



UNIVERSITÀ DEGLI STUDI DELL'AQUILA

Laboratorio di Programmazione ad Oggetti

Ph.D. Juri Di Rocco
juri.dirocco@univaq.it
<http://jdirocco.github.io>





Sommario

- › Classe Object
- › Classe System
- › Reflection
- › Numeri come oggetti
- › Autoboxing e autounboxing
- › Annotazioni
 - Introduzione
 - Definizioni e meta-annotazioni
- › Date prima e dopo Java 8



Classe Object (1)

- › Ogni classe in Java estende da `Object` anche se non si è obbligati a farlo ovvero
 - `public class Employee extends Object`
- › E' possibile
 - Dichiarare variabili di tipo `Object`
 - › `Object o = new Employee("Harry Hacker", 35000);`
 - Creare oggetti di tipo `Object`
 - › `Object o = new Object();`
- › Contiene una serie di metodi che possono essere utilizzati e/o sovrascritti (override)



Classe Object (2)

› Metodi che possono essere sovrascritti

- clone
- **equals**
- **hashCode**
- finalize
- **toString**

› Metodi final

- getClass
- notify
- notifyAll
- wait



Classe Object: equals (3)

› `public boolean equals(Object obj)`

- Verifica se un oggetto può essere considerato uguale ad un altro
- Operatore `==` applicato a tipi reference verifica se le due variabili **puntano** allo stesso oggetto
- Metodo classe `Object` verifica l'uguaglianza tra i due riferimenti (equivale a `==`)

```
public boolean equals(Object obj) {  
    return (this == obj);  
}
```

- Per modificare il comportamento è necessario effettuare l'override del metodo
- Nota: è necessario dichiarare il metodo dove il tipo del parametro formale è lo stesso (`Object`)



Classe Object: equals (4)

- › Regole per implementare `equals`
 - **Riflessiva:** `x.equals(x)` deve restituire `true`
 - **Simmetrica:** `x.equals(y)` è `true` se e solo se `y.equals(x)` è `true`
 - **Transitiva:** se `x.equals(y)` è `true` e `y.equals(z)` è `true` allora `x.equals(z)` è `true`
 - **Coerente:** se gli oggetti non cambiano allora `x.equals(y)` restituisce sempre lo stesso valore
 - `x.equals(null)` restituisce sempre `false`



Classe Object: equals (5)

- › Per rispettare tali regole è sufficiente implementare il metodo nel seguente modo
 - Utilizzare l'operatore `==` per verificare se l'argomento è un **riferimento** all'oggetto stesso

```
if ( this == obj) return true;
```
 - Utilizzare l'operatore `instanceof` per verificare se l'argomento è del tipo corretto. Se non lo è restituire `false`
 - Effettuare il casting al tipo corretto
 - Per ogni campo significativo della classe controllare se corrisponde a quello dell'argomento passato
 - › Se il test va a buon fine restituire `true` altrimenti `false`
 - Come ultimo passo vedere se sono valide le regole precedenti!!!



Classe Object: equals (6)

```
public class Punto {  
    private int x, y;  
    public Punto(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    @Override  
    public boolean equals(Object o) {  
        if (o == this) return true;  
        if (!(o instanceof Punto)) return false;  
        Punto p = (Punto)o;  
        return this.x == p.x && this.y == p.y;  
    }  
}
```




Classe Object: hashCode (7)

- › Generalmente se si effettua l'override di `equals` lo si fa anche di `hashCode`
- › Metodo viene utilizzato all'interno di alcune classi di Collection (esempio `Map`)
- › Metodo che restituisce un intero che *rappresenta* l'oggetto
- › Regole per implementare `hashCode`
 - Se `o1.equals(o2)` restituisce `true` allora `o1` e `o2` devono avere lo stesso hash code
 - Il viceversa non è vero, ovvero se `o1.hashCode() == o2.hashCode()` non è detto `o1.equals(o2)` restituisce `true`
 - Invocazione successiva di `hashCode` deve ritornare lo stesso valore
 - Metodo per generare l'hash code deve utilizzare le stesse variabili di istanza di `equals`



Classe Object: hashCode (8)

- › Metodo implementato all'interno di `hashCode` deve ridurre il numero di *conflitti*
- › Esempio non buono

```
public int hashCode() {  
    return x + y;  
}
```

Esempio buono

```
@Override  
public int hashCode() {  
    final int prime = 31;  
    int result = 1;  
    result = prime * result + x;  
    result = prime * result + y;  
    return result;  
}
```



Classe Object: toString (9)

- › `public String toString()`
 - E' buona norma effettuare l'override di tale metodo
 - Implementazione di default della classe `Object` restituisce il nome della classe seguito da `@` e dalla rappresentazione esadecimale senza segno di ciò che restituisce `hashCode`

› Esempio

```
@Override
public String toString() {
    return "Punto [x=" + x + ", y=" + y + " ]";
}
```



Classe System (1)

- › Classe che appartiene al package `java.lang`
- › E' dichiarata `final` per ragioni di sicurezza
- › Contiene solo variabili e metodi statici
- › Variabili statiche
 - `out`: di tipo `PrintStream` (package `java.io`), rappresenta lo stream standard di output ovvero la console
 - `err`: di tipo `PrintStream` (package `java.io`), rappresenta lo stream standard di errore ovvero la console
 - `in`: di tipo `InputStream` (package `java.io`), rappresenta lo stream standard di input ovvero la console



Classe System (2)

› Classico esempio per leggere da console

```
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class Test {
    public static void main(String[] args) {
        String linea = null;

        try (BufferedReader br = new BufferedReader(new InputStreamReader(System.in))) {
            while ((linea = br.readLine()) != null) {
                System.out.println("linea=" + linea);
                if ("x".equals(linea)) {
                    return;
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```



Classe System: Metodo printf (1)

- › Metodi `print` e `println` utilizzati per stampare stringhe sullo standard output (`System.out`)
- › `System.out.println()` è efficiente per stampare una linea di testo
- › Se la linea di testo deve essere formattata usare `System.out.printf()`

› Esempi

```
String s = "Hello World";
```

```
System.out.printf("The String object %s is at hash code %h%n", s, s);
```

OUTPUT: The String object Hello World is at hash code cc969a84

```
System.out.printf("Total is: $%,.2f%n", dblTotal);
```

```
System.out.printf("Total: %-10.2f: ", dblTotal);
```

```
System.out.printf("%4d", intValue);
```

```
System.out.printf("%20.10s\n", stringVal);
```

Manda a capo





Classe System: Metodo printf (2)

Java **printf()** Method Quick Reference

```
System.out.printf( "format-string" [, arg1, arg2, ... ] );
```

Format String:

Composed of literals and format specifiers. Arguments are required only if there are format specifiers in the format string. Format specifiers include: flags, width, precision, and conversion characters in the following sequence:

% [flags] [width] [.precision] conversion-character (square brackets denote optional parameters)

Flags:

- : left-justify (default is to right-justify)
- + : output a plus (+) or minus (-) sign for a numerical value
- 0 : forces numerical values to be zero-padded (default is blank padding)
- , : comma grouping separator (for numbers > 1000)
- : space will display a minus sign if the number is negative or a space if it is positive

Width:

Specifies the field width for outputting the argument and represents the minimum number of characters to be written to the output. Include space for expected commas and a decimal point in the determination of the width for numerical values.

Precision:

Used to restrict the output depending on the conversion. It specifies the number of digits of precision when outputting floating-point values or the length of a substring to extract from a String. Numbers are rounded to the specified precision.

Conversion-Characters:

- d : decimal integer [byte, short, int, long]
- f : floating-point number [float, double]
- c : character Capital C will uppercase the letter
- s : String Capital S will uppercase all the letters in the string
- h : hashCode A hashCode is like an address. This is useful for printing a reference
- n : newline Platform specific newline character- use %n instead of \n for greater compatibility



Classe System: Proprietà di sistema (1)

- › Programmi Java vengono eseguiti all'interno di un ambiente che contiene *attributi di sistema*
 - Directory corrente
 - Tipo sistema operativo
 - Versione corrente della JVM
 -
- › Programma Java può anche settare attributi configurabili (*attributi di programma*)
 - Opzioni allo start-up
 -



Classe System: Proprietà di sistema (2)

- › Attributi composto da due parti
 - Nome
 - Valore
- › Esempio
 - Attributo `os.name=Windows XP`
- › Classe `Properties` del package `java.util` gestisce un insieme di coppie chiave/valore
- › Classe `System` gestisce gli attributi di sistema mediante metodi
 - `public static String getProperty(String key)`
 - `public static String getProperty(String key, String default)`
 - `public static Properties getProperties()`



Classe System: Proprietà di sistema (3)

- › Utilizzando opzione `-D` allo startup è possibile poi recuperare la proprietà tramite metodo `getProperty`

```
java -DtestP=Hello Test
```

```
class Test{  
    public static void main(String[] args) {  
        String proprieta = System.getProperty("testP");  
        System.out.println("Proprietà:" + proprieta);  
    }  
}
```

- › E' possibile settare anche le proprietà
 - `public static String setProperty(String key, String value)`
 - `public static setPropertys(Properties prop)`



Classe System: Proprietà di sistema (4)

Più comuni

| Key | Meaning |
|-------------------|---|
| "file.separator" | Character that separates components of a file path. This is "/" on UNIX and "\" on Windows. |
| "java.class.path" | Path used to find directories and JAR archives containing class files. Elements of the class path are separated by a platform-specific character specified in the <code>path.separator</code> property. |
| "java.home" | Installation directory for Java Runtime Environment (JRE) |
| "java.vendor" | JRE vendor name |
| "java.vendor.url" | JRE vendor URL |
| "java.version" | JRE version number |
| "line.separator" | Sequence used by operating system to separate lines in text files |
| "os.arch" | Operating system architecture |
| "os.name" | Operating system name |
| "os.version" | Operating system version |
| "path.separator" | Path separator character used in <code>java.class.path</code> |
| "user.dir" | User working directory |
| "user.home" | User home directory |
| "user.name" | User account name |



Classe System: Proprietà di sistema (5)

- › Classe `Properties` fornisce metodi per le seguenti operazioni
 - Caricamento delle coppie chiave/valore da uno stream (es. file) ad un oggetto `Properties`
 - Recupero valore dalla sua chiave
 - Recupero di tutte le chiavi e i loro valore
 - Enumerazione sulle chiavi
 - Salvataggio delle proprietà su uno stream (es. file)



Esempio pratico



Classi Class e Reflection

- › Classe `Class` astrae il concetto di classe in Java
- › Viene utilizzata per effettuare la cosiddetta reflection (introspezione delle classi)
- › Permette la creazione di oggetti, invocazione di metodi durante l'esecuzione senza conoscere il nome della classe i nomi dei metodi, ecc.
- › Metodi `getConstructor`, `getMethods`, `getFields`, `isAnnotationPresent`. ritornano oggetti di tipo `Constructor`, `Method`, `Field`, ecc. che astraggono i concetti di costruttore, metodo, variabile



Istanziare un oggetto con Class

› Esempio

```
try {  
    Class stringa = Class.forName("java.lang.String");  
} catch (ClassNotFoundException e) {  
    e.printStackTrace();  
}
```



Esempio completo (1)

```
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public class Test {
    public static void main(String[] args) {
        try {
            Class<?> stringa = Class.forName("java.lang.String");
            Object obj = stringa.getDeclaredConstructor(String.class).newInstance("Ciao mondo");
            Method method = stringa.getDeclaredMethod("toUpperCase");
            Object result = method.invoke(obj);
            System.out.println(result);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

.....



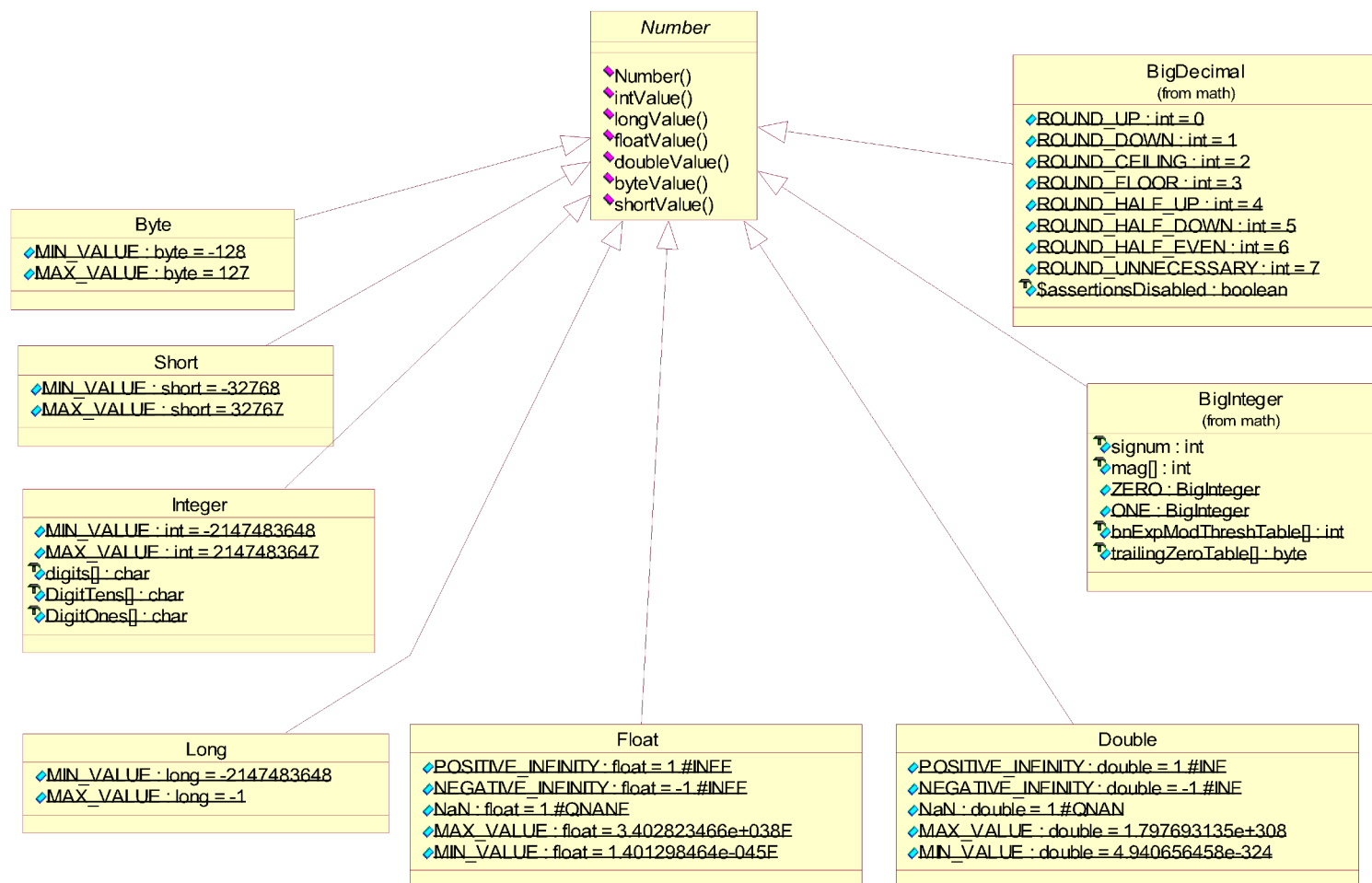
Esempio completo (2)

...

```
    catch (InstantiationException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    } catch (IllegalArgumentException e) {
        e.printStackTrace();
    } catch (InvocationTargetException e) {
        e.printStackTrace();
    } catch (NoSuchMethodException e) {
        e.printStackTrace();
    } catch (SecurityException e) {
        e.printStackTrace();
    }
}
```



Numeri come Oggetti (1)





Numeri come Oggetti (2)

› Integer

- Wrappa un valore di un tipo primitivo `int`
- Valore è **immutabile** ovvero una volta creato l'oggetto non è più possibile modificarlo
- Particolarmente utile per le collezioni (`ArrayList`, `HashMap`, ...) in quanto è possibile inserire soltanto oggetti
- Sono presenti una serie di metodi per convertire `String` in `int` e viceversa

› Esempi

```
Integer i = new Integer(10);
```

```
int i = Integer.parseInt("10");
```



Autoboxing e autounboxing (1)

- › Non è possibile inserire tipi primitivi (ad esempio `int`) all'interno delle collection (e in generale dei generics)
- › Collections possono contenere soltanto tipi **reference**
- › E' necessario utilizzare le classi Wrapper per i tipi primitivi (`Integer` per `int`)
- › Java fornisce un meccanismo automatico di conversione tra i tipi primitivi ai tipi reference Wrapper
- › Quando utilizzare l'autoboxing e unboxing?
 - Utilizzarlo soltanto quando c'è un mismatching tra tipi primitivi e tipi reference (ad esempio all'interno delle collection)
 - Inappropriato per calcolo scientifico o altro codice dove le performance numeriche sono importanti



Autoboxing e autounboxing (2)

```
import java.util.*;

public class AutoBoxing {
    public static void main(String[] args) {
        List<Integer> l = new ArrayList<>();
        for (int i=0; i<10; i++) {
            l.add(i);
        };
        for (int i : l) {
            System.out.println("elemento i-esimo" + i);
        }
    }
}
```



Autoboxing e autounboxing (3)

```
import java.util.*;

public class Frequency {
    public static void main(String[] args) {
        List<Float> list = new ArrayList<>();
        list.add(new Float(5.3F));
        float primitiveFloat = list.get(0);
        Integer i = new Integer(22);
        int j = i++;
        Integer k = (new Integer(10) + j);
        int t = k + j + i;
        Double d = 2; //Errore in compilazione. Ci vuole 2D
    }
}
```



Autoboxing e autounboxing (4)

- › Essendo i wrapper immutabili le istanze vengono poste in un pool della VM (come le stringhe)
- › Sono in questo pool
 - Tutti i tipi `byte`
 - I tipi `short` e `int` con valori compresi tra -128 e 127
 - I tipi `char` con con valori compresi tra 0 e 127
 - I tipi `boolean`
- › Inoltre

```
Boolean b = null;
```

```
boolean bb = b; //Compila ma in esecuzione NullPointerException
```



Autoboxing e autounboxing (5)

```
public class Comparison {  
    public static void main(String args[]) {  
        Integer a = 1000;  
        Integer b = 1000;  
        System.out.println(a==b);  
        Integer c = 100;  
        Integer d = 100;  
        System.out.println(c==d);  
        int e = 1000;  
        Integer f = 1000;  
        System.out.println(e==f);  
        int g = 100;  
        Integer h = 100;  
        System.out.println(g==h);  
    }  
}
```




Autoboxing e autounboxing (6)

› Overloading

```
public void metodo(Integer i);
```

```
public void metodo(float i);
```

```
metodo(123); //Quale viene invocato?
```



Autoboxing e autounboxing (6)

› Overloading

```
public void metodo(Integer i);  
public void metodo(float i);
```

```
metodo(123); //Quale viene invocato?
```

Risposta: quello con float. Per ragioni di compatibilità all'indietro



Autoboxing e autounboxing (7)

› Metodi statici di Integer, Byte, ...

```
public class Test {  
    public static void main(String args[]) {  
        int i = Integer.parseInt("100");  
        System.out.println(i);  
  
        byte b = (byte) (128);  
        System.out.println(Byte.valueOf(b));  
        System.out.println(Byte.toUnsignedInt(b));  
  
        int i1 = (int) 3000000000L;  
        System.out.println(Integer.toUnsignedString(i1, 10));  
        System.out.println(Integer.toUnsignedString(i1, 2));  
        System.out.println(Integer.toUnsignedString(i1, 8));  
        System.out.println(Integer.toUnsignedString(i1, 16));  
    }  
}
```



Annotazioni (1)

- › Meccanismo formale per aggiungere al codice informazioni utilizzabili in momenti successivi
- › Integrazione di meta-dati (informazioni su informazioni) nei file di codice sorgente invece di manipolare tali informazioni in file esterni
- › Sintassi
 - Aggiunta del simbolo @ (an annotation) con il nome dell'annotazione
- › Vengono utilizzate per diversi scopi tra i quali
 - Dal compilatore per determinare errori o sopprimere i warning
 - Processamento a tempo di **compilazione** oppure a tempo di deployment: software possono processare le annotazioni per generare codice, file xml e così via
 - Sono disponibili a tempo di **esecuzione** per essere esaminate
- › Possono essere applicate alle classi, alle variabili, ai metodi e ad altri elementi di un programma



Annotazioni (2)

- › Java fornisce seguenti annotazioni definite all'interno del package `java.lang`
- › `@Override`: utilizzato nella definizione di un metodo indica che il metodo effettua l'override della classe base; genera un errore di compilazione qualora la segnatura del metodo non è corretta
- › `@Deprecated`: è un warning al compilatore per indicare che l'elemento è deprecato
- › `@SuppressWarnings`: disattiva gli avvertimenti inadeguati
- › `@SafeVarargs`: applicato a metodo o costruttore, asserisce che il codice non esegue operazione unsafe su argomenti variabili
- › `@FunctionalInterface`: introdotto in Java 8



Annotazioni (3)

› Esempio

```
@Author (  
    name = "Benjamin Franklin",  
    date = "3/27/2003"  
)  
class MyClass {  
}
```



Annotazioni: Definizione (4)

› Esempio

```
public class Testable {  
    public void execute() {  
        System.out.println("Executing");  
    }  
  
    @Test void testExecute() {execute();}  
}
```

› Definizione

```
@Target(ElementType.METHOD)  
@Retention(RetentionPolicy.RUNTIME)  
public @interface Test{}
```



Annotazioni: Definizione (5)

› Esempio

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface UseCase {
    public int id();
    public String description default "no description";
}

public class PasswordUtils {
    @UseCase(id=47,
        description = "Passwords must be contain at least one numeric")
    public boolean validatePassword(String password) {
        .....
    }
}
```




Annotazioni: meta-annotazioni (6)

- › @Target: Indica dove può essere applicata l'annotazione. Gli argomenti possibili di `ElementType` sono
 - `ANNOTATION_TYPE`
 - `CONSTRUCTOR`
 - `FIELD`
 - `LOCAL_VARIABLE`
 - `METHOD`
 - `PACKAGE`
 - `PARAMETER`
 - `TYPE`: dichiarazione di classe, interfaccia



Annotazioni: meta-annotazioni (7)

- › `@Retention`: Indica per quanto tempo vengono mantenute le informazioni di annotazione. Gli argomenti possibili di `RetentionPolicy`
 - `SOURCE`: sono scartate dal compilatore
 - `CLASS`: vengono registrate nel file di classe del compilatore, tuttavia non devono necessariamente essere conservate dalla VM durante l'esecuzione
 - `RUNTIME`: sono conservate dalla VM durante l'esecuzione e possono essere lette mediante reflection

- › `@Documented`: Include le annotazioni nel javadoc

- › `@Inherited`: Consente alle sottoclassi di ereditare le annotazioni dal genitore



Annotazioni: meta-annotazioni (8)

- › `@Repeatable`: Introdotta in Java SE 8, indica che l'annotazione può essere applicata più di una volta
 - Valore tra parentesi è il tipo dell'annotazione contenitore che il compilatore usa per memorizzare le annotazioni ripetute

```
@Repeatable(Schedules.class)
public @interface Schedule {
    String dayOfMonth() default "first";
    String dayOfWeek() default "Mon";
    int hour() default 12;
}

public @interface Schedules {
    Schedule[] value();
}

@Schedule(dayOfMonth="last")
@Schedule(dayOfWeek="Fri", hour="23")
public void doPeriodicCleanup() { ... }
```



Esempio Junit

- › <https://github.com/junit-team/junit5>
- › <https://github.com/junit-team/junit5/blob/b41ae69659e8dc3fa230f97d8a751d81e69d06d7/junit-jupiter-api/src/main/java/org/junit/jupiter/api/Test.java>
- › <https://github.com/junit-team/junit5/blob/b41ae69659e8dc3fa230f97d8a751d81e69d06d7/junit-vintage-engine/src/main/java/org/junit/vintage/engine/discovery/IsPotentialJUnit4TestMethod.java#L16>