

# DEBUGGING



# DEBUGGING

- Attività di ricerca e correzione dei difetti che sono causa di malfunzionamenti.
- È l'attività consequenziale alla scoperta di un malfunzionamento. Comprende due fasi:
  - **Ricerca del difetto**
  - **Correzione del difetto**
- Il debugging è ben lungi dall'essere stato formalizzato
  - Metodologie e tecniche di debugging rappresentano soprattutto un elemento dell'esperienza del programmatore/tester



# RICERCA E LOCALIZZAZIONE DEI DIFETTI

- Ridurre la distanza tra difetto e malfunzionamento
  - Mantenendo un'immagine dello stato del processo in esecuzione in corrispondenza dell'esecuzione di specifiche istruzioni
    - Watch point e variabili di watch (Sonde)
      - Un watch, in generale, è una semplice istruzione che inoltra il valore di una variabile verso un canale di output
        - L'inserimento di un watch (sonda) è un'operazione invasiva nel codice: anche nel watch potrebbe annidarsi un difetto
        - In particolare l'inserimento di sonde potrebbe modificare sensibilmente il comportamento di un software concorrente
    - Asserzioni, espressioni booleane dipendenti da uno o più valori di variabili legate allo stato dell'esecuzione
      - Possiamo realizzare una asserzione con una interruzione programmata



# ESEMPIO DI SONDA

- Con le due `System.out.println` verifichiamo il capitale prima e dopo l'acquisto
- Avremmo potuto scrivere su di un file anzichè sullo schermo per poter avere alla fine un *log* di informazioni relative all'esecuzione

```
public float calcolaCapitale() {  
    float capitale=liquidita;  
    System.out.println(capitale);  
    for (Acquisto a : acquisti) {  
        capitale+=a.getAzione().getSocieta().getValoreAzione()*a.getQuantita();  
    }  
    System.out.println(capitale);  
    return capitale;  
}
```



# ESEMPIO DI ASSERTZIONE

- Con questo controllo vigiliamo che la liquidità non diventi negativa e in tal caso blocchiamo il programma in modo da sapere da che punto in poi le cose vanno male

```
public boolean acquista(String nomeSocieta, int quantita) {  
    boolean ok=true;  
    for (Societa s:l.getSocieta())  
        if (s.getNome().contentEquals(nomeSocieta)) {  
            if (ok) {  
                g.acquista(quantita, s.getPrezzoAzione(), s);  
                if(g.getLiquidita()<0)  
                    throw new RuntimeException("Errore: non avevi soldi per questo acquisto");  
                return true;  
            }  
        }  
    return false;  
}
```

- Problema: l'interruzione improvvisa può lasciare il database in uno stato inconsistente
- Anzichè RuntimeException potremmo utilizzare una eccezione personalizzata che fornisca dati relativi all'asserzione fallita



# AUTOMATIZZAZIONE DEL DEBUGGING

- Il debugging è un'attività estremamente intuitiva, che però deve essere operata nell'ambito dell'ambiente di sviluppo e di esecuzione del codice
- Strumenti a supporto del debugging sono quindi convenientemente integrati nelle piattaforme di sviluppo (IDE), in modo da poter accedere ai dati del programma, anche durante la sua esecuzione, senza essere invasivi rispetto al codice
  - In assenza di ambienti di sviluppo, l'inserimento di codice di debugging invasivo rimane l'unica alternativa




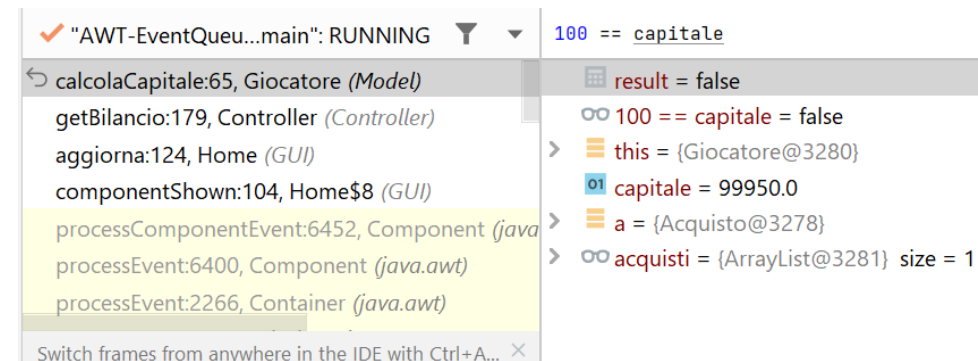
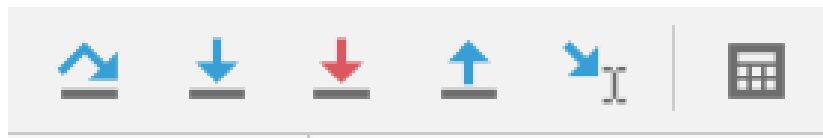
# FUNZIONALITÀ DI DEBUGGING

- Inserimento break point
  - Tramite IntelliJ è possibile attivare un breakpoint semplicemente con un click di fianco al numero della riga (si attiva un cerchio rosso che indica la presenza del breakpoint)
  - In aggiunto nel menu di contesto del tasto destroy è possibile accedere ad un completo menu con proprietà avanzate dei breakpoint
- Esecuzione passo passo del codice
  - Entrando o meno all'interno dei metodi chiamati
  - Uscendo da un metodo verso il chiamante



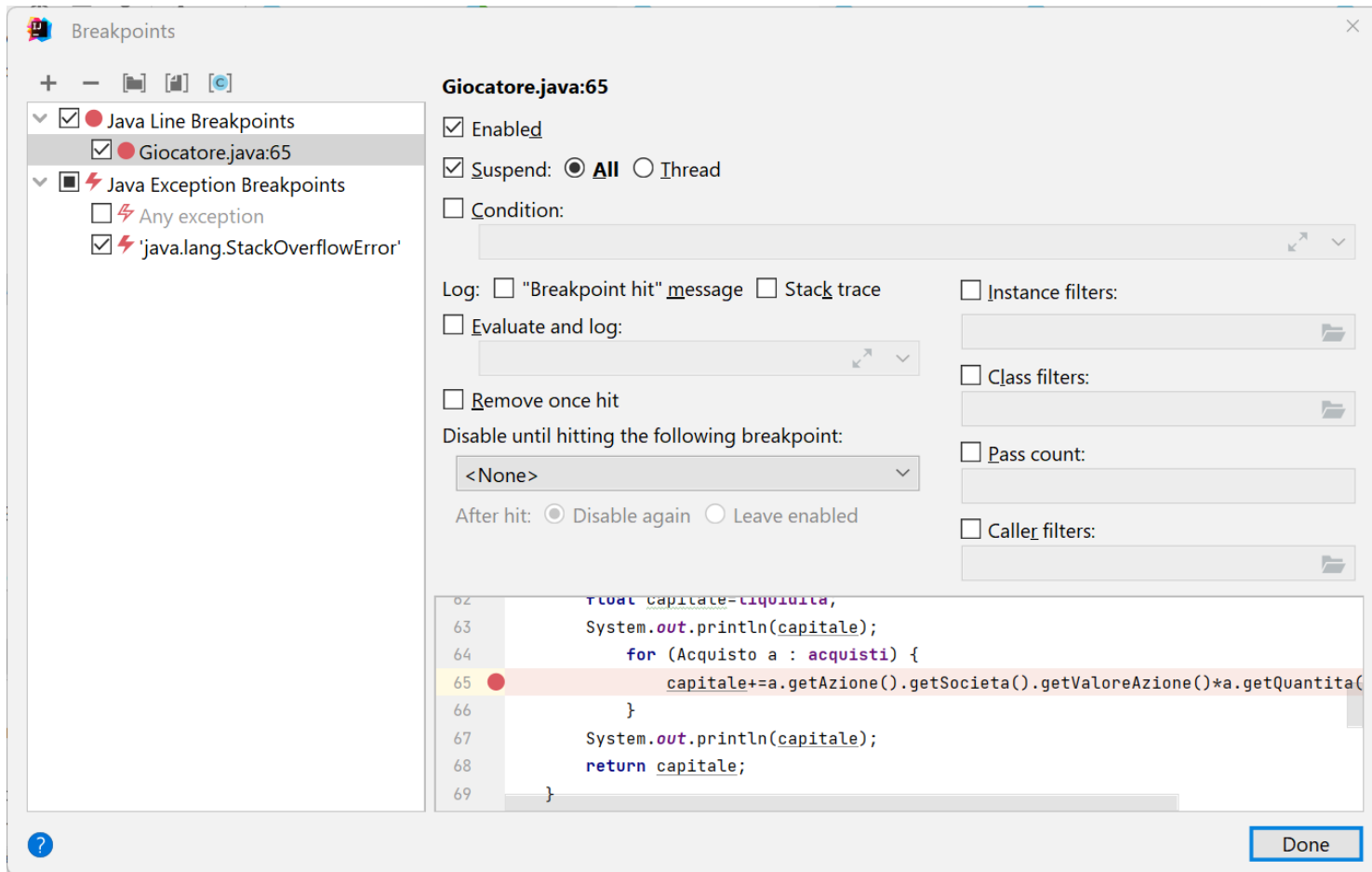
# ESECUZIONE STEP BY STEP

- Possibile quando si esegue il programma in modalità debugging 
- Quando il programma si ferma su di un breakpoint è possibile:
  - Farlo continuare (pulsante play)
  - Proseguire all'interno della funzione chiamata nella riga attuale
  - Proseguire con la riga successiva
  - Proseguire fino alla fine dell'esecuzione della funzione corrente
- Durante l'esecuzione step by step possiamo vedere facilmente i valori di tutte le variabili e chiedere di calcolare eventuali espressioni





# PROPRIETÀ AVANZATE DEI BREAKPOINT



# PROPRIETÀ AVANZATE DEI BREAKPOINT

- **Condition**
  - Il breakpoint ferma l'esecuzione solo se una certa espressione booleana è soddisfatta
    - Ad esempio se capitale è inferiore a 0
- **Log**
  - Genera un log del passaggio per il breakpoint
    - Ci permette di avere una sonda senza modificare il codice sorgente
- **Instance filters / Class filters / Caller filters**
  - Ulteriori filtri per limitare i casi nei quali il breakpoint si attiva
- **Pass count**
  - Il breakpoint si attiva solo quando viene eseguito un certo numero di volte
    - Ad esempio, se vogliamo fermare un ciclo alla decima esecuzione scriviamo 10
- **Exception Breakpoint**
  - Il breakpoint viene eseguito in seguito ad una eccezione



# WATCHPOINT

- Un watchpoint è un breakpoint collegato non ad una riga ma ad un attributo di una classe
- Si inserisce come un breakpoint, puntando sulla riga di dichiarazione dell'attributo
- Fa sospendere l'esecuzione ogni volta che l'attributo è letto (**Field Access**) o scritto (**Field Modification**)
- Tramite le Properties è possibile personalizzare quando effettuare la sospensione, con le stesse opzioni che ci sono per i breakpoint



# WATCHPOINT

