Alma Mater Studiorum-Università di Bologna
Scuola di Ingegneria

Package e Spazi di nomi

Corso di Laurea in Ingegneria Informatica

Anno accademico 2021/2022

Prof. ENRICO DENTI

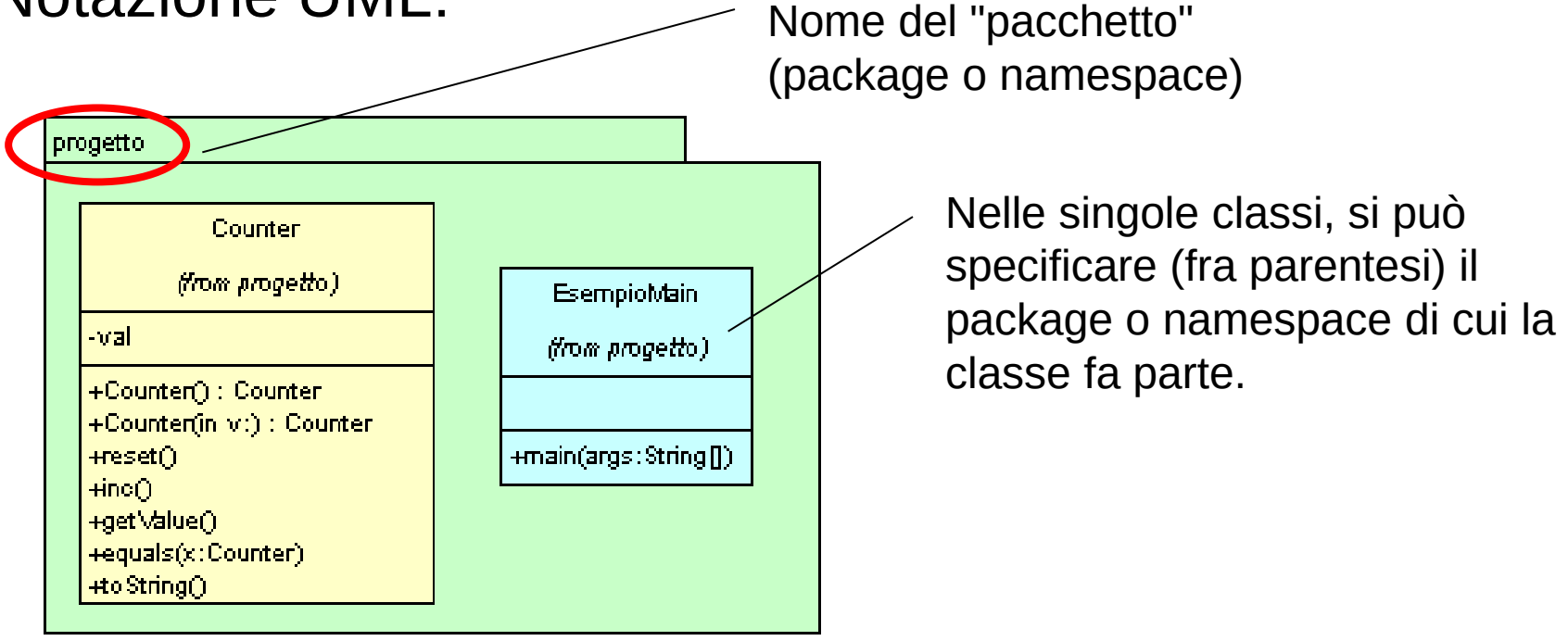
Dipartimento di Informatica – Scienza e Ingegneria (DISI)

STRUTTURAZIONE DI APPLICAZIONI

- Una applicazione complessa è tipicamente composta di *molte classi* e *librerie*
  - rischio di *conflitti di nome (name clash)*
  - necessità di *caratterizzare* gruppi di classi che costituiscono concettualmente un *"pacchetto software"*
- Necessità di uno *spazio di nomi strutturato*
  - ingestibilità di un insieme "piatto" di nomi
  - stesso problema dei nomi di file in un file system
- Costrutto *package* in Java & co. / *namespace* in C#
  - da Java 9, ulteriore costrutto *modulo*

PACKAGE e NAMESPACE

- I costrutti *package* (Java, Scala, Kotlin) e *namespace* (C#) nascono per permettere di definire e delimitare un *pacchetto software fatto di più classi*
- Notazione UML:



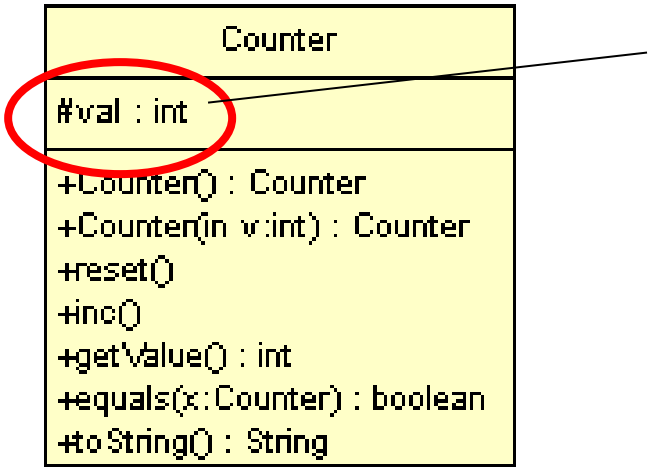
PACCHETTI SOFTWARE e VISIBILITÀ (1/2)

- FINORA abbiamo sempre distinto fra:
  - entità pubbliche, visibili a tutti
  - entità private, visibili solo entro la classe

  che è utile, ma.. molto *"bianco o nero", "all or nothing".*
- L'esperienza indica che possono essere utili *intelligenti vie di mezzo*, opportune *"gradazioni di grigio"* fra "tutto bianco" (tutto pubblico) e "tutto nero" (tutto privato).
- L'idea di *pacchetto software* suggerisce in modo naturale una di queste «vie di mezzo»
  - le classi di uno stesso pacchetto, *progettate intenzionalmente* per funzionare insieme, potrebbero utilmente beneficiare di un *grado di visibilità specifico* per la loro specifica situazione.

PACCHETTI SOFTWARE e VISIBILITÀ (2/2)

- Per questo, si introduce un *livello di visibilità intermedio* fra pubblico e privato: la *visibilità di package/namespace*
- I livelli di visibilità diventano quindi tre:
  - entità pubbliche, visibili a tutti → **`public`**
  - entità visibili a tutte le classi del pacchetto → **???**
  - entità private, visibili solo entro la classe → **`private`**



In UML, il qualificatore # indica un qualsiasi livello di *visibilità intermedio fra pubblico e privato*
(non solo la visibilità di package: qualunque gradazione di "grigio")

PACKAGE E VISIBILITÀ

- I diversi linguaggi fanno scelte diverse su quale sia il livello predefinito di visibilità per classi, oggetti, metodi
- In Java, il livello predefinito è la *visibilità di package*
  - non esiste una keyword per indicarla: è semplicemente il default
  - equivale a **`private`** per classi definite in altri package
  - equivale a **`public`** per classi definite nello stesso package
- In C#, invece, il livello predefinito è **`private`**
  - l'equivalente alla visibilità di package di Java è la *visibilità di assembly* indicata dalla keyword **`internal`**
- In Scala e Kotlin, infine, il livello predefinito è **`public`**
  - è il livello più usato, quindi renderlo il default implica «less typing»
  - la visibilità di package è espressa dalla keyword **`internal`**7

PACKAGE & FILE SYSTEM

- A differenza del C, il file rimane quindi *solo un contenitore fisico*, non definisce più uno scope di visibilità
  - GIUSTO: in un design pulito, un aspetto linguistico (la visibilità) deve avere una soluzione *nel linguaggio*, non all'esterno di esso
- Non ha quindi senso pensare di definire una classe o funzione visibile solo *in un certo «file»*, appunto perché il file *non è un costrutto linguistico*
  - ci si dovrà chiedere invece in quale package debba essere posta e da chi debba essere visibile o usata (dipendenze)

Spazi di nomi strutturati: cosa sono e come si usano

SPAZI DI NOMI STRUTTURATI (1)

- Un package introduce uno *spazio di nomi strutturato,* che può comprendere classi definite su file separati
- In Java, Scala, Kotlin, i nomi di package sono *minuscoli*
  - ESEMPI: **`matrix`**, **`edenti`**, etc
- In C#, i namespace hanno *iniziale maiuscola*
  - ESEMPI: **`System`**, **`MyCompany`**, etc
- Cosa significa *spazio di nomi strutturato?*
  - significa che per referenziare una entità in esso contenuta si deve usare il suo *nome assoluto (strutturato)*, non il solo nome *relativo*
  - ESEMPIO: la classe **`Book`** *(nome relativo)* del package **`edenti`** ha come *nome assoluto* **`edenti.Book`**

SPAZI DI NOMI STRUTTURATI (2)

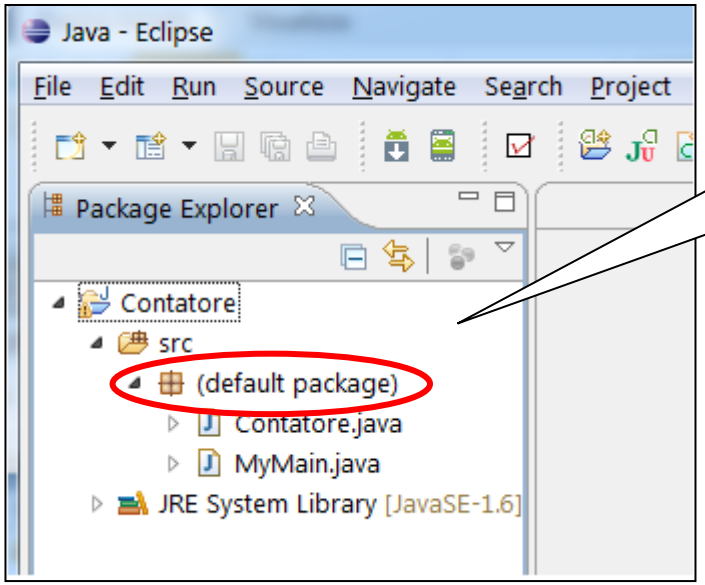
- Per permettere la definizione di nomi di package *unici a livello mondiale*
  - altrimenti, sai quante "utilities"..

  il nome strutturato può essere *multi-livello*
  - così si *evidenzia anche la provenienza (e l'azienda)*
- Inutile reinventare la ruota: c'è Internet, *e guarda caso i nomi dei domini sono già unici → si riusano rovesciati*
  - UniBo, che possiede il dominio **`unibo.it`**, potrebbe usare come prefisso naturale dei suoi nomi strutturati **`it.unibo`**
  - un package **`utilities`** di UniBo avrebbe quindi come nome strutturato multi-livello **`it.unibo.utilities`**
  - una classe **`Point`** in esso avrebbe perciò come nome assoluto **`it.unibo.utilities.Point`**

SPAZI DI NOMI STRUTTURATI (3)

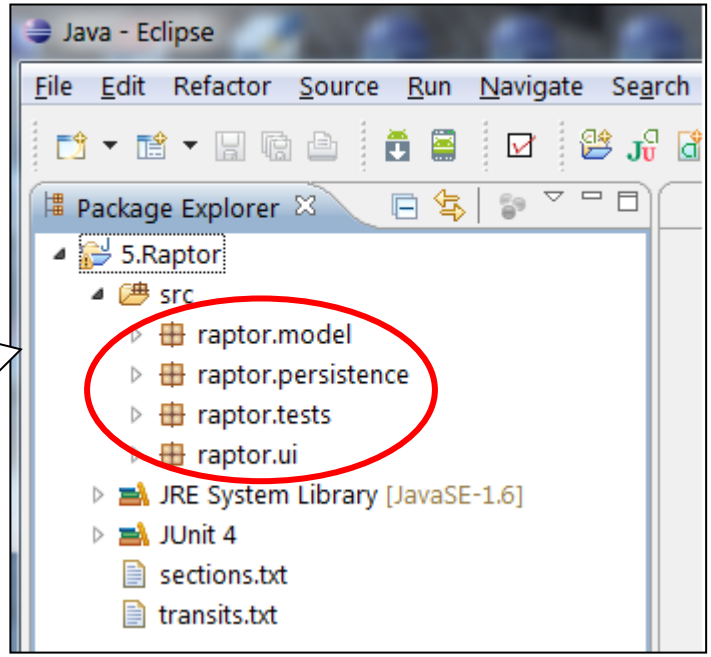
- Per riferirsi a una entità (classe, funzione, oggetto) di un certo package occorre quindi scriverne il nome *per esteso*
  - ESEMPI:
```
it.unibo.utilities.Point p;
p = new it.unibo.utilities.Point(x,y);
```
- Se non si specifica alcun nome di package/namespace, le entità vengono assegnate al *default package / namespace*
  - è il caso delle classi che abbiamo definito fino ad oggi
  - il default package va *assolutamente evitato in pratica*, perché le sue entità *non hanno nome assoluto:* di conseguenza, è *impossibile usarle da un altro package/namespace*, perché sono «innominabili»
    - NB: Scala offre l'identificatore `_root_` per riferirsi al top-level package, ma tale concetto *non ha nulla a che vedere* col default package: serve solo a disambiguare omonimie in casi molto particolari.

PACKAGE in ECLIPSE



Progetto organizzato sul *solo package di default*

Progetto organizzato su *quattro package* (non usa il default namespace)



IMPORTAZIONE DI NOMI (1/2)

- Però, i nomi strutturati (molto lunghi) sono *scomodi se la classe è usata spesso.*
- Si rimedia *importando i nomi pubblici* di un dato package o namespace nell'applicazione corrente
  - in Java, Scala, Kotlin: direttiva **`import`**
  - in C#: direttiva **`using`**
- Ciò permette di *scrivere il nome relativo (corto)* della classe *invece del nome completo (lungo)*
  - ovviamente, può fare solo se non ci sono omonimie
  - NB: la classe da importare *non può appartenere al default package* perché, dato che esso non ha nome, le sue classi *sono "innominabili" dall'esterno di esso* e quindi non sono importabili altrove.

IMPORTAZIONE DI NOMI (2/2)

- ESEMPI:
  - in Java: `import it.unibo.utilities.*;`
  - in C#: `using it.unibo.Utilities;`

  Se serve una sola classe, si può importare solo quella:
  - in Java: `import it.unibo.utilities.Point;`
  - in C#: `using UPoint = it.unibo.Utilities.Point;`
- In caso di omonimie:
  - Java permette una sola import: per l'altra classe si richiede il nome assoluto. Ad esempio, se si importa `java.awt.Point`, per riferirsi a `it.unibo.utilities.Point` occorrerà scriverla per esteso.
  - C#, Scala e Kotlin permettono invece di specificare un *alias*

    C#: `using UPoint = it.unibo.Utilities.Point;`
    Scala: `import it.unibo.Utilities.{Point => UPoint}`
    Kotlin: `import it.unibo.Utilities.Point as UPoint`

COMPILAZIONE & ESECUZIONE

- Alcune domande:
  - come si riflettono questi nomi nel file system?
  - come si invoca il compilatore ?
  - come si attiva l'esecuzione ?
- Package naming & file system structure
  - in Java bisogna seguire alcune *ben precise regole,* che estendono quelle sui nomi delle classi e dei file
  - in C# non ci sono regole particolari, valgono le regole generali sulla creazione di EXE e DLL in .NET
  - In Scala e Kotlin è fortemente raccomandato seguire *regole analoghe a quelle di Java* (anche per interoperabilità..)

IL COSTRUTTO PACKAGE in Java, Scala e Kotlin (1)

- In Java, Scala* e Kotlin, un *package* si dichiara scrivendo

  `package` *<nomepackage> ;*

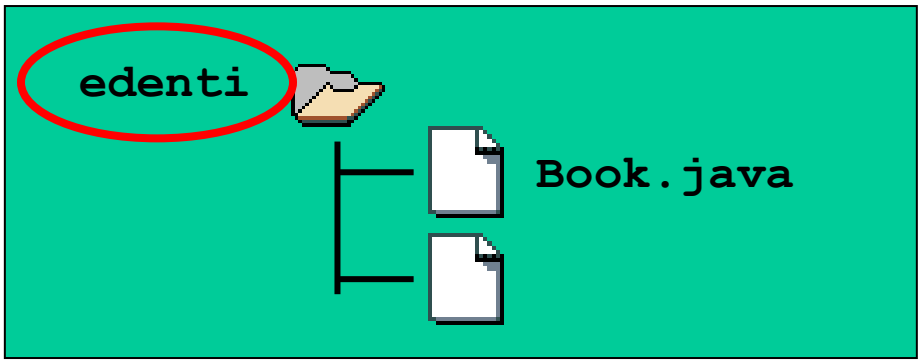
  – se presente, tale dichiarazione dev'essere *all'inizio* di un file

  – (*) Scala ammette anche una sintassi alternativa (v. oltre)

- Java pretende una *corrispondenza obbligatoria* fra:
  – nome del package (anche multi-livello)
  – nome e percorso della cartella in cui porre le classi.

Ad esempio, al *package* **`edenti`** deve corrispondere una *cartella* di nome **`edenti`**.

La classe **`Book`** di tale package deve *trovarsi fisicamente in essa.*

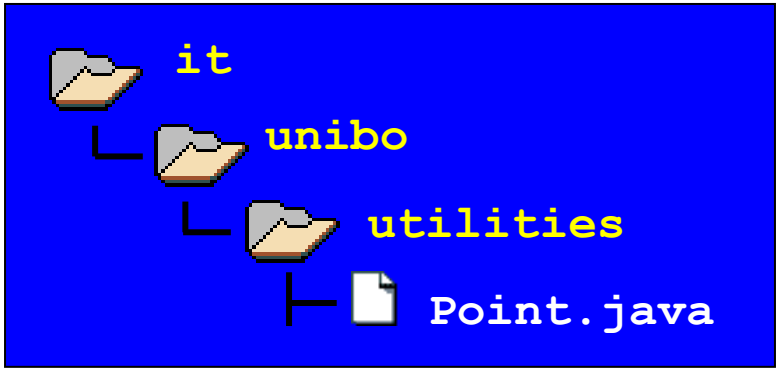


IL COSTRUTTO PACKAGE in Java, Scala e Kotlin (2)

- Regola aurea: ai nomi *multi-livello*, come

  `it.unibo.utilities`

  deve corrispondere una *struttura di cartelle innestate* con gli stessi nomi



UN PRIMO ESEMPIO (1/4)

La dichiarazione del package **`edenti`** deve apparire *all'inizio di ogni file* che contenga classi di quel package:

```
package edenti;
public class Book {
   ...
}
```

File **`Book.java`** nella cartella **`edenti`**

- questa classe ha come *nome assoluto strutturato* il nome **`edenti.Book`**
- eventuali altre classi (non pubbliche) contenute nello stesso file faranno parte dello stesso package
- Scala e Kotlin: sintassi analoga

UN PRIMO ESEMPIO (2/4)

Un possibile cliente (non appartenente a quel package):

```
public class MyClient {
   public static void main(String args[]){
      edenti.Book c = new edenti.Book(...);
   }
}
```

nome assoluto

File **MyClient.java**

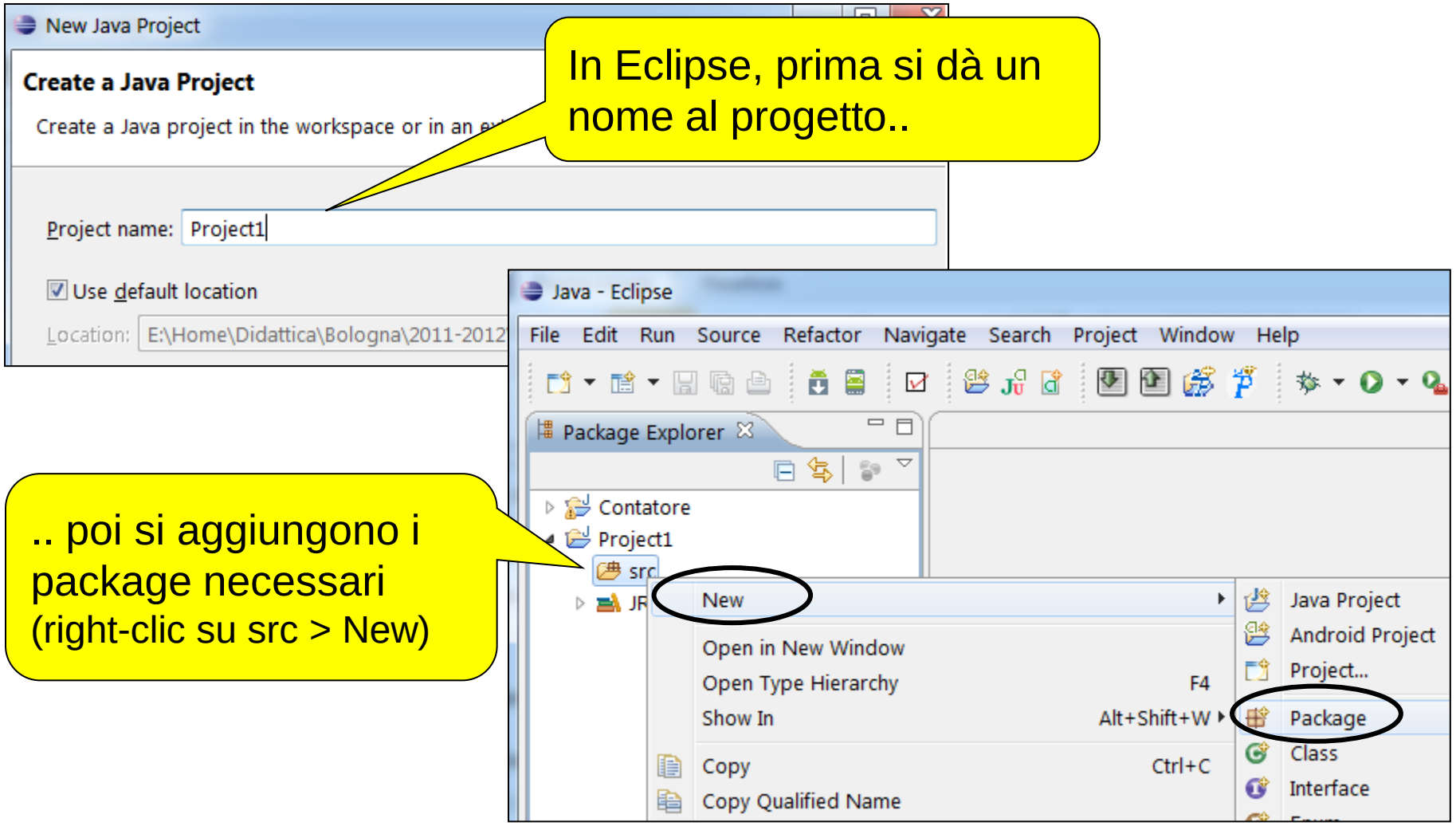
In alternativa, il cliente potrebbe importare il package:

```
import edenti.*;
public class MyClient {
   public static void main(String args[]){
      Book c = new Book(...);
   }
}
```

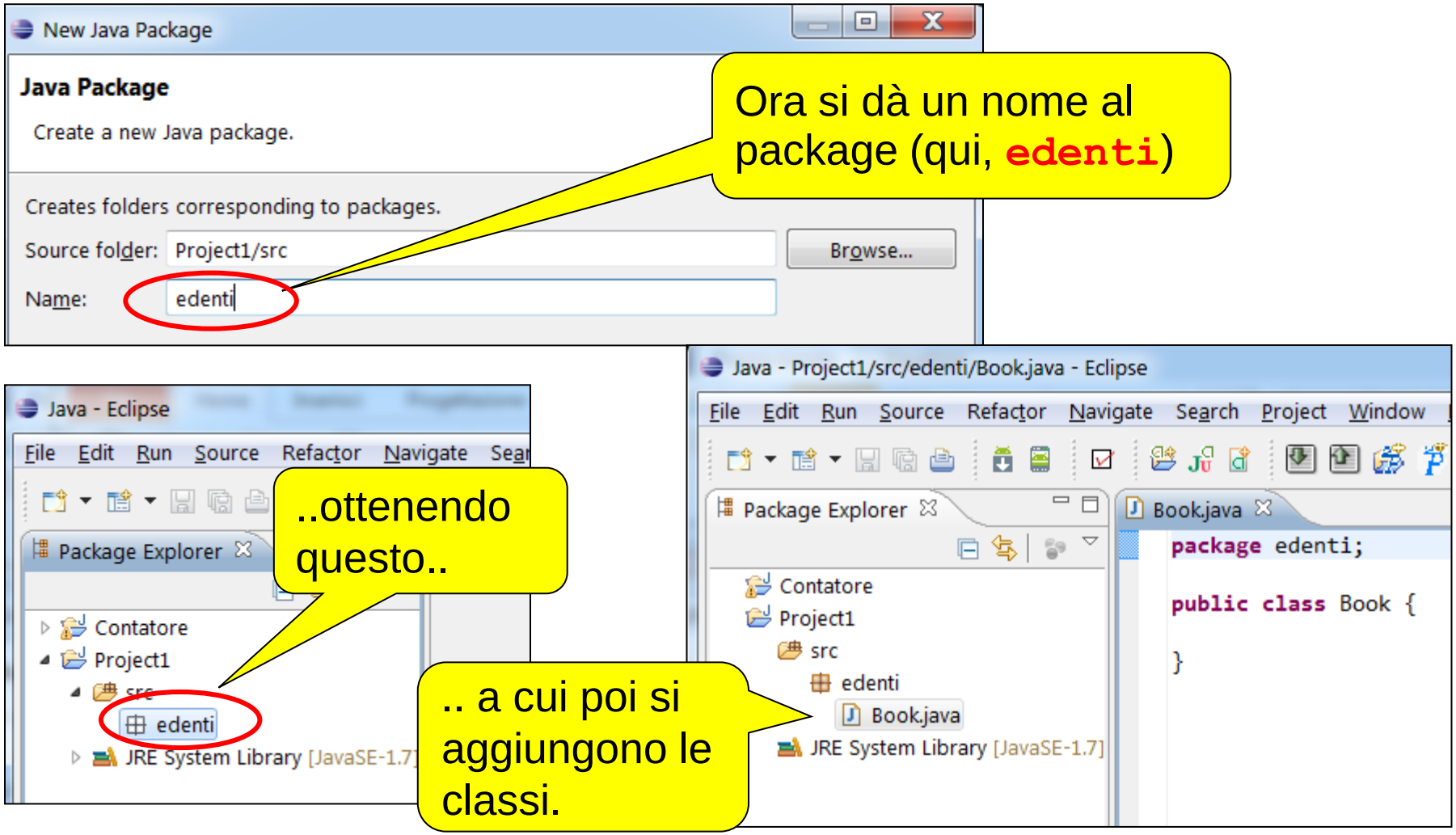
nome relativo perché importato

File **MyClient.java**

UN PRIMO ESEMPIO (3/4)



UN PRIMO ESEMPIO (4/4)



UN SECONDO ESEMPIO

Un cliente però può anch'essere *interno* al package:

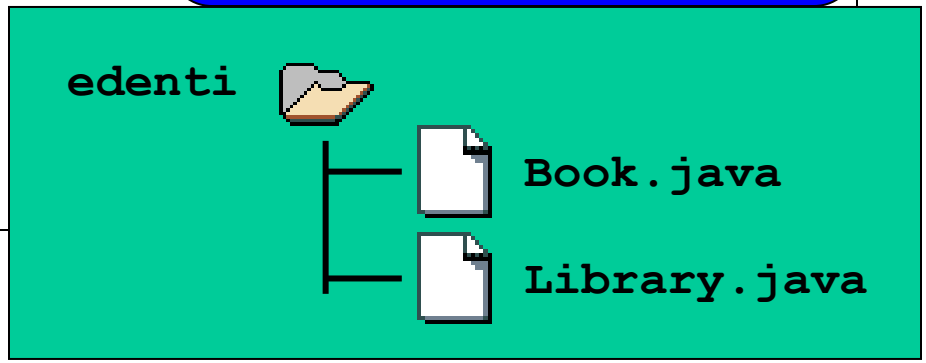
```
package edenti;
public class Book {
   ...
}
```

File **Book.java** nella cartella **edenti**

```
package edenti;
public class Library {
   public Library(){
      Book c = new Book(...);
   }
   // main
}
```

nome relativo perché interno al package

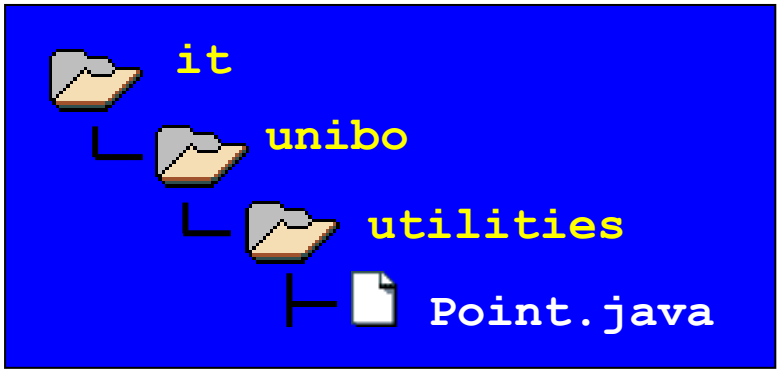
File **Library.java** nella cartella **edenti** *(contiene anche il main)*



VARIANTE: NOME STRUTTURATO

Usiamo il nome di package **`it.unibo.utilities`**

- l'obbligo di corrispondenza fra *nome del package* e *nome e percorso della cartella* comporta *tre cartelle innestate:*



- ATTENZIONE: *questo non significa che ci siano tre package uno dentro l'altro,* è solo un nome multi-livello
- c'è comunque *un unico* package!

NOMI MULTI-LIVELLO "SIMILI"… … = PACKAGE MULTIPLI ?

- Sebbene un nome multi-livello indichi un solo package, *a volte ci sono davvero più package dai nomi simili*

```
java.util
java.util.concurrent
java.util.concurrent.atomic
java.util.concurrent.locks
java.util.jar
java.util.logging
java.util.prefs
java.util.regex
java.util.zip
```

  - OCCHIO: questi package *non* sono contenuti uno nell'altro!
  - i loro nomi si somigliano per indicare *vicinanza concettuale*, ma sono tutti package *indipendenti* gli uni dagli altri

PACKAGE IN Scala: SINTASSI ALTERNATIVA

- Oltre alla sintassi standard «Java like», Scala ammette una sintassi a blocchi innestati

Sintassi standard

```
package edenti;
class Book {
   ...
}
```

Scala: sintassi a blocchi innestati

```
package edenti {
 class Book {
   ...
 }
}
```

- Tale sintassi ha vari vantaggi:
  - evidenzia meglio visivamente «l'appartenenza» a un package
  - consente package multipli nello stesso file
  - consente veri package *innestati concettualmente*

PACKAGE IN Scala: SINTASSI ALTERNATIVA

package multipli

```
package edenti {
  class Book {
    ...
  }
}
package mrossi {
  object Matrix {
    ...
  }
}
```

package definiti: **edenti**, **mrossi**

package innestati

```
package edenti {
  class Book {
    ...
  }
  package util {
    class Frazione {
      ...
    }
  }
}
```

package definiti: **edenti**, **edenti.util**

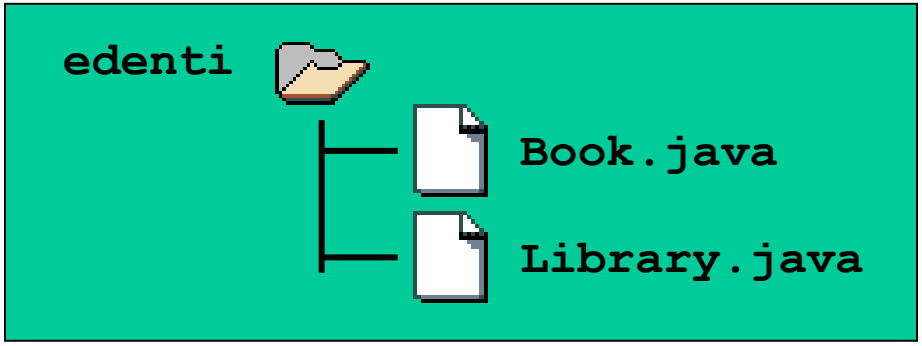
Compilazione ed esecuzione di applicazioni con package

COMPILAZIONE (1/2)

- Per *compilare* una classe **`Book`** che faccia parte di un package **`edenti`** occorre:
  - porsi nella cartella *superiore* a **`edenti`**
  - invocare *da lì* il compilatore, specificando il *percorso completo* della classe
  - ESEMPIO in Java:

```
javac edenti/Book.java
javac edenti/Library.java
```

ATTENZIONE: ogni altro modo di invocare il compilatore è *errato* e causerà problemi



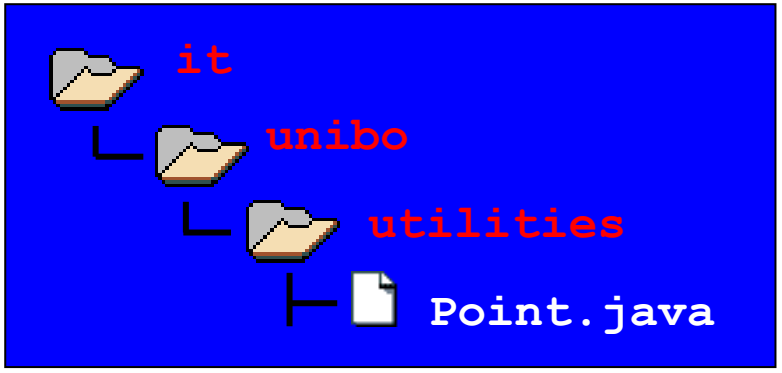
COMPILAZIONE (2/2)

- Analogamente, nel caso di cartelle annidate:

```
javac it/unibo/utilities/Point.java
```

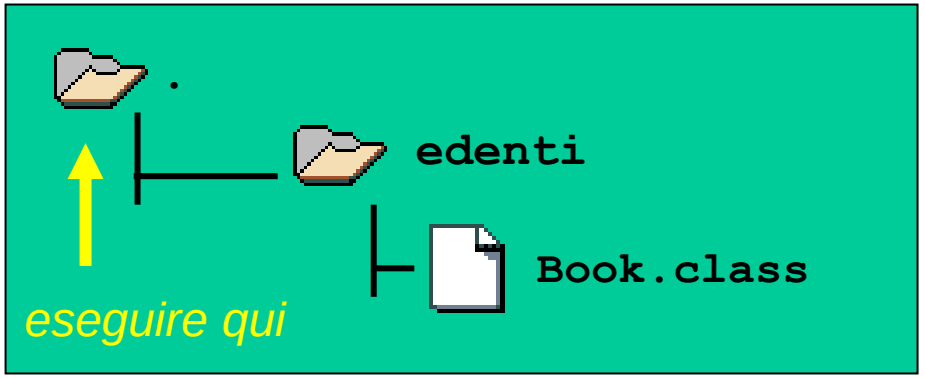
Va scritto così! OCCHIO ALLE BARRE…
Ogni altro modo di invocare il compilatore è errato

  - ATTENZIONE: invocare il compilatore *da dentro* la sotto-cartella non darà errore *immediato,* ma produrrà un file **`class`** con nome interno della classe *non corrispondente alle attese*
  - L'errore si verificherà subito dopo, a run time!



ESECUZIONE

- Per *eseguire* un programma con package occorre:
  - porsi nella *cartella superiore* a **`edenti`**
  - invocare *da lì* l'interprete, specificando il *nome assoluto* della classe che contiene il main
  - NB: nel caso del default package, si usa il *nome relativo* della classe, perché il nome assoluto semplicemente *non esiste!*
  - in alternativa, o qualora non sia possibile porsi nella cartella superiore al package, occorrerà specificare dove trovare il package tramite l'opzione **`-cp`**

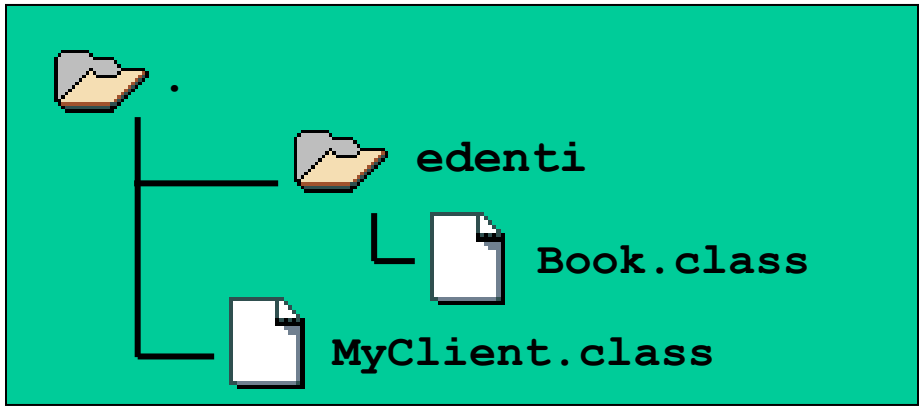


PRIMO ESEMPIO: ESECUZIONE

- Supponiamo che il main
  - sia nella classe **`MyClient`**, che appartiene al default package
  - usi la classe **`edenti.Book`**
- Affinché la classe **`edenti.Book`** sia trovata, l'esecuzione va lanciata *dalla cartella superiore a* ***`edenti`***

```
public class MyClient {
  // il main usa edenti.Book
}
```

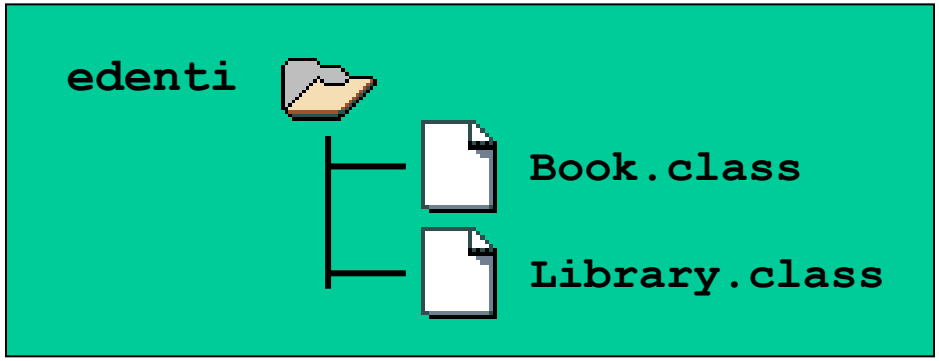
```
package edenti;
public class Book {
   ...
}
```



SECONDO ESEMPIO: ESECUZIONE

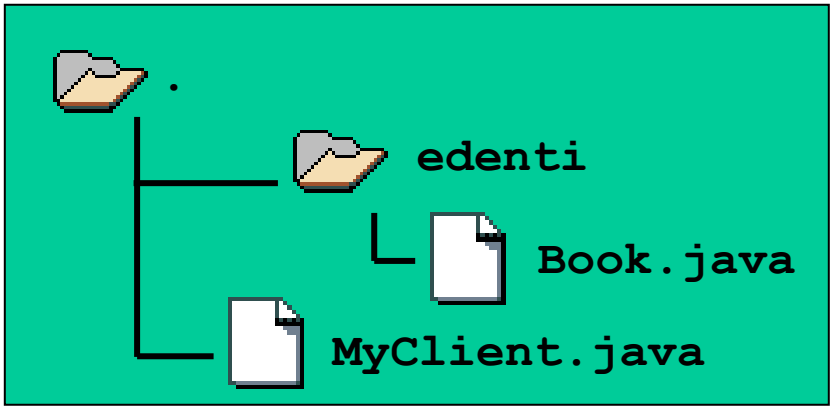
- Supponiamo ora che il main sia:
  - non più nella classe **`MyClient`** del default package
  - ma nella classe **`Library`** del package **`edenti`**
- L'esecuzione va lanciata:
  - ancora dalla cartella superiore a **`edenti`**
  - MA *indicando il nome assoluto* della classe che contiene il main
  - ESEMPIO in Java: **`java edenti.Library`**

```
package edenti;
public class Library {
  // usa edenti.Book
}
```



SPAZIO DI NOMI DI DEFAULT

- Come è noto, una classe che non preveda una diversa dichiarazione appartiene allo *spazio di nomi di default.*
- Pessima scelta: *darà solo problemi!*
  - in particolare, le sue classi saranno *inaccessibili* da altri package, perché il default package *non ha un nome e quindi non c'è modo di fare riferimento ad esse*
- Corrispondenza nel file system
  - il default package corrisponde, per convenzione, alla *cartella corrente ('.')*



UN CASO PIÙ COMPLESSO

- PROBLEMA: se un cliente fa uso di più package, la cartella *superiore* può non essere unica!
  - dipende da *quali package ci sono* e *come si chiamano*,
  - ognuno di essi starà dove gli compete nel file system
- In questi casi occorre specificare *l'elenco delle posizioni*, tramite l'opzione *classpath*
  - ESEMPIO in Java:

    `javac -cp` *listapercorsi* **`MyMain.java`**

    `java -cp` *listapercorsi* **`MyMain`**

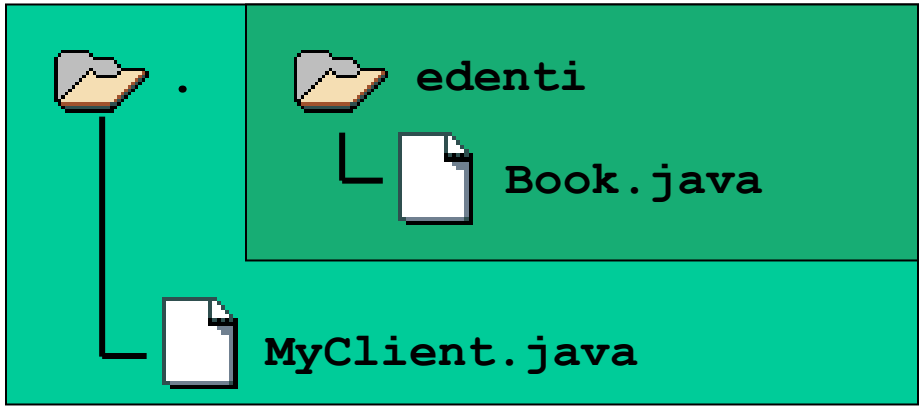
  - *listapercorsi* specifica *dove reperire* le classi e i package usati
  - il separatore dei vari percorsi è `;` su Windows, `:` su Mac/Linux/Unix

UN TERZO ESEMPIO (1/3)

- Come nel primo esempio, il main
  - è nella classe **`MyClient`**, che appartiene al default package
  - usa la classe **`edenti.Book`**
- Però, qui la cartella **`edenti`** *non è una subdirectory* della cartella in cui si trova il main
  - occorre quindi specificare il classpath, ad esempio in Java:

```
javac -cp … MyClient.java
```

Il classpath deve includere:
- la cartella corrente '.' in cui si trova la classe **`MyClient`**
- la cartella superiore a **`edenti`**

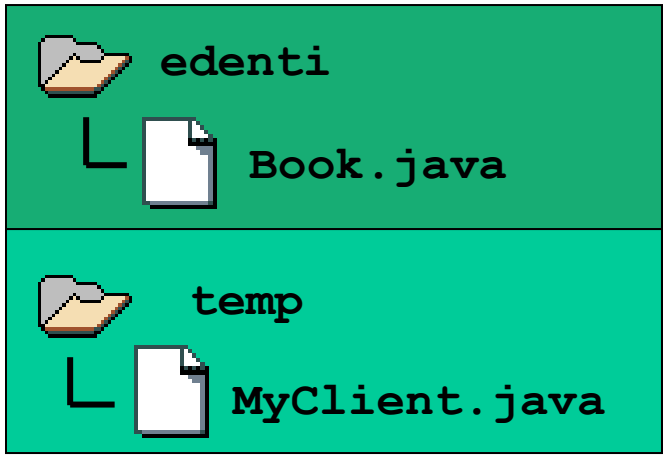


UN TERZO ESEMPIO (2/3)

- Per concretizzare, supponiamo che:
  - la classe **`MyClient`**, che contiene il main, si trovi nella cartella di lavoro **`temp`** (che non corrisponde ad alcun package)
  - la cartella **`edenti`** si trovi allo stesso livello di **`temp`**
- In questo caso, il compilatore dovrà essere invocato dalla cartella **`temp`**, scrivendo:

```
javac -cp .;.. MyClient.java
```

Il percorso comprende:
- la cartella corrente '`.`' (cioè **`temp`**)
- la cartella superiore a **`edenti`**, che è la cartella '`..`' perché per ipotesi **`edenti`** è allo stesso livello di **`temp`**

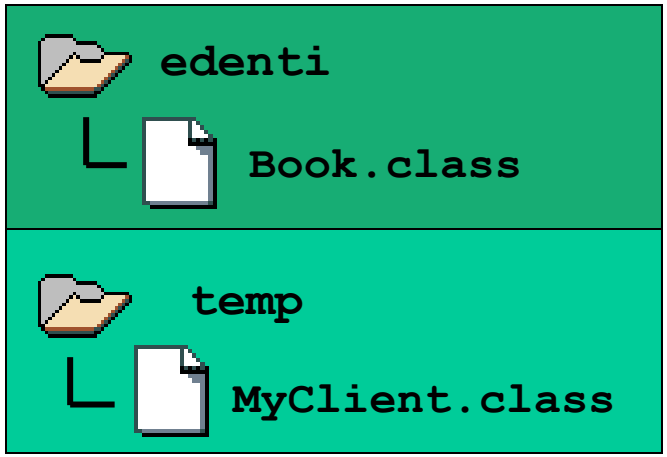


UN TERZO ESEMPIO (3/3)

- Idem per quanto riguarda l'esecuzione:

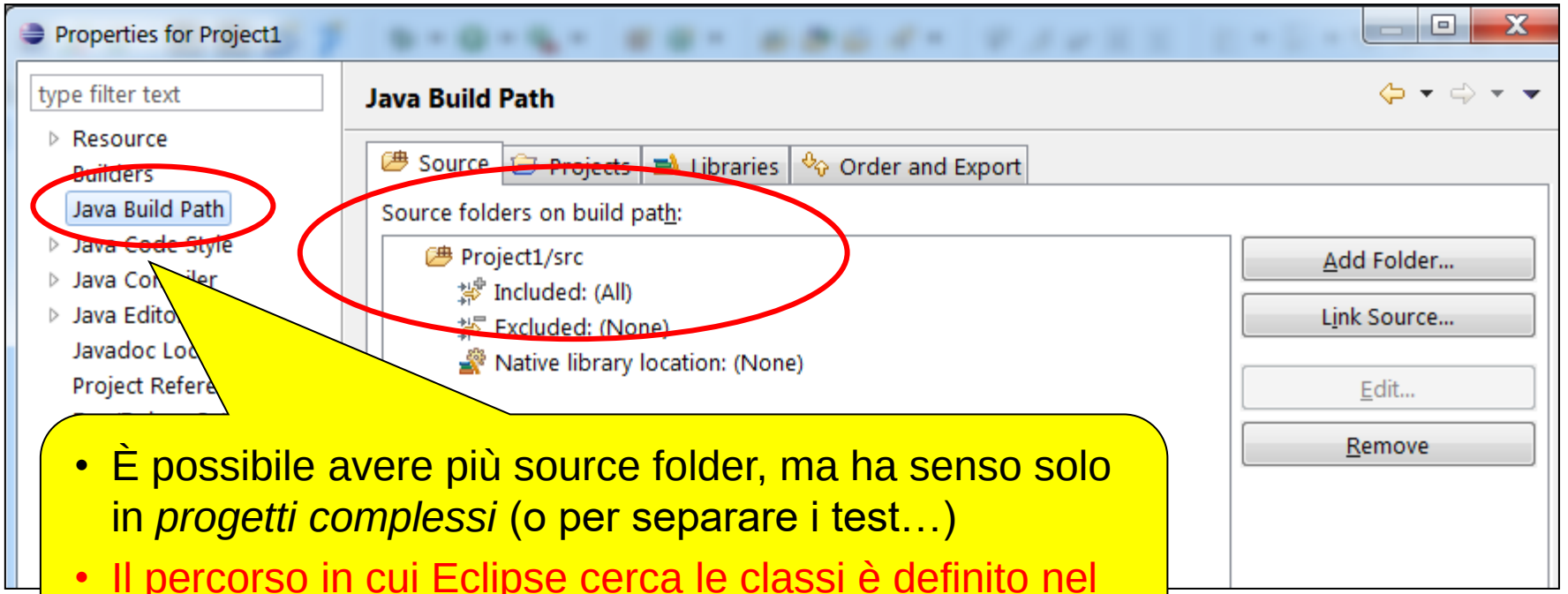
```
java -cp .;.. MyClient
```

Il percorso comprende:
- la cartella corrente '.' (cioè **temp**)
- la cartella superiore a **edenti**, che è la cartella '..' perché per ipotesi **edenti** è allo stesso livello di **temp**



GESTIONE DEI PERCORSI IN ECLIPSE

- Eclipse gestisce in modo automatico i percorsi nelle diverse cartelle, *purché si adotti la sua organizzazione:*
  → tutti i sorgenti dentro *il source folder* **`src`**

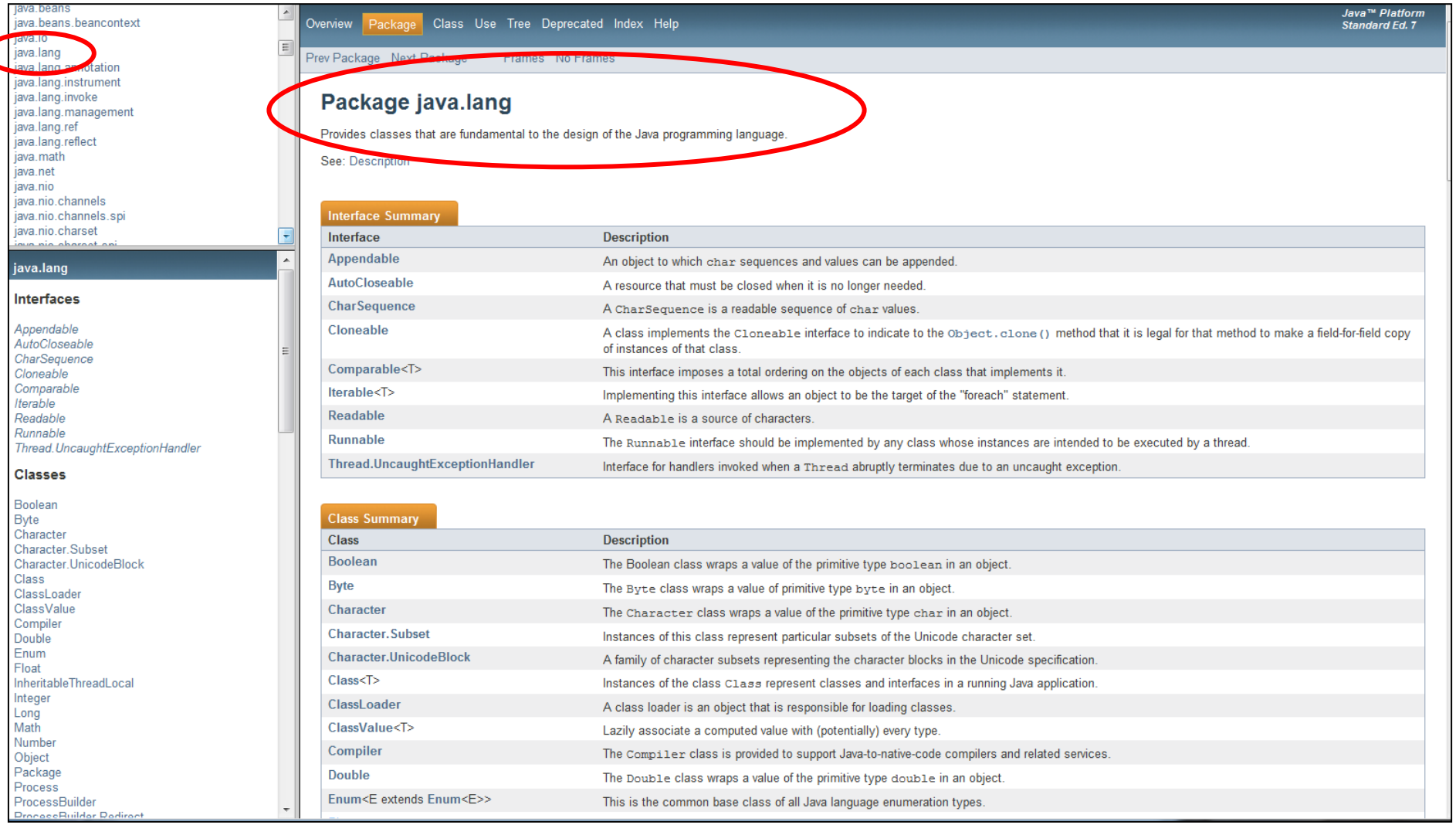


- È possibile avere più source folder, ma ha senso solo in *progetti complessi* (o per separare i test…)
- Il percorso in cui Eclipse cerca le classi è definito nel *Java Build Path*

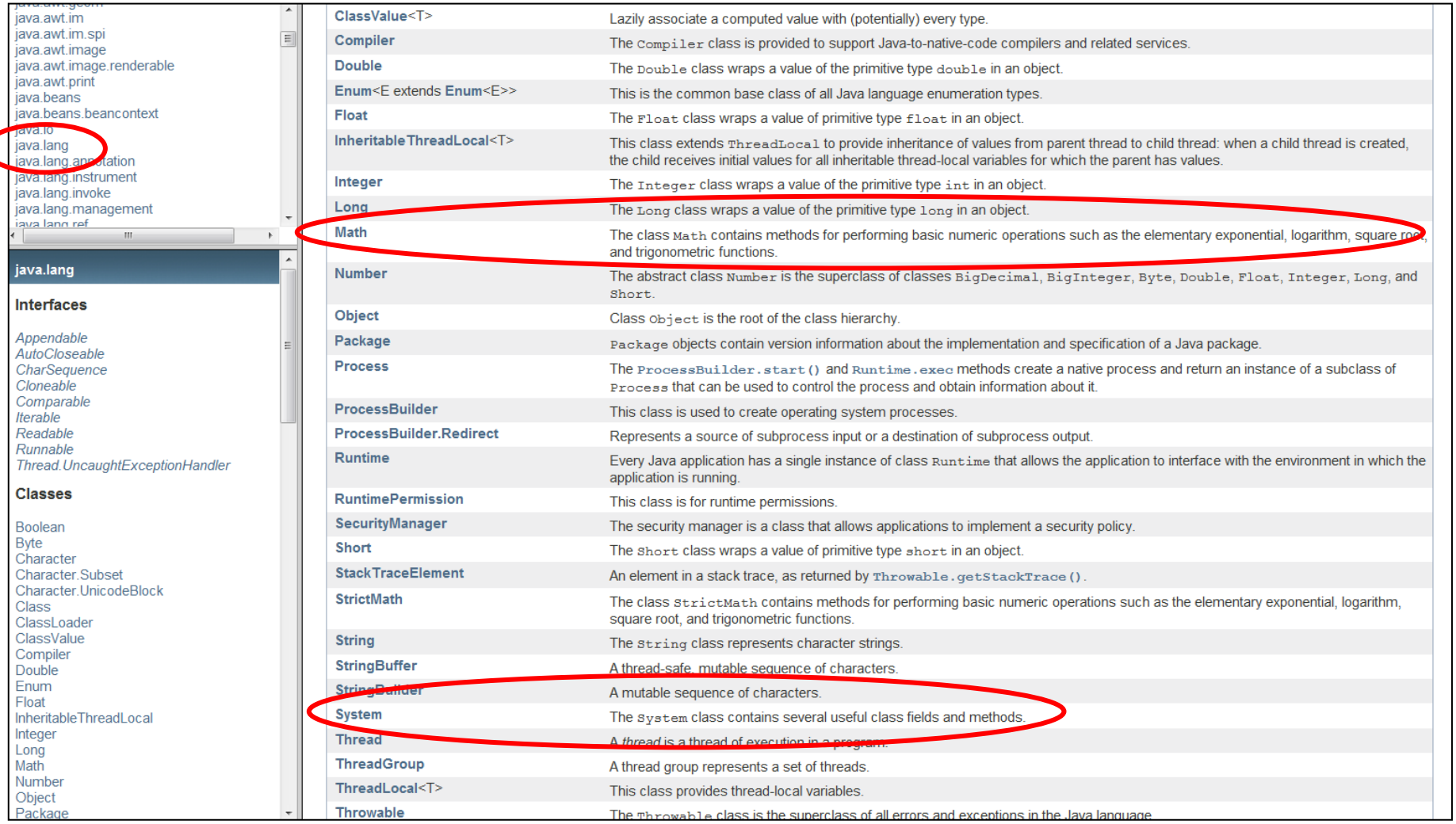
IL PACKAGE `java.lang` (1)

- Il nucleo centrale di Java è definito nel package **`java.lang`**
  - è importato automaticamente:
    la frase **`import java.lang.*`** è *sempre sottintesa*
  - definisce buona parte della classi di sistema
    (ad esempio, **`String`** è in realtà **`java.lang.String`**)
  - molte altre classi sono definite altrove:
    ci sono decine di package, che forniscono i servizi più vari:
    `java.util`, **`java.io`**, **`java.text`**, **`javafx`** …
- Anche Scala e Kotlin importano automaticamente i loro core
  - Scala: importa **`java.lang`**, **`scala`** e l'oggetto singleton **`Predef`**
  - Kotlin: importa **`java.lang`** e una parte dei package **`kotlin.*`**

IL PACKAGE `java.lang` (2)



IL PACKAGE `java.lang` (3)



java.awt.im
java.awt.im.spi
java.awt.image
java.awt.image.renderable
java.awt.print
java.beans
java.beans.beancontext
java.io
java.lang
java.lang.annotation
java.lang.instrument
java.lang.invoke
java.lang.management
java.lang.ref

java.lang

Interfaces

Appendable
AutoCloseable
CharSequence
Cloneable
Comparable
Iterable
Readable
Runnable
Thread.UncaughtExceptionHandler

Classes

Boolean
Byte
Character
Character.Subset
Character.UnicodeBlock
Class
ClassLoader
ClassValue
Compiler
Double
Enum
Float
InheritableThreadLocal
Integer
Long
Math
Number
Object
Package

ClassValue<T>	Lazily associate a computed value with (potentially) every type.
Compiler	The `Compiler` class is provided to support Java-to-native-code compilers and related services.
Double	The `Double` class wraps a value of the primitive type `double` in an object.
Enum<E extends **Enum**<E>>	This is the common base class of all Java language enumeration types.
Float	The `Float` class wraps a value of primitive type `float` in an object.
InheritableThreadLocal<T>	This class extends `ThreadLocal` to provide inheritance of values from parent thread to child thread: when a child thread is created, the child receives initial values for all inheritable thread-local variables for which the parent has values.
Integer	The `Integer` class wraps a value of the primitive type `int` in an object.
Long	The `Long` class wraps a value of the primitive type `long` in an object.
Math	The class `Math` contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.
Number	The abstract class `Number` is the superclass of classes `BigDecimal`, `BigInteger`, `Byte`, `Double`, `Float`, `Integer`, `Long`, and `Short`.
Object	Class `Object` is the root of the class hierarchy.
Package	`Package` objects contain version information about the implementation and specification of a Java package.
Process	The `ProcessBuilder.start()` and `Runtime.exec` methods create a native process and return an instance of a subclass of `Process` that can be used to control the process and obtain information about it.
ProcessBuilder	This class is used to create operating system processes.
ProcessBuilder.Redirect	Represents a source of subprocess input or a destination of subprocess output.
Runtime	Every Java application has a single instance of class `Runtime` that allows the application to interface with the environment in which the application is running.
RuntimePermission	This class is for runtime permissions.
SecurityManager	The security manager is a class that allows applications to implement a security policy.
Short	The `Short` class wraps a value of primitive type `short` in an object.
StackTraceElement	An element in a stack trace, as returned by `Throwable.getStackTrace()`.
StrictMath	The class `StrictMath` contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.
String	The `String` class represents character strings.
StringBuffer	A thread-safe, mutable sequence of characters.
StringBuilder	A mutable sequence of characters.
System	The `System` class contains several useful class fields and methods.
Thread	A *thread* is a thread of execution in a program.
ThreadGroup	A thread group represents a set of threads.
ThreadLocal<T>	This class provides thread-local variables.
Throwable	The `Throwable` class is the superclass of all errors and exceptions in the Java language.

Importazione statica di nomi in Java e C#

IMPORTAZIONE STATICA DI NOMI in Java e C#

- La direttiva import «standard» importa *nomi di classi (tipi): non si applica ai singoli metodi*
  - ESEMPIO: **`Math`** fa parte di **`java.lang`** (importato automaticamente) Quindi si può scrivere solo **`Math`** invece di **`java.lang.Math`**
- Ma per usare costanti o metodi statici di libreria occorre comunque specificarne il nome «dalla classe in poi»:

```
Math.sin(Math.PI/3)
```

  - se le classe ha un nome lungo, ciò può essere faticoso
- Per ovviare, in Java e C# esiste la direttiva **`import static`**
  - in C#, **`using static`**
  - importa i membri statici di una singola classe
  - consente l'uso del *nome relativo* di tali membri, senza citare la classe

IMPORTAZIONE STATICA DI NOMI in Java e C#

- ESEMPIO: anziché dover scrivere

`Math.sin(Math.PI/3)`

- la direttiva **`import static`** consente di importare i membri statici di **`Math`**, così da poter scrivere semplicemente

`sin(PI/3)`

```
import static java.lang.Math.*;
...
  sin(PI/3); // anziché Math.sin(Math.PI/3)
```

- MA ci sono luci e ombre… non a caso, fu molto discussa!

IMPORTAZIONE STATICA DI NOMI in Java e C#

- Il problema è che la direttiva *nasconde il fornitore del servizio*
  - non si capisce più *chi* fornisca *cosa*
  - il debugging può diventare una caccia al tesoro ☹

```
...
sin(PI/3);
```

Da dove vengono questa funzione e questa costante, non definite localmente?

- Possibile perdita di trasparenza e di leggibilità
  - vale la pena, per risparmiare qualche carattere?
  - compromesso: supporto da parte dell'ambiente integrato (Eclipse)
- In Scala e Kotlin si usa sempre la direttiva **`import`**
  - non c'è alcuna **`import static`**
  - **`import`** funziona su tutto (in Kotlin, anche funzioni, proprietà, etc.)

Namespace in C#

IL COSTRUTTO NAMESPACE

- In C#, una dichiarazione di namespace ha la forma:

```
namespace nome {
        ...
}
```

  - assomiglia alla sintassi Scala a blocchi innestati: il costrutto *racchiude con un blocco* le entità a cui si applica
  - a differenza di Java & co., non esiste una *corrispondenza obbligatoria* fra nome e cartelle
  - a differenza di Java, il livello di visibilità predefinito è **`private`**
  - la keyword **`internal`**, che indica visibilità nell'assembly, può essere sfruttata se un namespace coincide con un assembly

GLI ESEMPI IN C#

- Supponiamo di definire un namespace per racchiudere la classe **`Book`** :

Come in Java, il *nome strutturato* della classe è **`EDenti.Book`**

```
namespace Edenti {
   public class Book {
        ...
   }
}
```

- Il cliente che voglia usare la classe **`Book`** può:
  - usare il nome assoluto, **`Edenti.Book`**
  - importare il namespace tramite la direttiva **`using`**, usando poi il semplice nome relativo **`Book`**

```
using Edenti;
public class MyClient {
   ... // nome relativo
}
```

DEFINIZIONE DI ALIAS (1/2)

- In presenza di *omonimie* fra classi di namespace diversi da importare *(name clash)*:
  - in Java l'unica via è importarne una sola (la più usata), usando il nome assoluto per l'altra
  - in C#, Scala, Kotlin si può *stabilire un alias* per una classe importata, così da poterla distinguere dall'altra omonima

    C#: `using UPoint = it.unibo.Utilities.Point;`
    Scala: `import it.unibo.Utilities.{Point => UPoint}`
    Kotlin: `import it.unibo.Utilities.Point as UPoint`
  - ESEMPIO: se i namespace **Graph2D** e **Graph3D** definissero entrambi una classe **Circle** e occorresse importarli entrambi

```
using Graph2D;
using Circle3D = Graph3D.Circle;
// uso di Circle e Circle3D
```

Ora **Circle** è quella di **Graph2D**, mentre **Circle3D** è la **Circle** di **Graph3D**

DEFINIZIONE DI ALIAS (2/2)

- In C# è anche possibile attribuire un alias a un namespace, per poter fare riferimento ad esso con un *nome più corto.*
  - ad esempio, se volessimo stabilire un alias per il namespace

    `Edenti.Utilities.Formatters.BinaryFormatter`

```
using MyFormatter =
   Edenti.Utilities.Formatters.BinaryFormatter;
```

  - ATTENZIONE: questa **`using`** *non importa* i nomi del namespace, si limita a *stabilire un alias più corto* per il namespace in sé
  - Ergo, per usare le classi in esso contenute (ad es. **`BinForm`**) si deve indicarne il nome assoluto, usando l'alias ora definito (es. **`MyFormatter.BinForm`**)

Java 9:
dai package ai moduli

JAVA 9: OLTRE I PACKAGE

- Strutturare le applicazioni è fondamentale per dominare la complessità e favorirne il riuso *garantendo incapsulamento*
  - nei linguaggi a oggetti, le *classi* sono *l'elemento base*
  - i *package* (o namespace) costituiscono il *livello successivo*
  - ognuno di essi ammette i qualificatori private/(package)/public
- Tuttavia, l'esperienza ha mostrato che neppure i package sono sufficienti quando l'applicazione è vasta
  - il livello di protezione è troppo on/off, a grana grossa
  - package progettati per funzionare assieme *sono costretti a rendere pubbliche determinate funzionalità solo per renderle accessibili agli altri package*
  - …ma così facendo *qualunque altro package può vederle* ☹

IL CASO PRATICO DELL'INFRASTRUTTURA JAVA

- L'infrastruttura Java (JRE & JDK) ne è un esempio
  - cresciuta molto col passare degli anni
  - elementi obsoleti non rimuovibili per retrocompatibilità
  - molte applicazioni ne usano solo una piccola parte, ma il JRE dev'essere presente (e installato) nella sua interezza
- Sarebbe invece desiderabile che si potessero *separare e raggruppare le funzionalità* in parti..
  - ..in modo da non obbligare a portarsi sempre dietro tutto..
- e magari poter *esplicitare cosa rendere effettivamente pubblico* di quelle parti
  - ottenendo un nuovo livello di incapsulamento, a grana più grossa

IL CONCETTO DI MODULO (1)

- Java 9 ha introdotto a questo scopo il concetto di *modulo*
- Un modulo è:
  - concettualmente una *collezione di package*
  - praticamente un piccolo file **`module-info.java`**
- che specifica:
  - quali package siano *accessibili dall'esterno* **`exports`**
  - da quali altri moduli questo *dipenda* **`requires`**

`module-info.java`

```
module pippo {
   exports  package1, package2, … ;
   requires moduleA, moduleB, … ;
}
```

se manca **`exports`**, non esporta nulla

se manca **`requires`**, non richiede nulla

IL CONCETTO DI MODULO (2)

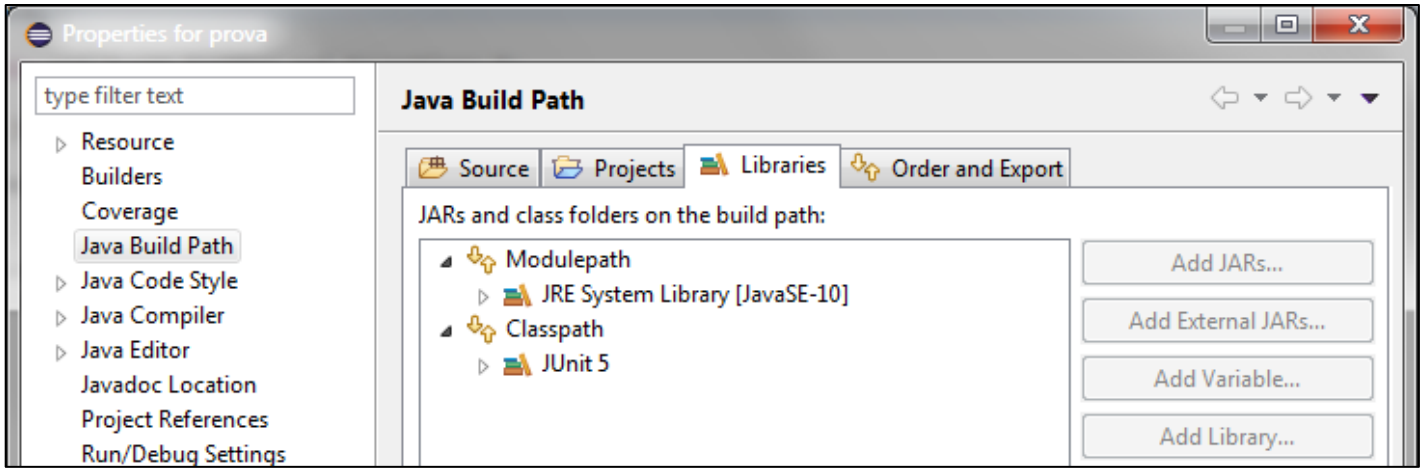
- Come un package, un modulo ha un *nome* che può essere, e tipicamente è, *strutturato per livelli*
  - sintassi: ***`identifier1.identifier2.identifier3`***
  - si adotta la solita convenzione "reverse Internet naming"
- Come nel caso dei package, l'uso di un nome strutturato *non significa includere logicamente altri moduli*
- Per convenzione, un modulo solitamente assume il *nome* del suo *package di top-level*
  - la convenzione mira a prevenire conflitti
    - poiché ogni package può stare in un solo modulo, se i package iniziano col nome del modulo e quest'ultimo è univoco, anche i package lo saranno
  - ma non c'è alcun obbligo netto al riguardo

IL CONCETTO DI MODULO (3)

- La JVM supporta nativamente il nuovo concetto
  - l'accesso a package *non esplicitamente esportati* viene impedito → *strong encapsulation*
  - ovviamente, però, in modo *retrocompatibile*
- Nuovo concetto: *module path*
  - in sostituzione (o aggiunta..) al più classico *class path*
  - i JAR menzionati nel module path sono *trattati come moduli* anziché come semplici JAR «vecchio stile»
- In un'applicazione *non sono ammessi package duplicati*
  - un package deve provenire da un solo modulo: altrimenti, *conflitto*
  - per questo più moduli non possono esportare lo stesso package
  - OCCHIO ai moduli automatici: possibili clash di nomi

JAVA BUILD PATH "MIXED"

- Di conseguenza, da Java 9 in poi, se guardate le *proprietà* di un qualunque progetto, troverete librerie
  - sia nel module path (se modularizzate): ad esempio, il JRE
  - sia nel class path (se JAR tradizionali): ad esempio, JUnit



RETROCOMPATIBILITÀ E MIGRAZIONE DI APPLICAZIONI

- NON è obbligatorio modularizzare un'applicazione
- Esiste un *modulo di default (unnamed module)* in cui finiscono le classi non appartenenti a un modulo esplicito
  - proprio *quelle elencate nel class path tradizionale*
  - in tal modo il module system è «virtualmente disattivato» e il comportamento rimane quello classico
- Nel nostro corso, faremo esattamente così!
  - inutile complicare le cose ora
  - i moduli non sono una feature essenziale della OOP
- *tranne per la grafica in JavaFX*
  - Java 11 ha separato JavaFX da Java "core"
  - OpenJFX è un vero e proprio *modulo* e andrà usato come tale

APPLICAZIONI TRADIZIONALI e MODULO DI DEFAULT

- Quindi, se si lancia in Java 9+ un'applicazione tradizionale *senza usare il module path,* ma solo il classico *class path*

  classico comando **`java -cp … MainClass`**

  classico comando **`java -jar MyApp.jar`**

  – il module system è «virtualmente disattivato»:
  tutte le classi finiscono nel modulo di default (unnamed module)

- Il modulo di default (*unnamed module*)
  – può accedere *a ogni altro modulo* ed esporta *tutti i suoi package*
  → comportamento finale identico a Java 8 (pre-moduli)
  – MA *non può essere acceduto dai moduli con nome,* esattamente come il package di default non è accessibile dai package con nome

VOLENDO, COMUNQUE…

- Non è difficile *modularizzare* applicazioni pre-esistenti, *purché siano già organizzate in package*
  - non si può migrare un'applicazione senza package!
- In generale, modularizzare un'applicazione significa:
  - concettualmente — stabilire quali e quanti moduli ci siano e *che dipendenze abbiano fra loro*
  - praticamente — strutturare l'app in *package* e predisporre i vari **`module-info.java`** collocandoli nella *directory base* dei package
- Per facilitare la migrazione esistono i *moduli automatici*
  - ogni JAR elencato nel module path è *convertito automaticamente* in un modulo con lo stesso nome (circa…)