Debug con Stream

Gli Stream possono essere usati anche per effettuare debugging. Ogni operazione produce uno stream diverso rispetto a quello che ha ricevuto in input.

```
//Esempio
List numbers = Arrays.asList(2, 3, 4, 5);
numbers.stream()
                .map(x -> x + 17)
                .filter(x -> x % 2 == 0)
                .limit(3)
                .forEach(System.out::println);
// Output: 20 22
```

Se si vuole capire cosa sta succedendo durante le operazioni sopra scritte, prima di forEach (che risulta l'unica operazione utile che riesce a mostrarmi l'ooutput), allora si può usare `peek()`.

`peek()` ***non modifica lo Stream*** su cui opera ed è un'operazione che solitamente si usa solo per il debug. Quindi è possibile mostrare cosa sta succedendo nelle varie operazioni dello Stream:

```
List numbers = Arrays.asList(2, 3, 4, 5);
numbers.stream()
                .peek(x -> System.out.println("from stream: " + x))
                .map(x -> x + 17)
                .peek(x -> System.out.println("after map: " + x))
                .filter(x -> x % 2 == 0)
                .peek(x -> System.out.println("after filter: " + x))
                .limit(3)
                .peek(x -> System.out.println("after limit: " + x))
                .collect(Collectors.toList());

// Output:
// from stream: 2
// after map: 19
// from stream: 3
// after map: 20
// after filter: 20
// after limit: 20
// from stream: 4
// after map: 21
// from stream: 5
// after map: 22
// after filter: 22
// after limit: 22
```

> *Usare la `peek()` comporta delle modifiche al comportamento che ogni singolo elemento fa durante l'esecuzione. Può accadere che solo 2 (o più) fanno prima la map e poi la filter (per poi tornare alla map). Insomma, l'ordine delle operazioni è un po' più "per i fatti suoi" quando si usa la `peek()` ma comunque sia il risultato in output è sempre quello desiderato.*

Esercizi con gli Stream

1

Data una lista di stringhe {“author”, “auto”, “autocorrect”, “begin”, “big”, “bigger”, “biggish”} produrre una lista che contiene solo le stringhe che cominciano con un certo prefisso noto
Esempio: se il prefisso è “au”, la lista prodotta è {“author”, “auto”, “autocorrect”}

- Il metodo `startsWith(String prefix)` restituisce true se il parametro passato costituisce la parte iniziale della stringa
- Usare il metodo `substring(int beginIndex, int endIndex)` della classe String che restituisce la sottostringa che inizia a beginIndex e termina a endIndex
  - Esempio: “ciao” `.substring(0, 2)` restituisce “ci”

```
List<String> l1 = List.of(“author”, “auto”, “autocorrect”, “begin”, “big”, “bigger”, “biggish”);

final String pref = "aut";

List<String> res = l1.stream()
                                        .filter(s -> s.startsWith(pref))
                                        .collect(Collectors.toList());
risultato.forEach(System.out::println);

//Applicazione equivalente che produce lo stesso risultato
//List<String> risultato = l1.stream()
//                                        .filter(s -> s.startsWith(pref))
//                                        .toList();
//risultato.forEach(s -> System.out.printl(s));
```

2

Data una lista di stringhe : {“to”, “speak”, “the”, “truth”, “and”, “pay”, “your”, “debts”} produrre una stringa contenente le iniziali di ciascuna stringa della lista

- Esempio: per la lista sopra si produrrà la stringa “tsttapyd”

```
List<String> l1 = List.of(“to”, “speak”, “the”, “truth”, “and”, “pay”, “your”, “debts”);

String res = l1.stream()
        .map(s->substring(0,0))
        .reduce("", (r, v) -> r.concat(v));
```

3

Data una lista di terne di numeri interi, per ciascuna terna verificare se essa costituisce un triangolo. Restituire la lista dei perimetri per le terne che rappresentano triangoli

- In un triangolo, ciascun lato è minore della somma degli altri due
- Si può rappresentare la terna come un array di tre elementi interi
- `int[] t = new int[] { 2, 3, 4 };`
- Si può rappresentare la lista di terne come lista di array di interi `List<int[]> lista;`

```
List<int[]> lista = Arrays.asList(new int[] { 2, 2, 3 }, new int[] { 3, 2, 3 }, new int[] {
3, 3, 3 }, new int[] { 3, 4, 5 }, new int[] { 5, 2, 3 });

List<int[]> risultato = lista.stream
                                                .filter( v -> v[0] < v[1] + v[2])
                                                .filter( v -> v[1] < v[2] + v[0])
                                                .filter( v -> v[2] < v[0] + v[1])
                                                .map(v -> v[0] +v[1] +v[2]) //perimetro
                                                .toList();
```

4

• Data una lista di numeri interi
Verificare se ciascuna terna formata prendendo dalla lista tre numeri contigui costituisce un triangolo
• Esempio: lista {2, 3, 5, 7, 8}, terne {2, 3, 5}, {3, 5, 7}, {5, 7, 8}
• Restituire la lista delle terne che rappresentano triangoli
• Esempio: terne {3, 5, 7}, {5, 7, 8}

```
private List lista = List.of(2, 2, 4, 6, 3, 6, 3, 3, 4, 5);

private List verifica() {
        return IntStream.rangeClosed(0, lista.size() - 3)
                                        .mapToObj(i -> new int[]{lista.get(i),
lista.get(i+1), lista.get(i+2)})
                                        .filter(t -> t[0] < t[1] + t[2])
                                        .filter(t -> t[1] < t[0] + t[2])
                                        .filter(t -> t[2] < t[0] + t[1])
                                        .collect(Collectors.toList());
}
```

5

• Data una lista di numeri interi positivi • Verificare se la lista è ordinata
• Suggerimenti
• Si generano gli indici da 0 a n-1
• Per ciascun valore dell’indice i, si confrontano l’elemento con indice i ed il successivo, se il secondo è minore del primo la lista non è ordinata e si può fermare la verifica

- Soluzione 1
  - Ogni iterazione accede a due elementi della lista
  - L'operazione `filter` emette l'indice `i` dell'elemento che è più grande del successivo
  - Appena `filter` trova un elemento, con l'operazione `findAny` ferma la ricerca
  - **Nessuna operazione conserva uno stato globale**

```
private List<Integer> lista = Arrays.asList(2, 2, 4, 6, 12, 3);
private boolean isOrdinata() {
  return IntStream.rangeClosed(0, lista.size() - 2)
    .filter(i -> lista.get(i) > lista.get(i+1))
    .peek(v -> System.out.print(lista.get(v) + " > " + lista.get(v+1)))
    .findAny()
    .isEmpty();
}
```

- Soluzione 2
  - Si conserva uno stato che è condiviso fra varie iterazioni

```
private List<Integer> lista = Arrays.asList(2, 2, 4, 6, 3, 6, 3, 3, 4, 5);
private int prec; // conserva l'elemento precedente, e' lo stato condiviso
private boolean isOrdinata() {
   prec = lista.get(0);
   return lista.stream()
      .filter(v -> seMinoreDiPrec(v)) // modifica prec, scarta valori false
      .findAny() // si ferma quando vi e' un false
      .isEmpty();
}
private boolean seMinoreDiPrec(int x) { // non puo' eseguire in parallelo
   int p = prec;
   prec = x; // modifica lo stato
   return x < p; // ritorna true se elemento corrente > elemento prec
}
```

Considerazioni esercizio 5

- Per la soluzione 1, si ha

```
.filter(i -> lista.get(i) > lista.get(i+1))
```

- l'espressione lambda passata a `filter` legge due numeri dalla lista e dà in output un `boolean`, senza altri effetti collaterali (side-effect-free), ovvero **non modifica uno stato condiviso**. Tale espressione lambda è una **funzione pura**

- Per la soluzione 2, si ha

```
.filter(v -> seMinoreDiPrec(v))
```

ovvero

```
.filter(v -> { int p = prec; prec = v; return v < p; })
```

- l'espressione lambda passata a `map` modifica un attributo. Tale modifica è un effetto collaterale (voluto), quindi non è una funzione pura

- Si noti che: (i) `prec` è un attributo, definito al di fuori dell'espressione lambda che può essere acceduto da essa (accessi al contesto sono consentiti); (ii) la modifica di uno stato condiviso non permette l'esecuzione parallela

In generale:

- La seconda soluzione non può essere usata per una soluzione parallela (parallel) perchè modifica uno stato globale e quindi uno stato condiviso.
- La prima soluzione funziona anche per una possibile esecuzione parallela visto che non vi è alcuna modifica ad uno stato globale(condiviso)

Function<T,R>

Una map() prende in ingresso una `Function<T, R>` prende in input un T e restituisce R

```
Function<String,Integer> stringLength = x -> x.length();

int result = Stream.of("truth", "flows", "to", "them", "sweetly", "by", "nature") // ->
Stream<String>
        .map(stringLength) // -> Stream<Integer>
        .reduce(0, Integer::sum); // 31
```

Function ha al suo interno un metodo chiamato `apply()` e quando si passa l'espressione lambda viene usata proprio questa.

BiFunction

Invece, una BI-FUNCTION è una Function del tipo `BiFunction<T1, T2, R>` e prende in ingresso T1e T2 e restituisce R.

Supplier<T>

E' un'interfaccia funzionale che ha un singolo metodo chiamato `get()`. **Non prende in input nessun parametro** e **restituisce un unico valore** di tipo T.

- Applica un'espressione lambda del tipo `() -> //code`

```
Supplier sup = () -> "ciao ciao";
String s = sup.get(); // s: ciao ciao
```

- `sup` non sta tenendo una stringa ma il codice che lo genera.

Factory Method (Ripasso)

Il Factory fa un lavoro del genere:

`Prodotto p = Creator.getProdotto(“primo");`

```
public class Creator {
   public static Prodotto getProdotto(String name) {
      switch (name) {
      case "primo": return new ProdottoA();
      case "secondo": return new ProdottoB();
      case "terzo": return new ProdottoC();
      case "quarto": return new ProdottoD();
      default: return new ProdottoA();
      }
   }
}
```

Quindi il Creator sceglie quale classe istanziare.

Factory con Supplier

- Usando un Supplier

```
Supplier<Prodotto> prodSupplier = ProdottoA::new;
```

- La linea di codice sopra è equivalente a

```
Supplier<Prodotto> suppl = () -> new ProdottoA();
```

- Per avere istanze di sottotipi di Prodotto, si chiama get() sul Supplier

```
Prodotto p1 = prodSupplier.get();
```

- Quindi si crea una mappa che fa corrispondere al nome di un Prodotto la sua creazione

```
Map<String, Supplier<Prodotto>> map = Map.of("primo", ProdottoA::new, “secondo",
ProdottoB::new, "terzo", ProdottoC::new);
```

- Si usa la mappa per istanziare sottotipi di Prodotto

```
public static Prodotto getProdotto(String name) {
  Supplier<Prodotto> s = map.get(name);
  if (s != null)
     return s.get();
  return new ProdottoA();
}
```

- Il frammento di codice sopra è equivalente a

```
public static Prodotto getProdotto(String name) {
    return map.getOrDefault(name, ProdottoA::new).get();
}
```

Si usa un Supplier per ogni ConcreteProduct che servono.

- l'istanza viene creata alla chiamata di `get()`.

- Si ***evita di scrivere molte condizioni*** e ciò equivale a scrivere codice migliore
- La mappa che si crea non contiene istanze ma avrà solo la coppia (chiave, valore) dove la chiave rappresenterà il parametro che un possibile client passa al metodo in modo da sapere quale istanza creare

Se si vuole evitare anche la parte condizionale seguente:

```
if (s != null)
        return s.get();
return new ProdottoA();
```

Si può usare un metodo della Map stessa nel seguente modo:

```
public static Prodotto getProdotto(String name) {
        return map.getOrDefault(name, ProdottoA::new).get();
}
```

- Il metodo `getOrDefault()` **ritorna il valore riferito alla chiave** passata `name` altrimenti, se non esiste, **ritorna un default** definito dal programmatore

Esempio Classe Persona

```java
public class Persona {
   private String nome, ruolo;
   private int eta, costo;

   public Persona(String n, int e, String r, int c) {
      nome = n;
      eta = e;
      ruolo = r;
      costo = c;
   }

   public int getCosto() {
      return costo;
   }

   public int getEta() {
      return eta;
   }

   public String getNome() {
      return nome;
   }

   public String getRuolo() {
      return ruolo;
   }
}
```


- Data una lista di istanze di Persona, stampare e contare i nomi dei programmatori

```java
private List<Persona> team = List.of(
      new Persona("Al", 28, "Architect", 44),
      new Persona("Claire", 29, "Programmer", 38),
      new Persona("Ed", 26, "Programmer", 36),
      new Persona("Pam", 25, "Programmer", 35),
      new Persona("Ted", 32, "Tester", 40));

public void conta(String ruolo) {
   System.out.print("Hanno ruolo " + ruolo + ": ");
   long c = team.stream()
         .filter(p -> p.getRuolo().equals(ruolo))
         .peek(p -> System.out.print(p.getNome() + ", "))
         .count();
   System.out.println("\nCi sono " + c + " " + ruolo);
}
// Output
// Hanno ruolo Programmer: Claire, Ed, Pam,
// Ci sono 3 Programmer

// chiamante
conta("Programmer");
```

- Data una lista di istanze di Persona, stampare i ruoli presenti e per ciascun ruolo la lista delle persone aventi quel ruolo

```
public void scriviRuoli() {
  team.stream()
    .map(p -> p.getRuolo())
    .distinct()
    .peek(r -> System.out.print("\nRuolo " + r + ": "))
    .forEach(r -> team.stream()
                  .filter(p -> p.getRuolo().equals(r))
                  .forEach(p -> System.out.print(p.getNome() + "  ")));
}

// Output
// Ruolo Architect: Al
// Ruolo Programmer: Claire  Ed  Pam
// Ruolo Tester: Ted
```

Esempio Classe Pagamento

```
public class Pagamento {
   private Persona pers;
   private int importo;

   public Pagamento(Persona p, int v) {
      pers = p;
      importo = v;
   }

   public Persona getPers() {
      return pers;
   }

   public int getImporto() {
      return importo;
   }
}
```

- Data una lista di nomi di persona, creare la lista di istanze di Pagamento con il costo calcolato in base a ciascuna persona, e stampare i pagamenti

```
private List<String> daPagare = List.of("Pam", "Ed", "Ted");

private List<Pagamento> pagati = new ArrayList<>();

public void pagamenti() {
   pagati = team.stream()
     .filter(p -> daPagare.contains(p.getNome()))
     .map(p -> new Pagamento(p, p.getCosto() * 30))
     .peek(v -> System.out.print(v.getPers().getNome() +
                                 " " + v.getImporto() + "  "))
     .collect(Collectors.toList());
}
// Output
// Ed 1080  Pam 1050  Ted 1200
```

```java
public class BustaPaga {
   private Persona pers;
   private int totale;

   public BustaPaga(Persona p) {
      pers = p;
   }

   public void calcolaCostoBase() {
      totale = pers.getCosto() * 30;
   }

   public void aggiungiBonus() {
      totale = (int) Math.round(totale * 1.1);
   }

   public Persona getPersona() {
      return pers;
   }

   public void stampa() {
      System.out.println(pers.getNome() + "\t " + totale + " euro");
   }

   public int getImporto() {
      return totale;
   }
}
```

- Data una lista di istanze di Persona, creare una lista con istanze di BustaPaga con l'importo calcolato in base al costo di ciascuna persona, e ordinare la lista per nome persona

```
private List<BustaPaga> buste;

public void generaBustePaga() {
   buste = team.stream()
      .map(p -> new BustaPaga(p))
      .peek(b -> b.calcolaCostoBase())
      .peek(b -> b.aggiungiBonus())
      .sorted(Comparator.comparing(b -> b.getPersona().getNome()))
      .collect(Collectors.toList());
}
```

- Data la lista di istanze di BustaPaga, stampare il nome di ciascuna persona e l'importo e calcolare la somma degli importi

```
public int calcolaSomma() {
   return buste.stream()
               .peek(b -> b.stampa())
               .mapToInt(b -> b.getImporto())
               .sum();
}

// Output
// Al         1452 euro
// Claire     1254 euro
// Ed         1188 euro
// Pam        1155 euro
// Ted        1320 euro
// Totale:    6369 euro


// chiamante
System.out.println("Totale:\t " + calcolaSomma() + " euro");
```

Test

Se il test rileva degli errori dipende tutto dal tipo di esecuzione che si è avviata all'interno del test.
Se il test **non segnala errori non si può dire che il programma è corretto** ma si dice che il test non rileva errori. Questo perchè non vengono analizzate tutte le possibili esecuzioni di un programma.

Per dire che un software è corretto **si devono avere dei test esaustivi**: essi scandiscono il funzionamento per ogni tipo di esecuzione e ciò è **estremamente difficile**. Quindi si conclude che ***i test esaustivi sono impraticabili***

In genere non si ha il tempo necessario per eseguire tutti i possibili test.

- Se si avesse un metodo che prende in ingresso 2 interi, allora servirebbero circa 2^{32} valori per ogni intero.

- dovrebbe essere eseguita$2^{32} * 2^{32}$ volte, ovvero circa $1.8 * 10^{19}$ volte
- Se la funzione esegue in $1ns = 10^{-9}$s occorrono $1.8 * 10^{10}$s ovvero, essendo $1Y = 3 * 10^{7} \rightarrow 600$ anni!

Strategie di Test

Si possono scrivere test Efficaci (non esaustivi) e cercare in tutti i modi possibili dove si potrebbe annidare il difetto.
Per scrivere test efficaci, allora, ci sono delle tecniche:

- Si devono controllare gli input che si aspetta il sistema
- Si crea un ***set di input VALIDI*** che è lecito dare ad un'esecuzione di un pezzo di programma
- Si deve pensare anche a un ***set di input NON VALIDI***
- I difetti, spesso, si manifestano in gruppi e ciò comporta che bisogna indagare ulteriormente perché un codice che presenta errori, ne porta altri di seguito. Questo perché se ci sono errori, vuol dire che la soluzione messa in atto non era molto leggibile e non si è applicata la strategia migliore. Quindi ***UN CODICE COMPLICATO PRODUCE ERRORI***
- i test devono poter essere riusati ogni volta che si aggiunge nuovo codice al sorgente
- Ogni test **deve invocare un solo metodo** così che si può riusare molto facilmente. Ha solo un determinato insieme di input e un solo insieme di output stimati.
- i test **non devono dipendere** gli uni dagli altri perché altrimenti *viene meno il concetto di riusabilità* del singolo test

Strategia di test white-box (*glass-box o strutturale*)

In questo caso **si analizza il codice e le relative specifiche** per quel codice.
Quindi dal codice si capisce il tipo di test da eseguire

Strategia di test black-box

Non si guarda il codice ma solo le specifiche. Per il codice deve solo sapere i punti di aggancio, quindi solamente le chiamate che si devono fare per invocare il test. Inoltre, ovviamente, chi scrive i test, conosce anche i possibili input che si devono dare in pasto al test e i relativi output stimati.

> *Visto che non si può scandagliare 2^{64} valori per i possibili output di un codice, fra questi valori, quali si scelgono?*

Partizionamento in classi equivalenti

Il progettista, ragionando su quello che deve eseguire un metodo, sa che **un metodo si comporta allo stesso modo per una gran parte di tipo di input**.

> *Esempio: un metodo si comporta allo stesso modo se gli si passa, **come input, una stringa**. Oppure, un altro esempio di partizione è il seguente: i valori in **input da usare devono essere compresi fra (0-1000)***

Quindi ci si trova nella stessa categoria (***PARTIZIONE***) di input. Chi scrive il codice sceglie un tipo di input da tutte le partizioni che si "*vengono a definire*".

- Aver partizionato concede **molti vantaggi nella scelta degli input** per un determinato test.
- Se si nota che l'algoritmo si comporta in maniera diversa con un altro tipo di input, allora si va a **scegliere un input di una diversa partizione**

Si parla di **BOUNDED-CONDITION** ovvero condizioni racchiuse in un certo partizionamento

Chi fa il test deve considerare sia classi di equivalenza ***VALIDE*** che classi di equivalenza ***NON VALIDE***. Una classe di equivalenza non valida rappresenta input inaspettati o errati.

I **test si devono eseguire con gli input provenienti dalle ZONE LIMITE** di un loro partizionamento:

> *Esempio: Se il partizionamento degli input è un range fra {0-100} allora si faranno test con i seguenti possibili input: (-1, 0, 1) e (99, 100, 101)*

Dobbiamo valutare più percorsi di testing per effettuare il testing per ogni percorso.

Test del percorso

In fase di test si deve considerare anche la possibilità di prendere più percorsi. Per tale motivo si deve fare un test per ogni possibile percorso.
La ***complessità ciclomatica*** dà il numero di test di percorso che si devono fare.

> *In input, comunque, devono essere usati input che creano esclusivamente un percorso evitando cosi che si copra un percorso 2 volte.*

La complessità ciclomatica cc = numero_archi - numero_nodi +2

Test sotto stress

Per gli input ***PIU' PROBABILI*** e le esecuzioni del programma più frequenti si devono sicuramente scrivere dei test.

Si devono eseguire dei test anche sotto stress ovvero in situazioni limite del software:

> *Se la memoria della macchina si sta esaurendo, il sistema riesce comunque a funzionare o si blocca? E se il disco è pieno?*
> *Se nel software ci sono 100 record funziona. Ma se ce ne sono 1000000000 funziona allo stesso modo?*

Copertura del codice

Se si combinano molti pezzi di codice, si ha una combinazione di flussi esponenziale.

La copertura (***COVERAGE***) indica la percentuale di codice eseguito rispetto al totale del programma.

Più copertura si ha, più il codice è stato testato. Si deve capire se c'è una maggiore copertura del codice se si implementa un nuovo test, altrimenti vorrà dire che il test è sbagliato e ripercorro un test già percorso.

Una copertura 100% non significa i test sono esaustivi ma che si ha una copertura con determinati parametri in input (*quindi non tutti i possibili*)

Si deve **arrivare almeno al 60% di code coverage** di solito per considerare il codice "testato bene"

Criteri di copertura del codice

Alcuni criteri sono:

- Si devono coprire con test tutte le funzioni
- si devono testare tutte le linee di codice (copertura del costrutto)

- Se tutto il sistema è come un grafo, si devono testare tutti i possibili archi del grafo, quindi tutte le possibili chiamate all'interno del sistema

> *Esempio: **un costrutto condizionale** deve essere attraversato usando **2 input diversi** che logicamente percorrono il costrutto con una **condizione vera** e una **condizione falsa***

Copertura delle decisioni

Si devono **coprire anche tutte le possibili condizioni** (*se per caso ce ne sono più di una*) quindi devono usare tutti i possibili input usati come condizione

Trend dei difetti scoperti

Quando si eseguono dei test si scopre se il codice è corretto.
Un criterio per continuare a scrivere test è la copertura: più copertura si ha, più si è soddisfatti e vuol dire che il sistema è stato testato più sufficientemente

Un altro criterio è IL TREND DEI DIFETTI: si calcola il numero di difetti che man mano si scoprono con le scritture dei test.

> *Scrivo 100 test, quanti difetti scopro?*

Man mano che scopro i difetti (e li correggo man mano), il residuo dei difetti presenti è difficile da rivelare.
Pertanto, il criterio è: ***man mano che si aggiungono test, il numero di difetti deve decrescere***
stabilendo una soglia e quando raggiungo il minimo definito dalla soglia, ci si ferma con la stesura di test.