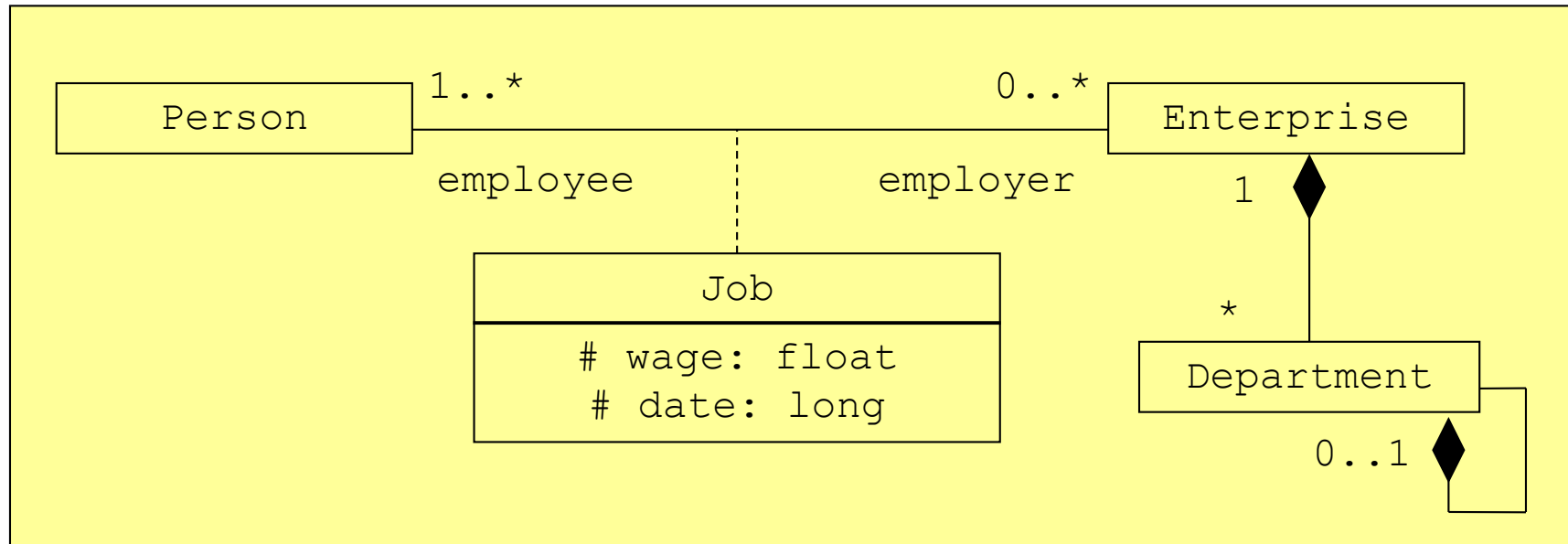
UML diagrams

- UML v2 makes available 13 diagrams, however, in OOP class only 1 is studied:
 - Structural diagrams:
 - **Class diagram**: classes, interfaces e relationships.
 - **Object diagram**: objects and relationships.
 - **Package diagram**: packages.

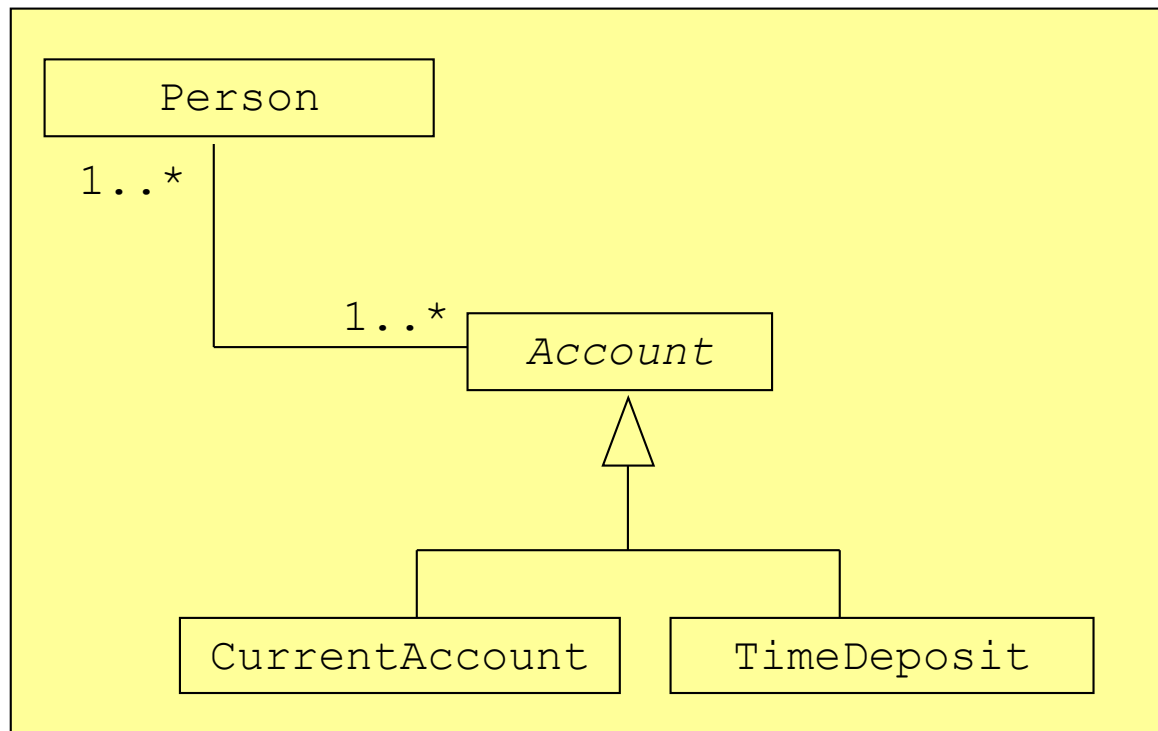
Class diagrams – definition

- **Class diagrams** are used to model:
 - Collaboration between classes.
 - Data base schemas:
 - Usually, information systems contains persistent objects.
 - The class diagram is a superset of **ER diagrams** (*entity-relationship*), a common model for logical database design. The ER diagrams focus only the data, whereas class diagrams allow, besides the data, to model behaviour.
- Typically, class diagrams contain classes, interfaces and relationships. It may also contain packages and objects.
- Class diagrams are the most common diagrams when modelling object oriented systems.

Class diagrams – UML (1)



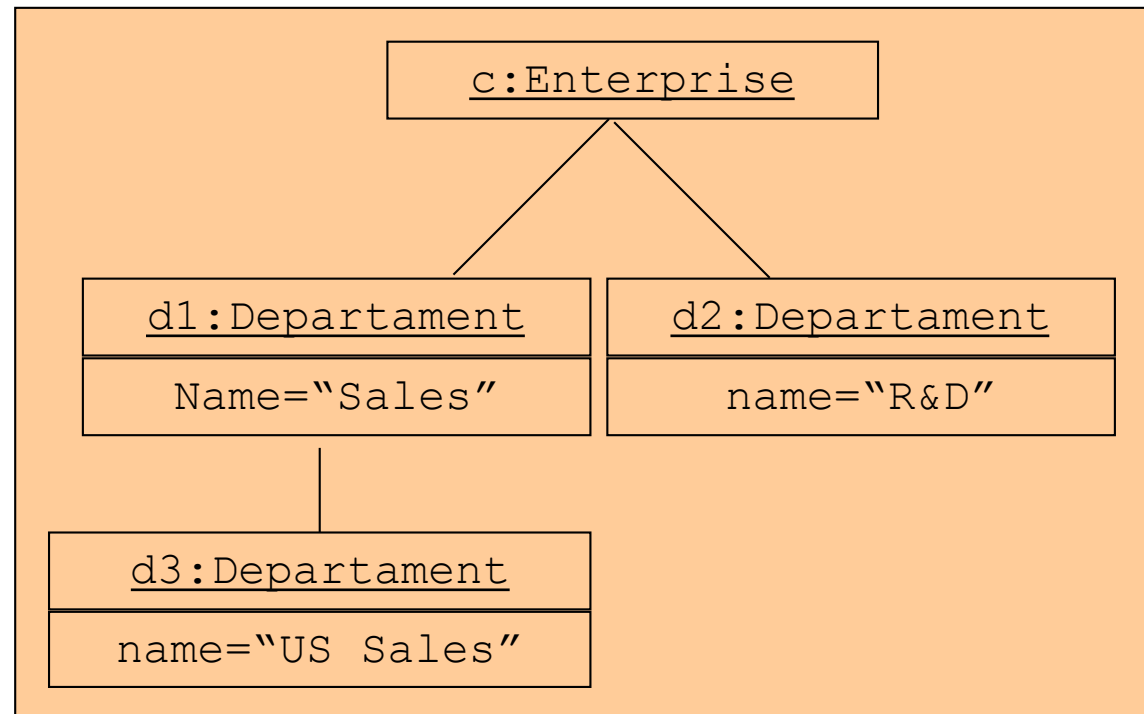
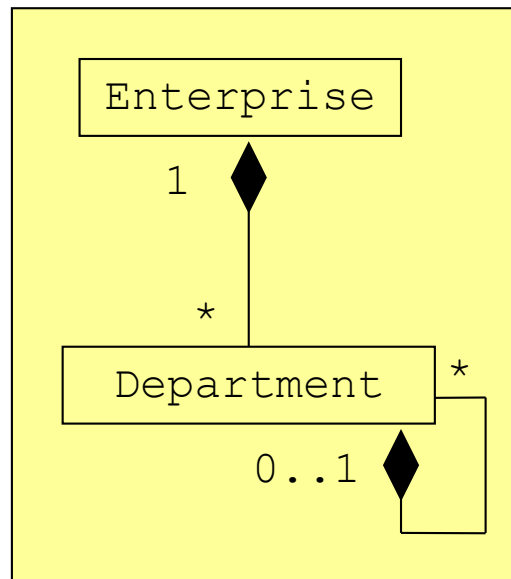
Class diagrams – UML (2)



Object diagrams – definition

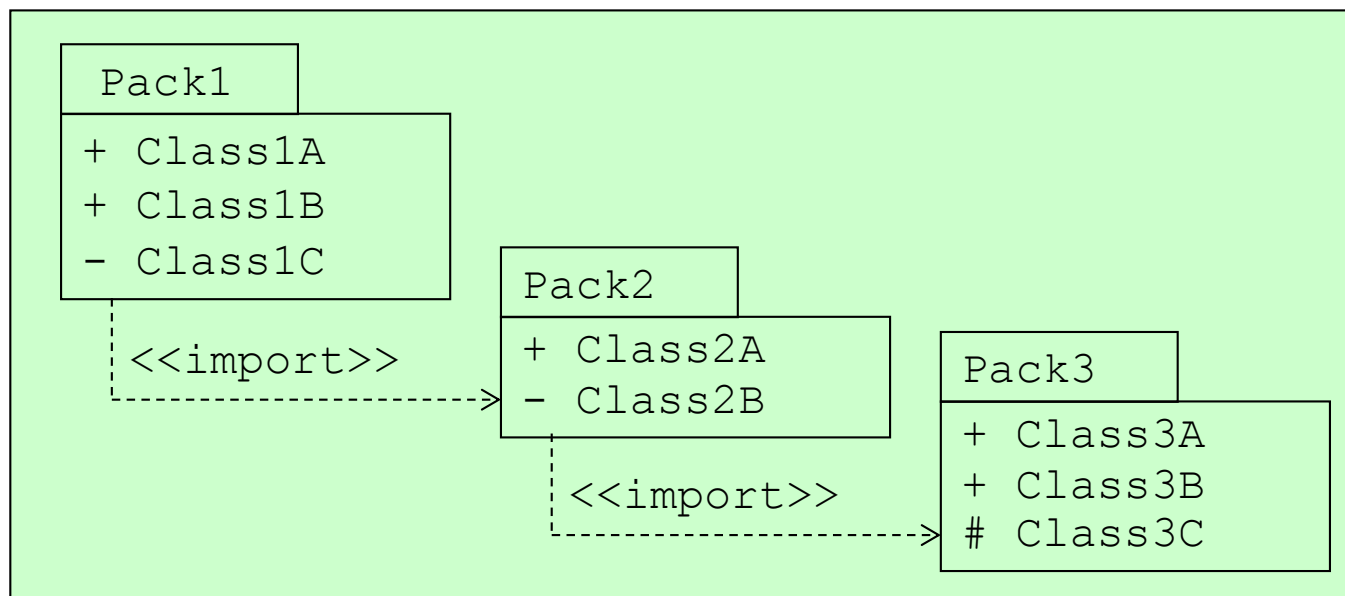
- **Object diagrams** shows a set of objects and their relationships (a snapshot at a certain time instant).
- A class diagram captures a set of abstractions that are interesting, exposing their semantics and their relationships to other abstractions.

Object diagrams – UML



Package diagrams – UML

- A package diagram shows the decomposition of the model itself into organization units (packages) and their dependencies (imports).



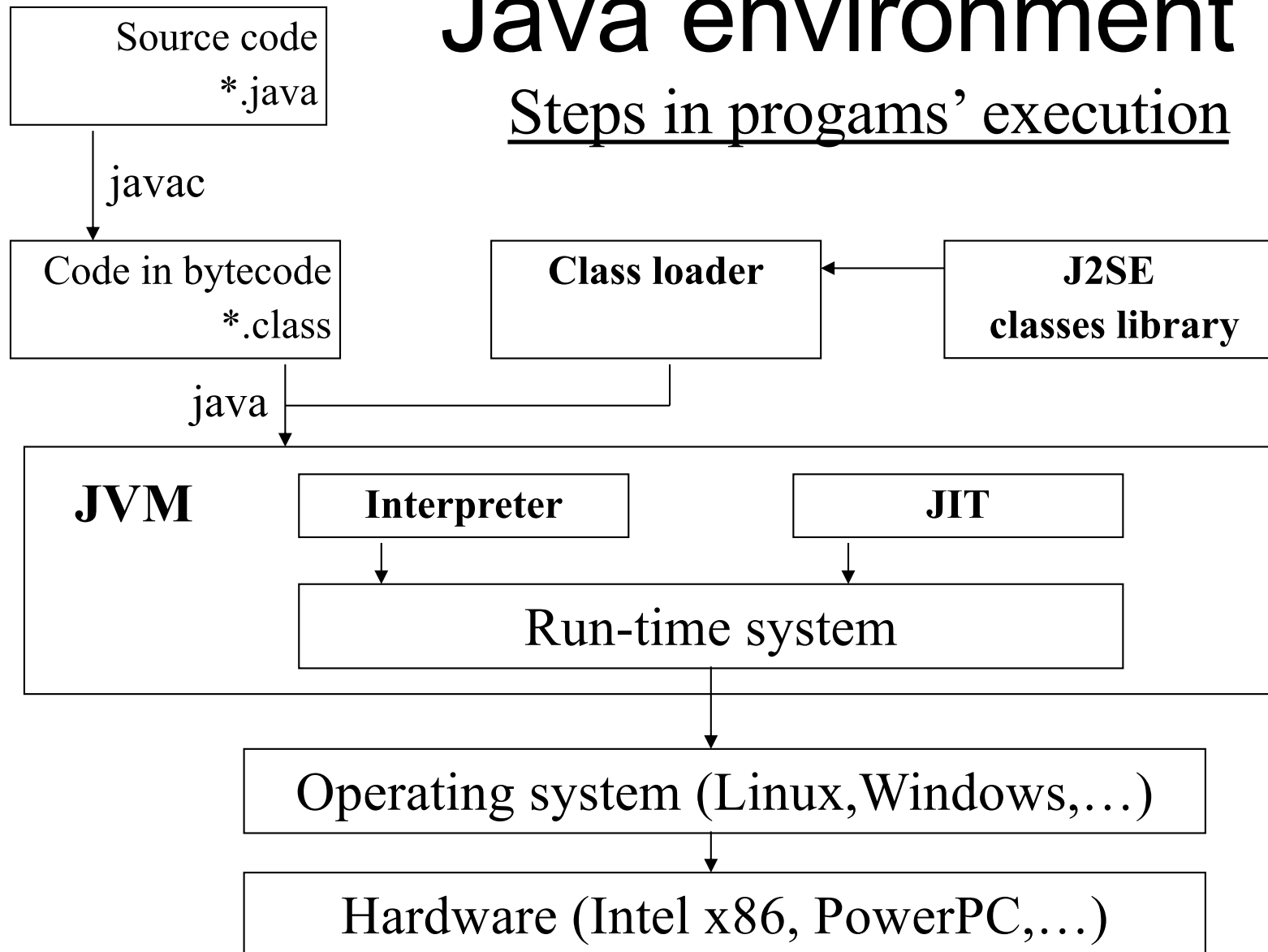
Object oriented programming

Java

Part 1: Introduction

Java environment

Steps in programs' execution



Java language

- Language particularities, comparatively to C/C++:
 - There are no pointers; instead, there are **references**.
 - There is **new**, but not **free/delete** (**garbage collector**).
 - The **parameters are passed by value** to the methods.
 - There isn't *operator overloading*.
 - There are no multiple inheritances of classes, only of interfaces.
 - There are no *preprocessor* or *header files*.
 - There are no global variables.
 - It is **strongly typed**.
 - Variables can be declared in any place inside the method, and not only in the beginning.
 - There are no **goto**, **typedef**, **union**, or **struct**.
 - There may exist more than one **main** (but only one per class).

Java: references (1)

- Java does not use pointers:
 - There are references, which indeed are **implicit pointers**.
 - **There is no pointer arithmetic**, the implicit pointers are never explicitly used as in C/C++.
 - They are treated like any other ordinary variable.
 - Every object in Java is found in the heap.

Java: references (2)

- Java primitive types (`char`, `int`, `long`, etc) are treated differently from objects:
 - **Primitive types:**

`int iVar;`

- Integer variable called `iVar`.
- The actual value of the variable is stored in a memory address called `iVar`.
- Before any assignment, it stores a default value: 0.

Java: references (3)

- **Objects:**

BankAccount baVar;

- **baVar** is a **reference to an object** of type **BankAccount**.
- The memory address called **baVar** does not store the object itself, but a reference to an object of that type; the object is stored elsewhere in memory.
- Before any assignment, it stores a reference to a special object: **null**.

Java: new operator

- Any object in Java must be created using the **new** operator:

```
BanckAccount baVar1;  
baVar1 = new BankAccount();
```

- The **new** returns a reference (not a pointer).
 - The programmer does not know the object memory address.
- It's not necessary to free memory.
 - Java verifies periodically every block of memory allocated with a **new** to check if there is still a valid reference to it (**garbage collector**).
 - Avoids **memory leaks**.

Java: assignment

- When assigning references, two references to the same object exist:

```
BankAccount baVar1, baVar2;  
baVar1 = new BankAccount();  
baVar2 = baVar1;
```

- Both variables are references to the same object.
- If on both variables a withdrawal of 1000€ is done then, in the end, the bank account in question will have less 2000€ than initially.

Java: equality/identity (1)

- **Primitive types: ==**
 - The equality operator (==) tells us whether two variables have the same value, as in C/C++.

```
int iVar1 = 27;  
int iVar2 = iVar1;  
if (iVar1==iVar2)  
    System.out.println("They're equal!");
```

Java: equality/identity (2)

- **Objects: identity** with `==`

```
BankAccount baVar1 = new BankAccount();  
BankAccount baVar2 = baVar1;  
if (baVar1==baVar2)  
    System.out.println("They're identical!");
```

- The equality operator (`==`), when applied to objects, tells us whether two references are identical. That is, whether they refer to the same object.

Java: identity/equality (3)

- **Objects: identity** with **equals**

```
BankAccount baVar1 = new BankAccount();  
BankAccount baVar2 = new BankAccount();  
if (baVar1.equals(baVar2))  
    System.out.println("They're equal!");
```

- The method **equals** is related to the identity of objects, that is, to check whether two objects have the same data/state.
- By default, the method **equals** returns the same as the operator **==**, but it should be redefined if identity between objects is needed.

Java: parameters

- In Java, parameters are always passed by value.
 - The object is never copied, only the reference is copied, referring both to the same object.

```
void method1() {  
    BankAccount baVar = new BankAccount();  
    method2(baVar);  
}  
void method2(BankAccount baArg) {}
```

- Both references **baVar** and **baArg** refer to the same object.
- In C/C++ arguments are passed by value, but the object is copied too. If this is not desired a pointer to an object must be used.
 - In C++ **baArg** would be a new object, copied from **baVar**.

Java: input/output (1)

- Output:
 - Any primitive type (numbers and chars), as well as **String** type objects, is printed in the following form:

```
System.out.print(var);  
System.out.println(var);
```

- The **print** method prints the value of **var**.
- The **println** method prints the value of **var** and terminates the current line.
- Variables/literals may also be separated by the **+** operator:

```
System.out.println("The answer is " + var);
```

- The result would be: "The answer is 15"
(is the value of **var** is 15).

Java: input/output (2)

- Input:
 - It is mandatory to import in the beginning of the java source file:

```
import java.io.*;
```

- Abnormal situations may occur, for instance, a file not found, therefore reading methods typically throw exceptions of type **IOException**.
- From the input stream an object of type **String** is read. If one needs to read any another type, for instance, a character or a number, it is necessary to convert the **String** into the desired type.

Java: input/output (3)

There is the class `Scanner`, useful for breaking down formatted input into tokens and translating individual tokens according to their data type:

```
import java.util.Scanner;
...
Scanner scan = null;
try{
    scan = new Scanner(System.in);
    System.out.print("Enter student number:");
    int nb = scan.nextInt();
    scan.nextLine(); //the \n is not read from nextInt
    System.out.print("Enter student name:");
    String name = scan.nextLine();
    System.out.println("Student\n -nb: "+nb+"\n -name: "+name);
} // catch ...
```

Java: main

- All classes in a Java application may have a **main** method.

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello, world!");  
    }  
}
```

- Each Java source file should contain only one public class, and the name of the file must be exactly the name of the class with the extension .java (in this example **HelloWorld.java**).
- The *Java virtual machine* (JVM) interpreter executes the **main method** of the class indicated in the command line.

Object Oriented Programming

Java: installation, configuration and tools

Tools – revision

- **J2SE JDK (last version)**
<http://www.oracle.com/technetwork/java/javase/downloads/index.html>
- **Eclipse, for java developers**
<http://www.eclipse.org/downloads/>
 - **IntelliJ IDEA (Eclipse alternative)**
<https://www.jetbrains.com/idea/download>
 - **NetBeans, Java SE (Eclipse alternative)**
<http://www.netbeans.org/downloads/index.html>
- **Visual paradigm**
<https://si.tecnico.ulisboa.pt/en/software/visual-paradigm/>

Java

- Download **JDK**.
- Inspect version installed in the terminal:
 - To obtain version, execute `java -version`

Java Platform

- Java technology is distributed for 3 platforms:
 - J2EE (*Enterprise Edition*), to develop enterprise applications.
 - J2ME (*Micro Edition*), to embedded devices (mobiles and PDA's).
 - **J2SE (*Standard Edition*), to desktops and servers.**

Java Platform

- Inside Java platform there is:
 - **J2xx Runtime Environment (JRE):**
 - JVM interpreter, environment classes, ...
 - Used only to run applications.
 - **J2xx Development Kit (JDK):**
 - JRE, compiler, utility classes (Swing,...), ...
 - Used for application development.

Java Platform

- Java 2 API consists of different classes organized within packages and sub-packages.
- Basic packages:
 - **java.lang**: environment classes (automatically imported)
 - **java.util**: utility classes (data types, etc)
 - **java.io**: I/O classes
 - java.net: classes for networking applications (TCP/IP)
 - java.sql: classes for database connection via JDBC
 - java.awt: native graphical user interface
 - **javax.swing**: graphical user interface (lighter than java.awt)
- HTML Documentation:
<https://docs.oracle.com/en/java/javase/18/docs/api/java.base/module-summary.html>

Java – Linux (1)

- Usually, **in Linux Java is located at:**
 - **/usr/java/jdkYOURVERSION**
 - **Compiler (javac) and JVM Interpreter (java)**
 - **/usr/java/jdkYOURVERSION/bin/**
 - **JVM Interpreter (java)**
 - **/usr/java/jreYOURVERSION/bin/**
- Pre-defined Java classes are organized in the system file following the name of the packages where they are found:
 - **/usr/java/jdkYOURVERSION/src.zip**
For instance, class `String` is defined in package `java.lang` therefore it is found in the zip file `src.zip` inside `java/lang/String.java`

Java – Linux (2)

- Configuration:
 - Update **PATH variable** if you want to run JDK executables are needed (`javac`, `java`, `javadoc`, etc) from the working directory without having to use the complete path to the executables.
 - Similarly, update **CLASSPATH variable** if the directory to other classes are needed (for instance, to a library) and you do not want to use the complete path to them.
 - Are you using the updated PATH?
 - % **which java**
 - % `java: Command not found`

Java – Linux (3)

– For C shell (csh):

- Add to the startup file (`~/ .cshrc`) the directory of the compiler and the JVM interpreter:

```
setenv Ljava /usr/java/jdk1.5.0_06  
set path=($path $Ljava/bin)
```

- Load the startup file and check java path:

```
source ~/.cshrc  
which java
```

- Directory to other classes (for instance, environment classes) are indicated in the environment variable CLASSPATH

```
setenv CLASSPATH .:$Ljava
```

Java – Linux (4)

– For ksh, bash or sh:

- Add to the startup file (`~/ .profile`) the directory of the compiler and the JVM interpreter:

```
PATH=/usr/java/jdk1.5.0_06/bin:$PATH
```

– Load the startup file and check java path:

```
. $HOME/.profile  
which java
```

- Directory to other classes (for instance, environment classes) are indicated in the environment variable `CLASSPATH`

```
CLASSPATH=/usr/java/jdk1.5.0_06:$CLASSPATH
```

Java – Windows (1)

- Usually, **in Windows Java is located at:**
 - **C:\Program Files\Java**
 - **Compiler (javac) and JVM interpreter (java)**
 - **C:\Program Files\Java\jdkVERSION\bin**
 - **Interpreter JVM (java)**
 - **C:\Program Files\Java\jreVERSION\bin**
- Pre-defined Java classes are organized in the system file following the name of the packages where they are found:
 - **C:\Program Files\Java\jdkVERSION\src.zip**
For instance, class `String` is defined in package `java.lang` therefore it is found in the zip file `src.zip` inside `java/lang/String.java`

Java – Windows (2)

- Configuration:
 - Update **PATH variable** if you want to run JDK executables are needed (`javac`, `java`, `javadoc`, etc) from the working directory without having to use the complete path to the executables.
 - Similarly, update **CLASSPATH variable** if the directory to other classes are needed (for instance, to a library) and you do not want to use the complete path to them.

Java – Windows (3)

- Include in PATH variable the directory of Java executables (typically in `User variables` or `System variables`).
- Directory to other classes (for instance, environment classes) are indicated in the environment variable `CLASSPATH` (typically in `Environment Variables` + `User variables`).

The `main` method – revision

- The JVM interpreter executes the **`main` method** of the class indicated in the command line:
 - Qualifiers: **`public static`**
 - Return: **`void`**
 - Parameters: **`String[] args`**
- All classes in the application may have a `main` method. The `main` method to execute is specified each time the program is run.

Executing Java programs (1)

- Steps to execute a Java program:

1) Create a directory

`dir`

This correspond
to the Java package!!

2) In `dir` edit

`File.java`

- The first line of this file must be `package dir;`
- `File.java` contain class `File`.
- All extra classes should be in the directories indicated in the CLASSPATH.

3) In `dir` compile

`javac File.java`

Or in the parent directory of `dir` `javac dir/File.java`

- The option `-cp` can be used to indicate needed directories that were not indicated in the CLASSPATH.
- After compiling, a `File.class` is created.

4) Go back to the parent directory of `dir`

Executing Java programs (2)

5) Execute `java dir.File`

You need to provide the full path from the root package (and sub-packages) until the class that contains the main!!

- The class `File` should contain the `main` method.
- The option `-cp` can be used to indicate needed directories that were not indicated in the CLASSPATH.
 - In Windows, for instance:
`java -cp %CLASSPATH%;C:\libs\lib.jar Fich`
 - In Linux, for instance:
`java -cp \usr\libs\lib.jar:$CLASSPATH Fich`
- The option `-verbose` can be used, that list all steps and loaded classes.

The jar tool (1)

- The **jar tool** (*Java archive tool*) manages archive files `.class`, preserving directory hierarchy.
 - The directory hierarchy should preserve the package hierarchy. For instance, the class `String` of package `java.lang`, is defined inside `/java/lang/String.java`

The jar tool (2)

- The JAR archive may contain the directory

`META-INF/`

where the following file can be found

`MANIFEST.MF`

with the information about the class to run:

- Directives (example: version, tool)
- Main class (class to run)
- White line

`Manifest-Version: 1.0`

`Created-By: 1.5.0_01 (Sun Microsystems Inc.)`

`Main-Class: project.Simulator`

↑ ↑
Sub-directory (package) Main file .class

The jar tool (3)

Command	Objective
<code>jar cf archive.jar file-list</code>	Creates a jar archive with a default manifest
<code>jar cfm archive.jar manifest-file file-list</code>	Creates a jar archive with a given manifest
<code>jar tf archive.jar</code>	List archive contents
<code>jar xf archive.jar [file-list]</code>	Extract files

The jar tool (4)

- The JVM interpreter can also run a jar archive:
`java -jar project.jar`
The class having the main method to run is indicated in the file
`META-INF/MANIFEST.MF`
- The jar program, is developed in C, and it is available in JDK.
- In Windows, the JAR archive can be opened by WinRAR.

The jar tool (5)

- **Executable:**
 - To make a jar archive file executable the **MANIFEST.MF** file should contain a line corresponding to the **Main-Class**.
 - To execute a jar archive use **option -jar**.
- **Libraries:**
 - To distribute a library one just need to make available a **jar archive with the compiled classes** (in that case the **MANIFEST.MF** file should not contain a line corresponding to the **Main-Class**).
 - To use a library one just need to compile/execute the program having the **library jar archive in the CLASSPATH**.

Object Oriented Programming

Java

Part 2: Classes and objects

Classes (1)

Syntax

Modifier* class Ident

```
[ extends IdentC] [ implements IdentI [,IdentI]* ] {  
  [ Fields | Methods ]*  
}
```

- **Modifier**: modifier (visibility, among others)
- **Ident**: class name
- **extends IdentC**: specialization of the superclass
- **implements IdentI**: implementation of interfaces

Classes (2)

- **Class modifiers:**
 - **public**: a public class is publicly accessible
 - Anyone can declare references to objects of the class or access its public members.
 - **abstract**: an abstract class is considered incomplete and no instances of the class may be created.
 - **final**: a final class cannot be subclassed.
- Without the modifier **public**, a class is only accessible within its own package.
- A class declaration can be preceded by several modifiers. However, cannot be both **abstract** and **final**.

Classes (3)

```
public class Account{  
    /* fields */  
    /* methods */  
}
```

Fields (1)

Syntax

Modifier* Type Ident [= Expr] [, Ident = Expr]* ;

- **Modifier**: modifier (visibility, among others)
- **Ident**: field name
- **Type**: field type
- **Expr**: field initialization

Fields (2)

- Field possible types:
 - Primitives:
 - `boolean`
 - `char`
 - `byte`
 - `short`
 - `int`
 - `long`
 - `float`
 - `double`
 - References: classes and interfaces defined by Java, for instance, the class `String`, and classes and interfaces defined by the programmer.

Fields (3)

- **Field modifiers:**
 - Visibility:
 - **public**: accessible anywhere the class is accessible.
 - **private**: accessible only in the class itself.
 - **protected**: accessible in subclasses of the class, in classes in the same package, and in the class itself.
 - **static**: class variable.
 - **final**: constant field, whose value cannot be changed after initialized.
 - **transient**: field which is not going to be serialized.
- If visibility is omitted, the field is accessible in the classes of the same package.
- With the exception of visibility modifiers, a field may contain more than one modifier.

Fields (4)

- **Principle of encapsulation and data hiding:**
 - Fields should not be accessible outside the object where they belong; instead, they should only be updated via methods (setters).
 - Fields' visibility should be **private** or **protected** (or package). The **public** modifier should be avoided.

Fields (5)

- **Initialization:**

- Expr might be a constant, another field, a call to a method, or an expression combining those.
- By default, a newly created object is given an initial state, where the fields are initialized with their default values depending on their types:
 - **boolean** – false
 - **char** – '\u0000'
 - **byte, short, int, long** – 0
 - **float, double** – +0.0
 - Reference to an object – null
- A field can be (explicitly) initialized:
 - Directly when it is declared in the class.
 - When the class is loaded to the JVM (in the case of **static** fields), or in the creation of the object (in the case of non-static fields).

Fields (6)

- A constant has the modifiers **static final**.

```
public static final double PI = 3.141592;
```

- A **final** field needs to be explicitly initialized. One that does not have an initializer is termed **blank final**.
 - Blank finals are used when simple initialization is not appropriate.
 - Blank finals must be initialized once the class has been loaded to the JVM (in the case of a static field) or once an object has been fully constructed (for non-static fields).
 - The compiler will ensure that this is done!

Fields (7)

```
public class Account {  
    /* fields */  
    private static long nbNextAccount = 0;  
    protected long nbAccount; // account number  
    protected String owner;   // account owner  
    protected float balance;  // actual balance  
    /* methods */  
}
```

Fields (8)

- A field in a class is accessed via the dot operator (“.”) in the form `reference.field`.
- The `reference` is an identifier of:
 - an object, for a non-static field.
 - a class, for a **static** field.

```
System.out.println(Account.nbNextAccount);
```

Objects

Syntax

Ident = new IdentClass ([Expr [, Expr]*]);

- **Ident**: reference to the new object
- **IdentClass**: type of the object to create
- **Expr**: constructor parameters

Garbage collector

- In Java, an object is created with the **new** operator, but the object is never deleted explicitly.
- The **garbage collector** manages memory and objects that cannot be used any longer have their space automatically reclaimed without programmer intervention.
- If the programmer no longer needs an object it should cease referring to it:
 1. With local variables in methods, this can be as simple as returning from the method.
 2. More durable variables, such as object fields, must be set to null.

Constructors (1)

- A **constructor** is a block of statements that are executed to initialize an object before the reference to it is returned by `new`.
 - They have the same name as the class.
 - Like methods, they take zero or more arguments.
 - Unlike methods, they have no return type, not even `void`.
 - They are commonly used to initialize the values of the fields, when more than simple initialization is needed.
- A class may contain more than one constructor.
 - The type and the number of arguments being passed to the constructor determine the constructor to be used.

Constructors (2)

```
public class Account {
    /* Fields */
    private static long nbNextAccount = 0;
    protected long nbAccount; // account number
    protected String owner;   // account owner
    protected float balance;  // actual balance
    /* Constructors */
    Account(String s) {
        nbAccount = nbNextAccount++;
        owner = s;
        balance = 100; //minimum amount to open an
account
    }
    Account(String s, float q) {
        nbAccount = nbNextAccount++;
        owner = s;
        balance = q;
    }
    /* Methods */
}
```


Constructors (3)

- If no constructor is provided (and only in this case), Java provides a **default no-arg constructor** (no-arg=no arguments).
- A **copy constructor** takes an argument of the current object type and constructs the new object to be a copy of the passed in object.
 - Usually, this is simple a matter of assigning the same values to all fields, but sometimes the semantics of the class dictate more sophisticated actions.

```
/* copy constructor */
Account(Account c) {
    nbAccount = c.nbAccount;
    owner= c.owner;
    balance = c.balance;
}
```

Constructors (4)

- One constructor can invoke another constructor from the same class by using the `this()`. This is called **explicit constructor invocation**.
- If the constructor has `N` parameters, these should be passed to the explicit invocation as `this(param1, ..., paramN)`.
- The argument list determines which version of the constructor is invoked.
- If provided, the explicit invocation must be the first statement in the constructor.
- Any expressions that are used as arguments for the explicit constructor invocation must not refer to any fields or methods of the current object.

Constructors (5)

```
public class Account {  
    /* Fields */  
    private static long nbNextAccount = 0;  
    protected long nbAccount; // account number  
    protected String owner;   // account owner  
    protected float balance;  // actual balance  
    /* Constructors */  
    Account(String s) {  
        this(s,100); //explicit method invocation  
    }  
    Account(String s, float q) {  
        nbAccount = nbNextAccount++;  
        owner = s;  
        balance = q;  
    }  
    /* Methods */  
}
```

Initialization of non-static fields (1)

- A newly created object is given an initial state:
 - Initialization by default.
 - Initialization when they are declared.
 - When more than a simple initialization is required:
 - **Constructors**: used to initialize an object before the reference to the object is returned by `new`.
 - **Initialization blocks**: executed as if they were placed at the beginning of every constructor in the class.
 - It provides guarantee of correction with blank final fields.

Initialization of non-static fields (2)

- The constructor is invoked after:
 - Initialization, by default, of the non-static fields.
 - Initialization of the non-static fields where they are declared.

Initialization of non-static fields (3)

```
public class Account {  
    /* Fields */  
    private static long nbNextAccount = 0;  
    protected long nbAccount; // account number  
    protected String owner;   // account owner  
    protected float balance;  // actual balance  
    /* Constructors */  
    Account(String s) {  
        this(s,100); //explicit method invocation  
    }  
    Account(String s, float q) {  
        nbAccount = nbNextAccount++;  
        owner = s;  
        balance = q;  
    }  
    /* Methods */  
}
```

Initialization of non-static fields (4)

```
public class Account {  
    /* Fields */  
    private static long nbNextAccount = 0;  
    protected long nbAccount; // account number  
    /* Initialization block */  
    {  
        nbAccount = nbNextAccount++;  
    }  
    protected String owner; // account owner  
    protected float balance; // actual balance  
    /* Constructors */  
    Account(String s) {  
        this(s,100); //explicit method invocation  
    }  
    Account(String s, float q) {  
        owner = s;  
        balance = q;  
    }  
    /* Methods */  
}
```

Initialization of static fields

- The static fields can be initialized:
 - when they are declared.
 - in **static initialization blocks**.
 - Declared as **static**.
 - It can only refer to static members of the class.
- The static initializers are executed after the class is loaded to the JVM, but before it is actually used.

Object Oriented Programming

Java

Part 3: Methods

Methods (1)

Syntax

```
Modifier* Type Id ( [ TypeP IdP [, TypeP IdP]* ] ) {  
    [ Local_variable | Statement ]*  
}
```

- **Modifier**: modifier (visibility, among others)
- **Type**: return type of the method
- **Id**: method name
- **TypeP**: type of the parameters of the method
- **IdP**: name of the parameters of the method
- **{ [Local_variable | Statement]* }**: body of the method

Methods (2)

- **Modifiers:**
 - Visibility:
 - **public**: accessible anywhere the class is accessible.
 - **private**: accessible only in the class itself.
 - **protected**: accessible in subclasses of the class, in classes in the same package and in the class itself.
 - **abstract**: method without body.
 - **static**: class method, invoked on behalf of the entire class.
 - **final**: cannot be overridden in subclasses.

Methods (3)

- If the visibility is omitted, the method is accessible in the class and in classes in the same package only.
- With the exception of visibility modifiers, a method may have more than one modifier. However, it cannot be at the same time **abstract** and **final**.
- A static method can only access static fields and static methods.

Methods (4)

- The return type of a method is mandatory and it can be:
 - primitive type (boolean, char, byte, short, int, long, float and double)
 - references (classes and interfaced defined by Java, for instance, class `String`, and classes and interfaced defined by the programmer)
 - `void`
- The result is returned to the caller by the `return` statement.

Methods (5)

- A method might have zero, one, or more parameters:
 - Possible types for the parameters:
 - **primitive types** (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float` and `double`)
 - **references** (classes and interfaces defined by Java, for instance, class `String`, and classes and interfaces defined by the programmer)

Methods (6)

- In Java, all parameter methods are **passed by value**:
 - The values of the parameters are always copies of the values passed by argument.
 - The method can change its parameters' value without affecting values in the code that invoked the method.
 - When the parameter is an object reference, it is the object reference, not the object itself, that is passed by value.
 - One can change which object a parameter refers to inside the method without affecting the reference that was passed.
 - But if one change any fields of the object or invoke methods that change the object's state, the object is changed for every part of the program that holds a reference to it.
 - A parameter can be defined as **final**, meaning that the value of the parameter will not change while the method is executing.
 - When the parameter is a reference, the final only applies to the reference, and not the object itself.

Methods (9)

```
public class Demo {  
    static void foo(int arg1) {  
        arg1 = 10;  
    }  
    public static void main(String args[]){  
        int arg = 5;  
        foo(arg);  
        System.out.println("arg = " + arg);  
    }  
}
```

In the terminal it is printed `arg = 5`

Methods (10)

```
public class Xpto {  
    public int var = 1;  
}
```

```
public class Demo {  
    public static void foo(Xpto x) {  
        x.var += 10;  
    }  
    public static void main(String args[]) {  
        Xpto arg = new Xpto();  
        foo(arg);  
        System.out.println("arg.var = " + arg.var);  
    }  
}
```

In the terminal it is printed `arg.var = 11`

Methods (11)

```
public static void printAnimals(final int i, final String[] s) {  
    for (; i<s.length; i++)  
        System.out.println(s[i]);  
}
```

**final parameter i may not be assigned
for (; i<strings.length; i++)
1 error**

```
public static void printAnimals(final int i, final String[] s) {  
    s[i]="Snake";  
    for (int j=i; j<s.length; j++)  
        System.out.println(s[j]);  
}
```

```
printAnimals(1,new String[]{"Lion","Tiger","Bear"});
```

prints in the terminal

Snake
Lion

Methods (12)

- A method is invoked with the dot operator (“.”) via references in the form `reference.method(params)`.
- The `reference` is an identifier of:
 - an object, in a non-static method.
 - a class, in a **static** method.

Methods (13)

- Sequential invocation:
 - A method may return an object, from which another method can be invoked. There are two ways to perform sequential invocation:

1. Store the object in a variable to invoke the method from it.

```
Classe var = obj.method1();  
var.method2();
```

2. Invoke it directly.

```
obj.method1().method2();
```

Methods (14)

- Inside a non-static method, the object on which the method was invoked is referenced by `this`.
- There is no `this` reference inside static methods.
- Usually, the `this` reference is used to pass the object on which the method was called as argument to other methods.

Methods (15)

```
public void deposit(float value){  
    balance += value;  
}
```

```
public void deposit(float value){  
    this.balance += value;  
}
```

- The `this` reference is needed in the presence of a **field hidden** by a local variable or a parameter.

```
public void deposit(float balance){  
    this.balance += balance;  
}
```

Methods (16)

- The signature of a method is given by its name and number and type of its parameters.
- Two methods can have the same name if they have different number or type of its parameters (and thus different signatures).
- This feature is called **overloading**.
- The compiler uses the number and type of the arguments to find the best match from the available overloads.
- Overloading is typically used when a method (or constructor) can accept the same information presented in different forms, or when it can use some parameters with default values (and so they are not needed to be supplied).

The `main` method (1)

- The JVM interpreter executed always the **`main method`** of the class indicated in the command line:
 - Modifiers: **`public static`**
 - Return: **`void`**
 - Parameters: **`String[] args`**
- An application can have any number of `main` methods (because each class in an application can have one). The `main` method to execute is specified each time the program is run.

The main method (2)

```
Class Echo {  
    public static void main(String[] args) {  
        for(int i=0; i<args.length; i++)  
            System.out.print(args[i]+ " ");  
        System.out.println();  
    }  
}
```

```
> javac Echo.java  
> java Echo I am here  
> I am here
```

The `main` method (3)

- The program terminates when it is executed the last instruction in the `main`.
- If one needs to anticipate the termination, the following method should be invoked:

`System.exit(int status);`

where the parameter `status` identifies the terminating code (success - 0 in Linux, 1 in Windows).

Local variables

- Declaring a local variable is similar to declaring a field, but **final** is the only modifier applicable.

Modifier* Type Id [= Expr] [, Id = Expr]* ;

- Local variables need to be initialized before being used, in their declaration or before they are used.
 - Unlike fields, there are no default initialization of local variables.
- Like fields, when the initialization of a local variable declared as **final** is not made in its declaration, this local variable is termed **blank final**.

Statements (1)

- A **block** groups zero or more statements.
- A block is delimited by the parenthesis { and }.
- A local variable exists only as long as the block containing its declaration is executing.
- A local variable can be declared in any point inside a block, but always before being used.

Statements (2)

Assignment

Var = Expr [, Var = Expr]*;

- **Var**: local variable
- **Expr**: expression
- Recall that:
 - The assignment `Var = Var op Expr` is equivalent to `Var op= Expr`.

Statements (3)

- The assignment of references provides two distinct references to the same object.

```
Account c1 = new Account(), c2;  
c1.deposit (1000);  
c2 = c1; // c2 and c1 are references to the same object  
System.out.println("Balance of c1 = " + c1.balance());  
System.out.println("Balance of c2 = " + c2.balance());  
c1.deposit(100);  
c2.deposit(200);  
System.out.println("Balance of c1 = " + c1.balance());  
System.out.println("Balance of c2 = " + c2.balance());
```

In the terminal is printed

```
Balance of c1 = 1000  
balance of c2 = 1000  
Balance of c1 = 1300  
Balance of c2 = 1300
```

Statements (4)

Conditional execution

if (Expr-Bool) statement1 [else statement2]

- **Expr_Bool**: Boolean expression

Statements (5)

```
char c;  
/* identifies char category */  
if (c>='0' && c<='9')  
    System.out.println("Digit!");  
else if ((c>='a' && c<='z') || (c>='A' && c<='Z'))  
    System.out.println("Character!");  
else  
    System.out.println("Other!");
```


Statements (6)

Value selection

```
switch (Expr) {  
    case literal: statement1  
    case literal: statement2  
    ...  
    default: statementN  
}
```

- Value of the expression **Expr** (char, byte, short or int, or corresponding wrapper class, or enum) is compared with **literals**.
- The **break** statement is use to terminate the processing of a particular case within the switch statement.

Statements (7)

```
int i = 3;
switch (i) {
    case 3: System.out.print("3, ");
    case 2: System.out.print("2, ");
    case 1: System.out.print("1, ");
    case 0: System.out.println("Boom!");
    default: System.out.println("A number, please!");
}
```

In the terminal is printed

3, 2, 1, Boom!
A number, please!

Statements (8)

```
int i = 3;
switch (i) {
    case 3: System.out.print("3, ");
    case 2: System.out.print("2, ");
    case 1: System.out.print("1, "); break;
    case 0: System.out.println("Boom!");
    default: System.out.println("A number, please!");
}
```

In the terminal is printed 3, 2, 1,

Statements (9)

```
int i = 4;
switch (i) {
    case 3: System.out.print("3, ");
    case 2: System.out.print("2, ");
    case 1: System.out.print("1, ");
    case 0: System.out.println("Boom!");
    default: System.out.println("A number, please!");
}
```

In the terminal is printed

A number, please!

Statements (10)

Conditional cycle

while (Expr-Bool) body-statements

- The **body-statements** are executed while the **Expr-Bool** is evaluated to `true`.

do body-statements while (Expr-Bool)

- The test may be executed only after **body-statements** are executed.
- In both conditional cycles:
 - Inside the **body-statements** the program may transfer control to the evaluation of **Expr-Bool** with **continue**.
 - A **break** may be used to exit from the cycle.

Statements (11)

Iterative cycle

**for (initialization; Expr-Bool; update)
body-statements**

- The for loop is used to iterate a variable over a range of values until some logical end to that range is reached.
- The **initialization** is executed before entering the cycle.
- If the Boolean expression **Expr-Bool** is evaluated to `true` the **body-statements** are executed.
- After executing **body-statements** the **update** is executed and then the loop-expression **Expr-Bool** is reevaluated.
- The cycle repeated until **Expr-Bool** is found to be `false`.

Statements (12)

- Loop variables may be declared directly in the **initialization**.
- The **initialization** and **update** parts of the loop may be comma-separated list of expressions.
- All the expressions in the for construct are optional.
 - By default, the **Expr-Bool** expression is evaluated `true`.
 - The statements **`for(;;)`** and **`while(true)`** are equivalent.
- Inside the **body-statements** the program may transfer control to the evaluation the **update** expression followed by the evaluation of **Expr-Bool** with **`continue`**.
- A **`break`** may be used to exit from the cycle.

Statements (13)

```
/* Print even numbers until 20 */  
for(int i=0, j=0; i+j<=20; i++, j++)  
    System.out.print(i+j + " ");  
System.out.println();
```

In the terminal is printed 0 2 4 6 8 10 12 14 16 18 20

Statements (14)

Iterative cycle (variant): *for-each loop*

for (Type loop-var: set-expr) body-statements

- The **set-expr** must evaluate to an object that defines a set of values (an array or an object that implements the `java.lang.Iterable` interface, as collections provided by Java) over which we intend to iterate through.
- Each time though the loop **loop-var** takes on the next value from the set, and **body-statements** are executed.
- This continues until no more values remain in the set.

Statements (15)

```
static double mean(int[] values) {  
    double sum= 0.0;  
    for (int val : values)  
        sum+= val;  
    return sum/ values;  
}
```

```
int[] values={20,19,18,17,16,15,14,13,12};  
System.out.println("The mean is" + mean(values));
```

In the terminal is printed

The mean is 16.0

Statements (16)

- The `break` statement may be used to exit from any block.
- There are two forms of the `break` statement:
 - **Unlabeled:** `break;`
 - **Labeled:** `break label;`
- An unlabeled `break` terminates the innermost `switch`, `for`, `while` or `do` statements, and so it can only be used in those contexts.
- A labeled `break` can terminate any labeled statement.

Statements (17)

```
public boolean updateValue(float[][] matrix, float val) {
    int i, j;
    boolean found = false;
findval:
    for (i=0; i<matrix.length; i++) {
        for (j=0; j<matrix[i].length; j++) {
            if (matrix[i][j]==val) {
                found = true;
                break findval;
            }
        }
    }
    if (!found)
        return false;
    //update matrix[i][j] as wished
    return true;
}
```

Statements (18)

```
public boolean updateValue(float[][] matrix, float val) {
    int i, j;
    findval:
    {
        for (i=0; i<matrix.length; i++) {
            for (j=0; j<matrix[i].length; j++) {
                if (matrix[i][j]==val)
                    break findval;
            }
        }
        //if we reach this then we have not found val
        return false;
    }
    //update matrix[i][j] as wished
    return true;
}
```

Statements (19)

- The **continue** statement can be used only within a loop (`for`, `while` or `do`) and transfers control to the end of the loop's body to continue on with the loop.
- This causes the following expression to be the next thing evaluated:
 - The **Expr-Bool**, in the case of `while` and `do`.
 - The **update** followed by **Expr-Bool**, in the case of a basic `for`.
 - The next value in the set of elements, if there is one, in the case of an enhanced for-each loop.

Statements (20)

- The `continue` statement also has two forms:
 - **Unlabeled: `continue`;**
 - **Labeled: `continue label`;**
- In the unlabeled form, `continue` transfers control to the end of the innermost loop's body.
- The labeled form transfers control to the end of the loop with the label. The label must belong to a loop statement.

Statements (21)

```
static void duplicateSymmMatrix(int[][] matrix) {  
    int dim = matrix.length;  
    column:  
    for (int i=0; i<dim; i++) {  
        for (int j=0; j<dim; j++) {  
            matrix[i][j]=matrix[j][i]=matrix[i][j]*2;  
            if (i==j)  
                continue column;  
        }  
    }  
}
```


Statements (22)

- A **return** statement terminated execution of a method and returns to the invoker.
 - If the method returns no value (`void`), use simply `return;`
 - If the method has a return type, the `return` must include an expression of a type that could be assigned to the return type.
- A `return` may also be used to exit a constructor; in this case, as constructors have no return type, use only `return;`

Object Oriented Programming

Java

Part 4: Environment classes

The java.lang package

- The **java.lang package** is automatically imported:
 - Interfaces:
 - **Cloneable**
 - **Runnable**
 - Classes:
 - **Class** e **Object**
 - **Boolean**, **Number** (and subclasses), **Character**, **Void**
 - **Math**
 - **Process**, **Thread**, **System** e **Runtime**
 - **String** and **StringBuffer**
 - **Throwable** and **Exception** (and subclasses)

The Object class (1)

- The `Object` class is the root of the class hierarchy in Java.
 - Every class directly or indirectly extends `Object`.
 - A variable of type `Object` can refer to any object.

The Object class (2)

- **Methods of the Object class:**

- **public int hashCode()**
Returns a hash code for this object.
- **public String toString()**
Returns a textual description for this object.
- **public boolean equals(Object obj)**
Returns equality between this object and `obj`.

Recall that:

- **Identity between object:** two references to the same object.
- **Equality between objects:** two objects with the same state (with the same field's values).

The Object class (3)

- Both the `equals` and `hashCode` should be overridden if one wants to provide a notion of equality different from the default implementation provided by the `Object` class.
 - **By default, `equals` implements identity between objects** (with distinct objects it returns `false`).

Note: The `==` and `!=` operators test always for identity between objects. That is, if the programmer overrides the `equals` method it continues to be able to test for identity through `==` and `!=`.

- **By default, two distinct objects usually return a different `hashCode`.**

The Object class (4)

- If the `equals` method is overridden to implement equality between objects, then the `hashCode` method should also be overridden accordingly, that is, it should be overridden in such a way that two objects evaluated as equal by `equals` must return the same value from `hashCode`.
 - The mechanism used by hashed collections relies on `equals` returning `true` when it finds a key of the same value in the table (for instance, `Hashtable` and `HashMap`); and the key is computed from `hashCode`.

The Object class (5)

- Usually classes override the `toString` method.
- The method has several uses:
 - Debugging.
 - Providing a textual description of the object.

Classe Object (5)

- **Methods of the Object class (cont):**

- **protected void finalize()**

Invoked by the garbage collector when the object is ceased to be referred.

Note: In Java, an object exists while it is being referred. The garbage collector reclaims the memory occupied by objects that are no longer being used by the program.

- **protected Object clone()
throws CloneNotSupportedException**

Builds and returns a copy of the calling object, with exactly the same fields. However, if the class of the calling `Object` does not implements the `Cloneable` interface, then the `CloneNotSupportedException` is thrown.

Note: For any `obj`, we have that

```
obj.clone() != obj;
```

Primitive types (1)

- **Primitive data types:**
 - **boolean** 1-bit (`true` or `false`)
 - **char** 16-bit Unicode UTF-16 (unsigned)
 - **byte** 8-bit signed integer
 - **short** 16-bit signed integer
 - **int** 32-bit signed integer
 - **long** 64-bit signed integer
 - **float** 32-bit IEEE 754 floating point
 - **double** 64-bit IEEE 754 floating point

Primitive types (2)

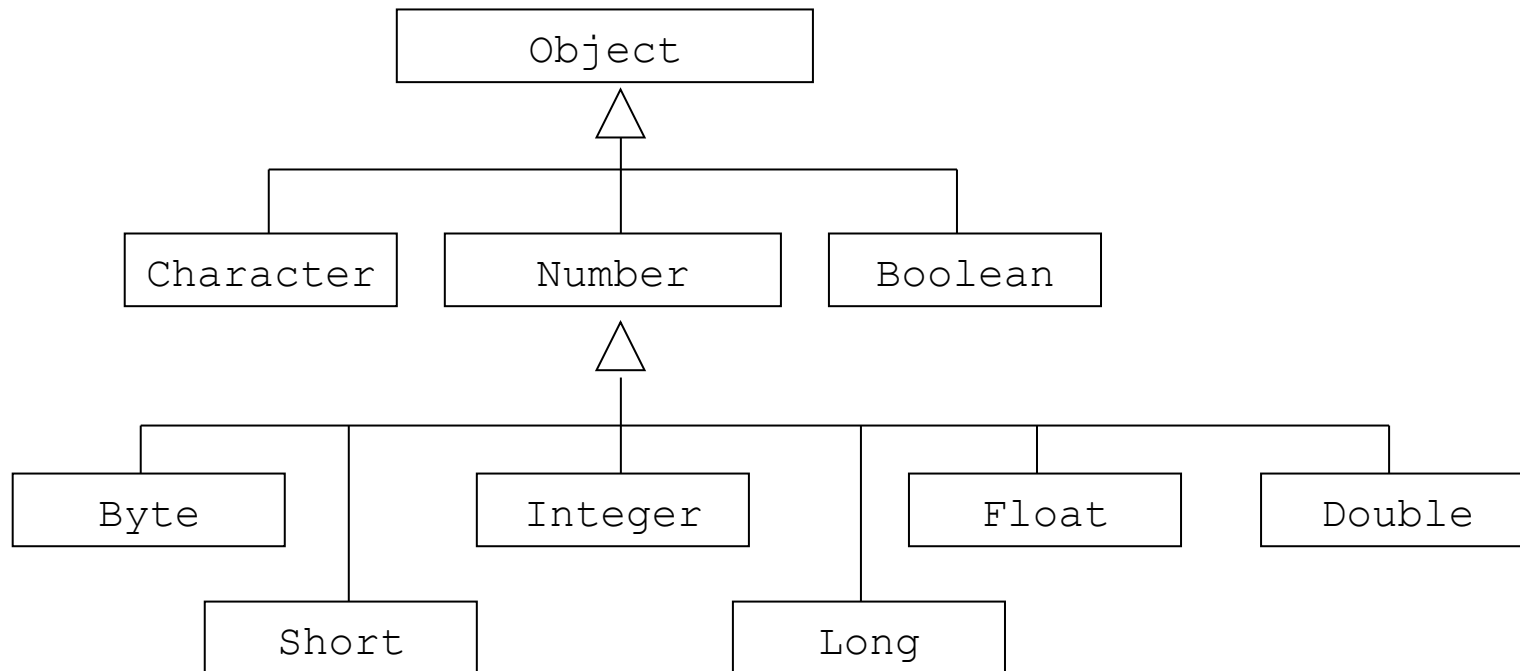
- In Java, for each primitive type there is in `java.lang` package a corresponding **wrapper class**.
- These wrapper classes, **Boolean**, **Character**, **Byte**, **Short**, **Integer**, **Long**, **Float** and **Double** provide a home for method and variables related to the type, e.g. string conversion and value range constants.
- Primitive data types:
 - Are fast and more convenient than reference types.
 - Occupy always the same space, independently from the machine where they are running on.

Primitive types (3)

- **Type casting:**
 - The java performs **implicit casting** of primitive types, in the order: `byte`->`short`->`int`->`long`->`float`->`double`.
 - An expression with distinct types results in a value of the superior type (for instance, `5+3.0` results in the value `8.0`).
 - When the implicit casting is not possible, usually an **explicit casting** is used (for instance, when casting a `float` into an `int` the fractional part is lost by rounding towards zero and `(int)-72.3` results in the value `-72`).

Wrapper classes (1)

- **Type hierarchy of wrapper classes:**



Wrapper classes (2)

- Instances of a given wrapper class contain a value of the corresponding primitive type.
- Each wrapper class defines an **immutable object** for the primitive value that is wrapping, that is, once the object is created the value represented by that object can never be created.
- The language provides **automatic conversion** between primitives and their wrappers:
 - **Boxing**: converts a primitive value to a wrapper object.
 - **Unboxing**: extracts a primitive type from a wrapper object.

```
Integer val = 3;
```

Wrapper classes (3)

- In the following, `Type` refers to the wrapper class, and `type` is the corresponding primitive type.
- Each wrapper class, `Type`, has the following methods:
 - `public static Type valueOf(type t)`
returns an object of the specified `Type` with the value `t`.
 - `public static Type valueOf(String str)`
returns an object of the specified `Type` with the value parsed from `str` (except for `Character` class).
 - `public type typeValue()`
returns the primitive value corresponding to the current wrapper object.

```
Integer.valueOf(6).intValue();
```

Wrapper classes (4)

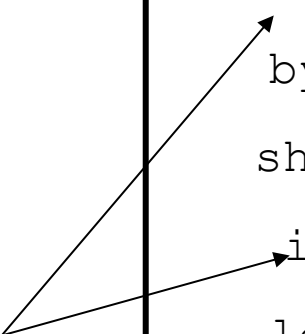
- `public static type parseType(String str)`
converts the string `str` to a value of the specified primitive `type`.
- `public static String toString(type val)`
returns a string representation of the given primitive value.
- All wrapper classes, with the exception of `Boolean`, define three constant fields:
 - `public static final type MIN_VALUE`
the minimum value the `type` can have.
 - `public static final type MAX_VALUE`
the maximum value the `type` can have.
 - `public static final int SIZE`
the number of bits used to represent the `type`.

Only semantic needs to be known.

Wrapper classes (5)

Primitive type	Wrapper class	Getting the value
boolean	Boolean	<code>booleanValue()</code>
char	Character	<code>charValue()</code>
byte	Byte	<code>byteValue()</code>
short	Short	<code>shortValue()</code>
int	Integer	<code>intValue()</code>
long	Long	<code>longValue()</code>
float	Float	<code>floatValue()</code>
double	Double	<code>doubleValue()</code>

Note:
Not all
names
match!



```
Integer I = new Integer(3);  
int i = I.intValue(); // i passa a ter 3  
String str_i = Integer.toString(i); // str_i has value "3"
```

Only semantic needs to be known.

Character class (1)

Static methods	Description
<code>boolean isLowerCase(char)</code>	Is the char a lowercase letter?
<code>boolean isUpperCase(char)</code>	Is the char an uppercase letter?
<code>boolean isDigit(char)</code>	Is the char a digit?
<code>boolean isSpace(char)</code>	Is it a <code>'\t'</code> , <code>'\n'</code> , <code>'\f'</code> or <code>' '</code> ?
<code>char toLowerCase(char)</code>	Converts the char to lowercase letter.
<code>char toUpperCase(char)</code>	Converts the char to uppercase letter.
<code>int digit(char,int)</code>	Returns the numeric value of char digit in the specified radix.

```
char c = Character.toUpperCase('g');
```

Only semantic needs to be known.

Character class (2)

- The Character class also has the following method:
 - `public static int getType(char)`
returns the `char`'s Unicode type - the return value is one of the following constants:
 - `CURRENCY_SYMBOL`
 - `LOWERCASE_LETTER`
 - `UPPERCASE_LETTER`
 - `MATH_SYMBOL`
 - `SPACE_SEPARATOR`
 - ...

Only semantic needs to be known.

Byte, Short, Integer and Float classes

- Byte, Short, Integer and Float classes also have the following method:
 - `public static type parseType(String str, int radix)`
converts `str` to a primitive type value of type `type` in the specified `radix` (decimal, by default).

Operators (1)

- **Binary Arithmetic operators:**
 - + addition
 - Subtraction
 - * multiplication
 - / division
 - % remainder
- The arithmetic operators can operate on any primitive numerical type (including chars, where the Unicode code is used).

Operators (2)

- **Increment/decrement operators:**
 - `++` increment
 - `--` decrement
- The increment/decrement operators can be applied to any primitive numerical type (including chars, next/previous Unicode code).

Operators (3)

```
int var = 5;
```

Statement	Result	Value after the statement
<code>System.out.println(var++);</code>	5	6
<code>System.out.println(++var);</code>	6	6
<code>System.out.println(var--);</code>	5	4
<code>System.out.println(--var);</code>	4	4
<code>System.out.println(var%3);</code>	2	5

Operators (4)

- **Relational operators:**

- > greater than
- >= greater than or equal to
- < less than
- <= less than or equal to

- **Equality operators:**

- == equal to
- != not equal to

- The relation and equality operators return a Boolean value, and they can be applied to primitive numerical types and chars.

Operators (5)

- The equality operators can be applied to any Boolean types.
- The equality operators can also be used to test identity between references:
 - In this case it refers to identity between objects, not equivalence:
 - **Identity:** `ref1==ref2` is `true` iff the two references refer to the same object (or if both are `null`).
 - **Equality:** `ref1.equals(ref2)` tests if the two references refer to (possibly distinct) objects with the same state (the same field values).
 - By default `equals` implements `==`.
 - If equality is needed, `equals` need to be overridden.

Operators (6)

- **Logical operators:**
 - ! logical negation
 - & logical AND
 - | logical inclusive OR
 - ^ logical exclusive OR (XOR)
 - && conditional AND (with lazy evaluation)
 - || conditional OR (with lazy evaluation)
- The logical operators combine Boolean expressions to yield Boolean values.

Operators (7)

- The **instanceof** operator evaluates whether a reference refers to an object that is an instance of a particular class or interface:
 - **Ref instanceof Ident**
verify if the reference **Ref** is of type **Ident**.

Operators (8)

- **Bit manipulation operators:**

& bitwise AND

| bitwise inclusive OR

^ bitwise exclusive OR (XOR)

~ complement (toggles each bit in its operand)

<< shift bits left, filling with 0 bits on the right-hand side

>> shift bits right, preserving the sign

>>> shift bits right, filling with 0 bits the left-hand side

- The bitwise operators apply only to integer types (including char).

Operators (9)

- The **conditional operator ? :** provides a single expression that yields one of the two values based on a Boolean expression:
 - **Expr-Bool ? Expr1 : Expr2**
if Expr-Bool is true then returns Expr1 else returns Expr2.

Operators (10)

- **Assignment operators:**

= simple assignment

op= composed assignment

- The left-operand must be a variable. The right-operand is an expression.
- The **op** operator might be any arithmetic, logic or bitwise operator.

Operators (11)

- **String concatenation operator:** +

```
String s1 = "boo";  
String s2 = s1+"hoo";  
s2 += "!";  
System.out.println(s2);
```

- The **new operator** creates an instance of a class or array.

Operators (12)

- **Operator precedence:**

1. Unary operators

`++ -- + - ~ !`

2. New or cast

`new (type)`

3. Multiplications

`* / %`

4. Additions

`+ -`

5. Shift

`<< >> >>>`

6. Relational

`< > >= <= instanceof`

7. Equality

`== !=`

8. AND

`&`

9. Exclusive OR

`^`

10. OR

`|`

11. Conditional AND

`&&`

12. Conditional OR

`||`

13. Conditional

`?:`

14. Assignment

`= += -= *= /= %= >>= <<= >>>= &= ^=`

`++x>3&&!b`
is equivalent to
`((++x)>3) && (!b)`

Operators (13)

- When two operators have the same precedence, the **operator associativity** determines the order of the operator evaluation.
 - **Left associative**: `expr1 op expr2 op expr3` is equivalent to `(expr1 op expr2) op expr3`.
 - **Right associative**: `expr1 op expr2 op expr3` is equivalent to `expr1 op (expr2 op expr3)`.
- Assignment operator is right associative. All other binary operators are left associative.
- The conditional operator is right associative.

Arrays (1)

- An **array** is an object containing a fixed number of elements, all from the same base type.
 - Arrays are objects themselves so they extend `Object`.
 - The base type can be a primitive or a reference type (including a references to other arrays).
 - J2SE also makes available unlimited collections, for instance, `Vector`, `Stack`, ...

Arrays (2)

Syntax

Base_type[] Ident = new Base_type [length]

- Array dimension is omitted in type declaration, the number of components in an array is determined when it is created using **new**.
- An array length is fixed at its creation and cannot be changed.
- The square brackets in type declaration may appear after the variable name Ident instead or after the type Base_type:
 - **Base_type [] Ident** or **Base_type Ident []**

Arrays (3)

- **Multidimensional arrays:**
 - Declared with several [].
 - The first (left-most) dimension of an array must be specified when the array is created.
 - Specifying more than the first dimension is a shorthand for a nested set of **new** statements.

```
float[][] mat = new float[4][4];
```

```
float[][] mat = new float[4][];  
for (int i=0; i < mat.length; i++)  
    mat[i] = new float[4];
```

Arrays (4)

- In a multidimensional array, each nested array can have a different size, and so we can have:
 - Triangular arrays
 - Rectangular arrays
 - ...

Arrays (5)

- The length of an array is available from its `length` field
`public final int length`
- The access to an element is done by using the name of the array and the index enclosed between [and]:
`Ident[pos]`.
- The first element of the array has index 0, and the last element of the array has index `length-1`.
- The access to indexes outside the proper range throws an exception `ArrayIndexOutOfBoundsException`.

```
int[] ia = new int[3];  
... //inicialização de ia  
for (int i=0; i < ia.length; i++)  
    System.out.println(i + ": " + ia[i]);
```

Arrays (6)

- An array might have dimension 0, and so it is said to be an **empty array**.
 - There is a big difference between a `null` array reference and a reference to an empty array.
 - Useful in methods' return.
- The usual modifiers can be applied to array variables and fields.
 - The modifiers apply only to the array reference and not to its elements.
 - An array variable that is declared **final** means that the array reference cannot be changed after initialization; it does not mean that array elements cannot be changed!

Arrays (7)

- **Array initialization:**

- When an array is created, each element is set to the default initial value of the type (zero for numeric types, false for `boolean`, `null` for references, etc).
- Arrays can be initialized in two different ways:
 1. With comma separated values inside braces:
 - There is no need to explicitly create the array using `new`.
 - The length of the array is determined by the number of initialization values given.

```
String[] animals = {"Lion", "Tiger", "Bear"};
```


Arrays (8)

- The `new` operator can be used explicitly, but in that case the dimension must be omitted (because, again, the length of the array is determined by the number of initialization values given).

```
String[] animals = new String[]{"Lion", "Tiger", "Bear"};
```

- The last value of the initialization list is allowed to have a comma after it.
- Multidimensional arrays can be initialized by nesting array initializers.

Arrays (8)

2. By direct assignment of its values:

```
String[] animals = new String[3];  
animals[0] = "Lion";  
animals[1] = "Tiger";  
animals[2] = "Bear";
```

Arrays (9)

- The `System` class offers a method that copies the values of an array into another:
 - `public static void arraycopy`
`(Object src, int srcPos,`
`Object dst, int dstPos,`
`int count)`
copies the content of `src` array, starting at `src[srcPos]`, to the destination array `dst`, starting at `dst[dstPos]`; exactly `count` elements will be copied.

Arrays (10)

- **Arrays as extension of Object class:**
 - Arrays do not introduce new methods, they only inherit methods from `Object`.
 - The `equals` method is always based on identity and not in equality.
 - The `deepEquals` method from the utility class `java.util.Arrays` allows to compare for equality between arrays.
 - Checks for equivalence between two `Object[]` recursively, taking into account the equivalence of nested arrays.

Arrays (11)

```
String[] animals = {"Lion", "Tiger", "Bear", };  
String[] aux = new String[animals.length];  
System.arraycopy(animals, 0, aux, 0, animals.length);  
  
for (int i=0; i<aux.length; i++)  
    System.out.println(i + ": " + aux[i]);  
  
System.out.println(aux.equals(animals));  
System.out.println(java.util.Arrays.deepEquals(aux, animals));
```

In the terminal is printed

```
0: Lion  
1: Tiger  
2: Bear  
false  
true
```

Arrays (12)

```
int[][] pascalTriangle1 = {
    { 1 }, { 1, 1 }, { 1, 2, 1 }, { 1, 3, 3, 1 }, { 1, 4, 6, 4, 1 } };

int[][] pascalTriangle2 = new int[5][];
pascalTriangle2[0] = new int[]{ 1 };
pascalTriangle2[1] = new int[]{ 1, 1 };
pascalTriangle2[2] = new int[]{ 1, 2, 1 };
pascalTriangle2[3] = new int[]{ 1, 3, 3, 1 };
pascalTriangle2[4] = new int[]{ 1, 4, 6, 4, 1 };

System.out.println(pascalTriangle1.equals(pascalTriangle2));
System.out.println(java.util.Arrays.deepEquals(
    pascalTriangle1, pascalTriangle2));
```

In the terminal is printed

false
true

String class (1)

- A **string** is an object that represents character strings.
 - Character strings are instances of the **String** class.
 - Strings are constant, their values cannot be changed after they are created.
 - If mutable string are required, use **StringBuffer** class.
 - A string is delimited by quotation marks (" and ").
 - The **+** operator concatenates two strings.

String class (2)

- **Building strings:**
 - Implicitly, by the use of a literal, or by operators such as **+** and **+=** on two objects of type **String**.
 - Explicitly, by the **new** operator (only some constructors, see Java documentation):
 - **public String()**
Allocates a newly creates String that represents the empty char sequence ("").
 - **public String(String valor)**
Copy constructor.

String class (3)

- **public String (char[] value)**
Allocates a new `String` that represents the specified char array.
- **public String(char[] valor, int offset, int count)**
Allocates a new `String` that represents a subarray of the specified char array. The **offset** is the index of the first char of the subarray. The **count** specifies the length of the subarray.

String class (4)

```
String s1 = "Bom";  
String s2 = s1 + " dia";  
String vazia = "";
```

```
String vazia = new String();  
String s1 = new String("Bom dia");  
char valor[] = {'B', 'o', 'm', ' ', 'd', 'i', 'a'};  
String s2 = new String(valor);  
String s3 = new String(valor, 4, 3);
```

String class (5)

- **Public methods from String:**
 1. String properties:
 - **int length()**
Length of this string.
 - **int compareTo(String str)**
Compares the this string with the specified string lexicographically. The comparison is based on the Unicode value of each char in the strings. Returns a negative integer if this string lexicographically precedes the specified string; a positive integer if this string lexicographically follows the specified string; and the result is 0 if the strings are equal.

String class (6)

2. Examining individual chars:

- **char charAt(int)**
Char at the specified index.
- **char[] toCharArray()**
Returns this string as a char array.
- **int indexOf(char)**
First index where the specified char occurs in this string.
- **int lastIndexOf(char)**
Last index where the specified char occurs in this string.
- **String substring(int, int)**
Substring of this string between specified positions.
- **String substring(int)**
Substring from specified position until the end of this string.

The first char of the string is in index 0.

String class (7)

3. Usual string operations:

- **String replace(char oldChar, char newChar)**
Returns a new string resulting from replacing all occurrences of `oldChar` in this string with `newChar`.
- **String toLowerCase()**
Converts all the chars in this string to lower case.
- **String toUpperCase()**
Converts all the chars in this string to upper case.
- **String trim()**
Returns a copy of the string, with leading and trailing whitespaces omitted.
- **String concat(String)**
Concatenates the specified string to the end of this string.

String class (8)

```
String s = "/home/asmc/oop-lecture.ppt";  
...  
int inicio, fim;  
inicio = s.lastIndexOf('/');  
fim = s.lastIndexOf('.');  
System.out.println(s.substring(inicio+1,fim));
```

In the terminal is printed oop-lecture

Only semantic needs to be known.

String class (9)

- Conversion between primitive type and String:

Type	To String	From String
boolean	<code>String.valueOf(boolean)</code>	<code>Boolean.parseBoolean(String)</code>
int	<code>String.valueOf(int)</code>	<code>Integer.parseInt(String, int)</code>
long	<code>String.valueOf(long)</code>	<code>Long.parseLong(String, int)</code>
float	<code>String.valueOf(float)</code>	<code>Float.parseFloat(String)</code>
double	<code>String.valueOf(double)</code>	<code>Double.parseDouble(String)</code>

String class (10)

- **Conversion between char array and String:**
 - In the `String` class:
 - `public char[] toCharArray()`
 - In the `System` class:
 - `public static void arraycopy(Object src, int srcPos, Object dst, int dstPos, int count)`
copies the content of `src` array, starting at `src[srcPos]`, to the destination array `dst`, starting at `dst[dstPos]`; exactly `count` elements will be copied.

String class (11)

```
public static String squeezeOut(String from, char toss){
    char chars[]=from.toCharArray(); // char array from String
    int len=chars.length;           // length of char array

    for (int i=0; i<len; i++) {
        if (chars[i]==toss) {
            --len; // final string has one less char
            System.arraycopy(
                chars, i+1, chars, i, len-i); // shift right end
            --i; // to continue searching in the same position
        }
    }
    return new String(chars, 0, len);
}
```

System.out.println(squeezeOut("Programação por Objetos", 'o'));
prints in the terminal Prgramaçã pr Objets

String class (12)

- The `String` class overrides the `equals` method from `Object` to return `true` iff two strings have the same content.
- It also overrides `hashCode` to return a hash based on the contents of the `String` so that two strings with the same content have the same `hashCode`.

```
String s1 = new String("abc"), s2 = "abc";
```

Expression	Result	Justification
<code>s1==s2</code>	false	Distinct objects
<code>s1.equals(s2)</code>	true	Same content

Only semantic needs to be known.

Math class (1)

- The Math class contains methods for performing basic numeric operations and mathematical constants.

Constant	Meaning
PI	π
E	e

```
System.out.println("Pi=" + Math.PI);
```

Only semantic needs to be known.

Math class (2)

- All methods are static (num is used for int, long, float or double)

Method	Description
<code>double sin(double)</code>	Sine
<code>double pow(double, double)</code>	1st arg raised to the power of the 2nd arg
<code>num abs(num)</code>	Absolute value
<code>num max(num, num)</code>	Maximum
<code>num min(num, num)</code>	Minimum
<code>int round(float)</code> <code>long round(double)</code>	Round to the nearest number
<code>double sqrt(double)</code>	Square root

Object Oriented Programming

Java

Part 5: Associations

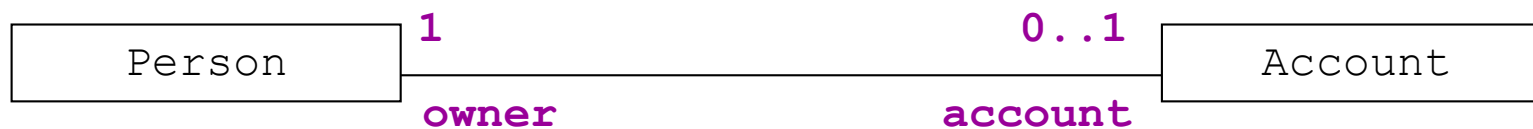
Association – revision

- An **association** represents a reference between objects.
- In an association are defined:
 - **Identifier** – term that describes the association.
 - **Role** – roles represented by the association in each of the related classes.
 - **Multiplicity** – number of objects associated in each of the related association.

Association (1)

- **Association**: represented by fields of the associated type.
 1. The association is established when both fields are initialized.
 2. Deleting an association requires that both fields cease to have a reference to the associated object.

Association (2)



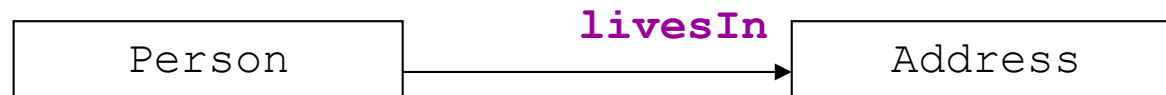
```
public class Person{
    Account account;
    ...
    Person() {
        account = null;
        ...
    }
    void associateAccount(Account a)
    {
        account = a;
    }
}
```

```
public class Account{
    Person owner;
    ...
    Account(Person o) {
        owner = o;
        ...
    }
}
```


Association (3)

- **Directed associations:** in Java, only the from-class contains a field to the to-class.
- **Association multiplicities:** in Java, multiplicity distinct from 0..1 and 1 is implemented with fields of type array, `Vector`, ... in the associated classes.
- **Associations with extra information:** in Java, the associated classes have an extra field to the association class.

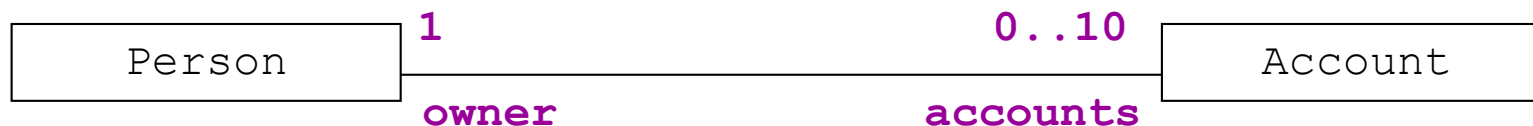
Association (4)



```
public class Person{
    Account account;
    Address livesIn;
    ...
    Person(Address a) {
        account = null;
        livesIn = a;
        ...
    }
}
```

```
public class Address{
    String street;
    ...
}
```

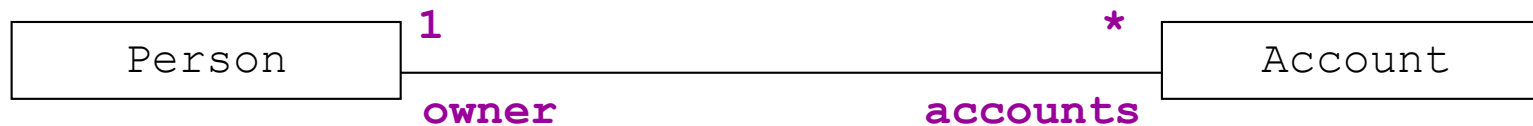
Association (5)



```
public class Person{
    int idxNextAccount=0;
    static final int nbMaxAccount=10;
    Account[] accounts;
    ...
    Person() {
        accounts = new Account[nbMaxAccount];
    }
    void associateAccount(Account c) {
        if (idxNextAccount<nbMaxAccount)
            accounts[idxNextAccount++] = c;
        else
            System.out.println("Maximum number attained!");
    }
}
```

```
public class Account{
    Person owner;
    ...
    Account(Person o) {
        owner = o;
    }
}
```

Association (6)

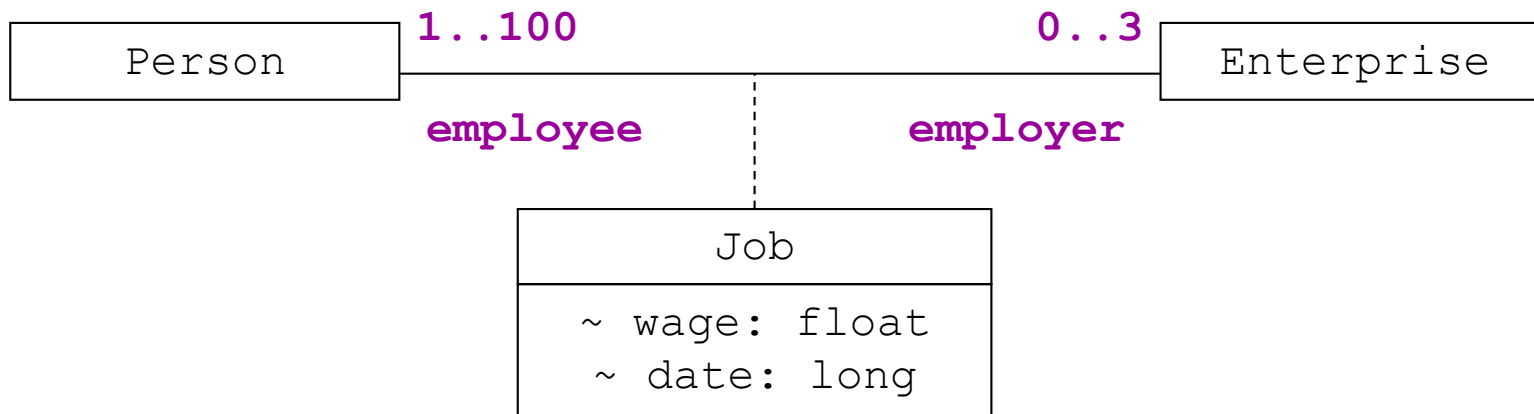


```
import java.util.LinkedList;

public class Person{
    LinkedList<Account> accounts;
    ...
    Person() {
        accounts = new LinkedList<Account>();
    }
    void associateAccount(Account c) {
        accounts.add(c);
    }
}
```

```
public class Account{
    Person owner;
    ...
    Account(Person o) {
        owner = o;
    }
}
```

Association (7)



```
public class Person{
    Enterprise[] employers;
    Job[] jobs;
    ...
    void associateJob(
        Enterprise employer,
        Job job){...}
}
```

```
public class Job{
    float wage;
    long date;
    ...
}
```

Association (8)

```
public class Enterprise{
    int idxNextJob=0;
    static final int nbMaxJobs=100;
    Person[] employees;
    Job[] jobs;
    ...
    Enterprise() {
        employees = new Person[nbMaxJobs];
        jobs = new Emprego[nbMaxJobs];
        ...
    }
    void newJob(Person person, Job job) {
        if (idxNextJob<nbMaxJobs) {
            employees[idxNextJob]=person;
            jobs[idxNextJob++]=job;
        } else System.out.println("Maximum number attained!");
    }
}
```

Association (9)

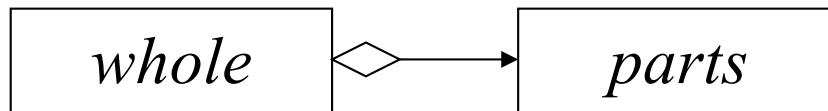
- It is the responsibility of the programmer to ensure the proper establishment of associations and maintenance of their consistency.

Aggregation/Composition – revision (1)

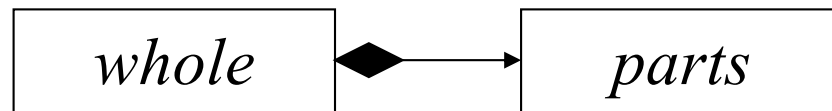
- An **aggregation/composition** is an association which denotes that the *whole* is formed by *parts*.
- The aggregation/composition is said to be a relationship of “**has-a**”.

Aggregation/Composition – revision (2)

- The **aggregation** is an association which denotes that the *whole* is formed by *parts*.



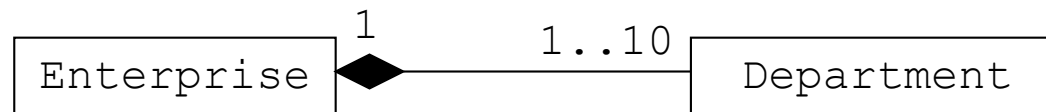
- In **composition** there is no-sharing, that is, when the owning object is destroyed, so are the contained objects.



Aggregation/Composition (1)

- An **aggregation/composition** is implemented in Java as an association, with fields which are references to the associated objects.
- **Regarding composition, it is still necessary to ensure consistency over the disappearance of the *whole*, which in turn implies the disappearance of the *parts*.**

Aggregation/Composition (2)



```
public class Enterprise{
    int idxNextDepartment = 0;
    static final int nbMaxDepartments = 10;
    Department[] departments;
    ...
    Enterprise() {
        departments = new Department[nbMaxDepartments];
        ...
    }
}
```

Aggregation/Composition (3)

```
// ... Continues previous slide

void newDepartment () {
    if (idxNextDepartment < nbMaxDepartments)
        departments[nbNextDepartment++]
            = new Department(this);
    else System.out.println("Maximum number attained!");
}
}
```

```
public class Department {
    Enterprise enterprise;
    ...
    Department(Enterprise e) {
        enterprise = e;
    }
}
```

Object Oriented Programming

Java

Part 6: Inheritance and polymorphism

Inheritance – revision (1)

- Inheritance is a mechanism where the **subclass** constitute a specialization of a **superclass**. The superclass might be seen as a **generalization** of its subclasses.
- Inheritance is said as a relationship of “**is-a**”.
- Subclasses inherit the attributes and methods of superclasses. The inherit methods might be modified. New attributes and methods might be added to the subclasses.

Inheritance – revision (2)

- **Polymorphism** occurs when there is **redefinition** of methods (methods with the same signature) of the superclass in the subclass.
- In OO, polymorphism is usually implemented through **dynamic binding**, i.e., the method being executed is determined only in runtime (and not in compile time).

Inheritance – revision (3)

- In **simple inheritance** each subclass has only a direct superclass.
- In **multiple inheritance** a subclass might have more than one direct superclass.

Inheritance (1)

- Java adopts the following strategies regarding inheritance:
 - There is only **simple inheritance of classes**.
 - **All classes are subclasses of Object** (explicitly or implicitly).
 - **The subclass inherits all public and protected (but not private) fields and methods of the superclass.**
 - In the case of fields and methods without a visibility modifier (package visibility), the subclass inherits these fields and methods if it is defined in the same package as the superclass, and only in that case.
 - The constructors are not methods, so they are not inherited.

Inheritance (2)

- If the subclass declares a method with the same identifier and parameters (number and type) as one of its superclasses, then the **subclass overrides/redefined that method**.
- If the subclass declares a field with the same identifier as one of its superclasses, then the field of the **subclass hides the field of the superclass** (but it continues to exist!)

Inheritance (3)

Syntax

Modifier* class Ident

```
[ extends IdentC] [ implements IdentI [,IdentI]* ] {  
  [ Fields | Methods ]*  
}
```

- **Modifier**: modifier (visibility, among others)
- **Ident**: class name
- **extends IdentC**: specialization of the superclass
- **implements IdentI**: implementation of interfaces

Inheritance (4)

```
public class SavingAccount extends Account {
    private static float interestRate=0.05;
    private long begin;
    private int interval;

    public void interestRate() {
        long today=System.currentTimeMillis;
        if(today==begin+interval) {
            balance *= (1+interestRate);
            begin = today;
        }
    }
}
```

Constructors in subclasses (1)

- The subclass constructor should initialize its own fields but only the superclass knows how to correctly initialize its fields.
- The subclass constructor must delegate construction of the inherited fields by either implicitly or explicitly invoking a superclass constructor.
- An **explicit invocation** of a superclass constructor is done with `super()`.
- If the superclass constructor has `N` parameters, these should be passed in the explicit invocation: `super(param1, ..., paramN)`.
- If provided, the explicit invocation must be the first statement in the constructor.

Constructors in subclasses (2)

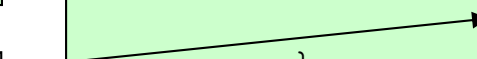
- The choice of the superclass constructor can be deferred, invoking explicitly one of the class constructors, using `this()` (instead of `super()`).
- If no superclass constructor is called, or if no class constructor is called, as the first statement of a constructor, the no-arg constructor from the superclass is implicitly called before any statement of that constructor.
- If the superclass does not have a no-arg constructor, the superclass constructor invocation is mandatory.

Constructors in subclasses (3)

```
public class A {  
    protected int a;  
    A() {  
        a = 5;  
    }  
    A(int var) {  
        a = var;  
    }  
}
```

It is not necessary to call **explicitly** `super()` as `super()` is implicitly called!

```
public class DuplicateA extends A {  
    DuplicateA() {  
        a *= 2;  
    }  
    DuplicateA(int var) {  
        super(var);  
        a *= 2;  
    }  
}
```



Constructors in subclasses (4)

```
public class A {  
    protected int number;  
    A(int num) {  
        number=num;  
    }  
}
```

It is mandatory to explicitly call `super(-1)` because the superclass does not have a no-arg constructor!

```
public class B extends A {  
    protected String name="not-defined";  
    B() {  
        super(-1);  
    }  
    B(int num) {  
        super(num);  
    }  
    B(int num, String str) {  
        this(num);  
        name= str;  
    }  
}
```


Constructors in subclasses (5)

- When an object is created, memory is allocated for all its fields, including those inherited from superclasses.
- Those fields are set to default initial values for their respective types (zero for all numeric types, false for boolean, '\u0000' for char, and null for object references).
- After this, construction has three phases:
 1. Invoke a superclass constructor (through an implicit or explicit invocation).
 - If the explicit `this()` constructor invocation is used then the chain of such invocations is followed until an implicit or explicit superclass constructor invocation is found.
 - The superclass constructor is executed in the same three phases; this process is applied recursively, terminating when the constructor for `Object` is reached.

Constructors in subclasses (6)

- Any expressions evaluated as part of an explicit constructor invocation are not permitted to refer to any of the members of the current object.
2. Initialize the fields using their initializers and any initialization blocks.
 - In this second stage all the field initializers and initialization blocks are executed in the order in which they are declared.
 - At this stage references to other members of the current object are permitted, provided they have already been declared.
 3. Execute the body of the constructor.

Constructors in subclasses (7)

- When an object of type B is created...

```
public class A {  
    protected int a=1;  
    protected int total;  
    A() {  
        total=a;  
    }  
}
```

```
public class B extends A {  
    protected int b=2;  
    B () {  
        total+=b;  
    }  
}
```

1. Fields with default values
2. Constructor of B is called
3. Constructor of A is called (`super()`)
4. Constructor of `Object` is called
5. Initialization of fields in A
6. Constructor of A is executed
7. Initialization of fields in B
8. Constructor of B is executed

a	b	total
0	0	0
0	0	0
0	0	0
0	0	0
1	0	0
1	0	1
1	2	1
1	2	3

Inheritance and redefinition (1)

- In a subclass it might:
 - be added new fields and methods to the class.
 - be overridden/redefined methods from the superclass.

Inheritance and redefinition (2)

- A subclass method is an **override** of a superclass method if:
 - Both have the same identifier and parameters (number and type).
 - The return type might be **covariant**:
 - If the return type is a reference type then the overriding method can declare a return type that is a subtype of that declared by the superclass method.
 - If the return type is a primitive type, then the return type of the overriding method must be identical to that of the superclass method.

Inheritance and redefinition (3)

- **A method can be overridden only if it is accessible.**
 - If the method is not accessible then it is not inherited, and if it is not inherited it can't be overridden.
 - If a subclass defines a method that coincidentally has the same signature and return type as a superclass private method, they are completely unrelated, the subclass method does not override the superclass private method.
 - Invocations of private methods always invoke the implementation of the method declared in the current class.

Inheritance and redefinition (4)

- The overriding methods have their own access specifies. A subclass can change the access of a superclass method, but only to provide more access.
 - A method declared protected in the superclass can be redeclared **protected** (the usual thing to do) or declared **public**, but it cannot be declared **private** or have **package** access.
 - Making a method less accessible than it was in a superclass violates the contract of the superclass, because an instance of the subclass would not be usable in place of a superclass instance.

Inheritance and redefinition (5)

- **An instance method cannot have the same identifier and parameters (number and type) as an inherited static method, and vice-versa.**
- **The overriding method can, however, be made abstract, even though the superclass method was not.**

Inheritance and redefinition (6)

- **A subclass can change whether a parameter in an overriding method is final** (this is just an implementation detail).
- **The overriding method can be final**, but obviously the method it is overriding cannot.

Inheritance and redefinition (6)

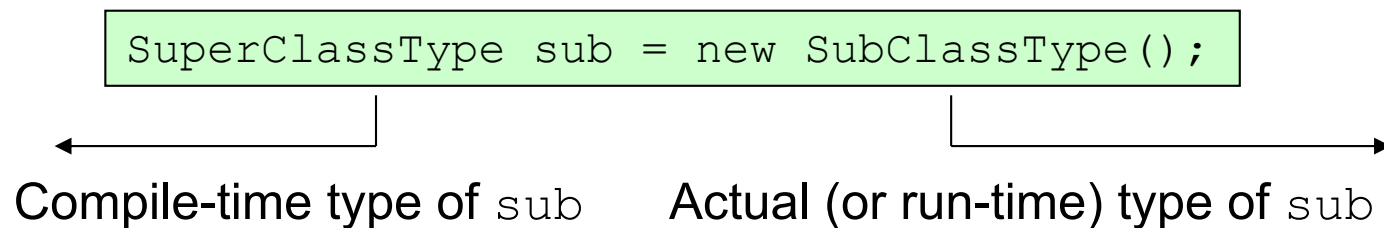
- If a method in the subclass has
 - the same identifier,
 - but different number or parameter types,of a (visible) method in the superclass, then it is an **overloading**.
- If a method in the subclass has
 - the same identifier,
 - the same number or parameter types,
 - but the return is not covariantof a (visible) method in the superclass, then it is a **compile-time error**.

Inheritance and redefinition (7)

- Fields cannot be overridden, they can only be **hidden**.
 - If a field is declared in a subclass with the same name (regardless of type) as one in the superclass, that other field (of the superclass) still exists (in the subclass), but it can no longer be accessed directly by its simple name.
 - In this case, the **super** reference (see slide 32) or another reference of the superclass type must be used to access it.

Polymorphism (1)

- In a class hierarchy, when there is redefinition the implementation of the method is replaced.
 - When an object invokes a method that has been redefined in a subclass, to which method it refers, the superclass method or the subclass method?
 - **When a method is invoked through an object reference, the actual type of the object governs which implementation is used.**



Polymorphism (2)

- **The declared type of a reference is determined in compile time.**
 - The compiler has access to that information directly from the declaration.

```
SuperClass sub = new SubClass();
```

```
SuperClass sub;  
...  
//in any point of the program  
sub = new SubClass();
```

- **Static type:** declared type or explicit cast.

Polymorphism (3)

- The instantiation type (**actual or dynamic type**) of an object is only determined in run time.
 - The actual type maybe a subclass of the declared type.
 - The instantiation of an object can be done in a different point of the program from the point where the declaration statement was made.
 - Only the program flow will dictate the actual class.

```
SuperClass sub;  
...  
//in other point of the program  
sub = new SubClass();
```

Polymorphism (4)

```
public class SuperClass {  
    protected String str = "SupCStr";  
    public void print() {  
        System.out.println("SupCPrint(): "+str);  
    }  
}
```

```
public class SubClass extends SuperClass {  
    protected String str = "SubCStr";  
    public void print() {  
        System.out.println("SubCPrint(): "+str);  
    }  
    // continues in the next slide
```

Polymorphism (5)

```
// continued from previous slide

public static void main(String[] args) {
    SubClass sub = new SubClass();
    SuperClass sup = sub;
    sup.print();
    sub.print();
    System.out.println("sup.str = "+sup.str);
    System.out.println("sub.str = "+sub.str);
}
}
```

In the terminal is printed

```
SubCImprime(): SubCStr
SubCImprime(): SubCStr
sup.str = SupCStr
sub.str = SubCStr
```


Polymorphism (6)

- Relatively to the previous slide:
 - The declared and instantiation type of `sub` is `SubClass`.
 - The `sub` reference was declared as being of type `SubClass`.
 - Memory was allocated for `sub` as being an object of type `SubClass`.
 - The declared type of `sup` is `SuperClass`.
 - The `sup` reference was declared as being of type `SuperClass`.
 - The instantiation type of `sup` is `SubClass`.
 - Memory was allocated for `sup` as being an object of type `SubClass` (assignment of references, see slide 22 of Java - part 3: methods).
 - The same would happen if

```
sup = new SubClasse();
```

The `super` reference (1)

- The `super` keyword is available in all non-static methods of a class.
- In field access and method invocation, `super` acts as a reference to the current object as an instance of its superclass.
 - Using `super` is the only case in which the type of the reference governs selection of the method implementation to be used.
- An invocation of `super.method` uses always the implementation of the method the superclass defines (or inherits).
 - It does not use any overriding of that method further down the class hierarchy.

The super reference (2)

```
public class SuperClass {  
    protected void name() {  
        System.out.println("SuperClass");  
    }  
}
```

```
public class SubClass extends SuperClass {  
    protected void name() {  
        System.out.println("SubClass");  
    }  
    // continues in the next slide
```

The super reference (3)

```
// continued form previous slide

protected void printName() {
    SuperClass sup = (SuperClass) this;
    System.out.print("this.name(): ");
    this.name();
    System.out.print("sup.name(): ");
    sup.name();
    System.out.print("super.name(): ");
    super.name();
}
}
```

In the terminal is printed

```
this.name(): SubClass
sup.name(): SubClass
super.name(): SuperClass
```

Static members (1)

- **Static members within a class, whether fields or methods, cannot be overridden, they are always hidden.**
- If a reference is used to access a static member then, as with instance fields, static members are always accessed based on the declared type of the reference, not the actual type of the object referred to.

Static members (2)

```
public class SuperClass {  
    protected static String str = "SupCStr";  
    public static void print() {  
        System.out.println("SuperCPrint(): "+str);  
    }  
}
```

```
public class SubClass extends SuperClass {  
    protected static String str = "SubCStr";  
    public static void print() {  
        System.out.println("SubCPrint(): "+str);  
    }  
    // continues in the next slide
```

Static members (3)

```
// continued from previous slide

public static void main(String[] args) {
    SubClass sub = new SubClass ();
    SuperClass sup = sub;
    sup.print();
    sub.print();
    System.out.println("sup.str = "+sup.str);
    System.out.println("sub.str = "+sub.str);

}
}
```

In the terminal is printed

```
SupCPrint(): SupCStr
SubCPrint(): SubCStr
sup.str = SupCStr
sub.str = SubCStr
```

Explicit conversion (1)

- A cast is used to inform the compiler that the casted expression should be interpreted as being declared by the type specified by the cast.
 - **Upcast**: cast of a class to any superclass type (from subclass to superclass, not necessarily a direct superclass).
 - **Downcast**: cast of a class to any subclass type (from superclass to subclass, not necessary a direct subclass).
- The upcast is also known as **safe cast**, because it is always valid.

Explicit conversion (2)

- In Java, a field or local variable declared with a superclass type can refer any actual subclass object. However, it can only be directly:
 - invoked methods of the superclass.
 - accessed fields declared in the superclass.

```
public class A {  
    void foo() {...}  
    ...  
}
```

```
public class B extends A {  
    void b() {...}  
    ...  
}
```

```
public class Warehouse {  
    A var[] = new A[2];  
    void xpto() {  
        var[0] = new A();  
        var[0].foo();  
        var[1] = new B();  
        var[1].foo();  
    }  
}
```

Explicit conversion (3)

- Relatively to the previous slide:
 - References `var[0]` and `var[1]` do not have (direct) access to method `b()` of `B` (even being `var[1]` instantiated as an object of type `B`).
 - However, `var[1]` can invoke method `b()` with a downcast: `((B)var[1]).b();`
 - On the other hand, the statement `((B)var[0]).b();` is valid in compile time, but it throws an exception in runtime!
 - If the subclass `B` overrides the method `foo()`, which implementation will be invoked with `var[1].foo()`?
 - The actual type of `var[1]` governs the method that will be invoked, in this case the `foo()` from `B` (see slide 24).

Object Oriented Programming

Java

Part 7: Interfaces

Interfaces – revision (1)

- An **interface** is a collection of operations (without implementation) that are used to specify a service of a class:
 - Interfaces do not contain attributes, except for constants.
 - An interface may be realized (or implemented) by a **concrete class**. In this **implementation** or **realization**:
 - The attributes needed to the correct implementation of the interface are determined.
 - The code of the methods made specified by the interface is provided.

Interfaces – revision (2)

- An interface may inherit the definitions of another interface.
 - Interfaces may use polymorphism.
 - If a class realizes more than one interface, and different interfaces have methods with the same signature, the class should provide only one implementation of those methods.
- An interface cannot be instantiated.

Interfaces (1)

- Both classes and interfaces define **types** (the fundamental unit of OO programming).
- In Java, a class can only extend another class, but it might implement more than one interface.
- For a given class the extended class and implemented interfaces are its **supertypes**. The class itself is the **subtype**.
- A reference to an object of the subtype can always be used when a reference to an object of the supertypes (classes or interfaces) is required.

Interfaces (2)

- Java provides a set of interfaces, being the most used:
 - `Comparable`: objects of this type have an associated order making possible to compare them.
 - `Iterable`: objects of this type provide an iterator and so they can be used in a enhanced for-each loop.
 - `Collection`: objects of this type store other objects.
 - `Set`, `List`, `Queue`, ...

Interfaces (3)

Syntax

```
Modifier* interface Ident [ extends IdentI [, IdentI ]* ] {  
    [ ModifierC* Type IdC = expression; ] *  
    [ ModifierM* Type IdM ( [TypeP IdP [, TypeP IdP ]*); ] *  
}
```

- **Modifier**: modifier (visibility, among others)
- **Ident**: interface name
- **extends IdentI**: interface specialization

Interfaces (4)

- **Interface modifiers:**
 - **public**: publicly accessible interface.
 - **abstract**: interfaces cannot be instantiated.
- When the **public** modifier is omitted the interface is only accessible in the package where it is defined.

Interfaces (5)

- All interfaces are implicitly **abstract**. By convention, the **abstract** modifier is omitted.
- All interface members are implicitly **public**. By convention, the **public** modifier is omitted.
- All interface constants are implicitly **public static final**. By convention, the modifiers are omitted.
- All interface methods are implicitly **public abstract**. By convention, the modifiers are omitted.
- No other qualifier is allowed to constants and methods of an interface.

Interfaces (6)

```
public interface Queue {  
    //methods  
    boolean empty();  
    Object top ();  
    boolean add(Object o);  
    void remove();  
}
```

Inheritance of interfaces (1)

- An interface may extend more than one interface.
- The extended interfaces are denominated **superinterfaces**, whereas the new interface is named **subinterface**.
- A subinterface inherits all constants declared in its superinterfaces.
- If a subinterface declares a constant with the same name as one inherited from its superinterfaces (independently of the type), the subinterface constant **hides** the inherited constant.
- In the subinterface the inherited constant is accessed only by its qualified name (**superinterface.constant**).

Inheritance of interfaces (2)

```
interface X {  
    int val = 1;  
    String strx = "X";  
}
```

```
interface Y extends X {  
    int val = 2;  
    int sum = val + X.val;  
    String stry = "Y extends" + strx;  
}
```

Inheritance of interfaces (3)

- If an interface inherits two or more constants with the same name, any non-qualified use of that constant is ambiguous and it results in a compile-time error.

```
interface A {  
    String str = "A";  
}
```

```
interface B extends A {  
    String str = "B";  
}
```

```
interface C extends A {  
    String str = "C";  
}
```

```
interface D extends B, C {  
    String d = str;  
}
```

```
interface D extends B, C {  
    String d = A.str+B.str+C.str;  
}
```

Compile-time error: which str?

Inheritance of interfaces (4)

- A subinterface all methods declared in its superinterfaces.
- If a subinterface declares a method with the same signature, up to a covariant return, as one or more methods inherited from the superinterfaces, the method in the subinterface is a **redefinition** of the inherited methods.
- If a subinterface inherits more than one method with the same signature, up to a covariant return, the subinterface contains only one method – the method that returns the common subtype (or one below in the hierarchy).

Inheritance of interfaces (5)

- If a subinterface method differs only in the return type of an inherited method, or two inherited methods differ only in the return type, and these returns are not covariant, there is a compilation error.
- If a subinterface method has the same name but different parameters of an inherited method, the subinterface method is an **overload** of the inherited method.

Inheritance of interfaces (6)

```
interface X {  
    void xpto();  
    Number foo1();  
    Number foo2();  
}
```

```
interface Y {  
    Object foo1();  
    Object foo2();  
}
```

```
interface Z extends X, Y {  
    void xpto(String s);  
    Integer foo1();  
}
```

- Methods of the interface Z:
 - `public void xpto()`
 - `public void xpto(String)`
 - `public Integer foo1()`
 - `public Number foo2()`

Implementation of interfaces (1)

- A class identifies the interfaces that implement, listing them after the keyword `implements`.

Syntax (revision)

Modifier* class Ident

**[extends IdentC] [implements IdentI [,IdentI]*] {
[Fields | Methods]*
}**

Implementation of interfaces (2)

- The interfaces that a class implements are denominated **superinterfaces** of the class.
- The class should provide an implementation for all methods defined in the superinterfaces, otherwise the class must be declared as **abstract**.

Implementation of interfaces (3)

- When a class implements an interface, the class can access the constants defined in the interface as if they were declared in the class.
- A class that implements more than one interface, or extends a class and implements one or more interfaces, suffers from the same problems of hidden constants and ambiguity that an interface which extends an interface (see slides 10, 11 and 12).

Implementation of interfaces (4)

```
interface X {  
    int val = 1;  
    String strx = "X";  
}
```

```
class Z implements Y {  
    int val = 3;  
}
```

```
interface Y extends X {  
    int val = 2;  
    int sum = val + X.val;  
    String stry = "Y extends " + strx;  
}
```

```
Z z = new Z();  
System.out.println(  
    "z.val=" + z.val +  
    " ((Y)z).val=" + ((Y)z).val + /* ou Y.val */  
    " ((X)z).val=" + ((X)z).val) /* ou X.val */;  
System.out.println("z.strx=" + z.strx + " z.stry=" + z.stry;
```

In the terminal is printed

```
z.val=3 ((Y)z).val=2 ((X)z).val=1  
strx=X stry=Y extends X
```

Implementation of interfaces (5)

```
interface A {  
    String str = "A";  
}
```

```
interface B extends A {  
    String str = "B";  
}
```

```
interface C extends A {  
    String str = "C";  
}
```

```
class D implements B, C {  
    String d = str;  
}
```

```
class D implements B, C {  
    String d = A.str+B.str+C.str;  
}
```

Compile-time error: which str?

Implementation of interfaces (6)

- If a class implements multiple interfaces with more than one method having the same signature class contains only one such method.
- If a class implements multiple interfaces with more than one method with the same signature, up to a covariant return, the implementation must define the method that returns the common subtype (otherwise results in a compile error).
- If a class implements multiple interfaces with more than a method that differs only in the return type, and these returns are not covariant, there is a compilation error.

Implementation of interfaces (7)

```
interface X {  
    void xpto();  
    Number foo1();  
    Number foo2();  
}
```

```
interface Y {  
    Object foo1();  
    Object foo2();  
}
```

```
interface Z extends X, Y {  
    void xpto(String s);  
    Integer foo1();  
}
```

```
class ClassZ implements Z {  
    public void xpto() {...}  
    public void xpto(String s) {...}  
    public Integer foo1() {...}  
    public Number foo2() {...}  
}
```

It is important to identify the methods to implement in an interface: if `ClassZ` does not provide an implementation of all methods defined in the superinterfaces it must be declared as **abstract** (see slide 17).

Implementation of interfaces (8)

- An implementation of the `Queue` interface may be done in two different ways:
 - Based on an array: `ArrayQueue`
 - Based on a linked list: `LinkedListQueue`
- The `Queue` interface corresponds to the **abstract data type**, whereas both `ArrayQueue` and `LinkedListQueue` correspond to the **data type** implemented in two different ways.

```
public interface Queue{  
    //methods  
    boolean empty();  
    Object top();  
    boolean add(Object obj);  
    void remove();  
}
```

Implementation of interfaces (9)

```
public class ArrayQueue implements Queue {  
  
    private final int MAX;  
    private Object queue[];  
    private int freePos; // first free position  
  
    public ArrayQueue(int max) {  
        MAX = max;  
        queue = new Object[MAX];  
        freePos = 0;  
    }  
    public boolean empty() {  
        return freePos==0;  
    }  
    public Object top() {  
        return freePos>0 ? queue[freePos-1] : null;  
    }  
}
```

Implementation of interfaces (10)

```
//continued from previous slide

public boolean add(Object obj){
    if (freePos<MAX-1){
        queue[freePos++] = obj;
        return true;
    }
    return false;
}

public void remove() {
    if (freePos>0)
        queue[--freePos] = null;
}

public int nbMaxElements() {
    return MAX;
}

}
```

Implementation of interfaces (11)

```
public class LinkedListQueue implements Queue {  
  
    private QueueElement base;  
    private int nbElements;  
  
    public LinkedListQueue() {  
        base = null;  
        nbElements = 0;  
    }  
    public boolean empty() {  
        return nbElements==0;  
    }  
    public Object top() {  
        return base!=null ? base.element : null;  
    }  
}
```

Implementation of interfaces (12)

```
//continued from previous slide

public boolean add(Object obj){
    base = new QueueElement(obj,base);
    nbElements++;
    return true;
}

public void remove () {
    if (base!=null) {
        base = base.next;
        nbElements--;
    }
}

public int nbElements() {
    return nbElements;
}

}
```

Implementation of interfaces (13)

```
public class QueueElement{  
  
    Object element;  
    QueueElement next;  
  
    public QueueElement(Object elem, QueueElement n){  
        element = elem;  
        next = n;  
    }  
}
```

Implementation of interfaces (14)

- As interfaces define a type, it is possible to declare variables/fields of that type:

```
Queue p = new ArrayQueue(100);
```

- However, references to an interface type, can only be used to access the members of the interface:

```
p.add(new Integer(100));  
p.add(new Character('a'));  
p.remove();
```

```
int max = p.nbMaxElements(); //INVALID!!!
```

- A cast can be used:

```
int max = ((ArrayQueue)p).nbMaxElements();
```

Implementation of interfaces (15)

- It is possible to invoke any method from `Object` with a reference to an interface:

```
String s = p.toString();
```


Implementation of interfaces (16)

- The programmer may instantiate any queue:

```
Queue p1 = new LinkedListQueue();  
Queue p2 = new LinkedListQueue();
```

```
p1.add(new Integer(5));  
p2.add(new Character('a'));
```

- If later the programmer decide to use instead an `ArrayQueue` only the instantiation needs to be updated, and not the declared type of the reference (of type `Queue`):

```
Queue p1 = new ArrayQueue(20);  
Queue p2 = new ArrayQueue(100);
```

```
p1.add(new Integer(5));  
p2.add(new Character('a'));
```

Object Oriented Programming

Java

Part 8: Generic types

Generic types – revision

- A **generic class** (or **parametric class**) is a class that receives as argument other classes. Instances of **generic classes** are denominated **parameterized classes**.
- Generic classes are commonly used to define collections (sets, lists, queues, trees, etc).

Motivation

- Before generics
(version 4 and previous versions):

```
List v = new ArrayList();  
v.add("test");  
Integer i = (Integer)v.get(0);
```

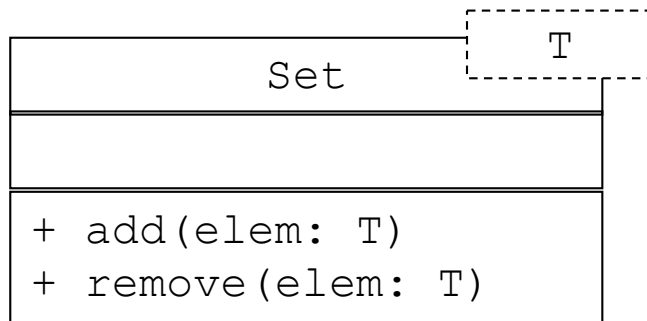
Compile without errors but
throws an exception (in run-
time) in the 3rd line!

- After generics
(version 5 and following versions):

```
List<String> v = new ArrayList<String>();  
v.add("test");  
Integer i = v.get(0);
```

Compile-time error
in the 3rd line 😊

Generic types (1)



```
public class Set<T> {  
    public void add(T elem) {...}  
    public void remove(T elem) {...}  
}
```

Generic types (2)

```
public class Element<T> {  
    protected T element;  
    protected Element<T> next;  
    public Element(T element) {  
        this.element = element;  
    }  
    public Element(T element, Element<T> next) {  
        this.element = element;  
        this.next = next;  
    }  
}
```

- To reference/build an object of type `Element` a **generic type invocation** must be performed, which replaces `T` with some concrete value, such as `String`:

```
Element<String> strElem = new Element<String>("Hello");
```

Generic types (3)

```
public class Set<T> {  
    protected int nbElems;  
    protected Element<T> head;  
    public void add(T elem) {  
        //verify if the element already exist in the set  
        for (Element<T> aux=head; aux!=null; aux=aux.next)  
            if (elem.equals(aux.element)) return;  
        //if not, add it in the beginning  
        head = new Element<T>(elem,head);  
        nbElems++;  
    }  
    public void remove (T elem) {...}  
}
```

Generic types (4)

- Once again, to reference/build an object of type `Set` it is necessary to perform a generic type invocation. For instance, a set of object of type `String` is referenced/built like:

```
Set<String> set = new Set<String>();
```

- Adding objects of type `String` to the set would be:

```
set.add("Hello");  
set.add("World");  
set.add("!");
```


Generic types (5)

- Declaring a variable `set` of type `Set<String>` tells the compiler that `set` is a reference to an object of type `Set<T>` where `T` is a `String`.
 - It is not created a class `Set<String>`.
 - The types `Set<String>` and `Set<Number>` are not two distinct classes.
- As for methods, in the generic types we also have the concept of parameter/argument:
 - In the context of `Set<T>`, `T` is said to be a **(type) parameter**.
 - In the context of `Set<String>`, `String` is said to be a **(type) argument**.

Generic types (6)

```
public class Set<T> {
    protected int nbElems;
    protected Element<T> head;
    private static int nbNextSet=0;
    private int nbSet;
    public Set() {
        nbSet = nbNextSet++;
    }
    public int nbSet() {
        return nbSet;
    }
    public static int nbNextSet() {
        return nbNextSet;
    }
    public void add(T elem) {...}
    public void remove (T elem) {...}
}
```

Generic types (7)

```
Set<String> set_s = new Set<String>();  
System.out.println("set_s has number " + set_s.nbSet());  
Set<Integer> set_i = new Set<Integer>();  
System.out.println("set_i has number " + set_i.nbSet());
```

In the terminal is printed

```
set_s has number 0  
set_i has number 1
```

Generic types (8)

- Consequences:
 - The type parameter T cannot be used in static contexts.
 - The access to static members of the parameterized classes cannot be done through the parameterized type:

```
Set<String>.nbNextSet(); //INVALID!!!
```

```
Set.nbNextSet();
```

Generic types (9)

- It is not possible to use the type parameter T to build objects (nor to build arrays).

```
public class Set<T> {  
    // ...  
    public T[] convertSetToArray() {  
        T[] res = new T[nbElems]; //INVALID!!!  
        //copy the elements from this set to the array  
    }  
}
```

Generic types (10)

- Solution 1: pass the array as a parameter to the method `convertSetToArray` which fills this array with the elements of the set.

```
public class Set<T> {  
    // ...  
    public T[] convertSetToArray(T[] array) {  
        int i = 0;  
        for (Element<T> aux=head; aux!=null; aux=aux.next)  
            array[i++] = aux.element;  
        return array;  
    }  
}
```

Generic types (11)

- When defining a generic type it is possible to restrict the type argument that is passed to the type parameter `T`:

```
interface OrderedCollection<T extends Comparable<T>> {...}
```

- The argument type passed to the type argument `T` implements the methods of the interface `Comparable<T>`.

```
interface CharSequenceOrderedCollection<T extends  
    Comparable<T> & CharSequence> {...}
```

- The argument type passes to the parameter type `T` implements the methods of the interface `Comparable<T>` and implements the methods of the interface `CharSequence`.

Generic types (12)

- The keyword `extends` is used in the type parameters of the generic types in a very general form:
 - It means `extends` if the type that follows is a class.
 - It means `implements` if the type that follows is an interface.

Parameterized types (1)

- The generic types might be used in the context of the declaration/instantiation of parameterized types:
 - Types of fields;
 - Types of local variables;
 - Types of method parameters;
 - Type of method return.

Parameterized types (2)

- Consider a method that sum the elements in a `Set<Number>`:

```
static double sum(Set<Number> set) {  
    double res = 0.0;  
    for (Element<Number> aux=set.head; aux!=null; aux=aux.next)  
        res+=aux.element.doubleValue();  
    return res;  
}
```

Parameterized types (3)

- If the `sum` method is invoked from a `Set<Integer>`, the code does not compile:

```
Set<Integer> set = new Set<Integer>();  
set.add(1);  
set.add(2);  
double sum = sum(set); //INVALID!!!
```

- Although `Integer` is a subtype of `Number`, `Set<Integer>` is not a subtype of `Set<Number>`.

Parameterized types (4)

- The solution is to define the parameter of method `sum` as a set of `Number` or of any subtype of `Number`:

```
static double sum(Set<? extends Number> set) {  
    double res = 0.0;  
    for (Element<? extends Number> aux=set.head;  
        aux!=null; aux=aux.next)  
        res+=aux.element.doubleValue();  
    return res;  
}
```

- The `? extends` in the type parameter refers to a `Number` or to a subtype of `Number`.

Parameterized types (5)

- It is also possible to define the type parameter as being of a certain type or of any supertype of that type.
 - `Set<? super Integer>` denotes the set of `Integer`, or of a supertype of `Integer`:
 - `Set<Integer>`
 - `Set<Number>`
 - `Set<Object>`
 - ...
- The `extends` and the `super` cannot be used simultaneously.

Parameterized types (6)

- It is also possible to define the type parameter as being of any type.
 - The `Set<?>` denotes a set of any type.
 - Implicitly, `Set<?>` refers to any type provided that it is an `Object` or any subtype of it.
 - The `Set<?>` is another way of writing `Set<? extends Object>` (and not `Set<Object>`).

Parameterized types (10)

- The ? can only be used in the context of the declaration of parameterized types:
 - Types of fields;
 - Types of local variables;
 - Types of method parameters;
 - Type of method return.

Generic methods and constructors (1)

```
public class Set<T> {  
    // ...  
    public T[] convertSetToArray(T[] array) {  
        //copy the elements of this set to the array  
        return array;  
    }  
}
```

- The method `convertSetToArray` as it is defined is **too restrictive** (see slide 11 and 12):
 - In an object of type `Set<Integer>` we need to pass as parameter an array of `Integer[]`.
 - In an object of type `Set<Integer>` we cannot pass as parameter an array of `Object[]`, even if it is valid to store an object of type `String` in such an array.

Generic methods and constructors (2)

- A **generic method** is declared by defining the type parameter between the modifiers and the return type of the method:

```
public class Set<T> {  
    //...  
    public <E> E[] convertSetToArray(E[] array) {  
        Object[] tmp = array;  
        int i = 0;  
        for (Element<T> aux=head; aux!=null; aux=aux.next)  
            tmp[i++] = aux.element;  
        return array;  
    }  
}
```

Generic methods and constructors (3)

```
Set<Integer> set = new Set<Integer>();  
Object[] array = new Object[set.nbElems];  
array = set.convertSetToArray(array);
```

```
Set<Integer> set = new Set<Integer>();  
Integer[] array = new Integer[set.nbElems];  
array = set.convertSetToArray(array);
```

Generic methods and constructors (4)

- A generic method or constructor does not need to be declared within a generic class, but if it is the type parameters are different.

Generic methods and constructors (5)

- The method `convertSetToArray` can be invoked as:

```
Set<Integer> set = new Set<Integer>();  
Integer[] array = new Integer[set.nbElems];  
array = set.<Integer>convertSetToArray(array);
```

- This **parameterized invocation** tells the compiler that the type parameter `E` of the method `convertSetToArray` must be treated as an `Integer`, and that the arguments and the return type must comply with that.

Generic methods and constructors (6)

- The parameterized invocation is rarely needed. The compiler is able, in general, to perform the **type inference**:

```
Set<Integer> set = new Set<Integer>();  
Integer[] array = new Integer[set.nbElems];  
array = set.convertSetToArray(array);
```

- The type inference is based on the **static type** of the argument passed to the generic method or constructor (and not on its dynamic type).
 - Declaration type of explicit cast.

Generic methods and constructors (7)

- The parameterized invocation must be always done through a qualified name:
 - `this.<Type>method(params) ;`
 - `super.<Type>method(params) ;`
 - `ref.<Type>method(params) ;`
 - `IdentC.<Type>method(params) ;`
to static methods, where `IdentC` is the name of the respective class.

```
String s1 = "Hello";  
String s2 = <String>passObject(s1); //INVALID!!!
```

Generic methods and constructors (8)

```
<T> T passObject(T obj) {  
    return obj;  
}
```

```
String s1 = "Hello";  
String s2 = this.<String>passObject(s1);
```

```
String s1 = "Hello";  
String s2 = passObject(s1);           // T -> String  
Object o1 = passObject(s1);           // T -> String  
Object o2 = passObject((Object)s1);   // T -> Object
```

```
String s1 = "Hello";  
s1 = passObject((Object)s1); //INVALID!!!
```

```
String s1 = "Hello";  
s1 = (String) passObject((Object)s1); // T -> Object
```

Subtypes and generic types (1)

- It is possible to:
 - Define a subtype (generic or not) from a non-generic supertype.
 - Define a subtype (generic or not) from a generic supertype.

```
class GeneralList<E> implements List<E> {...}
class StringList implements List<E> {...}
class NumberSet<T extends Number> extends Set<T> {...}
class IntegerSet extends Set<T> {...}
...
```


Subtypes and generic types (2)

- A class cannot implement two interfaces which are different parameterizations of the same interface.

```
class Value implements Comparable<Value> {...}  
class ExtendedValue extends Value  
    implements Comparable<ExtendedValue> {...} //INVALID!!!
```

- In this case, a class `ExtendedValue` should implement both interfaces `Comparable<Value>` and `Comparable<ExtendedValue>` (which is not allowed).

Object Oriented Programming

Java

Part 9: Utility classes

Introduction (1)

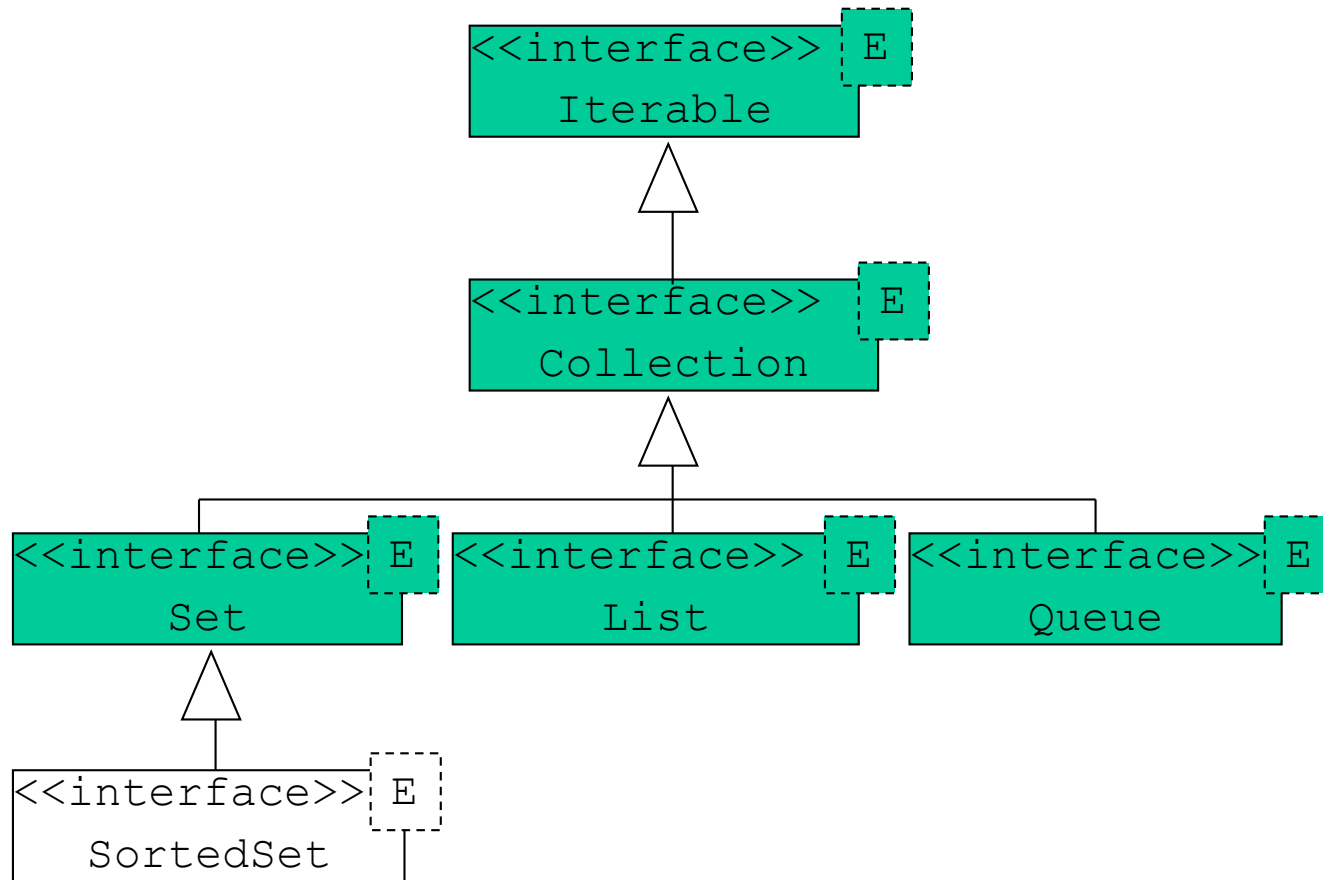
- Java provides a set of utility classes:
 - with important functionality to the programmer.
 - distributed in the development environment in different packages (inside **src.zip** file)
 - `src/java/lang` # classes of the language (`Integer`,...)
 » Automatically imported
 - `src/java/util` # diverse utilities (`Vector`,...)
 - `src/java/math` # `Math` class
 - `src/java/io` # I/O classes

Introduction (2)

- The J2SE provides several groups of interfaces. In these slides we focus 4 of them:
 1. **Comparator** and **Comparable** – describe comparison between objects (for instance, for sorting).
 2. **Collection** – describe collections of objects.
 3. **Map** – describe functions between objects.
 4. **Iterator** – describe iterations over collections of objects, without knowing the way objects are organized inside the collection.
- The code of the classes is available in:
<http://www.docjar.com>

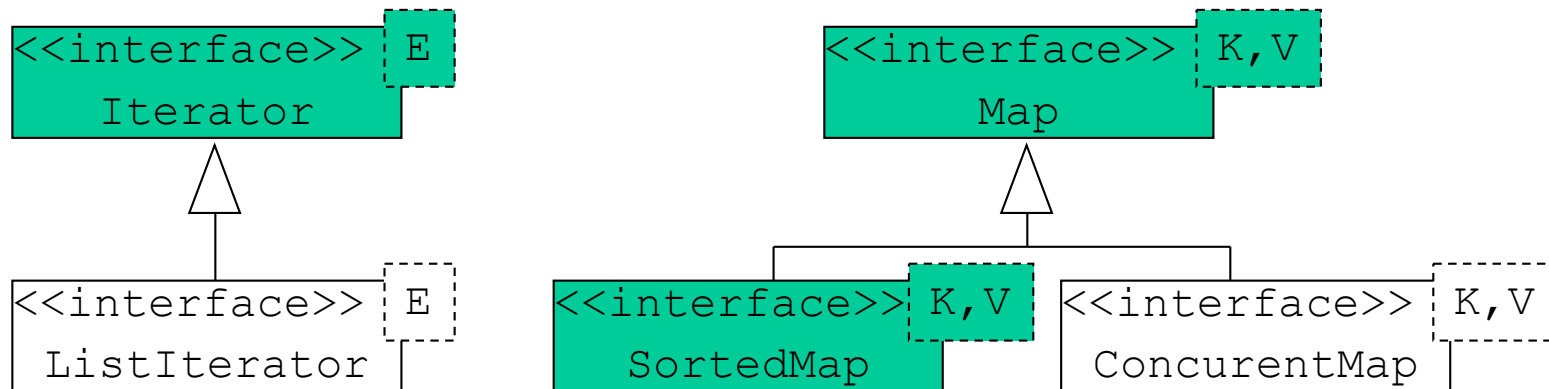
Introduction (3)

General hierarchy of interfaces for ADTs in J2SE 5



Introduction (4)

General hierarchy of interfaces for ADTs in J2SE 5



Sorting (1)

- Classes that require sorting implement one of two interfaces:
 - Comparable
 - Comparator

Comparable interface (1)

- Used when there is a **natural order** (e.g.: `Character`, `Integer`, `Date`).
- Implemented inside the class by the method **`compareTo`**, which implies **total order** inside the class.
- **Simpler to implement, but less flexible** than the `Comparator` interface.

Comparable interface (2)

```
public interface Comparable<T> {  
    public int compareTo(T other);  
}
```

- The value returned by the **compareTo** should be:
 - < 0 if this object is less than the object passed by parameter
 - = 0 if this object is equal (with **equals**) to the object received as parameter
 - > 0 otherwise

Comparable interface (3)

```
public class Account implements Comparable<Account> {
    private static long nbNextAccount = 0;
    protected long nbAccount; // account number
    protected float balance; // current balance
    //...
    public boolean equals(Object obj) {
        return nbAccount==((Conta)obj).nbAccount;
    }
    public int compareTo(Account other) {
        if (nbAccount > other.nbAccount) return 1;
        else if (nbAccount == other.nbAccount) return
0;
        else return -1;
    }
    //...
}
```

Comparable interface (4)

```
Account mc = new Account("Manuel Silva",1000);  
Account outra = new Account("Luís Silva",200);  
System.out.println(mc.compareTo(mc));  
System.out.println(mc.compareTo(outra));  
System.out.println(outra.compareTo(mc));
```

In the terminal is printed

0
1
-1

Comparable interface (5)

- Interfaces define types, so we can have:

```
Comparable<Account> cc;
```

- It is possible to define, for instance, a method to sort an array of `Comparable` objects (without knowing to which class these objects belong):

```
class Sort {  
    static Comparable<?>[] sort(Comparable<?>[] objs) {  
        // sort details ...  
        return objs;  
    }  
}
```

Comparable interface (6)

- The class `java.util.Arrays` provides a method that allows to sort objects in an `Object` array according to the natural ordering of its elements:

```
public static void sort(Object[] a,  
                        int fromIndex, int toIndex)
```

- Sort the objects in array `a` from index `fromIndex` (inclusive) to index `toIndex` (exclusive).
- All elements in `[fromIndex, toIndex]` must implement the `Comparable` interface.
- All elements in that range must be mutually comparable (that is, `obj1.compareTo(obj2)` must not throw an exception `ClassCastException`).

Comparable interface (7)

```
public class Sort {  
    static Comparable<?>[] sort(Comparable<?>[] objs) {  
        Comparable<?>[] res = new Comparable<?>[objs.length];  
        System.arraycopy(objs, 0, res, 0, objs.length);  
        java.util.Arrays.sort(res, 0, res.length);  
        return res;  
    }  
}
```

```
Account mc = new Account("Manuel Silva",1000);  
Account outra = new Account("Luís Silva",200);  
Comparable<?>[] accounts = new Comparable<?>[2];  
accounts[0]=outra;  
accounts[1]=mc;  
Accounts = Sort.sort(accounts);
```

Comparator interface (1)

- Used when there is an **application-dependent order** (e.g.: sorting a list of students of a certain course may be performed according to their number, name, or mark).
- Implemented outside the class (but it can use the `compareTo` over the class fields), realizing the `Comparator` interface.
- **More complex implementation but more powerful** than the one offered by the `Comparable` interface.

Comparator interface (2)

```
public interface Comparator<T> {  
    public int compare(T o1, T o2);  
}
```

- Value returned by **compare** must be:
 - < 0 if the object o1 is less than the object o2
 - = 0 if the object o1 is equal to the object o2
 - > 0 otherwise

Comparator interface (3)

- Despite not making sense to define a natural ordering of accounts by balance, you may need to sort accounts by balance somewhere in an application...

```
import java.util.Comparator;
public class ComparatorByBalance implements Comparator<Account> {
    public int compare(Account o1, Account o2) {
        if (o1.balance > o2.balance) return 1;
        else if (o1.balance == o2.balance) return 0;
        else return -1;
    }
}
```

Comparator interface (4)

- The class `java.util.Arrays` provides a generic method that allows to order the objects in an array according to an order induced by a `Comparator`:

```
public static <T> void sort(  
    T[] a, int fromIndex, int toIndex,  
    Comparator<? super T> c)
```

- Sorts the objects in array `a` from index `fromIndex` (inclusive) to index `toIndex` (exclusive).
- All elements in this range must be mutually comparable with the specified `Comparator` (that is, `obj1.compare(obj2)` must not throw an exception `ClassCastException`).

Comparator interface (5)

- An array of accounts could be ordered by balance in this way:

```
Account[] accounts = new Account[2];  
accounts[0] = new Account("Manuel Silva",1000);  
accounts[1] = new Account("Luís Silva",200);  
java.util.Arrays.sort(  
    accounts, 0, accounts.length,  
    new ComparatorByBalance());
```

Comparator interface (6)

- Given a list of students, the natural criterion would be to sort students by number. To impose an ordering by name:

```
import java.util.Comparator;
public class StudentComparatorByName
    implements Comparator<Student> {
    public int compare(Student o1, Student o2) {
        String name1 = o1.firstName();
        String name2 = o2.lastName();
        if (name1.equals(name2)) {
            name1 = o1.lastName();
            name2 = o2.lastName();
            return name1.compareTo(name2);
        } else return name1.compareTo(name2);
    }
}
```

Comparator interface (7)

```
public static void main(String[] args) {  
    Student[] students = new Student[args.length];  
    for(int i=args.length-1; i>=0; i--)  
        students[i] = new Student(args[i]);  
    System.out.println(students);  
    System.out.println("*** Ordered by name ***");  
    java.util.Arrays.sort(students,  
        new StudentComparatorByname());  
    System.out.println(students);  
}
```

Comparator interface (8)

- If one wants to sort the student's array according to other criteria, for instance, according to their marks, one should simply develop another implementation of `Comparator` and invoke the method `sort` from `java.util.Arrays`.

```
System.out.println("*** Ordered by mark ***");  
java.util.Arrays.sort(students,  
    new StudentsComparatorByMark());  
System.out.println(students);
```

Iterator interface

```
public interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    void remove();  
}
```

- The `Iterator` interface must be implemented by those classes that want to iterate over its elements, one by one.

Iterable interface

```
public interface Iterable<E> {  
    Iterator<E> iterator();  
}
```

- A class that implements the `Iterable` interface offers an `Iterator` which can then be used in for-each loops.

Collection interface (1)

- A **collection**, or **container**, is an object that contains diverse objects (eventually repeated) in a single unit.
- Prototypes of methods are grouped in:
 - Basic operations.
 - Bulk operations which perform an operation on the entire collection.
 - Operations that convert the collection into an array.

Collection interface (2)

```
public interface Collection<E> extends Iterable<E> {  
    // Basic operations  
    int          size();  
    boolean      isEmpty();  
    boolean      contains(Object elem);  
    boolean      add(E elem);  
    boolean      remove(Object elem);  
    Iterator<E>  iterator();  
    // Bulk operations  
    boolean      containsAll(Collection<?> coll);  
    boolean      addAll(Collection<? extends E> coll);  
    boolean      removeAll(Collection<?> coll);  
    boolean      retainAll(Collection<?> coll);  
    void         clear();  
    // Array operations  
    Object[]     toArray();  
    <T> T[]      toArray(T dest[]);  
}
```

Collection interface (3)

- **All methods that need the notion of equivalence between objects use the `equals` method**
(`contains`, `add`, `remove`, `containsAll`, `addAll`, `removeAll` e `retainAll`).
- The `Collection` interface does not make any restriction about adding `null` elements to the collection.

Collection interface (4)

- It is possible to use a for loop and the `Iterator` methods to step through the contents of a collection:
 - It is possible to `add/remove` objects to/from the collection during the iteration.
 - It possible to update the objects during the iteration.
 - It is possible to iterate over multiple collections.

```
public class RemoveShortStrings {  
    public static void remove(Collection<String> c) {  
        // remove strings with length 0 or 1  
        for (Iterator<String> i=c.iterator(); i.hasNext(); )  
            if (i.next().length()<2)  
                i.remove();  
    }  
}
```

Collection interface (5)

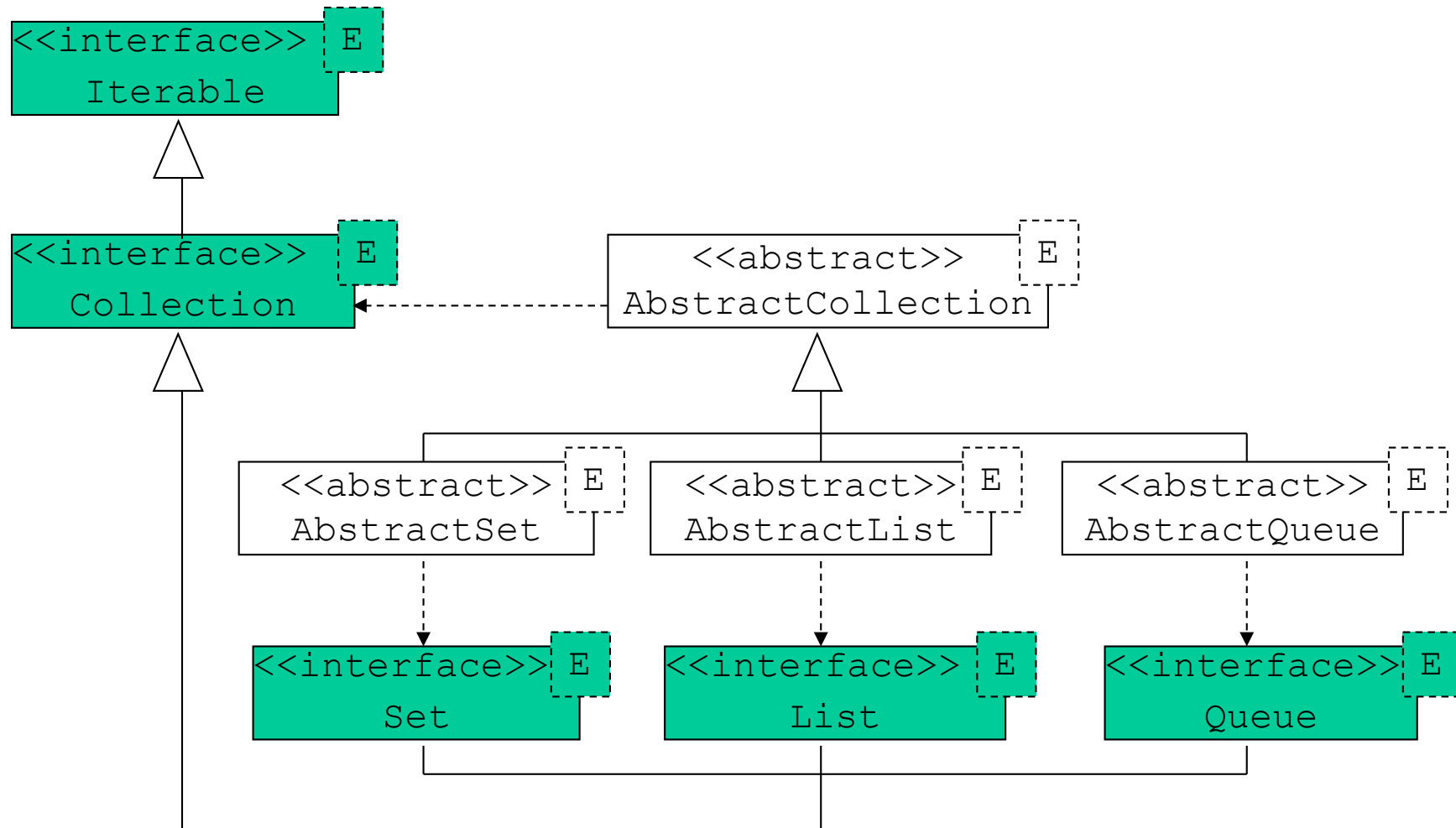
- Stepping through the elements of a collection can also be performed with a for-each loop:
 - The advantage of the for-each loop is purely syntactic.
 - It is not possible to add/remove objects to/from the collection during the iteration.
 - It possible to update the objects during the iteration.
 - It is not possible to iterate over multiple collections.

```
public class PrintShortStrings {  
    public static void print(Collection<String> c) {  
        for(String s:c)  
            if (s.length()<2)  
                System.out.println(s);  
    }  
}
```

Collection interface (6)

- From the **Collection** interface several interfaces are derived:
 - **Set**: collection without duplicate elements
 - **List**: list of elements
 - **Queue**: queue of elements

Collection interface (7)



Interfaces' implementation (1)

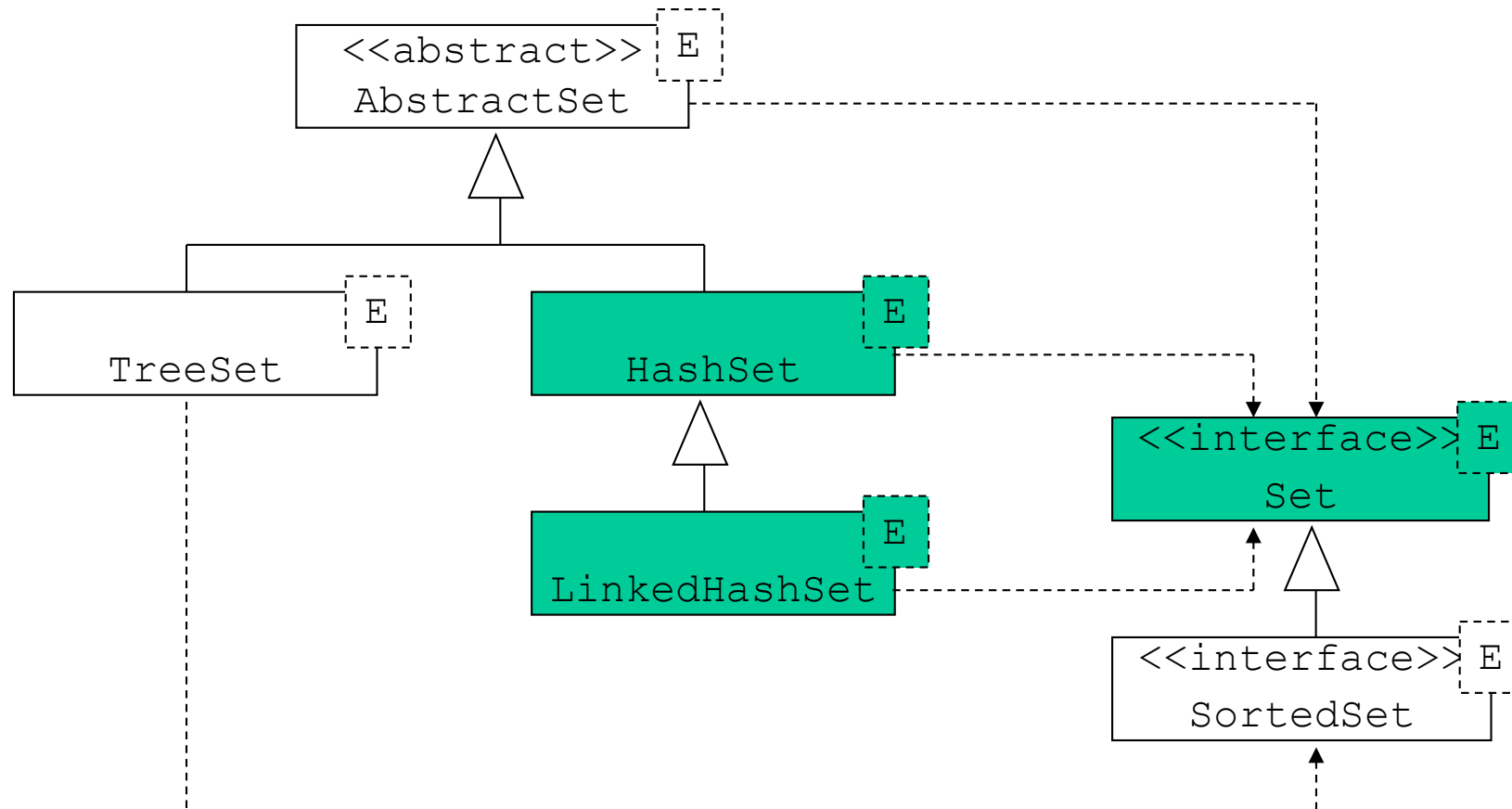
- There are diverse structures underlying data to implement interfaces:
 1. **Linear**: the objects are ordered in positions, each object has only a predecessor (except the first) and a successor (except the last).
 2. **Hierarchical**: each object has only a predecessor (except the root) and it might have a fixed number of successors.
 3. **Unsorted**: there are no relation between two objects.

Interfaces' implementation (2)

- The J2SE 5 implements the `Set/List/Map` interfaces through four data structures:
 - **Hash tables.**
 - **Variable length arrays.**
 - **Balanced tress.**
 - **Linked lists.**

Underlying data structure					
Interfaces	Hash tab.	Var. len. arrays	Bal. trees	Linked lists	Hash + linked list
Set	HashSet	---	TreeSet	---	LinkedHashSet
List	---	ArrayList	---	LinkedList	---
Map	HashMap	---	TreeMap	---	LinkedHashMap

Set interface (1)



Set interface (2)

- The `Set` interface adds no new method of its own:
 - It only provides the methods from `Collection`.
 - Extra restrictions are imposed to the `add` method in order to avoid duplicate elements in this collection.

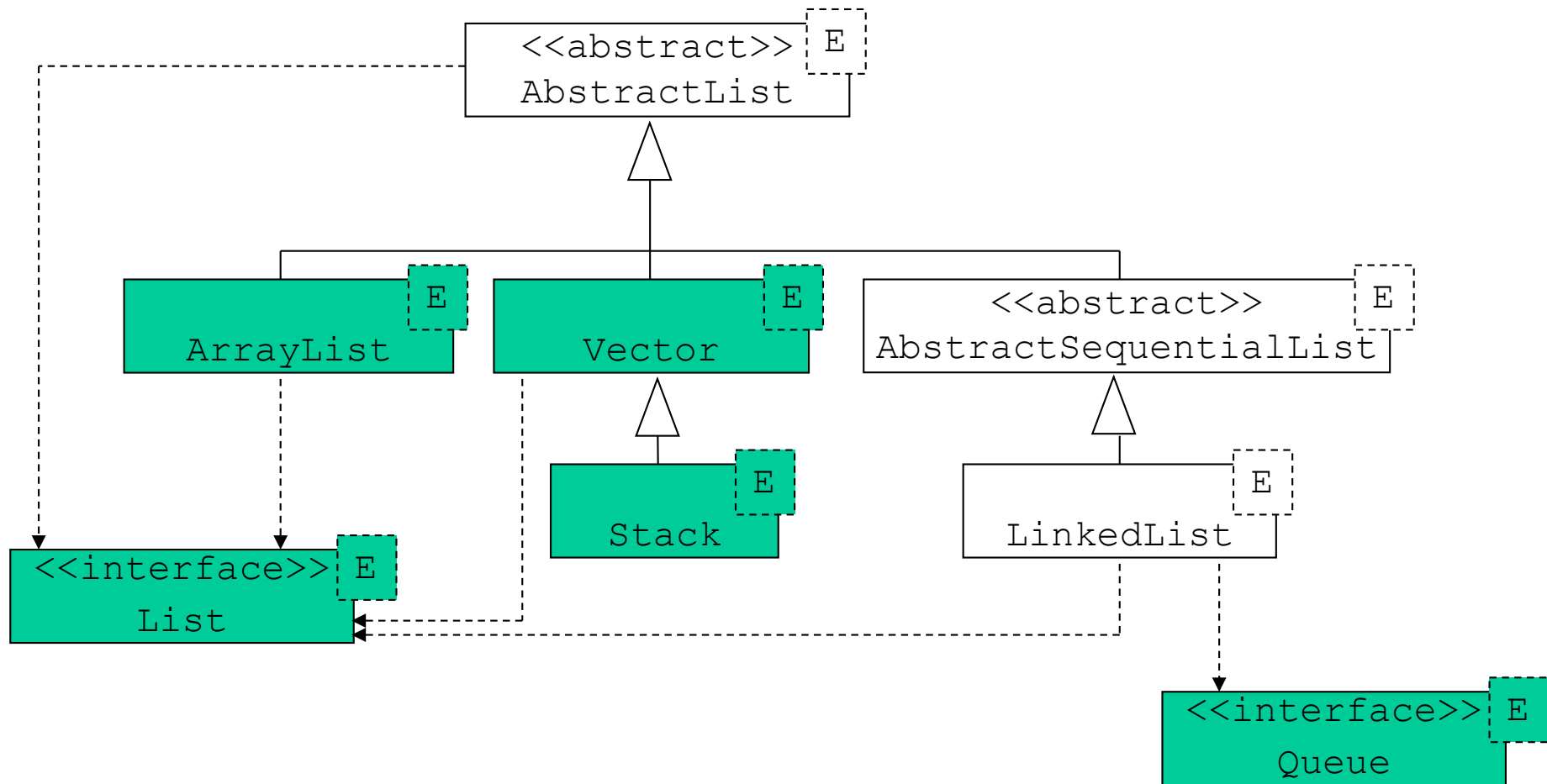
Set interface (3)

- Some implementations of the `Set` interface:
 - **HashSet**: the best for most of the uses.
 - **LinkedHashSet**: imposed order in the `Iterator` (insertion order).
 - **TreeSet**: imposed order in the `Iterator` (natural order or an order defined by a `Comparator`).
- Exposure of the implementation should be avoided:

```
Set<Integer> s = new HashSet<Integer>(); // preferable
```

```
HashSet<Integer> s = new HashSet<Integer>(); // to avoid!
```

List interface (1)



List interface (2)

```
public interface List<E> extends Collection<E> {  
    E get(int index);  
    E set(int index, E elem);  
    void add(int index, E elem);  
    E remove(int index);  
    int indexOf(Object elem);  
    int lastIndexOf(Object elem);  
    List<E> subList(int min, int max);  
    ListIterator<E> listIterator(int index);  
    ListIterator<E> listIterator();  
}
```

ArrayList class (1)

- It is an implementation of `List` that store its elements in an array:
 - The array has an initial capacity.
 - When the initial capacity is exceeded it is built a new array and its content is copied.
 - A correct value for the initial capacity of the `ArrayList` improves its performance.
- **Complexity:**
 - **Adding (in position i) and removing (in position i): $O(n-i)$ where n is the length of the list and $i < n$.**
 - **Add (in the end) and remove (from the end): $O(1)$**
 - **Accessing an element (in any position): $O(1)$**

ArrayList class (2)

```
public class ArrayList<E>
    extends AbstractList<E>
    implements List<E>, ...
{
    ArrayList() {...}
    ArrayList(int initialCapacity) {...}
    ArrayList(Collection<? extends E> coll) {...}
    void trimToSize() {...}
    void ensureCapacity(int minCapacity) {...}
}
```

- By default, the initial capacity of an `ArrayList` is 10.

LinkedList class (1)

- It is an implementation of `List` with a doubly linked list.
- The `LinkedList` class also implements the interface `Queue`.
- **Complexity:**
 - Add (in position i) and remove (from position i): $O(\min\{i, n-i\})$, where n is the length of the list.
 - Add (in the beginning or in the end) and remove (from the beginning or the end): $O(1)$.
 - Accessing an element in position i : $O(\min\{i, n-i\})$, where n is the length of the list.

LinkedList class (2)

- **From the complexity analysis one can conclude:**
 - **A `LinkedList` should be used wherever:**
 - The length of the list varies.
 - It is important to add or remove elements in arbitrary positions of the list.
 - **It is preferable to use an `ArrayList` whenever:**
 - New elements are added/removed to/from the end of the list.
 - It is important to access its elements very efficiently.

LinkedList class (3)

```
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Queue<E>, ...
{
    LinkedList() {...}
    LinkedList(Collection<? extends E> coll) {...}
    E getFirst() {...}
    E getLast() {...}
    E removeFirst() {...}
    E removeLast() {...}
    void addFirst(E elem) {...}
    void addLast(E elem) {...}
}
```

List implementations

Advantages

- Solve the drawback of the arrays' constant length.

Disadvantages

- It can only store objects (data of primitive type must be stored within wrapper classes).
- Access to arrays is more efficient.

Arrays class (1)

- The **Arrays class** is provided by the J2SE with static methods to manipulate arrays.
- The great majority of the methods has several overloads:
 - One for arrays for each primitive type.
 - One for `Object` arrays.
- There are also two variants of some methods:
 - One acting in the entire array.
 - One acting on a subarray specified by two supplied indexes.

Arrays class (2)

- The methods of the `Arrays` utility class are:
 - **`static void sort`**: sorts in ascending order, with parameters:
 1. Array to sort (mandatory)
 2. Two indexes that define the subarray (by default, coincides with the entire array)
 3. A `Comparator` object that induces an order in the array elements (by default, natural order defined by the `Comparable`)
 - **`static int binarySearch`**: binary search (the array must be sorted in ascending order), with parameters:
 1. Array where to search (mandatory)
 2. Value to search for (mandatory)
 3. A `Comparator` object that induces an order in the array elements (by default, natural order defined by the `Comparable`)

Arrays class (3)

```
Integer[] ints = new Integer[2];  
ints[0]=1;  
ints[1]=2;  
System.out.println(Arrays.binarySearch(ints,1));  
ints[0]=2;  
ints[1]=1;  
System.out.println(Arrays.binarySearch(ints,1));
```

In the terminal is printed

0
-1

Arrays class (4)

- **static void fill**: Fill the array entries, with parameters:
 1. Array to fill (mandatory)
 2. Two indexes that define the subarray (by default, coincides with the entire array)
 3. Value to insert (mandatory)
- **static boolean equals**: Test for equivalence between arrays (use `equals` on each non-`null` element of the array), with parameters:
 1. Two arrays of the same type (mandatory)
- **static boolean deepEquals**: Test for equivalence between multidimensional arrays (based on contents), with parameters:
 1. Two arrays of type `Object` (mandatory)

Arrays class (5)

- **static int hashCode**: returns the array hash code (use the `hashCode` of each non-`null` element).
- **static int deepHashCode**: returns the hash code of the `Object[]` array (based on contents, taking into account nested arrays).
- **static String toString**: returns a string that represents the textual content of the array received as parameters.
- **static String deepToString**: returns a string that represents the textual content, taking into account nested arrays, of the `Object[]` array received as parameter.
- **static <T> List<T> asList(T[] t)**: returns a `List` with the elements of the array received as parameter.
 - This method acts as bridge between arrays and collections (complements the `toArray` method in collections).

Arrays class (6)

```
Integer[][] ints = new Integer[2][5];
Arrays.fill(ints[0],0); Arrays.fill(ints[1],1);
System.out.println("ints="+Arrays.deepToString(ints));
Integer other[][] = new Integer[2][5];
Arrays.fill(other[0],0); Arrays.fill(other[1],1);
System.out.println("other="+Arrays.deepToString(other));
System.out.println(ints.hashCode()+"\t"+
    Arrays.hashCode(ints)+"\t"+
    Arrays.hashCode(ints[0])+"\t"+Arrays.hashCode(ints[1])+"\t"+
    Arrays.deepHashCode(ints));
System.out.println(other.hashCode()+"\t"+
    Arrays.hashCode(other)+"\t"+
    Arrays.hashCode(other[0])+"\t"+Arrays.hashCode(other[1])+"\t"+
    Arrays.deepHashCode(other));
```

In the terminal is printed

```
ints=[[0, 0, 0, 0, 0], [1, 1, 1, 1, 1]]
other=[[0, 0, 0, 0, 0], [1, 1, 1, 1, 1]]
16795905 922240875    28629151 29583456 917088098
12115735 676418749   28629151 29583456 917088098
```

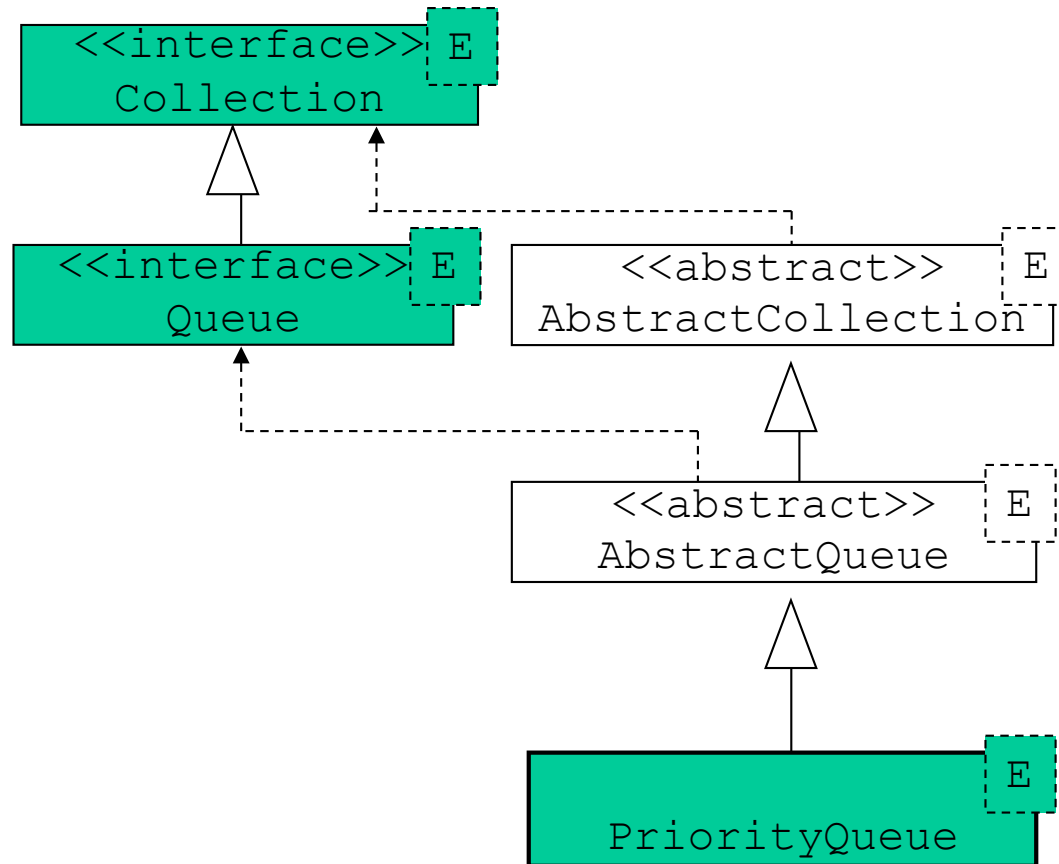
Arrays class (7)

```
List<?> list = new ArrayList<Object>();  
list.add(1);  
list.add("Hello");  
Object[] objects = new Object[2];  
objects = list.toArray();  
System.out.println(list.equals(Arrays.asList(objects)));  
System.out.println(objects.equals(list.toArray()));  
System.out.println(Arrays.equals(objects, list.toArray()));
```

In the terminal is printed

true
false
true

Queue interface (1)



Queue interface (2)

```
public interface Queue<E> extends Collection<E> {  
    E element();  
    E peek();  
    E remove();  
    E poll();  
    boolean offer(E elem);  
}
```

Queue interface (3)

- Although collections allow for `null` elements, a `Queue` must not contain `null` elements, as `null` is used in the return of the `peek` and `poll` methods to indicate that the `Queue` is empty.
- The `LinkedList` class is the simpler implementation of the `Queue` interface.
 - For historic reasons `null` elements are allowed in a `LinkedList`.
 - Inserting `null` elements in a `LinkedList` must be avoided whenever it is being used as a `Queue`.

PriorityQueue class (1)

- The `PriorityQueue` is other implementation of `Queue`.
- It is based on a priority heap.
- The head of the priority queue is the smallest element in it.
 - The smallest element is determined either by the elements' natural order or by a supplied comparator.
 - Whether the smallest element represents the element with the highest or lowest priority depends on how the natural order or the comparator is defined.

PriorityQueue class (2)

- The `PriorityQueue` iterator is not guaranteed to traverse the elements in priority order.
- But it guarantees that removing elements from the queue occurs in a given order.

PriorityQueue class (3)

- PriorityQueue constructors:

```
public PriorityQueue()  
public PriorityQueue(int initialCapacity)  
public PriorityQueue(int initialCapacity,  
                      Comparator<? super E> comp)  
public PriorityQueue(Collection<? extends E> coll)  
public PriorityQueue(SortedSet<? extends E> coll)  
public PriorityQueue(PriorityQueue<? extends E> coll)
```

- The capacity is unlimited, but the adjustment is computationally expensive.

PriorityQueue class (4)

```
public class Task {  
    String name; // identifier  
    int level;    // priority  
  
    public int level() {  
        return level;  
    }  
    public void newLevel(int value) {  
        level = value;  
    }  
    public Task(String name, int l) {  
        this.name=name;  
        level = l;  
    }  
}
```

PriorityQueue class (5)

```
private static class TaskComparator implements Comparator<Task> {  
    public int compare(Task l, Task r) {  
        return l.level() - r.level();  
    }  
}
```

PriorityQueue class (6)

```
PriorityQueue<Task> pq =  
    new PriorityQueue<Task>(10,new TaskComparator());  
Task t;  
for (char letter='A';letter<='G';letter++)  
    pq.add(new Task("Task "+letter,((letter-'A')%4));  
while (!pq.isEmpty()){  
    t=pq.poll();  
    System.out.println(t.toString()+ " priority="+t.level());  
}
```

In the terminal is printed:

```
Task A priority=0  
Task E priority=0  
Task B priority=1  
Task F priority=1  
Task C priority=2  
Task G priority=2  
Task D priority=3
```

Map interface (1)

- The `Map<K, V>` interface does not extend the `Collection` interface.
- Main characteristics of a `Map<K, V>`:
 - One does not add an element to a map, one adds a key/value pair.
 - A map allows to look up the value stored under a key.
 - A given key maps to one value or no values.
 - A value can be mapped to by many keys.
- A **map** establishes a partial function from keys to values.

Map interface (2)

- Basic methods of the `Map<K, V>` interface:

```
int size();  
boolean isEmpty();  
boolean containsKey(Object key);  
boolean containsValue(Object value);  
V get(Object key);  
V put(K key, V value);  
V remove(Object key);  
void putAll(Map<? extends K, ? extends V> otherMap);  
void clear();
```

Map interface (3)

- Some methods to see a `Map<K, V>` as a `Collection`:

```
Set<K> keySet();  
Collection<V> values();
```

- From the `Map` interface are derived other interfaces:
 - **SortedMap**: keys are ordered
 - **ConcurrentMap**

HashMap class (1)

- The `HashMap` class is an implementation of the `Map` interface by an hash table.
- It is very efficient.
 - With a well-written `hashCode` method, adding, removing or finding a key/value pair is $O(1)$.
- Constructors of the `HashMap` class:

```
public HashMap(int initialCapacity, float loadFactor)
public HashMap(int initialCapacity)
public HashMap()
public HashMap(Map<? extends K, ? extends V> map)
```


HashMap class (2)

```
import java.util.*;

String str;
Long l;
Map store = new HashMap(); // name is used as key

str = "Miguel"; l = new Long(1327);
store.put(str,l);
l = (Long) store.get(str);
if (l!=null)
    System.out.println("Codigo de "+str+"="+l.longValue());
str = "Luisa"; l = new Long(9261);
store.put(str,l);
l = (Long) store.get(str);
if (l!=null)
    System.out.println("Codigo de "+str+"="+l.longValue());
```

SortedMap interface

```
interface SortedMap<K,V> extends Map<K,V> {  
    Comparator<? super K> comparator();  
    K firstKey();  
    K lastKey();  
    SortedMap<K,V> subMap(K minKey, K maxKey);  
    SortedMap<K,V> headMap(K maxKey);  
    SortedMap<K,V> tailMap(K minKey);  
}
```

TreeMap class

- The `TreeMap` class is an implementation of the `SortedMap` interface by a binary balanced tree.
 - The access is not so efficient as with `HashMap`.
 - Adding, removing or finding a key/value pair is $O(\log n)$.
 - But the elements are always ordered.
- In the `HashMap` example just replace the declaration:

```
Map store = new TreeMap();
```

Object Oriented Programming

Java

Part 10: Packages and exceptions

Packages – revision (1)

- Packages are a mechanism to group information:
 - Packages may contain other packages, classes, interfaces and objects.
 - The package defines a **namespace**, so its members need to have unique identifiers (for instance, in a package it cannot exist two classes with the same name).
 - The identifier of a package may consist in a **simple name** or in a **qualified name**. The qualified name corresponds to the simple name prefixed with the name of the package where it resides, if it exists. It is common to use `::` to split the simple names.

Packages – revision (2)

- An **import** adds the content of the imported package to the namespace of the importing package, so that the members of the imported package need not to be used by their qualified name.
- Importing is not transitive.
 - If package B imports package A, and package C imports package B, in package C the members of A are not imported.
 - If package C also wants to import the members of A, two imports are needed, one to import package A and other to import package B.
- The set of methods of a package is referred as **API** (*Application Programmer Interface*).

Packages (1)

- Packages are useful for several reasons:
 - Group related interfaces and classes in the same package (which can then be stored in a jar file, together with the MANIFEST.MF describing the package).
 - Create a namespace that help avoid naming conflict between types defined inside and outside the package (possible use of popular names, e.g., List).
 - Provide a protected domain for application development (code within a package can cooperate using access to members of the classes and interfaces of the package that are unavailable to external code).

Packages (2)

- A class is inserted in a package with

```
package IdPacote;
```

- A `package` declaration must appear first in a source file, before any class or interface declaration (and before any `import`).
- Only one `package` declaration can appear in a source file.
- The package name is implicitly prefixed to each type name contained within the package.

Packages (3)

- If a type is not declared as being part of an explicit package, it is placed in an **unnamed package** (ease the implementation of small programs).

Packages (4)

- Type import are preformed as:

```
import IdPacote[IdSubPacote]*.(*|IdTipo);
```

- **Type import on demand:**

```
import java.util.*;
```

The * imports all public types in the corresponding package.

- **Single type import:**

```
import java.util.Set;
```

Packages (5)

- The `import` statement should be used after the declaration of the `package`, but before the declaration of the type.
- The package `java.lang` is automatically imported by Java (subpackage `lang` of the package `java`).
 - The separator “.” in Java corresponds to the “/” in Unix and “\” in Windows.

Packages (6)

- Code in a package, which needs types defined outside that package, has two options:
 - Use the qualified name of the type.
 - Import part or all the package.

```
package xpto;  
public class Xpto {  
    java.util.Set<String> strings;  
    //...  
}
```

```
package xpto;  
import java.util.Set;  
public class Xpto {  
    Set<String> strings;  
    //...  
}
```

Packages (8)

- There are only two options of visibility for classes and interfaces (non-nested ones) in a package: package and **public**.
 - A public class or interface is available on code outside the package.
 - By default a class or interface is accessible only in code within the same package.
 - The types are hidden out of the package.
 - The types are hidden for subpackages.

Packages (9)

- By default, a member of a class is visible within the corresponding package, and only inside this.
- Members of a class not declared **private** in a package are visible throughout the package.
- All members of an interface are implicitly **public**.

Packages (10)

- A method can only be redefined in a subclass if it is accessible (from the superclass).
- When a method is invoked, the runtime system must consider the accessibility of the method to decide which implementation to use ...

Packages (11)

```
package p1;

public abstract class AbstractSuperClass {
    private void pri() {print("AbstractSuperClass.pri()");}
    void pac() {print("AbstractSuperClass.pac()");}
    protected void pro() {print("AbstractSuperClass.pro()");}
    public void pub() {print("AbstractSuperClass.pub()");}
    public final void print() {
        pri();
        pac();
        pro();
        pub();
    }
}
```


Packages (12)

```
package p2;  
import p1.AbstractSuperClass;  
  
public class SubClass1 extends AbstractSuperClass {  
    public void pri() {print("SubClass1.pri()");}  
    public void pac() {print("SubClass1.pac()");}  
    public void pro() {print("SubClass1.pro()");}  
    public void pub() {print("SubClass1.pub()");}  
}
```

Invoking

```
new SubClass1().print();
```

prints in the terminal

```
AbstractSuperClass.pri()  
AbstractSuperClass.pac()  
SubClass1.pro()  
SubClass1.pub()
```

Packages (13)

```
package p1;  
import p2.SubClassa1;  
  
public class SubClass2 extends SubClass1 {  
    public void pri() {print("SubClass2.pri()");}  
    public void pac() {print("SubClass2.pac()");}  
    public void pro() {print("SubClass2.pro()");}  
    public void pub() {print("SubClass2.pub()");}  
}
```

Invoking

```
new SubClass2().print();
```

prints in the terminal

```
AbstractSuperClass.pri()  
SubClasse2.pac()  
SubClasse2.pro()  
SubClasse2.pub()
```

Packages (14)

```
package p3;
import p1.SubClass2;

public class SubClass3 extends SubClass2 {
    public void pri() {print("SubClass3.pri()");}
    public void pac() {print("SubClass3.pac()");}
    public void pro() {print("SubClass3.pro()");}
    public void pub() {print("SubClass3.pub()");}
}
```

Invoking

```
new SubClass3().print();
```

prints in the terminal

```
AbstractSuperClass.pri()
SubClass3.pac()
SubClass3.pro()
SubClass3.pub()
```

Exceptions – revision (1)

- Frequently, applications are subject to many kind of errors:
 - Mathematic errors (for instance, divide by 0 arithmetic).
 - Invalid data format (for instance, integer with invalid characters).
 - Attempt to access a null reference.
 - Open an unexisting file.
 - ...

Exceptions – revision (2)

- An **exception** is a signal that is thrown when an unexpected error is encountered.
 - The signal is synchronous if it occurs directly as a consequence of a particular user instruction.
 - Otherwise, it is asynchronous.
- Exceptions can be handled in different ways:
 - Terminating abruptly execution, with warning messages and printing useful information (unacceptable in critical systems).
 - Being managed in specific places, denominated **handlers**.

Handling exceptions (1)

- In Java, exceptions are objects of type **Exception**, which extends **Throwable**.
 - Exceptions are primarily **checked exceptions**, meaning that the compiler checks the exceptions a method declared to throw.
 - Checked exceptions represent conditions that, although exceptional, can reasonably be expected to occur.
 - The standard runtime exceptions and errors extend one of the classes **RuntimeException** and **Error**, making them **unchecked exceptions**.
 - Unchecked runtime exceptions represent conditions that generally reflect errors in the program's logic and cannot be reasonably recovered from at run time (e.g. **NullPointerException**, **ArrayIndexOutOfBoundsException**).

Handling exceptions (2)

- Exceptions are catch by enclosing code in **try** blocks with either at least one **catch** clause or the **finally** clause.
- The basic syntax is:

```
try {  
    statements:  
    if (test) throw new MyException(String);  
} catch (IdException01 e) {  
    statements  
} catch (IdException2 e) {  
    statements  
//... As many catches as needed  
} finally {  
    statements  
}
```

Handling exceptions (3)

- A `try` clause must have at least one `catch`, or a `finally`.
- Any number of `catch` clauses, including zero, can be associated with a particular `try` as long as each clause catches a different type of exception.
- **The body of the `try` statement is executed until either an exception is thrown or the body finishes successfully.**

Handling exceptions (4)

- If an exception is thrown, each `catch` clause is examined in turn, from first to last, to see whether the type of the exception object is assignable to the type declared in the `catch` .
 - When an assignable `catch` is found, its block is executed. No other `catch` clause is executed.
 - If no appropriate `catch` is found, the exception percolated out of the `try` statement into any outer `try` that might have a `catch` clause to handle it.
 - If this clause is never found the program ends abruptly.

Handling exceptions (5)

- **It is not possible to put a superclass catch clause before a catch of one of its subclasses.**
 - The first clause would always catch the exception, and the second clause would never be reached.
 - This is a compile-time error.
- **If a try has the finally clause, its code is executed after the code in the try is completed** (even if an unexpected exception occurs, or a `return` or `break` is executed).

Handling exceptions (6)

- Only one exception is handled in a `try` clause. If a `catch` or `finally` clause throw other exception, the `catch` clauses of the `try` are not reexamined.
 - The `catch` and `finally` clauses are out of the protection of the respective `try` clause.
 - Such exceptions are passed from method to method in the program stack and they can (or cannot) be handled in a `try` clause of one of those methods.

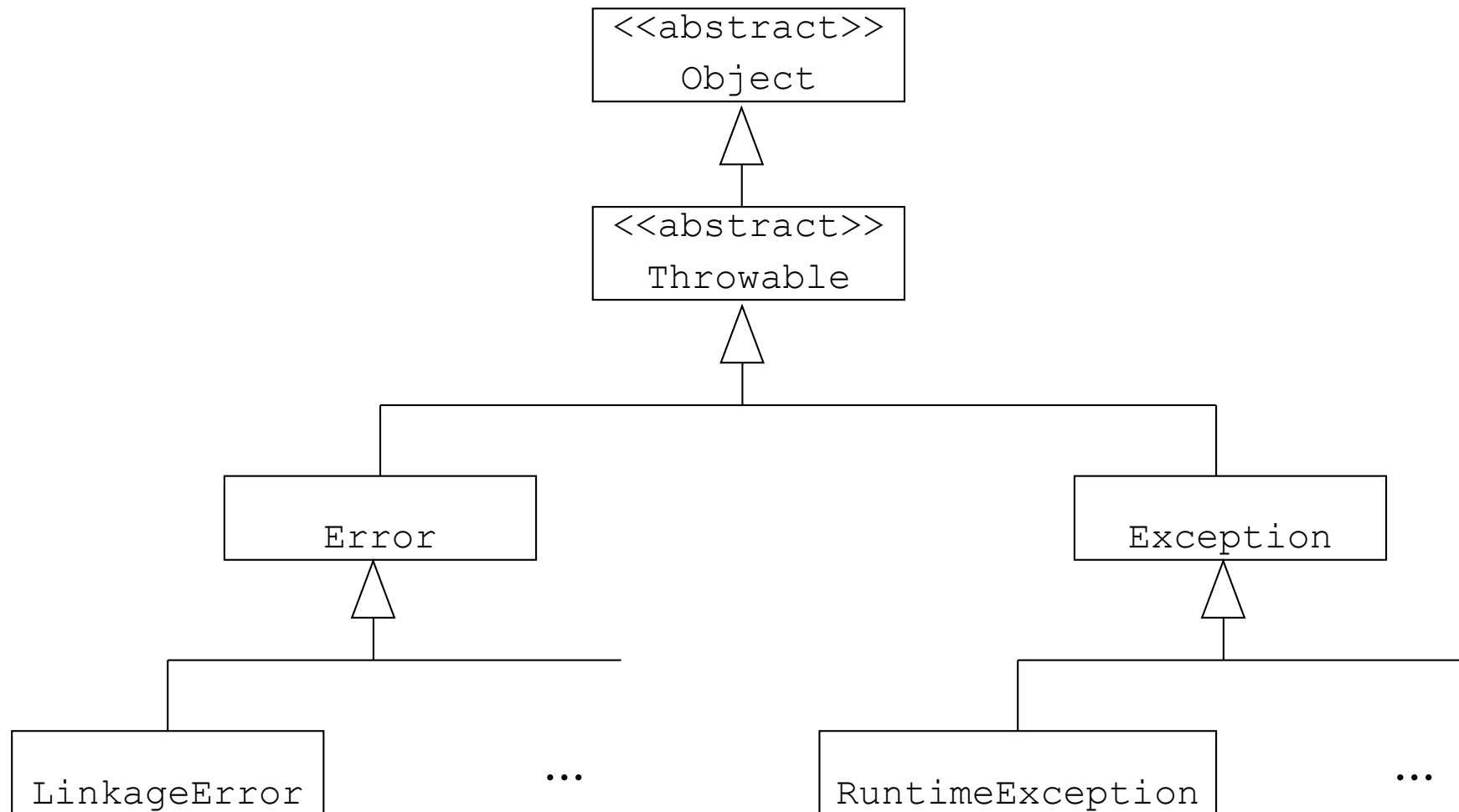
Handling exceptions (7)

- Usually, in the exception handlers:
 1. The error message is saved in a log.
 2. The object state is recovered.
 3. The method from where the exception was thrown is invoked.

Handling exceptions (8)

- If the program throws an exception, and in the code there is no `catch` clause to handle it, the JVM:
 1. Aborts the execution of the application.
 2. Prints in the `System.err` the exception thrown and the program stack.

Exception type hierarchy



Throwable class (1)

- Superclass of all errors and exceptions.
- Only objects of this class, or its subclasses, can be thrown in a `throw` statement and used as arguments of a `catch` clause.

Throwable class (2)

- Constructor:

Throwable()

Builds a new Throwable without detailed message.

Throwable(String message)

Builds a new Throwable with this detailed message.

Throwable(String message, Throwable cause)

Builds a new Throwable with this detailed message and cause.

Throwable(Throwable cause)

Builds a new Throwable with this cause and detailed message
(cause==null ? null : cause.toString()).

Throwable class (3)

- Some methods:

Throwable initCause(Throwable cause)

Initialize the cause of this `Throwable` with the cause received as parameter. This method can be invoked only once, usually is invoked directly in the constructor, or immediately after its call. If this `Throwable` was built with `Throwable(Throwable)` or `Throwable(String, Throwable)` this method cannot be invoked.

Throwable getCause()

Returns the cause of this `Throwable`, or `null`.

String getMessage()

Returns the associated detailed message.

void printStackTrace()

Prints in the `System.err` the program stack.

Error class

- Used to throw exceptions on failure of the JVM.
- Usually are not handled by the programmer.

Exception class (1)

- J2SE provides 55 subclasses:
 - `ClassNotFoundException`
 - `IOException` (contains 21 subclasses)
 - ...
- Programmer exceptions are subclasses of `Exception`.

Exception class (2)

- Constructors:

Exception()

Builds a new `Exception` without detailed message.

Exception(String message)

Builds a new `Exception` with this detailed message.

Exception(String message, Throwable cause)

Builds a new `Exception` with this detailed message and cause.

Exception(Throwable cause)

Builds a new `Exception` with this cause and detailed message
(`cause==null ? null : cause.toString()`).

- The `Exception` class does not provide new methods.

Exception chaining (1)

- Usually it is of interest the exception to be handled by the object that invoked the method (usually the recovery depends on the object that calls it).
- The method, where the exception may be thrown, should indicate in the header **throws IdException**
- In the example of division by zero, the method divide should be updated to:

```
public int divide(int op2) throws DivisionByZero{  
    if (op2==0) throw new DivisionByZero();  
    return op1/op2;  
}
```

Exception chaining (2)

```
public static void main(String args[]) {  
    if (args.length!=1) {  
        System.out.println("Only one number!");  
        System.out.exit(0);  
    }  
    int result;  
    Divide d = new Divide();  
    try {  
        result = d.divide(Integer.parseInt(args[0],10));  
        System.out.println(d.op1()+"/"+args[0]+"="+result);  
        System.exit(0);  
    } catch (DivisionByZero e) {  
        System.out.println(e);  
        System.exit(1);  
    }  
}
```