

Programmazione Orientata agli oggetti

Ambiente di Lavoro: Eclipse

A cura di Valerio Cetorelli

Sommario

- Introduzione
- Installazione
- Progetti
- Workspace
- Perspectives
- Views

Introduzione: IDE

- Un **IDE** (Integrated Development Environment) *open source* ideato da un consorzio di grandi società (Eclipse Foundation)
- Scritto in Java, ma utilizza librerie grafiche proprie al fine di rendere più veloce l'interazione con l'utente
 - (anche perché all'epoca il supporto alla grafica della piattaforma Java era inadeguato)

Introduzione: Cos'è Eclipse?

- L'intera piattaforma di sviluppo è incentrata sull'uso di *plug-in*: componenti software ideate per uno specifico scopo (es. la programmazione in Java)
 - Tutta la piattaforma è un insieme di plug-in che chiunque può sviluppare e/o modificare.
- Nella versione base è possibile programmare in Java, con funzioni di aiuto come:
 - completamento automatico
 - suggerimento dei tipi dei parametri dei metodi
 - riscrittura automatica del codice (*refactoring*)
 - ...
- Eclipse è disponibile per le piattaforme: Linux, macOS, Windows e non solo

Installazione

- Eclipse è molto facile da installare:
 1. scaricare la versione *Enterprise Edition* (la stessa usata in sede di esame)
<https://www.eclipse.org/downloads/packages/release/2019-12/r/eclipse-ide-enterprise-java-developers>
 2. scompattare l'archivio nella cartella che si preferisce

I Progetti

- Eclipse non è un editor, ma un ambiente di sviluppo
 - Non si può, ad esempio, aprire e compilare un qualunque file
- Bisogna organizzare il proprio lavoro in progetti
- Un progetto è l'insieme di file relativi ad una applicazione che si sta sviluppando
 - Nel caso di Java, ad esempio, il progetto conterrà non solo i file sorgenti .java, ma anche l'indicazione di quale JDK usare, eventuali librerie esterne (file .jar), etc. etc.

Workspace

- Un *workspace* è un contenitore di progetti ed è associato ad un singola cartella
 - quella in cui vengono altre cartelle, ciascuna associata a sua volta ad un singolo progetto
- È preferibile selezionare sempre una cartella esterna a quella dove è installato Eclipse
 - Si può successivamente installare e usare una nuova versione senza perdere i progetti sviluppati con la versione precedente

Perspectives

- Con Eclipse è possibile organizzare l'interfaccia grafica in base a ciò che si sta facendo
- Una *prospettiva* è un modo di vedere il progetto (cioè organizzare l'interfaccia grafica) nella maniera più comoda per uno specifico compito
 - *Programmare* nella prospettiva *Java*, *Organizzare* i file nella prospettiva *Resource*, *Debugging* in *Debug*, etc.
- Utile per avere sempre sottomano i comandi più immediatamente utili
- I numerosissimi plugin di Eclipse offrono funzionalità non solo testuali (editor, debug, ...) ma anche grafiche (come il plugin di supporto agli schemi UML)
- Per cambiare e personalizzare la prospettiva si usa il menu *Window*

Views

- Una prospettiva è composta da *viste*
- Ogni vista occupa un tab che visualizza un output o un qualche genere di struttura
 - Per visualizzare o nascondere le viste si usa *Window -> Show view.*
- Esempi di viste:
 - *Navigator*: visualizza i file contenuti in un progetto
 - *Console*: alla stregua della console riceve input e mostra l'output del programma
 - *Problems*: evidenzia i problemi di compilazione del programma


Views e Perspectives

- Le prospettive possono essere personalizzate spostando, aggiungendo o eliminando le viste.
 - In questo modo si può avere l'interfaccia più congeniale al lavoro che si sta facendo
- Le views possono essere chiuse, spostate, ridotte o ingrandite
- Con Ctrl+F6 ci si può spostare nelle tab dei file aperti nell'editor, tenendo premuto Ctrl e usando le frecce direzionali

Prima Esecuzione di Eclipse

- Come prima cosa viene chiesto di scegliere la locazione del workspace
 - Scegliere la cartella che si preferisce
- Una schermata di help propone vari informazioni e tutorial
 - Si può chiudere questa vista e riaprirla in seguito tramite il menu *Help -> Welcome*
- Durante il corso useremo la versione **1.7** di Java
 - Per impostare questa scelta di default andare su *Window -> Preferences... -> Java -> Compiler* e impostare **Compiler compliance level a 1.7.**

Occhiata all'Interfaccia

- In alto a destra è evidenziata la prospettiva corrente.
- Per cambiarla si può usare il menu *Window* o il simbolo 
- Sulla sinistra c'è la view *Package explorer*
 - Non appena creeremo dei progetti permetterà di esplorare la struttura della nostra applicazione Java
- La view *Hierarchy* serve a navigare la gerarchia delle classi (>>)
- Le view in basso hanno diverse funzioni legate all'output
- In alto c'è una toolbar che contiene comandi usati frequentemente (come *Run*)
- ESERCIZIO:
 - ✓ Passare alla prospettiva *Debug* e tornare a Java. L'interfaccia cambia radicalmente! Cosa c'è di nuovo?
 - ✓ Nella prospettiva Java non c'è ancora la vista *Console*, provare a farla comparire

Hello World! su Eclipse (1)

- Finalmente è giunto il momento di scrivere il canonico "Hello World!"
- Per creare un nuovo progetto selezionare *File -> New -> Project* e poi *Java Project*
- La schermata successiva permette di specificare il nome del progetto e altre opzioni tra cui il *Compiler Compliance Level* (impostato a 1.7 poco fa). A questo punto non ci interessano le altre opzioni e possiamo cliccare su *Finish*

Hello World! su Eclipse (2)

- A questo punto bisogna scrivere il metodo `main()`
- Un modo veloce per farlo è il seguente:
 - posizionare il cursore nella classe, indentare e scrivere "main"
 - premere ctrl-SPAZIO, Eclipse mostra una lista di "completion" possibili.
 - selezionare il primo (*main method*) e premere invio, magicamente appare il metodo `main()`
- Analogamente è possibile creare i metodi *Getter&Setter*, se necessario

Hello World! su Eclipse (3)

- Una volta terminato di scrivere il contenuto del metodo `main()` salvare il file con Ctrl+s.
- Una opzione comoda di Eclipse è la compilazione ad ogni salvataggio
 - Ulteriori configurazioni per la compilazione sono nel menu *Project*
 - Se ci sono degli errori (provare a metterne uno di proposito) vengono mostrati nella view *Problems*
- Se dopo aver scritto il codice o effettuato una modifica non si è salvato Eclipse te lo segnala con un asterisco vicino al nome nella tab


Hello World! su Eclipse (4)

- Nel Package Explorer si può vedere il nuovo progetto con al suo interno la JRE utilizzata.
- Con *File -> New -> Package* aggiungere un package chiamato **hello**
 - NB: Eclipse inserisce le classi senza package in un package virtuale chiamato *default package*.
- Con *File -> New -> Class* aggiungere una classe chiamata **HelloWorld** (specificando che fa parte del package *prova*)

Hello World! su Eclipse (5)

- Scritto il codice e compilato non resta che eseguire il programma.
- L'esecuzione è un po' meno intuitiva
 - Eclipse gestisce delle *run configurations*, ovvero dei set di impostazioni per l'esecuzione.
 - Per crearne uno al volo senza perdersi in dettagli seleziona la classe `HelloWorld` dalla view *Package Explorer* ed esegui *Run -> Run As -> Java Application*.
 - In basso nella view *Console* viene visualizzato l'output

Hello World! su Eclipse (6)

- Per far partire di nuovo l'applicazione si può ripetere la procedura o cliccare sull'icona di *Run* nella toolbar 
- Se si clicca sul triangolino nero si apre un menu a tendina con le varie opzioni di *Run* (*Run As*, *Run*, *Organize Favorites*), altrimenti viene fatta partire l'ultima applicazione eseguita

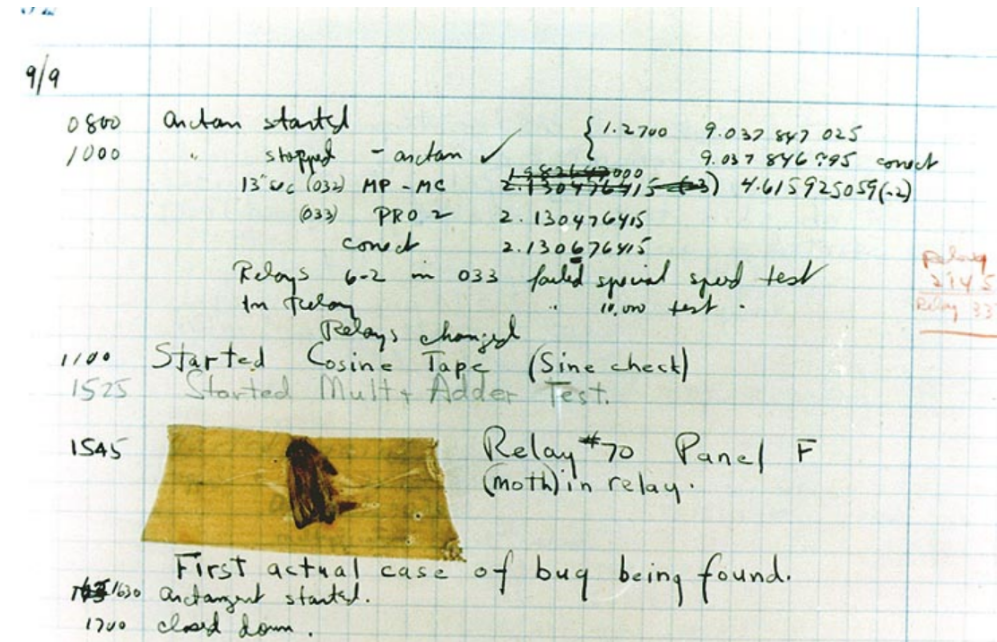
Importare Classi in Eclipse

- Crea un nuovo progetto
 1. dal menu File seleziona *Import*
 2. seleziona *general->filesystem*
 3. indicare la cartella che contiene i file da importare e sulla destra i file da importare
 4. fatto!
- In alternativa
 - copiare direttamente (dal sistema operativo) i file nella cartella del progetto (già creato) e poi aggiornare con *File->refresh*.

NB: saper fare refresh è necessario in sede d'esame

Cos'è un Debugger?

- Il debugger è un programma specificatamente progettato per l'analisi e l'eliminazione dei bug presenti in altri programmi
- L'uso del termine bug è legato ad un curioso aneddoto:
 - Nell'agosto del 1945 il tenente Hopper ed il suo gruppo stavano cercando la causa del malfunzionamento di un computer Mark II quando si accorsero che una falena si era incastrata tra i circuiti
 - Dopo aver rimosso l'insetto (in inglese *bug*), il tenente lo incollò sul registro del computer e annotò: "1545. Relay #70 Panel F (moth) nel relay. First actual case of bug being found"
 - Questo registro è conservato presso lo *Smithsonian National Museum of American History*



Classe Dummy (1)

- Supponiamo di dover scrivere una classe che offre metodi per manipolare array di stringhe
- La classe, chiamata **Dummy**, per ora deve offrire solo il metodo

```
public int search(String[] elenco, String parola)
```

- Restituisce la posizione della **String parola** nell'array **elenco** o -1 se tale **String** non è contenuta nell'array
- Il metodo **searchTest** verifica il funzionamento del metodo **search**

Classe Dummy (2)

```
package dummy;

public class Dummy {
    public int search(String[] elenco,String parola) {

        int i = 0;
        for(String stringaCorrente : elenco) {
            if(stringaCorrente == parola) {
                return i;
            }
            i++;
        }
        return -1;
    }

    public void searchTest() {

        System.out.println("test di search()");

        // CONTINUA...
```

La classe Dummy (2)

```
// inizializza l'elenco di stringhe
String[] elenco = new String[5];
elenco[0] = new String("anna");
elenco[1] = new String("carla");
elenco[2] = new String("sedia");
elenco[3] = new String("sei");
elenco[4] = new String("puma");
//verifica la presenza di tutte le parole
System.out.println("\t" + (this.search(elenco, new String("sedia")) == 2));
System.out.println("\t" + (this.search(elenco, new String("sei")) == 3));
System.out.println("\t" + (this.search(elenco, new String("carla")) == 1));
System.out.println("\t" + (this.search(elenco, new String("puma")) == 4));
System.out.println("\t" + (this.search(elenco, new String("anna")) == 0));
//verifica la mancanza di parole
System.out.println("\t" + (this.search(elenco, new String("cane")) == -1));
System.out.println("\t" + (this.search(elenco, new String("lume")) == -1));
System.out.println("\t" + (this.search(elenco, new String("gino")) == -1));
System.out.println("\t" + (this.search(elenco, new String("nota")) == -1));
}

public static void main(String[] args) {
    Dummy dummy = new Dummy();
    dummy.searchTest();
}
}
```

Errore nella Classe Dummy

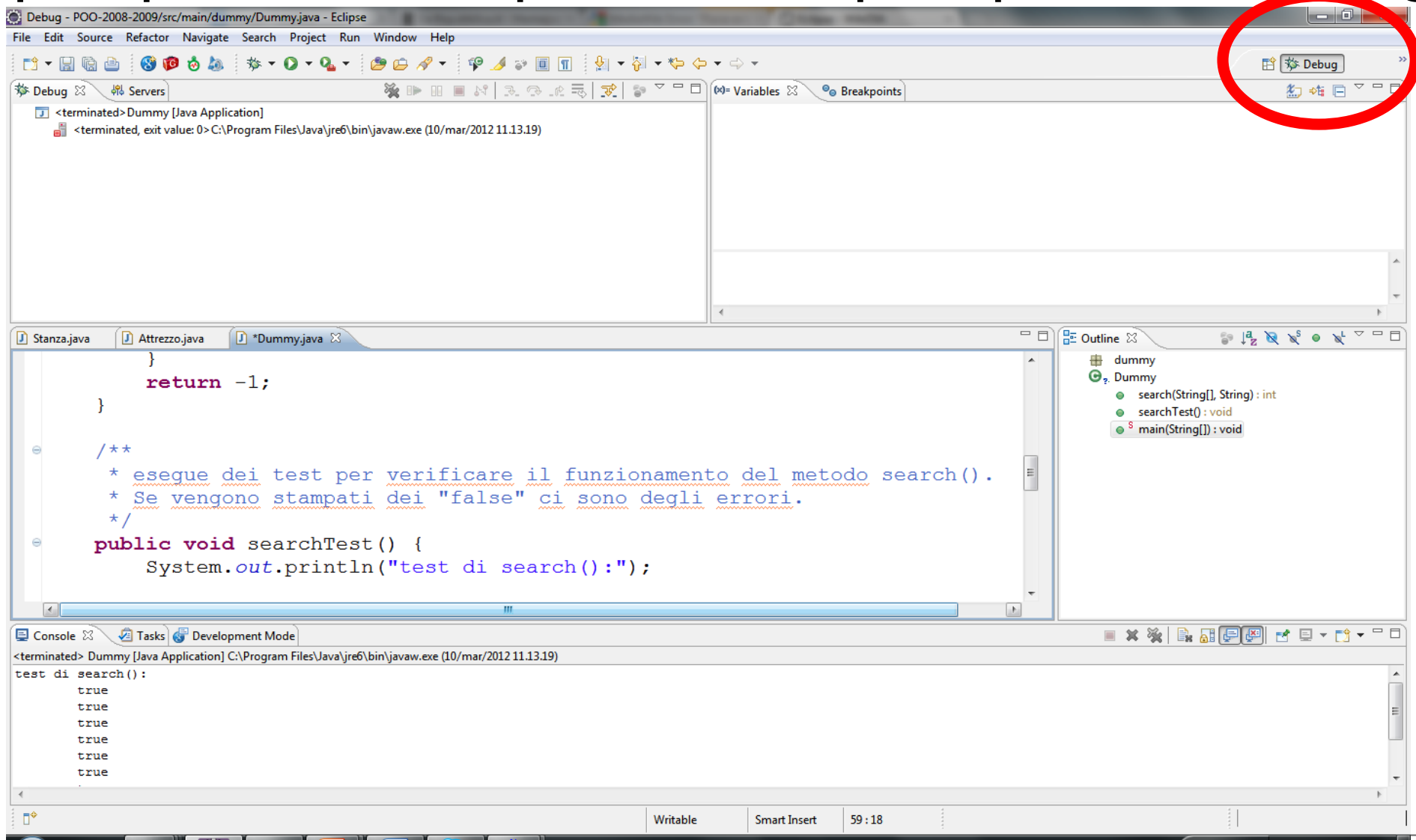
- Se tutto è stato scritto senza errori verrà stampata una sequenza di `true` che indica il successo in tutte le prove effettuate
- Se l'esecuzione del test mostra dei problemi, come nel nostro caso, siamo di fronte ad un errore logico
- Dobbiamo scovare e correggere l'errore
- Ipotezzando che il metodo di test sia stato scritto correttamente non rimane altro che controllare il metodo `search`

Il debugger di Eclipse (1)

- Un debugger offre la possibilità di eseguire il programma permettendo di:
 - far partire l'esecuzione
 - sospendere l'esecuzione e ispezionare lo stack di attivazione dei metodi (comprese le variabili)
 - eseguire solo una riga o un metodo
 - far ripartire il programma
- Per indicare al debugger dove sospendere l'esecuzione si devono indicare tramite dei **breakpoint** una o più righe di codice che (se raggiunte) bloccano il programma, permettendo al programmatore di analizzarne lo stato in cerca di errori
 - Con "impostare breakpoint alla riga x" si intende proprio questo concetto

Il debugger di Eclipse (2)

- Proviamo ora a usare il debugger di Eclipse: per prima cosa passa alla prospettiva *Debug*



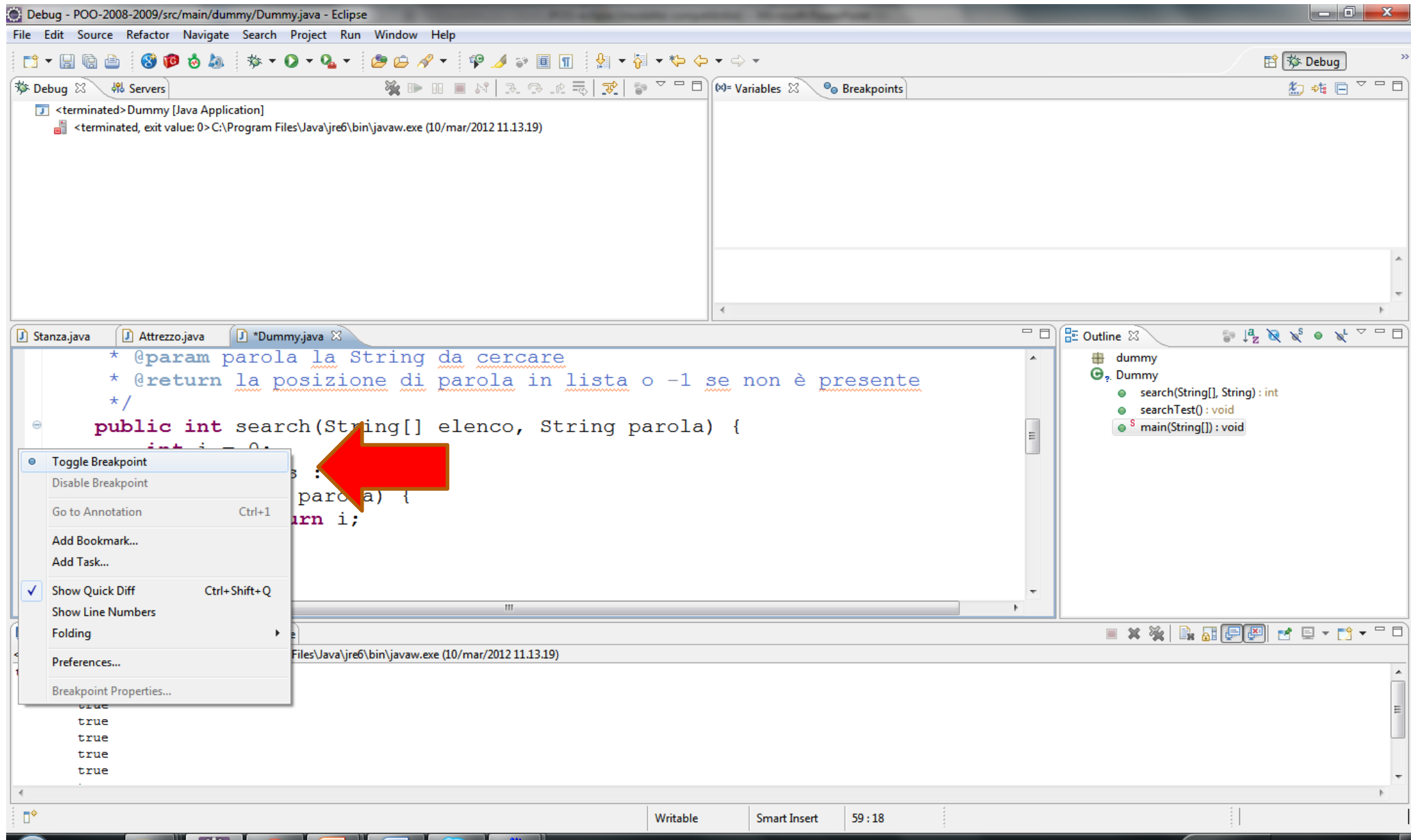
Il debugger di Eclipse (3)

- A questo punto bisogna impostare un breakpoint per poter analizzare il programma e capire dov'è l'errore
- Visto che probabilmente è nel metodo **search** è possibile impostarlo alla prima riga

```
int i = 0
```

- Per farlo si clicca col tasto destro a sinistra della riga desiderata e si seleziona *Toggle Breakpoint*

Impostare un Breakpoint



Esecuzione in Modalità Debugging (1)

- Per eseguire in modalità di debugging si esegue il programma con *Run -> Debug As... -> Java Application*
- Apparirà una schermata ad indicare che l'esecuzione è attiva, ma bloccata appena prima della riga su cui è stato impostato il breakpoint

Esecuzione in Modalità Debugging (2)

The screenshot shows the Eclipse IDE in Debug mode. The main editor displays the source code of `Dummy.java`, with the `search` method highlighted. A red arrow points to the line `int i = 0;`. The `Variables` view on the right shows the current state of the program, including the `this` object, the `elenco` array, and the `parola` variable. The `Outline` view on the right shows the class structure. The `Console` view at the bottom shows the output of the program.

Variables View:

Name	Value
this	Dummy (id=17)
elenco	String[5] (id=18)
parola	"sedia" (id=21)

Console View:

```
test di search():
```

Source Code:

```
@author Lucis
*
*/
public class Dummy {

    public int search(String[] elenco, String parola) {
        int i = 0;
        for (String stringaCorrente : elenco) {
            if (stringaCorrente == parola) {
                return i;
            }
        }
    }
}
```

Vista *Debug* di Eclipse (1)

- Nella view *Debug* (in alto a sinistra) è riportata la pila di attivazione corrente
- In *Variables* (in alto a destra) è possibile ispezionare le variabili
 - Selezionando la variabile `elenco` viene mostrato il contenuto dell'array
- In questo modo è possibile scoprire che in questa esecuzione di `search` stiamo cercando la stringa `sedia` nell'array `{"anna", "carla", "sedia", "sei", "puma"}`, quindi il metodo deve restituire il valore 2


Vista *Debug* di Eclipse (2)

- Premendo F6 il debugger esegue solo una riga di codice
- La riga corrente ora è quella successiva al breakpoint impostato e tra le variabili è comparsa la variabile `i` che non era ancora stata dichiarata
- Premendo due volte F6 si arriva alla riga
`If (stringaCorrente == parola) {`
che controlla se la stringa corrente è uguale alla stringa passata come parametro
- Ma quanto valgono queste `String`?
- Dall'ispezione di variabili parola vale `sedia` ed `stringaCorrente` vale `anna`. Ci aspettiamo che la riga successiva non sia un `return`, ma il prossimo ciclo nel `for`.
 - Funzionerà? Per scoprirlo basta premere F6...

Vista *Debug* di Eclipse (3)

- Effettivamente è andato tutto bene. Il metodo di test stampa `false` per questa attivazione di `search`, quindi l'errore sarà più avanti
- Continuare il debugging una riga alla volta fino a quando la variabile `stringaCorrente` non vale proprio `sedia`
 - quando viene eseguito l'if ci si aspetta che il metodo ritorni il valore della variabile `i` ad indicare la posizione dell'array
- Provare a farlo. Cosa succede?
- Non viene ritornato nessun valore, anzi si continua con l'esecuzione del `for`. Eppure sia la variabile `stringaCorrente` che il parametro `parola` memorizzano `sedia`

Vista *Debug* di Eclipse (4)

- L'errore logico è nel confronto tra oggetti **String** che non può essere fatto con l'operatore **==**
 - Tale operatore non verifica che due oggetti siano *equivalenti*, ma controlla che il loro indirizzo in memoria sia lo stesso (ovvero la loro *identità*)
- Quindi con due oggetti **String** che memorizzano **sedia** l'operatore **==** restituisce **false**.
- Per controllare correttamente l'equivalenza bisogna utilizzare il metodo **equals**
- Dopo aver modificato la riga del confronto utilizzando **equals** al posto di **==** non rimane che provare di nuovo ad eseguire il test
- Per interrompere il processo di debug si usa il  pulsante con il quadratino rosso (nella view *Debug*)

Vista *Debug* di Eclipse (5)

- Per usare meglio il debugger di Eclipse è necessario:
 - ✓ Capire a cosa servono i tasti F5, F7 ed F8
 - ✓ Impostare dei breakpoint durante il debug di un programma (non solo prima o dopo)