



UNIVERSITÀ DEGLI STUDI DI SALERNO
DIPARTIMENTO DI INFORMATICA
DIPARTIMENTO DI ECCELLENZA

Università degli Studi di Salerno

Dipartimento di Informatica

Programmazione ad Oggetti

a.a. 2022-2023

Espressioni Lambda

Docente: Prof. Massimo Ficco

E-mail: mficco@unisa.it

Le espressioni lambda

- Le **espressioni lambda** sono una importante funzionalità inclusa in Java SE 8.
- Offre meccanismi per supportare la programmazione in maniera funzionale.
- In matematica e informatica in generale, un'espressione lambda è una funzione.
- In Java, un'espressione lambda fornisce un modo per creare una funzione anonima, che può quindi essere passato come argomento o restituito in uscita nei metodi,
- Si tratta di un metodo senza una dichiarazione, una sorta di scorciatoia che consente di scrivere un metodo nello stesso posto dove ti serve.



Le espressioni lambda

- In generale le lambda ci permettono di scrivere codice più chiaro e meno verboso.
- Le lambda expressions sono particolarmente utili nei casi in cui serve definire una breve funzione che ha poche linee di codice e che verrà utilizzata una sola volta. Si risparmia di scrivere un metodo a parte con modificatore, nome, ecc.

..... ma prima di passare al dettaglio sulle lambda expression è necessario ricordare alcuni concetti del linguaggio Java...

Parliamo delle classi anonime interne (anonymous inner classes) e le interfacce funzionali (functional interfaces).



Calssi Anonime Interne

- In Java, le classi anonime interne forniscono un modo per implementare classi che vengono utilizzate una sola volta in un'applicazione.
- Un tipico uso di queste classi lo possiamo vedere in un'applicazione con interfaccia basata su swing o un'applicazione JavaFX in cui è richiesto un numero di gestori di eventi (Event Handler) per gli eventi di tastiera e mouse.



Calssi Anonime Interne

File *ClickListener.java*

```
public interface ActionListener
{
    void actionPerformed(ActionEvent event);
}
```

```
01: import java.awt.event.ActionEvent;
02: import java.awt.event.ActionListener;
03:
04: /**
05:     An action listener that prints a message.
06: */
07: public class ClickListener implements ActionListener
08: {
09:     public void actionPerformed(ActionEvent event)
10:     {
11:         System.out.println("I was clicked.");
12:     }
13: }
```

Tale classe per poter essere utilizzata da JButton deve implementare l'interfaccia ActionListener



Calssi Anonime Interne

File *Button2.java*

```
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;

public class Button2 extends JFrame {
    private JButton b1 = new JButton("Button 1");
    JButton b2 = new JButton("Button 2");
    private JTextField txt = new JTextField(10);

    public Button2(){
        super("2 pulsanti");
        setSize(400,300);
        JPanel p=new JPanel();
        ActionListener listener = new ClickListener();
        b1.addActionListener(listener);
        b2.addActionListener(new ClickListener());
        p.add(b1); p.add(b2); p.add(txt);}

    public static void main(String[] args){
        JFrame frame=new Button2();
        frame.show();
    }
}
```

Bisogna istanziare un ActionListener
e aggiungerlo al bottone



Calssi Anonime Interne

- In Java, le classi anonime interne forniscono un modo per implementare classi che vengono utilizzate una sola volta in un'applicazione.
 - Un tipico uso di queste classi lo possiamo vedere in un'applicazione con interfaccia basata su swing o un'applicazione JavaFX in cui è richiesto un numero di gestori di eventi (Event Handler) per gli eventi di tastiera e mouse.
- Piuttosto che scrivere una classe di gestione degli eventi separata per ogni evento, è possibile scrivere qualcosa del genere:

```
JBUTTON testButton = new JButton("Test Button");
testButton.addActionListener(new ActionListener(){
    @Override
    public void actionPerformed(ActionEvent ae){
        System.out.println("Test Button clicked")
    }
});
```



Calssi Anonime Interne

- Se non si facesse così sarebbe necessario scrivere n classi che implementano ActionListener, una per ogni evento da gestire.
- In questo modo invece:
 - definendo la classe dove serve;
 - il codice risulta più facile da leggere.



Classi Interne Anonime

- ▶ Le classi interne anonime offrono la possibilità di implementare una classe usata una sola volta senza dover realizzare un apposito file Java. La classi anonime implementano una interfaccia senza fornire un nome a questa implementazione. Sebbene sono più concise di una classe con nome, per le classi con un solo metodo, anche la classe anonima è vista problematica ed eccessiva, va scritto un po' di codice anche solo per definire un unico metodo.

```
interface HelloWorld {  
    public void greet();  
    public void greetSomeone(String someone);  
}
```

```
HelloWorld frenchGreeting = new HelloWorld() {  
    String name = "tout le monde";  
    public void greet() {  
        greetSomeone("tout le monde");  
    }  
    public void greetSomeone(String someone) {  
        name = someone;  
        System.out.println("Salut " + name);  
    }  
};
```

Interfacce funzionali

- Le interfacce come *ActionListener*, vengono chiamate in Java 8, **interfacce funzionali** (functional interface) e sono caratterizzate dalla presenza di un solo metodo.
- L'utilizzo di interfacce funzionali con le classi anonime interne sono un modello comune in Java.
- Oltre alle classi *EventListener*, interfacce come *Runnable*, *Comparator*, *Consumer*, *FileFilter*, *Predicate*, ... sono da considerarsi in modo simile.

```
public interface Predicate<T> {  
    boolean test(T t); }  
  
public interface Consumer<T>{  
    void accept(T t); }  
  
public interface Comparable<T>{  
    void compareTo(T t); }
```



Interfacce funzionali

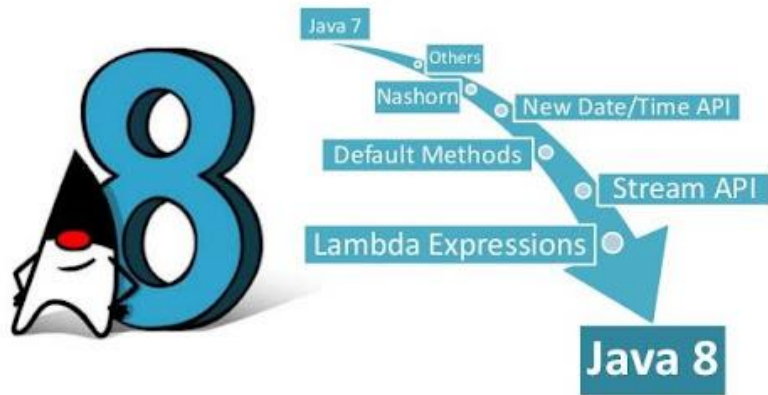
```
package java.awt.event;  
import java.util.EventListener;
```

```
public interface ActionListener extends EventListener {  
    public void actionPerformed(ActionEvent e);  
}
```

- Le interfacce funzionali sono sfruttate per l'utilizzo con le espressioni lambda.



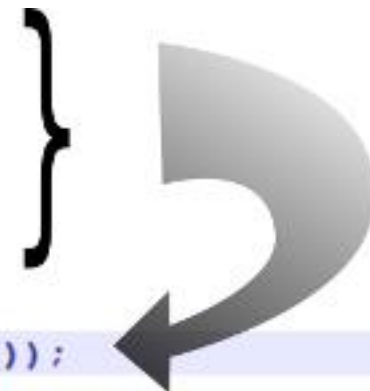
Espressioni Lambda



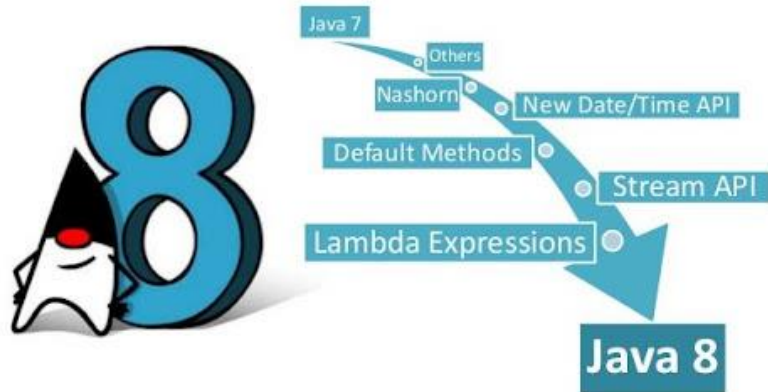
Le **Espressioni Lambda** sono una nuova e importante funzionalità inclusa in Java 8 e consentono allo sviluppatore di rendere una funzionalità come un argomento di metodo o il codice come un dato, con una sintassi migliore e più compatta rispetto alle classi interne anonime.

```
stuff.sort(new Comparator<Person>() {  
    @Override  
    public int compare(Person o1, Person o2) {  
        return o1.getAge() - o2.getAge();  
    }  
});
```

```
stuff.sort((o1, o2) -> o1.getAge() - o2.getAge());
```

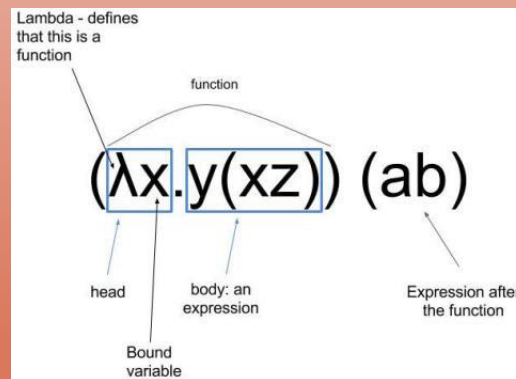


Espressioni Lambda



Le **Espressioni Lambda** sono una nuova e importante funzionalità inclusa in Java 8. Consentono allo sviluppatore di rendere una funzionalità come un argomento di metodo o il comportamento di un dato con una sintassi

Derivano dal Lambda Calculus, un sistema formale sviluppato per analizzare formalmente le funzioni e il loro calcolo.



```
stuff.sort(new Comparator<Person>() {  
    @Override  
    public int compare(Person o1, Person o2) {  
        return o1.getAge() - o2.getAge();  
    }  
});
```

```
stuff.sort((o1, o2) -> o1.getAge() - o2.getAge());
```



Espressioni Lambda

- Un'espressione lambda è come un metodo (anonimo), fornisce un elenco di parametri formali e un corpo (che può essere un'espressione o un blocco di codice).
- Le espressioni lambda risolvono il problema della verbosità delle classi interne permettendo una riduzione delle linee di codice da scrivere. La sintassi generale è la seguente:

(Lista degli argomenti) -> Espressione

oppure

(Lista degli argomenti)->{ istruzioni; }



Espressioni Lambda

```
// espressione che prende in input due interi e restituisce la somma  
(int x, int y) -> x + y
```

```
// espressione che prende in input una stringa e restituisce la sua lunghezza  
s -> s.length()
```

```
// espressione senza argomenti che restituisce il valore 50  
() -> 50
```

```
// espressione che prende in input una stringa e non restituisce nulla  
(String s) -> { System.out.println("Benvenuto ");  
                System.out.println(s); }
```

Nota: in un'espressione lambda è possibile omettere il tipo dei parametri.



Espressioni Lambda

ActionListener visto prima :

```
public class ListenerTest {  
  
    public static void main(String[] args) {  
  
        // Anonymous ActionListener  
        JButton testButton = new JButton("Test Button");  
        testButton.addActionListener(new ActionListener(){  
            @Override public void actionPerformed(ActionEvent ae){  
                System.out.println("Click Detected by Anon Class");  
            }  
        });  
  
        // Lambda ActionListener  
        testButton.addActionListener(e -> System.out.println("Click Detected by Lambda Listener"));  
  
        // Swing stuff  
        JFrame frame = new JFrame("Listener Test");  
        . . .  
    }  
}
```

Da notare in questo ultimo caso come la lambda expression sia passata come argomento di un metodo.



Possibili Usi

- Supponiamo di implementare un'applicazione di social networking, e si vuole consentire ad un amministratore di realizzazione ogni tipo possibile di azione sui membri dell'applicazione che soddisfano un determinato criterio.

Campo	Descrizione
Nome	Realizza un'azione di membri selezionati.
Attore Primario	Amministratore.
Precondizioni	L'amministratore è loggato nel sistema.
Postcondizioni	L'azione è realizzata sui membri che soddisfano una determinata condizione.
Principale scenario di successo	<ol style="list-style-type: none">1. L'amministratore specifica la condizione di scelta dei membri;2. L'amministratore specifica l'azione da realizzare;3. L'amministratore seleziona il bottone Submit;4. Il sistema trova tutti i membri che soddisfano la condizione;5. Il sistema esegue l'azione sui membri selezionati.
Estensioni	
Occorrenze	Molte volte durante il giorno.



Possibili Usi

- Ipotizziamo che tutti i membri nell'applicazione siano rappresentati dalla seguente classe Person e memorizzati in una List<Person>:

```
public class Person {  
    public enum Sex { MALE, FEMALE };  
    String name;  
    LocalDate birthday;  
    Sex gender;  
    String emailAddress;  
    public int getAge() {    // ...    }  
    public void printPerson() {    // ...    }  
}
```

Come implementare la caratteristica dell'applicazione?



Possibili Usi

- ▶ **Approccio 1: Creare metodi che identificano i membri che soddisfano una caratteristica**
- ▶ Una soluzione semplicistica è di creare vari metodi, ognuno cerca per i membri che soddisfano una data caratteristica, come il sesso o l'età. Il seguente stampa a video i membri che sono più vecchi di una data età:

```
public static void printPersonsOlderThan(List<Person> roster, int age) {  
    for (Person p : roster) {  
        if (p.getAge() >= age) {  
            p.printPerson();  
        }  
    }  
}
```



Possibili Usi

- ▶ **Approccio 1: Creare metodi che identificano i membri che soddisfano una caratteristica**
- ▶ Una soluzione semplicistica è di creare vari metodi, ognuno cerca per i membri che soddisfano una data caratteristica, come il sesso o l'età. Il seguente stampa a video i membri che sono più vecchi di una data età:

```
public static void printPersonsOlderThan(List<Person> roster, int age) {  
    for (Person p : roster) {  
        if (p.getAge() >= age) {  
            p.printPerson();  
        }  
    }  
}
```

Questa soluzione non è ottimale perchè rende la nostra applicazione precaria.



Possibili Usi

- ▶ **Approccio 1: Creare metodi che identificano i membri che soddisfano una caratteristica**
- ▶ Una soluzione semplicistica è di creare vari metodi, ognuno cerca per i membri che soddisfano una data caratteristica, come il sesso o l'età. Il seguente stampa a video i membri che sono più vecchi di una data età:

```
public static void printPersonsOlderThan(List<Person> roster, int age) {  
    for (Person p : roster) {
```

L'applicazione non funziona se si hanno modifiche alla classe Person o nuovi tipi di dati. Bisogna riscrivere varie parti dell'API se Person contiene nuovi campi, o se l'età è misurata in maniera diversa. Inoltre, la soluzione è troppo restrittiva in maniera non necessaria.

```
    }  
    Se vogliamo stampare i membri più giovani va modificato il metodo.
```



Possibili Usi

- ▶ **Approccio 2: Creare dei metodi di ricerca più generalizzati**
- ▶ Una soluzione differente è di avere dei metodi di ricerca più generalizzati, che stampano i membri all'interno di un intervallo d'età specificato:

```
public static void printPersonsWithinAgeRange(  
    List<Person> roster, int low, int high) {  
    for (Person p : roster) {  
        if (low <= p.getAge() && p.getAge() < high) {  
            p.printPerson();  
        }  
    }  
}
```



Possibili Usi

- ▶ **Approccio 2: Creare dei metodi di ricerca più generalizzati**
- ▶ Una soluzione differente è di avere dei metodi di ricerca più generalizzati, che stampano i membri all'interno di un intervallo d'età specificato:

```
public static void printPersonsWithinAgeRange(  
    List<Person> roster, int low, int high) {  
    for (Person p : roster) {  
        if (low <= p.getAge() && p.getAge() < high) {  
            p.printPerson();  
        }  
    }  
}
```

Sebbene più generica, anche questa soluzione non è ottimale.



Possibili Usi

- ▶ **Approccio 2: Creare dei metodi di ricerca più generalizzati**
- ▶ Una soluzione differente è di avere dei metodi di ricerca più generalizzati, che stampano i membri all'interno di un intervallo d'età specificato:

```
public static void printPersonsWithinAgeRange(  
    List<Person> roster, int low, int high) {  
    for (Person p : roster) {  
        if (low <= p.getAge() && p.getAge() < high) {
```

Se il criterio di ricerca è differente bisogna implementare un altro metodo generico, inoltre risulta complicato realizzare la combinazione tra due o più criteri di ricerca.

```
}
```



Possibili Usi

- ▶ **Approccio 3: Specificare il criterio di ricerca in una classe locale**
- ▶ Una migliore soluzione è quella di separare il codice che specifica il criterio di ricerca dall'azione da eseguire sui membri selezionati:

```
public static void printPersons(  
    List<Person> roster, CheckPerson tester) {  
    for (Person p : roster) {  
        if (tester.test(p)) {  
            p.printPerson();  
        }  
    }  
}
```



Possibili Usi

- ▶ **Approccio 3: Specificare il criterio di ricerca in una classe locale**
- ▶ Una migliore soluzione è quella di separare il codice che specifica il criterio di ricerca dall'azione da eseguire sui membri selezionati:

```
public static void printPersons(  
    List<Person> roster, CheckPerson tester) {  
    for (Person p : roster) {  
        if (tester.test(p)) {  
            p.printPerson();  
        }  
    }  
}
```

Questo metodo controlla ogni membro della lista di Person se soddisfa o meno il criterio di ricerca specificato nel parametro, invocando il suo metodo test(). Se il metodo restituisce true, allora l'azione viene eseguita.



Possibili Usi

- ▶ **Approccio 3: Specificare il criterio di ricerca in una classe locale**
- ▶ Una migliore soluzione è quella di separare il codice che specifica il criterio di ricerca dall'azione da eseguire sui membri selezionati:

```
public static void printPersons(  
    List<Person> roster, CheckPerson tester) {  
    for (Person p : roster) {  
        if (tester.test(p)) {  
            p.printPerson();  
        }  
    }  
}
```

Il criterio di ricerca è codificato in una classe che implementa l'interfaccia CheckPerson:

```
interface CheckPerson {  
    boolean test(Person p);  
}
```



Possibili Usi

- Ipotizziamo di voler implementare una condizione di ricerca in cui le istanze di Person sono eligibili per il Selective Service negli Stati Uniti: il metodo test() restituisce true se l'istanza è un membro maschile di età tra 18 e 25:

```
class CheckPersonEligibleForSelectiveService implements CheckPerson {  
    public boolean test(Person p) {  
        return p.gender == Person.Sex.MALE &&  
            p.getAge() >= 18 &&  
            p.getAge() <= 25;  
    }  
}
```



Possibili Usi

- Ipotizziamo di voler implementare una condizione di ricerca in cui le istanze di Person sono elegibili per il Selective Service negli Stati Uniti: il metodo test() restituisce true se l'istanza è un membro maschile di età tra 18 e 25:

```
class CheckPersonEligibleForSelectiveService implements CheckPerson {  
    public boolean test(Person p) {  
        return p.gender == Person.Sex.MALE &&  
            p.getAge() >= 18 &&  
            p.getAge() <= 25;  
    }  
}
```

Per usare questa classe basta istanziarla e passarne l'oggetto al metodo printPersons:

```
printPersons(  
    roster, new CheckPersonEligibleForSelectiveService());
```

Possibili Usi

- Ipotizziamo di voler implementare una condizione di ricerca in cui le istanze di Person sono elegibili per il Selective Service negli Stati Uniti: il metodo test() restituisce true se l'istanza è un membro maschile di età tra 18 e 25:

```
class CheckPersonEligibleForSelectiveService implements CheckPerson {  
    public boolean test(Person p) {  
        return p.gender == Person.Sex.MALE &&  
            p.getAge() >= 18 &&  
            p.getAge() <= 25;  
    }  
}
```

Sebbene questa soluzione rende l'applicazione meno instabile, richiede del codice aggiuntivo: un'interfaccia e una **classe locale** per ogni criterio di ricerca.



Possibili Usi

- ▶ **Approccio 4: Specificare il criterio di ricerca in una classe anonima**
- ▶ Rispetto alla soluzione precedente, invece di avere una classe locale per ogni criterio di ricerca, è possibile avere un'implementazione dell'interfaccia CheckPerson come una classe anonima:

```
printPersons( roster, new CheckPerson() {  
    public boolean test(Person p) {  
        return p.getGender() == Person.Sex.MALE  
            && p.getAge() >= 18  
            && p.getAge() <= 25;  
    }  
});
```



Possibili Usi

- ▶ **Approccio 4: Specificare il criterio di ricerca in una classe anonima**
- ▶ Rispetto alla soluzione precedente, invece di avere una classe locale per ogni criterio di ricerca, è possibile avere un'implementazione dell'interfaccia CheckPerson come una classe anonima:

```
printPersons( roster, new CheckPerson() {  
    public boolean test(Person p) {
```

Questa soluzione riduce il codice necessario per l'implementazione dell'interfaccia del criterio di ricerca, ma la sintassi è ingombrante e ridondante considerando che si deve implementare un solo metodo.

```
    }  
}
```



Possibili Usi

- Approccio 5: Specificare il criterio di ricerca con una espressione lambda
- L'interfaccia CheckPerson è detta **interfaccia funzionale**, dal momento che contiene un solo metodo astratto (ne può contenere molteplici di default o statici). Per questo caso è possibile ometterne il nome quando la si implementa. Per fare ciò, invece di una classe anonima, si usano le espressioni lambda:

```
printPersons(  
    roster,    (Person p) -> p.getGender() == Person.Sex.MALE  
                && p.getAge() >= 18  && p.getAge() <= 25  
);
```



Possibili Usi

- **Approccio 5: Specificare il criterio di ricerca con una espressione lambda**
- L'interfaccia CheckPerson è detta **interfaccia funzionale**, dal momento che contiene un solo metodo astratto (ne può contenere molteplici di default o statici). Per questo caso è possibile ometterne il nome quando la si implementa. Per fare ciò invece di una classe anonima si usano le espressioni lambda.

Quali tra queste sono interface funzionali?

```
public interface Adder {  
    int add(int a, int b);  
}  
  
public interface SmartAdder extends Adder {  
    int add(double a, double b);  
}  
  
public interface Nothing {  
    ...  
}
```



Possibili Usi

- **Approccio 5: Sp** **pressione lambda**
- **Ch** **dal momento che**
- in sol** **olteplici di default o**
- statici). Per que** **quando la si**
- implementa. Per fare ciò invece di una classe anonima si usano le espressioni**
- lamb**
- Quali tra queste sono interface funzionali?

```
public interface Adder {  
    int add(int a, int b);  
}  
  
public interface SmartAdder extends Adder {  
    int add(double a, double b);  
}  
  
);  
  
public interface Nothing {  
}
```



Possibili Usi

- **Approccio 5: Specificare il criterio di ricerca con una espressione lambda**
- L'interfaccia CheckPerson è detta **interfaccia funzionale**, dal momento che contiene un solo metodo astratto (ne può contenere molteplici di default o statici). Per questo caso è possibile ometterne il nome quando la si implementa. Per fare ciò, invece di una classe anonima, si usano le espressioni lambda:

```
printPersons(  
    roster, (Person p) -> p.getGender() == Person.Sex.MALE  
            && p.getAge() >= 18 && p.getAge() <= 25  
);
```

Espressione Lambda per
l'interfaccia CheckPerson.



Possibili Usi

- ▶ **Approccio 6: Specificare il criterio di ricerca con una espressione lambda e per mezzo di Interfacce Funzionali Standard**
- ▶ È possibile ridurre ulteriormente il codice necessario sostituendo all'interfaccia CheckPerson una funzionale standard:

```
interface CheckPerson {  
    boolean test(Person p);  
}
```



Possibili Usi

- ▶ **Approccio 6: Specificare il criterio di ricerca con una espressione lambda e per mezzo di Interfacce Funzionali Standard**
- ▶ È possibile ridurre ulteriormente il codice necessario sostituendo all'interfaccia CheckPerson una funzionale standard:

```
interface CheckPerson {  
    boolean test(Person p);  
}
```

Questa interfaccia è molto semplice, e il metodo che definisce non è particolarmente articolato (richiede un parametro e restituisce un valore booleano).



Possibili Usi

- **Approccio 6: Specificare il criterio di ricerca con una espressione lambda e per mezzo di Interfacce Funzionali Standard**

- È possibile ridurre il codice di esempio precedente Posso evitare di definire una propria interfaccia per l'applicazione di interesse, ma impiegare una di quelle generiche che mette a disposizione Java in **java.util.function**.

```
interface CheckPerson {  
    boolean test(Person p);  
}
```

```
interface Predicate<T> {  
    boolean test(T t);  
}
```

- Possiamo quindi modificare la firma del metodo di stampa a video, usando **Predicate<T>** al posto di **CheckPerson**.



Possibili Usi

La firma del metodo astratto di una interfaccia funzionale descrive la firma dell'espressione lambda.

La notazione () -> void rappresenta una funzione con una lista vuota di parametri che ritorna void, ed è ciò che rappresenta l'interfaccia Runnable.

Le espressioni lambda sono controllate nel tipo dal compilatore Java. Possono essere assegnate ad una variabile o passate a un metodo che richiedono una interfaccia funzionale come orgaomento, a condizione che l'espressione lambda ha la stessa firma del metodo astratto dell'interfaccia.

```
interface Predicate {  
    boolean test(Person p);  
}
```

```
interface Predicate<T> {  
    boolean test(T t);  
}
```



- Possiamo quindi modificare la firma del metodo di stampa a video, usando Predicate<T> al posto di CheckPerson.



Possibili Usi

```
public static void printPersonsWithPredicate(  
    List<Person> roster, Predicate<Person> tester) {  
    for (Person p : roster) {  
        if (tester.test(p)) {  
            p.printPerson();  
        }  
    }  
}
```

- In base a questa definizione, possiamo invocare il metodo come segue:

```
printPersonsWithPredicate(  
    roster, p -> p.getGender() == Person.Sex.MALE  
        && p.getAge() >= 18 && p.getAge() <= 25 );
```



Possibili Usi

- ▶ **Approccio 7: Usare le espressioni lambda in un'applicazione**
- ▶ Riconsideriamo l'implementazione del metodo `printPersonsWithPredicate` per vedere in quali altri modi possiamo impiegare le lambda espressioni:

```
public static void printPersonsWithPredicate(  
    List<Person> roster, Predicate<Person> tester) {  
    for (Person p : roster) {  
        if (tester.test(p)) {  
            p.printPerson();  
        }  
    }  
}
```

Il metodo passa in rassegna ogni istanza di `Person` nella lista che ha ricevuto in ingresso e se la condizione viene soddisfatta allora invoca il metodo `printPerson()`.



Possibili Usi

- ▶ **Approccio 7: Usare le espressioni lambda in un'applicazione**
- ▶ Riconsideriamo l'implementazione del metodo `printPersonsWithPredicate` per vedere in quali altri modi possiamo impiegare le lambda espressioni:

```
public static void printPersonsWithPredicate(  
    List<Person> roster, Predicate<Person> tester) {  
    for (Person p : roster) {  
        if (tester.test(p)) {  
            p.printPerson();  
        }  
    }  
}
```



Come specificare un'azione diversa?



Possibili Usi

- ▶ **Approccio 7: Usare le espressioni lambda in un'applicazione**
- ▶ Riconsideriamo l'implementazione del metodo `printPersonsWithPredicate` per vedere in quali altri modi possiamo impiegare le lambda espressioni:

```
public static void printPersonsWithPredicate(  
    List<Person> roster, Predicate<Person> tester) {  
    for (Person p : roster) {  
        if (tester.test(p)) {  
            p.printPerson();  
        }  
    }  
}
```

La soluzione è ricorrere alle espressioni lambda, usando l'interfaccia funzionale **Consumer<T>**, che presenta il metodo void **accept(T t)**; per sostituire l'invocazione di `printPerson()`.



Possibili Usi

```
public static void processPersons(List<Person> roster,  
    Predicate<Person> tester, Consumer<Person> block) {  
    for (Person p : roster) {  
        if (tester.test(p)) {  
            block.accept(p);  
        }  
    }  
}
```

- Questo metodo non solo rappresenta come un'espressione lambda il criterio di ricerca, ma anche l'azione da eseguire sui membri trovati. Tale metodo può essere invocato come segue:

```
processPersons(  
    roster, p -> p.getGender() == Person.Sex.MALE  
        && p.getAge() >= 18 && p.getAge() <= 25,  
    p -> p.printPerson() );
```



Possibili Usi

- Supponiamo di voler realizzare un'azione più complessa sui membri selezionati, magari estraendo degli opportuni dati e restituendo un determinato valore. **L'interfaccia Function <T,R>** contiene il metodo **R apply(T t)**.

```
public static void processPersonsWithFunction(  
    List<Person> roster, Predicate<Person> tester,  
    Function<Person, String> mapper, Consumer<String> block) {  
    for (Person p : roster) {  
        if (tester.test(p)) {  
            String data = mapper.apply(p);  
            block.accept(data);  
        }  
    }  
}
```



Possibili Usi

Questo metodo ha un'espressione per la ricerca, una per l'estrazione e una per l'elaborazione.

- Supponiamo di voler processare una lista di persone, magari estraendo degli oggetti e restituendo un determinato valore. **L'interfaccia Function** contiene il metodo R apply(T t):

```
public static void processPersonsWithFunction(  
    List<Person> roster, Predicate<Person> tester,  
    Function<Person, String> mapper, Consumer<String> block) {  
    for (Person p : roster) {  
        if (tester.test(p)) {  
            String data = mapper.apply(p);  
            block.accept(data);  
        }  
    }  
}
```



Possibili Usi

- Questo nuovo metodo possiamo impiegarlo per poter ottenere l'indirizzo e-mail per tutti i membri eligibili per il Selective Service e stamparlo a video:

```
processPersonsWithFunction(  
    roster, p -> p.getGender() == Person.Sex.MALE  
        && p.getAge() >= 18 && p.getAge() <= 25,  
    p -> p.getEmailAddress(),  
    email -> System.out.println(email)  
);
```

Da ogni elemento filtrato viene estratto un dato, in questo campo l'indirizzo e-mail.



Possibili Usi

- ▶ **Approccio 8: Uso più estensivo dei Generics**
- ▶ Riconsideriamo il metodo `processPersonsWithFunction()`, e realizziamone un'implementazione generica:

```
public static <X, Y> void processElements(  
    Iterable<X> source, Predicate<X> tester,  
    Function <X, Y> mapper, Consumer<Y> block) {  
    for (X p : source) {  
        if (tester.test(p)) {  
            Y data = mapper.apply(p);  
            block.accept(data);  
        }  
    }  
}
```



Possibili Usi

- **Approccio 8: Uso più estensivo dei Generics**
- Riconsideriamo il metodo `processPersonsWithFunction()`, e realizziamone un'implementazione generica:

```
public static <X, Y> void processElements(  
    Iterable<X> source, Predicate<X> tester,  
    Function<X, Y> mapper, Consumer<Y> block) {  
    for (X p : source) {  
        if (tester.test(p)) {  
            Y data = mapper.apply(p);  
            block.accept(data);  
        }  
    }  
}
```

In questa implementazione si è resa generica la tipologia di elementi da trattare e il tipo di dato da estrarre dalle istanze selezionate del primo tipo generico.



Possibili Usi

- **Approccio 8: Uso più estensivo dei Generics**
- Riconsideriamo il metodo `processPersonsWithFunction()`, e realizziamone un'implementazione generica:

```
public static <X, Y> void processElements(  
    Iterable<X> source, Predicate<X> tester,  
    Function<X, Y> converter, Consumer<Y> block) {  
    for (X p : source) {  
        if (tester.test(p)) {  
            Y data = m  
            block.accept  
        }  
    }  
}
```

L'interfaccia **Java Iterable** rappresenta una collezione di oggetti che è iterabile, che può essere attraversata. Implementazione del l'interfaccia **Iterable** consente a un oggetto di utilizzare il ciclo **for-each**. Lo fa chiamando internamente il metodo `iterator()` sull'oggetto.



Sintassi

► Una espressione lambda consiste dei seguenti elementi:

- Una lista di parametri formali separati da virgole e racchiusi tra parentesi. Il metodo `CheckPerson.test()` contiene un parametro `p`, che rappresenta un'istanza della classe `Person`. È possibile omettere il tipo del parametro e le parentesi se si ha un solo parametro:

```
p -> p.getGender() == Person.Sex.MALE && p.getAge() >= 18 &&
p.getAge() <= 25
```

- Il simbolo di freccia ->
- Un corpo, che è costituito da una singola espressione o un blocco di espressioni. Nel caso di una singola espressione, l'ambiente di esecuzione di Java valuta l'espressione e restituisce il valore calcolato. In alternativa, è possibile usare uno statement di return:

```
p -> { return p.getGender() == Person.Sex.MALE && p.getAge() >= 18
&& p.getAge() <= 25; }
```



Possibili Usi

Nelle espressioni lambda, gli statement vanno racchiusi in parentesi graffe.
Solo uno statement che restituisce void **non** deve essere racchiuso in tali parentesi, oppure espressioni:

email -> System.out.println(email)



Possibili Usi

Nelle espressioni lambda, gli statement vanno racchiusi in parentesi graffe.
Solo uno statement che restituisce void **non** deve essere racchiuso in tali parentesi, oppure espressioni:

email -> System.out.println(email)

Casi d'uso	Esempio di lambda
Espressione booleana	(List<String> list) -> list.isEmpty()
Creazione di oggetti	() -> new Apple(10);
Consumo di oggetti	(Apple a) -> System.out.println(a.getWeight());
Selezione/Estrazione da un oggetto	(String s) -> s.length();
Combinazione di due valori	(int a, int b) -> a*b
Confronto di due oggetti	(Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight())



Possibili Usi

Nelle espressioni lambda, gli statement vanno racchiusi in parentesi graffe.
Solo uno statement che restituisce void **non** deve essere racchiuso in tali parentesi, oppure espressioni:

email -> System.out.println(email)

- Le espressioni lambda somigliano a dichiarazioni di metodo, e infatti sono spesso considerate come dei metodi anonimi, senza un nome.



Possibili Usi

Nelle espressioni lambda, gli statement vanno racchiusi in parentesi graffe.
Solo uno statement che restituisce void non deve essere racchiuso in tali parentesi, oppure espressioni:

email -> System.out.println(email)

- Le espressioni lambda somigliano a dichiarazioni di metodo, e infatti sono spesso considerate come dei metodi anonimi, senza un nome.



Esempio

- Consideriamo un esempio di espressione lambda che assume più di un parametro formale e realizza una operazione matematica su due operandi interi.

```
public class Calculator {  
    interface IntegerMath {  
        int operation(int a, int b);  
    }  
    public int operateBinary(int a, int b, IntegerMath op) {  
        return op.operation(a, b);  
    }  
}
```



Esempio

- Consideriamo un parametro interi.

Interfaccia che definisce una generica operazione aritmetica a due operandi interi con ritorno intero.

```
public class Calcu
```

```
interface IntegerMath {  
    int operation(int a, int b);  
}
```

```
public int operateBinary(int a, int b, IntegerMath op) {  
    return op.operation(a, b);  
}
```



Esempio

- Consideriamo un esempio di espressione lambda che assume più di un parametro formale e realizza una operazione matematica su due operandi interi.

```
public class LambdaExample {  
    // Metodo che prende due interi in input, e vi applica una  
    // data operazione aritmetica definita per mezzo  
    // dell'interfaccia IntegerMath.  
    interface IntegerMath {  
        int operation(int a, int b);  
    }  
  
    public int operateBinary(int a, int b, IntegerMath op) {  
        return op.operation(a, b);  
    }  
}
```



Esempio

```
public static void main(String... args) {  
    Calculator myApp = new Calculator();  
    IntegerMath addition = (a, b) -> a + b;  
    IntegerMath subtraction = (a, b) -> a - b;  
    System.out.println("40 + 2 = " +  
        myApp.operateBinary(40, 2, addition));  
    System.out.println("20 - 10 = " +  
        myApp.operateBinary(20, 10, subtraction));  
}  
}
```



Esempio

```
public static void main(String... args) {  
    Calculator myApp = new Calculator();  
    IntegerMath addition = (a, b) -> a + b;  
    IntegerMath subtraction = (a, b) -> a - b;  
    System.out.println("40 + 2 = " +  
        myApp.operateBinary(40, 2, addition));  
    System.out.println("20 - 10 = " +  
        myApp.operateBinary(20, 10, subtraction));  
}  
}
```



Esempio

```
public static void main(String... args) {  
    Calculator myApp = new Calculator();  
    IntegerMath addition = (a, b) -> a + b;  
    IntegerMath subtraction = (a, b) -> a - b;  
    System.out.println("40 + 2 = " +  
        myApp.operateBinary(20, 10, subtraction));  
}
```

Definizione di due espressioni lambda con nome, parametri e definizione del corpo.



Esempio

```
public static void main(String... args) {  
    Calculator myApp = new Calculator();  
    IntegerMath addition = (a, b) -> a + b;  
    IntegerMath subtraction = (a, b) -> a - b;  
  
    System.out.println("40 + 2 = " +  
        myApp.operateBinary(40, 2, addition));  
    System.out.println("20 - 10 = " +  
        myApp.operateBinary(20, 10, subtraction));  
}  
}
```

Applicazione delle espressioni a determinati valori ed ottenimento del valore di ritorno, con sua stampa a video.



Sintassi



Quale delle seguenti espressioni è valida?

() -> {}

() -> "Raoul"

() -> { return "Mario"; }

(Integer i) -> return "Alan" + 1;

(String s) -> {"Iron Man"; }



Sintassi



Quale delle seguenti espressioni è valida?

`() -> {}`

`() -> "Raoul"`

`() -> { return "Mario"; }`

`(Integer i) -> return "Alan" + i;`

`(String s) -> {"Iron Man"; }`



La prima espressione è valida e corrisponde a un metodo con un corpo vuoto.

La seconda non ha parametri e restituisce una stringa.

La terza non ha parametri e restituisce una stringa.

La quarta non è valida, bisogna mettere le parentesi graffe: `(Integer i) -> { return "Alan" + i; }`

La quinta non è valida perché vanno tolte le parentesi: `(String s) -> "Iron Man";`



Cattura Variabili

- ▶ Anche le espressioni lambda possono catturare le variabili, come le classi locali e quelle anonime. Hanno lo stesso accesso alle variabili locali nel contesto in cui sono state incluse; ma a differenza di queste, le espressioni lambda non hanno alcun problema di ombreggiamento.
- ▶ Se una dichiarazione di un tipo (come il nome di un parametro o di una variabile membro) in un particolare contesto (come in una classe interna o in una definizione di metodo) ha lo stesso nome di un'altra dichiarazione in un contesto di inclusione, allora la prima dichiarazione ombreggia quella nel contesto di inclusione:

```
public class ShadowTest {  
    public int x = 0;  
    class FirstLevel {  
        public int x = 1;  
        void methodInFirstLevel(int x) {  
            System.out.println("x = " + x);  
            System.out.println("this.x = " + this.x);  
        }  
    }  
}
```



Cattura Variabili

- ▶ Anche le espressioni lambda possono catturare le variabili, come le classi locali e quelle anonime. Hanno lo stesso accesso alle variabili locali nel contesto in cui sono state incluse; ma a differenza di queste, le espressioni lambda non hanno alcun problema di ombreggiamento.
- ▶ Se una dichiarazione di un tipo (come il nome di un parametro o di una variabile membro) in un particolare contesto (come in una classe interna o in una definizione di metodo) ha lo stesso nome di un'altra dichiarazione in un contesto di inclusione, allora la prima dichiarazione ombreggia quella nel contesto di inclusione:

```
public class ShadowT
```

```
    public int x = 0;
```

```
    class FirstLevel {
```

```
        public int x = 1;
```

```
        void methodInFirstLevel(int x) {
```

```
            System.out.println("x = " + x);
```

```
            System.out.println("this.x = " + this.x);
```

x identifica sia una variabile membro della classe interna, sia di quella esterna, sia un parametro di ingresso al metodo della classe interna.



Cattura Variabili

- ▶ Anche le espressioni lambda possono catturare le variabili, come le classi locali e quelle anonime. Hanno lo stesso accesso alle variabili locali nel contesto in cui sono state incluse; ma a differenza di queste, le espressioni lambda non hanno alcun problema di ombreggiamento.
- ▶ Se una dichiarazione di un tipo (come il nome di un parametro o di una variabile membro) in un particolare contesto (come in una classe interna o in una definizione di metodo) ha di inclusione, allora la pr inclusione:

```
public class ShadowT  
    public int x = 0;  
    class FirstLevel {  
        public int x = 1;  
        void methodInFirstLevel(int x) {
```

Il metodo invoca diversamente una variabile x così da testare quale risultato viene restituito. Nel primo caso x identifica il parametro del metodo, per poter accedere alla variabile membro della classe interna bisogna ricorrere al this.

```
        System.out.println("x = " + x);  
        System.out.println("this.x = " + this.x);  
    }  
}
```



Cattura Variabili

```
        System.out.println("ShadowTest.this.x = " +ShadowTest.this.x);
    }
}
public static void main(String... args) {
    ShadowTest st = new ShadowTest();
    ShadowTest.FirstLevel fl = st.new FirstLevel();
    fl.methodInFirstLevel(23);
}
}
```



Cattura Variabili

```
System.out.println("ShadowTest.this.x = " +ShadowTest.this.x);
```

```
}
```

Per ottenere il valore della variabile membro della classe esterna ombreggiata da quella interna, bisogna invocare il this sulla classe esterna.

```
ShadowTest.FirstLevel fl = st.new FirstLevel();  
fl.methodInFirstLevel(23);
```

```
}
```

```
}
```



Cattura Variabili

```
        System.out.println("ShadowTest.this.x = " +  
ShadowTest.this.x);  
    }  
}  
public static void main(String... args) {  
    ShadowTest st = new ShadowTest();  
    ShadowTest.FirstLevel fl = st.new FirstLevel();  
    fl.methodInFirstLevel(23);  
}  
}
```

Ecco l'output che si ottiene
eseguendo il codice di esempio:

x = 23

this.x = 1

ShadowTest.this.x = 0



Cattura Variabili

- Le espressioni lambda sono lessicamente valutate, e le dichiarazioni nell'espressione sono interpretate così come lo sono nell'ambiente che le contiene.

```
public class LambdaScopeTest {  
    public int x = 0;  
    class FirstLevel {  
        public int x = 1;  
        void methodInFirstLevel(int x) {  
            Consumer<Integer> myConsumer = (y) -> {  
                System.out.println("x = " + x);  
                System.out.println("y = " + y); System.out.println("this.x = " + this.x);  
                System.out.println("LambdaScopeTest.this.x = " +  
                    LambdaScopeTest.this.x);    };  
            myConsumer.accept(x);    }    }  
}
```



Cattura Variabili

- Le espressioni lambda sono lessicamente valutate, e le dichiarazioni nell'espressione sono interpretate così come lo sono nell'ambiente che le contiene.

```
public class LambdaScopeTest {
```

```
    public int x = 0;
```

```
    class FirstLevel {
```

```
        public int x = 0;
```

```
        void method1()
```

```
        {  
            Consumer<Integer> myConsumer = (y) -> {
```

```
                System.out.println("x = " + x);
```

```
                System.out.println("y = " + y); System.out.println("this.x = " + this.x);
```

```
                System.out.println("LambdaScopeTest.this.x = " +
```

```
                    LambdaScopeTest.this.x);    };
```

```
            myConsumer.accept(x);    } }
```

Questa istruzione causa un errore di compilazione
"local variables referenced from a lambda
expression must be final or effectively final".



Cattura Variabili

- Le espressioni lambda sono lessicamente valutate, e le dichiarazioni nell'espressione sono interpretate così come lo sono nell'ambiente che le contiene.

```
public class LambdaScopeTest {
```

```
    public int x = 0;
```

Perchè va dichiarata final oppure essere effettivamente final senza dichiararlo (una sola assegnazione)?

```
    public int x = 1;
```

```
    void methodInFirstLevel(final int x) {
```

```
        Consumer<Integer> myConsumer = (y) -> {
```

```
            System.out.println("x = " + x);
```

```
            System.out.println("y = " + y); System.out.println("this.x = " + this.x);
```

```
            System.out.println("LambdaScopeTest.this.x = " +
```

```
                LambdaScopeTest.this.x);    };
```

```
        myConsumer.accept(x);    } }
```



Cattura Variabili

- Le espressioni lambda sono lessicamente valutate, e le dichiarazioni nell'espressione sono interpretate così come lo sono nell'ambiente che le contiene.

```
public class LambdaScopeTest {
```

```
    public int x = 0;
```

Perchè va dichiarata final oppure essere effettivamente final senza dichiararlo (una sola assegnazione)?

```
    public int x = 1;
```

```
    void methodInFirstLevel(final int x) {
```

Le variabili di istanza sono allocate nell'heap, mentre quelle locali nello stack.

Se la lambda avesse accesso alla variabile locale direttamente e fosse impiegata in un thread, allora il thread userebbe la lambda per cercare di accedere alla variabile, anche dopo che il thread che l'ha allocata potrebbe averla anche deallocata. Pertanto, per effetto della cattura la lambda lavora su una copia della variabile, ecco perchè deve essere final.



Cattura Variabili

- Le espressioni lambda sono lessicamente valutate, e le dichiarazioni nell'espressione sono interpretate così come lo sono nell'ambiente che le contiene.

```
public class LambdaScopeTest {  
    public int x = 0;  
    class FirstLevel {  
        public int x = 1;  
        void methodInFirstLevel(final int x) {  
            Consumer<Integer> myConsumer = (y) -> {  
                System.out.println("x = " + x);  
                System.out.println("y = " + y); System.out.println("this.x = " + this.x);  
                System.out.println("LambdaScopeTest.this.x = " +  
                    LambdaScopeTest.this.x);    };  
            myConsumer.accept(x);    }    }  
}
```



Cattura Variabili

```
public static void main(String... args) {  
    LambdaScopeTest st = new LambdaScopeTest();  
    LambdaScopeTest.FirstLevel fl = st.new FirstLevel();  
    fl.methodInFirstLevel(23);  
}  
}
```

Cosa succede?



Cattura Variabili

```
public static void main(String... args) {  
    LambdaScopeTest st = new LambdaScopeTest();  
    LambdaScopeTest.FirstLevel fl = st.new FirstLevel();  
    fl.methodInFirstLevel(23);  
}  
}
```

- Questo esempio se eseguito restituisce il seguente

x = 23

y = 23

this.x = 1

LambdaScopeTest.this.x = 0

Cosa succede?



Cattura Variabili

```
public static void main(String... args) {  
    LambdaScopeTest st = new LambdaScopeTest();  
    LambdaScopeTest.FirstLevel fl = st.new FirstLevel();  
    fl.methodInFirstLevel(23);  
}  
}
```

- ▶ Questo esempio se eseguito restituisce il seguente output:

x = 23

y = 23

this.x = 1

LambdaScopeTest.this.x = 0

- ▶ Proviamo a modificare la dichiarazione dell'espressione, indicando x come parametro invece che y:

```
Consumer<Integer> myConsumer = (x) -> { // ... }
```

Cosa succede?



Cattura Variabili

```
public static void main(String... args) {  
    LambdaScopeTest st = new LambdaScopeTest();  
    LambdaScopeTest.FirstLevel fl = st.new FirstLevel();  
    fl.methodInFirstLevel(23);  
}  
}
```

- ▶ Questo esempio se eseguito restituisce il seguente output:

x = 23

y = 23

this.x = 1

LambdaScopeTest

- ▶ Proviamo a compilare il seguente codice con il compilatore Java. Cons
Il compilatore restituisce un errore “variable x is already defined in method methodInFirstLevel(int)” perchè l’espressione non introduce un nuovo livello di scope nel naming delle variabili.



Cattura Variabili

- Nelle espressioni lambda è possibile accedere direttamente a campi, metodi e variabili locali dello scope che le include. Ma, come le classi locali ed anonime, un'espressione può accedere a variabili locali e parametri solo se definiti final o che lo sono effettivamente sebbene non dichiarati in quanto tali. Nel precedente esempio abbiamo dichiarato il parametro di ingresso al metodo `methodInFirstLevel()` come `final`. In alternativa avremmo potuto rendere `x` effettivamente `final`:

```
void methodInFirstLevel(int x) {  
    x = 99;  
    // ... }  
}
```

- A causa di questa assegnazione, la variabile `FirstLevel.x` non è più `final` effettivamente, e il compilatore solleva un errore “local variables referenced from a lambda expression must be final or effectively final” quando l'espressione vi si riferisce:

```
System.out.println("this.x = " + this.x);
```



Riferimento di Metodo

- ▶ L'uso delle espressioni lambda serve a creare metodi anonimi, ma alcune volte tali espressioni non fanno altro che invocare dei metodi dichiarati nel codice applicativo. In questi casi, è più chiaro riferirsi direttamente a tali metodi con il loro nome. I riferimenti ai metodi consentono di fare questo per mezzo di un approccio compatto.
- ▶ Consideriamo la classe Person:

```
public class Person {  
    public enum Sex {    MALE, FEMALE    }  
    String name; LocalDate birthday;  
    Sex gender; String emailAddress;  
    public int getAge() {    // ...    }  
    public Calendar getBirthday() {    return birthday;    }  
    public static int compareByAge(Person a, Person b) {  
        return a.birthday.compareTo(b.birthday);  
    }  
}
```



Riferimento di Metodo

- Ipotizziamo che i membri dell'applicazione di social networking siano contenuti in un array, e si vuole ordinare tale array per età:

```
List<Person> roster = Person.createRoster();  
Person[] rosterAsArray = roster.toArray(new Person[roster.size()]);  
class PersonAgeComparator implements Comparator<Person> {  
    public int compare(Person a, Person b) {  
        return a.getBirthDay().compareTo(b.getBirthDay());  
    }  
}  
  
Arrays.sort(rosterAsArray, new PersonAgeComparator());
```



Riferimento di Metodo

- Ipotizziamo che i membri dell'applicazione di social networking siano contenuti in un array, e si vuole ordinare tale array per età:

```
List<Person> roster = Person.createRoster();
```

```
Person[] rosterAsArray = roster.toArray(new Person[roster.size()]);
```

```
class PersonAgeComparator implements Comparator<Person> {  
    public int compare(Person a, Person b) {  
        return a.getBirthDay().compareTo(b.getBirthDay());  
    }  
}
```

```
}
```

```
Arrays.sort(
```

Classe locale che implementa **l'interfaccia funzionale Comparator<T>**, che dichiara un metodo per comparare due istanze del tipo T.



Riferimento di Metodo

- Ipotizziamo che i membri dell'applicazione di social networking siano contenuti in un array, e si vuole ordinare tale array per età:

```
List<Person> roster = Person.createRoster();
```

```
Person[] rosterAsArray = roster.toArray(new Person[roster.size()]);
```

```
class PersonAgeComparator implements Comparator<Person> {
```

```
    public i
```

```
    retur
```

```
}
```

```
}
```

Metodo statico della classe Arrays che accetta in ingresso un array e un'istanza dell'interfaccia Comparator<T> per ordinare l'array secondo la politica implementata nel metodo compare.

```
static <T> void sort(T[] a, Comparator <? super T> c)
```

```
Arrays.sort(rosterAsArray, new PersonAgeComparator());
```



Riferimento di Metodo

- Siccome l'interfaccia Comparator<T> è funzionale, possiamo impiegare un'espressione lambda invece di una classe locale:

```
Arrays.sort(rosterAsArray,  
    (Person a, Person b) -> {  
        return a.getBirthDay().compareTo(b.getBirthDay());  
    });
```



Riferimento di Metodo

- Siccome l'interfaccia `Comparator<T>` è funzionale, possiamo impiegare un'espressione lambda invece di una classe locale:

```
Arrays.sort(rosterAsArray,  
    (Person a, Person b) -> {  
        return a.getBirthDay().compareTo(b.getBirthDay());  
    });
```

- Notiamo, però, che il metodo che compara le date di due istanze di Person già esiste, quindi possiamo modificare l'espressione invocando direttamente tale metodo:

```
Arrays.sort(rosterAsArray, (a, b) -> Person.compareByAge(a, b));
```

