



# Reti e Laboratorio

## Modulo Laboratorio 3

### AA. 2024-2025

docente: Laura Ricci

[laura.ricci@unipi.it](mailto:laura.ricci@unipi.it)

## Lezione 2

### JAVA: eccezioni e collezioni

27/9/2024



# ERRORI ED ECCEZIONI IN JAVA

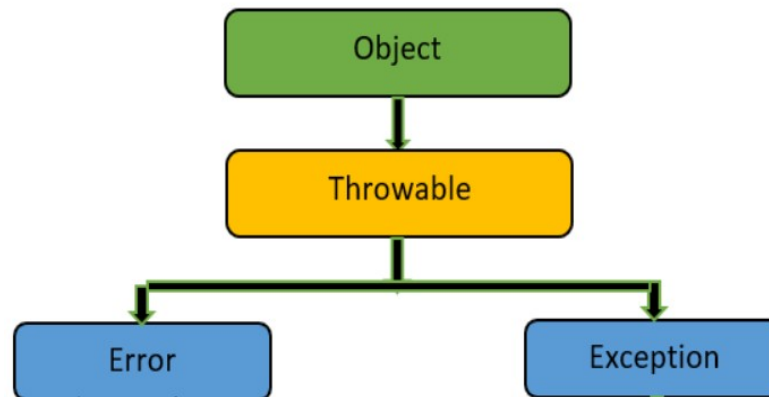
- errori ed eccezioni interrompono improvvisamente la normale esecuzione e generano un evento “non pianificato”
- Java consente di intercettare tali eventi e gestirli, anziché far crashare l'intero programma
- se non gestiti, gli errori e le eccezioni vengono propagati attraverso lo stack delle chiamate
- il programma si arresta improvvisamente e viene stampata una “stack trace”, che permette di individuare
  - il tipo di errore/eccezione
  - dove il programma si è interrotto

# ERRORI ED ECCEZIONI

- gli errori si verificano nella macchina virtuale di Java e non sono riparabili
  - no memory available
  - troppe chiamate ricorsive: stack overflow
  - librerie non presenti
- le eccezioni sono attivate dalla macchina virtuale o dal programma stesso e possono generalmente essere gestite per ristabilire la situazione normale
  - deferenziamento di un puntatore a null
  - divisione per zero
  - read/write errors da dispositivi
  - errori nella “business logic del programma”

# ECCEZIONI COME OGGETTI

- quando un programma Java esegue un'operazione illegale viene **lanciata un'eccezione**, che può essere generata
  - dalla logica del programma: `ArithmeticException`, `ArrayOutOfBoundsException`, `NumberFormatException`,
  - dalla interazione con la JVM e il SO: `IOException`, `FileNotFoundException`
- seguendo la filosofia generale del linguaggio, un'eccezione è un oggetto, che può essere predefinito o creato dal programmatore
- la gerarchia delle classi riguardanti le eccezioni è la seguente:



- la classe `Throwable` include un campo `String` che contiene una descrizione della eccezione sollevata

# ECCEZIONI NON GESTITE

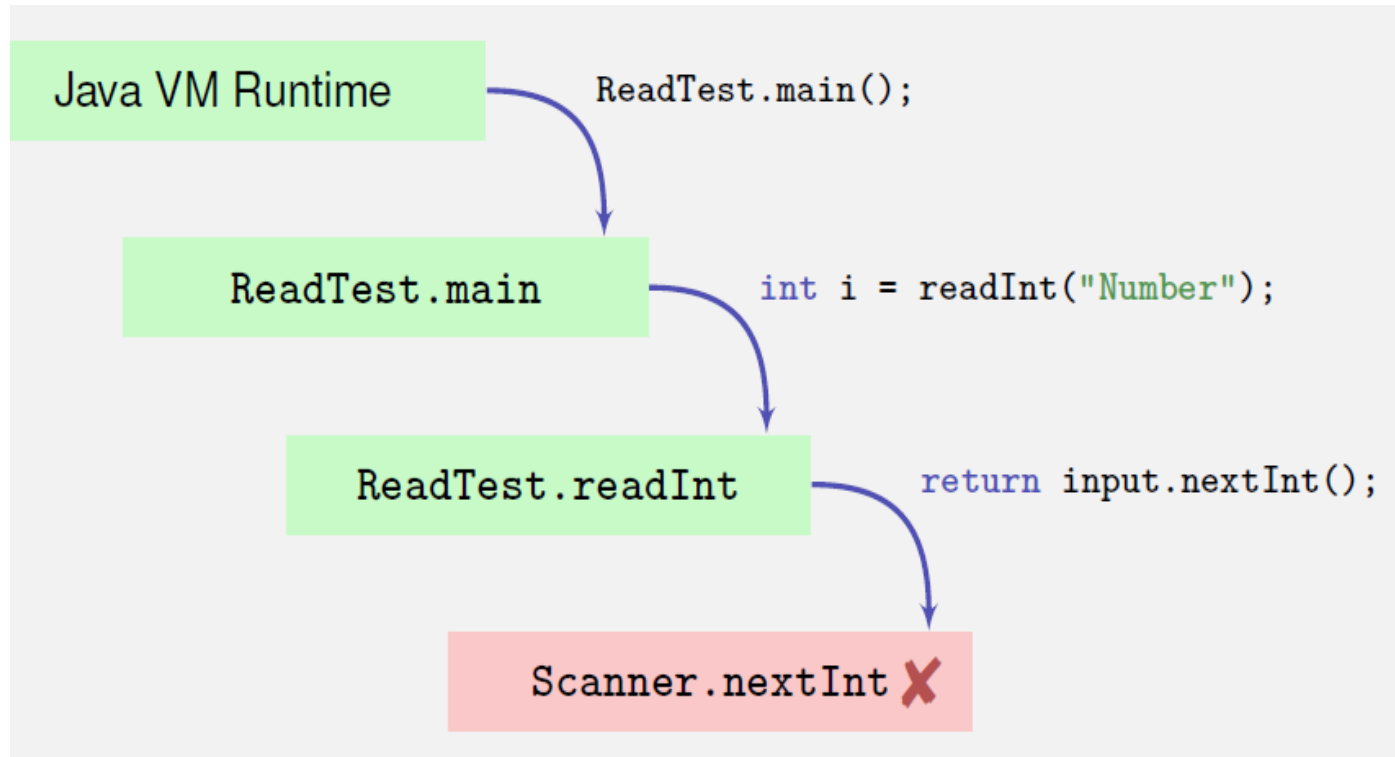
```
package UnhandledException;
import java.util.Scanner;
class ReadTest {
    public static void main(String[] args){
        int i = readInt("Number");
    }
    private static int readInt(String prompt){
        System.out.print(prompt + ": ");
        Scanner input = new Scanner(System.in);
        return input.nextInt ();
    }
}
```

con input abcd s, il programma si interrompe e stampa:

Number: abcd

```
Exception in thread "main" java.util.InputMismatchException
at java.base/java.util.Scanner.throwFor(Scanner.java:964)
at java.base/java.util.Scanner.next(Scanner.java:1619)
at java.base/java.util.Scanner.nextInt(Scanner.java:2284)
at java.base/java.util.Scanner.nextInt(Scanner.java:2238)
at UnhandleException/UnhandledException.ReadTest.readInt(ReadTest.java:12)
at UnhandleException/UnhandledException.ReadTest.main(ReadTest.java:7)
```

# PROPAGAZIONE DELLE ECCEZIONI



- l'eccezione viene propagata a ritroso, nello stack delle chiamate, fino a che non raggiunge il Java Runtime, che stampa il messaggio
- tutti i metodi nello stack delle chiamate falliscono

# ECCEZIONI NON GESTITE

```
1 package ArithmeticException;
2
3 public class AritmeticExceptionClass {
4
5     public static void main(String[] args) {
6         int a = Integer.parseInt(args[0]);
7         int b = Integer.parseInt(args[1]);
8         int quot = a / b;
9         System.out.println("The quotient is "+ quot);
10    }
11 }
```

con input 30, 0, il programma si interrompe e stampa:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    At ArithmeticException/ArithmeticException.AritmeticExceptionClass.main
    (AritmeticExceptionClass.java:8)
```

# TRY-CATCH STATEMENT

- **blocco try**
  - le chiamate ai metodi che possono causare un'eccezione devono essere inserite in un blocco try.
- **blocco catch**
  - segue il blocco try
  - gestisce una situazione “eccezionale” indica cosa deve accadere nel caso in cui si verifichi un'eccezione
- nel caso non si verifichino eccezioni nel blocco try
  - vengono seguite le istruzioni nel blocco try
  - si prosegue la esecuzione con l'istruzione che segue tutte le clausole catch
- nel caso si verifichino eccezioni nel blocco try
  - il controllo viene immediatamente trasferito la cui clausola corrisponde alla eccezione che è stata sollevata
  - dopo aver eseguito le istruzioni corrispondenti a quella clausola, il controllo viene trasferito alla prima istruzione che segue tutte le clausole catch



# CHECKED E NON CHECKED EXCEPTIONS

```
package HandlingException;

public class ArithmeticHandling {
    public static void main(String[] args) {
        int a = Integer.parseInt(args[0]);
        int b = Integer.parseInt(args[1]);
        try {
            int quot = a / b;
            System.out.println("The quotient is "+ quot);
        }
        catch (ArithmeticException e){
            System.out.println("0 not allowed as 2nd input");
        }
    }
}
```

con input 30, 0, il programma non solleva la eccezione e stampa:

0 not allowed as 2nd input

# L'OGGETTO ECCEZIONE

- quando si verifica un'eccezione, Java crea un oggetto eccezione che (nel precedente esempio è stato chiamato e) contiene informazioni sull'errore.
- ogni oggetto eccezione contiene un messaggio stringa che può essere utilizzato invece di stampare un messaggio personalizzato.

```
package HandlingException2;
public class ArithmeticHandling2 {
    public static void main(String[] args) {
        int a = Integer.parseInt(args[0]);
        int b = Integer.parseInt(args[1]);
        try {
            int quot = a / b;
        }
        catch (ArithmeticException e){
            System.out.println(e.getMessage());
        }
    }
}
```

con input 30, 0, il programma si interrompe e stampa:

/ by zero

# TRY-CATCH STATEMENT

- blocco catch
  - segue il blocco try
  - gestisce la situazione “eccezionale” indica cosa deve accadere nel caso in cui si verifichi un'eccezione
  - un blocco try può avere **più clausole catch associate** che gestiscono diverse eccezioni
- osservazioni:
  - le variabili definite nel blocco try sono **locali** e **non accessibili all'esterno**. In genere, è meglio definire e inizializzare le variabili al di fuori del blocco try.
  - se si vuole terminare il programma, si può utilizzare `System.exit(-1)` nel blocco catch
  - Il valore non zero, come `-1`, indica che il programma è terminato in modo anomalo.

# TRY-CATCH-FINALLY

```
try {  
    // Codice che potrebbe generare eccezioni  
} catch (TipoEccezione e) {  
    // Gestione dell'eccezione  
} finally {  
    // Codice che verrà eseguito sempre
```

- Il blocco finally è **opzionale**
- viene sempre eseguito, anche se si verifica una eccezione non gestita
- utilizzato spesso per chiudere risorse (file, connessioni di rete,...) utilizzate nelle try...catch

# TRY-CATCH-FINALLY

```
public class FinallyClass {
    public static void main(String args[]) {
        try {
            int a = 15;
            int b = 0;

            System.out.println("Value of a = " + a);
            System.out.println("Value of b = " + b);

            int c = a / b;
            System.out.println("a / b = " + c);
        }
        catch (Exception e) {
            System.out.println("Exception Thrown: " + e);
        }
        finally {
            System.out.println("Finally block executed!");
        } } }
```

Value of a = 15

Value of b = 0

Exception Thrown: [java.lang.ArithmeticException](#): / by zero

Finally block executed!

# TRY-CATCH-FINALLY

```
public class FinallyClass {
    public static void main(String args[]) {
        try {
            int a = 15;
            int b = 5;

            System.out.println("Value of a = " + a);
            System.out.println("Value of b = " + b);

            int c = a / b;
            System.out.println("a / b = " + c);
        }
        catch (Exception e) {
            System.out.println("Exception Thrown: " + e);
        }
        finally {
            System.out.println("Finally block executed!");
        } } }
```

```
Value of a = 15
Value of b = 5
a / b = 3
Finally block executed!
```

# UNCHECKED EXCEPTIONS

- le eccezioni mostrate negli esempi precedenti appartengono alla classe delle **unchecked exceptions** o **Run Time Exceptions**
  - ignorate dal compilatore, ma sollevate a run time dalla JVM
  - il compilatore non verifica se il programmatore ha gestito o meno la eccezione
  - a compilazione del seguente programma ha successo:

```
public static void main(String[] args) {  
    int a = Integer.parseInt(args[0]);  
    int b = Integer.parseInt(args[1]);  
    int quot = a / b;  
    System.out.println("The quotient is "+ quot);}}
```
  - appartengono alla classe `RuntimeException` o a una delle sottoclassi di questa classe e vengono anche indicate come `RuntimeException`
  - esempi:

`NullPointerException`, `ArithmeticException`,  
`ArrayIndexOutOfBoundsException`, `InputMismatchException`

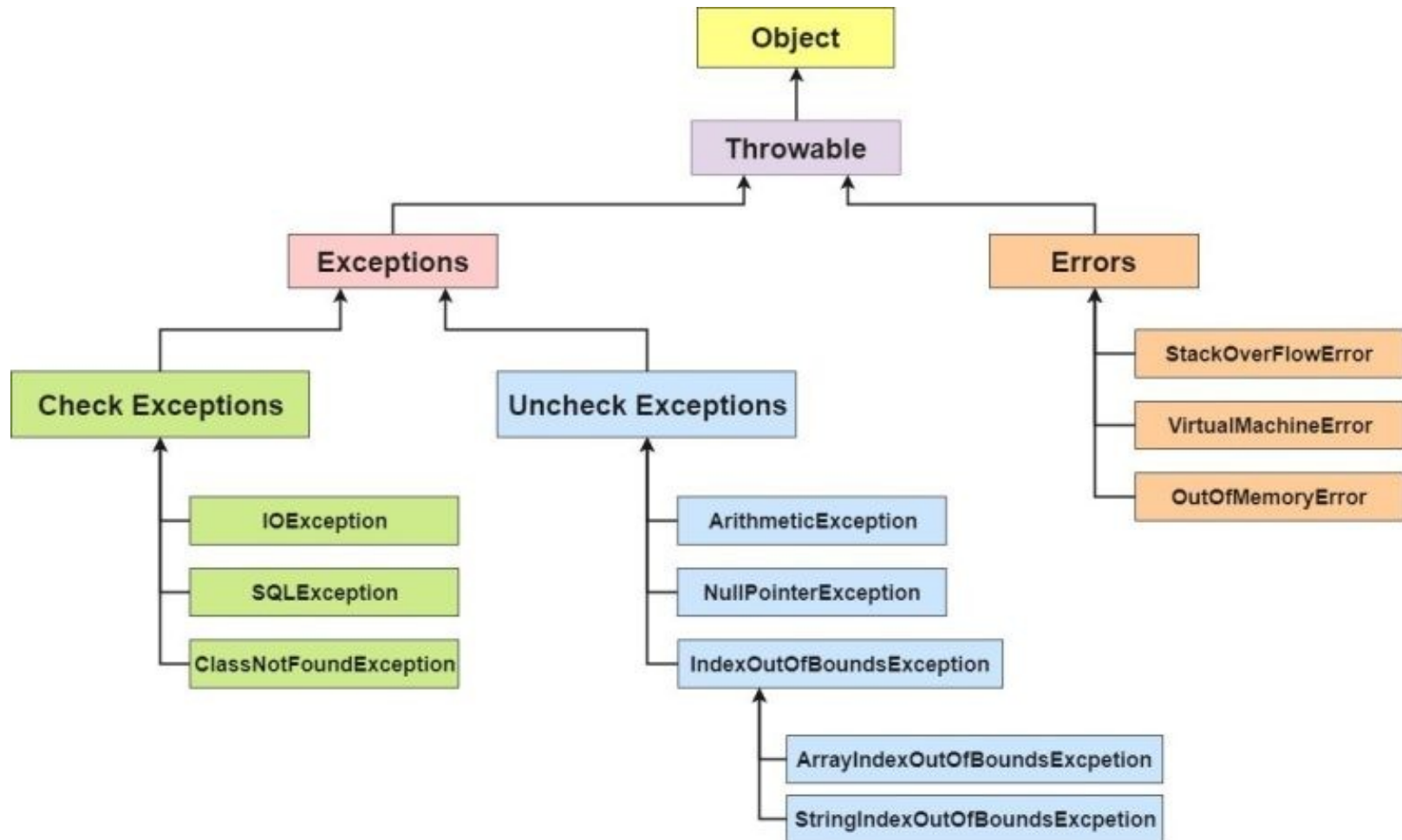
# CHECKED EXCEPTIONS

- un'altra classe di eccezioni, dette anche **compile time exceptions**
- il compilatore controlla se il programmatore ha gestito o meno una potenziale eccezione
- restituisce errore se la eccezione non è gestita
- ad esempio, nel programma sotto, il compilatore restituirebbe un errore nel caso la IOException non fosse gestita nella try ...catch

```
import java.io.*;
public class CheckedException {
    public static void main (String args[])
    {
        System.out.println("Enter a character:");
        try {
            int input = System.in.read();
            System.out.println((char) input);
        }
        catch (IOException e) {
            System.err.println("Error reading input: " + e.getMessage());
        }
    }
}
```



# ECCEZIONI: LA GERARCHIA DELLE CLASSI



# MULTIPLE-CATCH STATEMENT

- un blocco try può avere più clausole catch associate
- quando viene sollevata una eccezione, tutti i blocchi vengono esaminati in sequenza e viene eseguito il blocco che corrisponde alla eccezione

```
public class MultiCatchExample {
    public static void main(String[] args)
    { String[] s = {"abc", "123", null, "xyz"};
      for (int i = 0; i < 6; i++)
        { try
          { int a = s[i].length() + Integer.parseInt(s[i]);
            //This statement may throw NumberFormatException, NullPointerException
            and ArrayIndexOutOfBoundsException
          }
          catch(NumberFormatException ex) {
            System.out.println("NumberFormatException will be caught here");
          }
          catch (ArrayIndexOutOfBoundsException ex) {
            System.out.println("ArrayIndexOutOfBoundsException will be caught here");
          }
          catch (NullPointerException ex) {
            System.out.println("NullPointerException will be caught here");
          }
          System.out.println("After executing respective catch block, this statement
                              will be executed");}}}}
```

# RISULTATO DELL'ESECUZIONE

[NumberFormatException](#) will be caught here

After executing respective catch block, this statement will be executed

After executing respective catch block, this statement will be executed

[NullPointerException](#) will be caught here

After executing respective catch block, this statement will be executed

[NumberFormatException](#) will be caught here

After executing respective catch block, this statement will be executed

[ArrayIndexOutOfBoundsException](#) will be caught here

After executing respective catch block, this statement will be executed

[ArrayIndexOutOfBoundsException](#) will be caught here

After executing respective catch block, this statement will be executed

# CATCH MATCHING RULES

- se le eccezioni intercettate in una catch multipla non appartengono alla stessa gerarchia di classi, l'ordine delle catch non conta
- se due eccezioni intercettate nella catch sono in relazione superclasse- classe figlia, oppure, più in generale una classe è antenata dell'altra
  - ordinare le catch indicando per prime le eccezioni più specifiche, poi quelle più generali
  - le catch vengono valutate in ordine, dalla prima alla ultima, e viene eseguita la prima clausola in cui la classe dell'eccezione è compatibile con il tipo dell'eccezione sollevata
  - se si inverte l'ordine, l'eccezione più specifica non viene mai intercettata

# CATCH MATCHING RULES

- una clausola catch di una super classe viene soddisfatta anche da tutte le clausole catch delle sottoclassi
- se si vogliono intercettare eccezioni sia della superclasse che della sottoclasse, occorre inserire per prima la clausola relativa alla sottoclasse
- altrimenti la superclasse “intercetta” tutte le altre eccezioni

```
try {  
    // Code that may throw exceptions  
} catch (RuntimeException re) {  
    // Handle more general RuntimeException  
} catch (ArrayIndexOutOfBoundsException aioobe) {  
    // Handle specific ArrayIndexOutOfBoundsException  
}
```

- nell'esempio
  - ArrayIndexOutOfBoundsException è una classe figlia di RuntimeException
  - la seconda clausola catch non verrà mai eseguita
  - occorre invertire l'ordine della clausole catch

# LANCIARE ECCEZIONI

- try...catch utilizzato per intercettare eccezioni che vengono lanciate a tempo di esecuzione dalla JVM
- è possibile anche sollevare esplicitamente una “custom exception” mediante la clausola throw
- il seguente esempio solleva una unchecked exception esplicitamente

```
public class ThrowTest {
    static void checkAge(int age) {
        if (age < 18) {
            throw new ArithmeticException("Access denied - You must be at least 18 years
                old.");
        } else {
            System.out.println("Access granted - You are old enough!");
        }
    }

    public static void main(String[] args) {
        checkAge(Integer.parseInt(args[0]));
    }
}
```

# LANCIARE ECCEZIONI

```
public class Main {
    public int test(int n1, int n2) {
        if(n2 == 0){
            throw new ArithmeticException();
        } else {
            return n1/n2;
        }
    }
    public static void main(String[] args) {
        Main main = new Main();
        try{
            System.out.println(main.test(130, 0));
        }catch(ArithmeticException e){
            e.printStackTrace();
        } }
    }
```

java.lang.ArithmeticException

at ThrowsException1/ThrowsException1.Main.test(Main.java:7)

at ThrowsException1/ThrowsException1.Main.main(Main.java:16)

- il metodo test
  - lancia esplicitamente una eccezione unchecked
  - l'eccezione viene intercettata dal metodo che ha invocato test (il main) mediante try...catch

# LA PAROLA CHIAVE THROWS

- se un metodo solleva esplicitamente delle **checked exceptions** durante la sua esecuzione, occorre dichiarare queste eccezioni nella segnatura del metodo
- in questo modo chiunque invochi il metodo può conoscere quale eccezioni deve gestire

```
public class ThrowsExample {
    static void testMethod() throws Exception {
        String test = null;
        test.toString();
    }
}
```

```
public static void main(String[] arg) {
    try {
        testMethod();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

[java.lang.NullPointerException](#): Cannot invoke "String.toString()" because "test" is null  
at ThrowsException3/ThrowsException3.ThrowsExample.testMethod(ThrowsExample.java:5)  
at ThrowsException3/ThrowsException3.ThrowsExample.main(ThrowsExample.java:10)



# DIFFERENZE TRA THROW E THROWS

- throw
  - usata per sollevare esplicitamente una eccezione in un blocco di codice
  - riferisce l'istanza di una sottoclasse della classe Exception che si vuole sollevare
  - può essere usata da sola per propagare una unchecked exception
- throws
  - utilizzata nella segnatura del metodo per dichiarare una eccezione che può essere sollevata dal metodo durante la esecuzione
  - può essere usata per dichiarare sia checked che unchecked exceptions
  - riferisce il tipo di eccezione che può essere sollevata

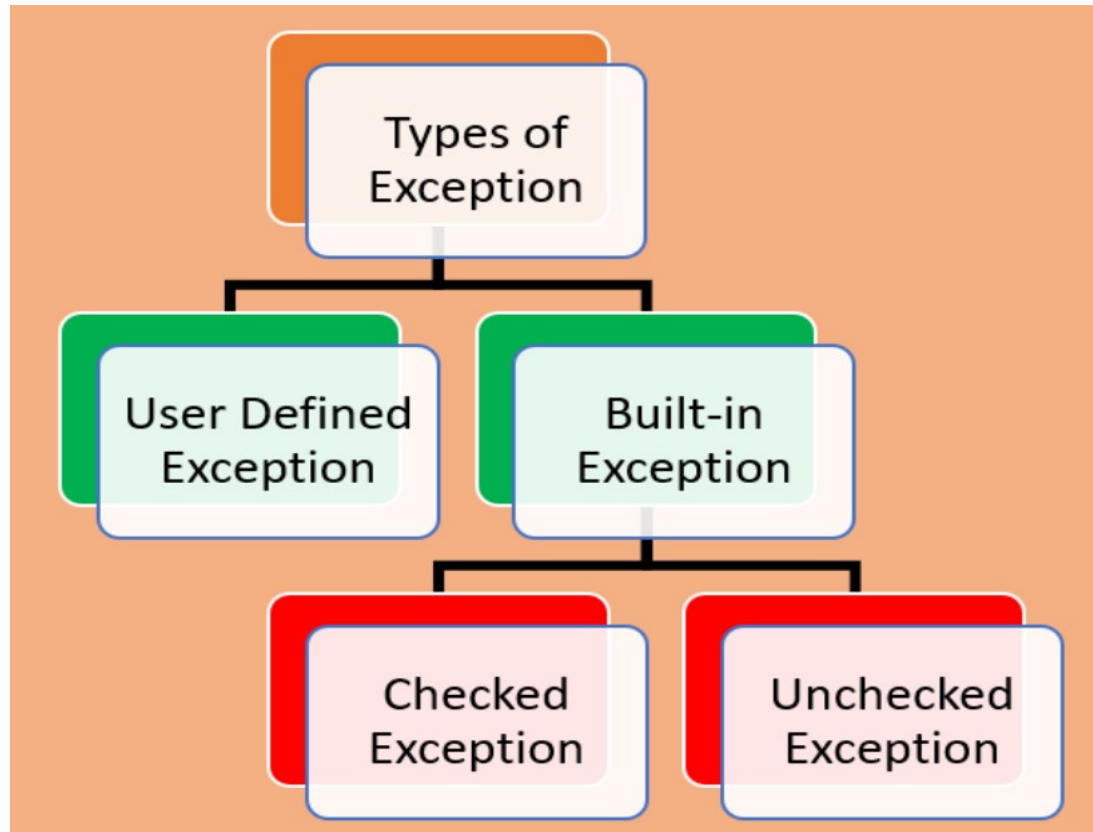
# CREARE NUOVE ECCEZIONI

- **user defined exceptions**: il programmatore può creare le proprie eccezioni
- poiché una eccezione è un oggetto, è sufficiente estendere la classe `Exception`

```
class CustomException extends Exception {
    public CustomException(String s)
    { // Chiamata al costruttore della SuperClasse Exception
        super(s);
    }
}
// Una classe che usa la customException
public class MyException {
    public static void main(String args[])
    {
        try {
            // Solelva una istanza della custom exception
            throw new CustomException("LaMiaEccezione");
        }
        catch (CustomException ex) {
            System.out.println("Intercettata");
            System.out.println(ex.getMessage()); } } }
```

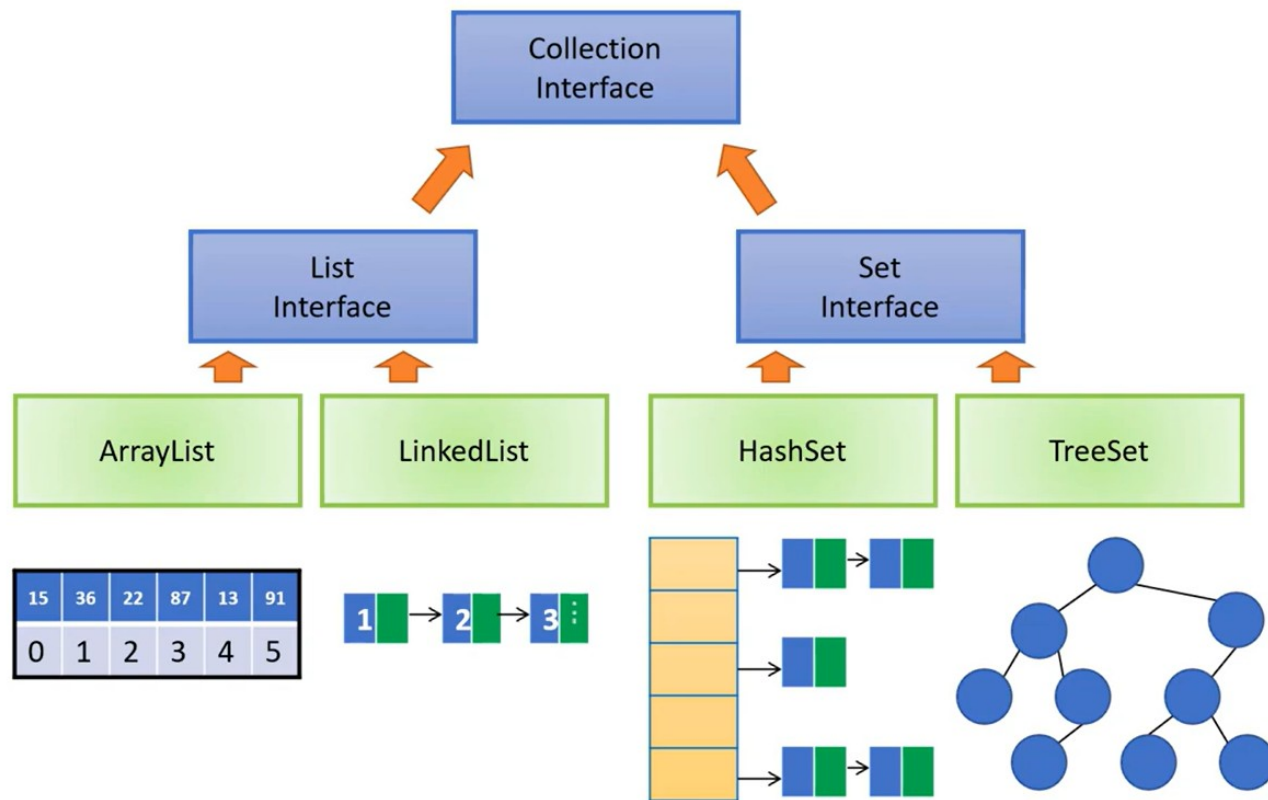
Intercettata  
LaMiaEccezione

# CLASSIFICAZIONE DELLE ECCEZIONI



# JAVA COLLECTIONS FRAMEWORK

- la libreria standard di Java fornisce una serie di classi che consentono di lavorare con gruppi di oggetti (collezioni)
- tali classi prendono il nome di Java Collections Framework
- sono contenute nel package `java.util` della libreria standard di Java



# JAVA COLLECTIONS: LISTE E INSIEMI

- un insieme
  - corrisponde a un gruppo di oggetti tutti distinti tra loro
  - gli elementi possono (facoltativamente) essere mantenuti secondo qualche ordinamento
  - HashSet implementa l'interfaccia Set non mantenendo gli elementi ordinati
  - TreeSet implementa l'interfaccia Set mantenendo un ordinamento
- una lista
  - corrisponde a un gruppo di oggetti in cui possiamo avere elementi ripetuti
  - gli oggetti sono generalmente mantenuti in ordine di inserimento
  - Vector e ArrayList implementano l'interfaccia List tramite **array dinamici** (ridimensionabili)
  - LinkedList implementa l'interfaccia List mediante **liste concatenate**

# ARRAYLIST: INTRODUCTION

- simile ad un array, però
  - la struttura è dinamica
  - automatic resizing

```
import java.util.ArrayList;  
ArrayList list = new ArrayList();
```

Warning: unsafe or unchecked operations

ArrayList is a raw type. References to generic type ArrayList<E> should be parametrized

- il warning avverte che dovremmo specificare il tipo degli elementi dell'ArrayList, mediante i **generics**

```
ArrayList <String> list = new ArrayList<String>();
```

- la ArrayList può contenere esclusivamente oggetto, non tipi di dato primitivi

```
ArrayList <int> list = new ArrayList<int>();
```

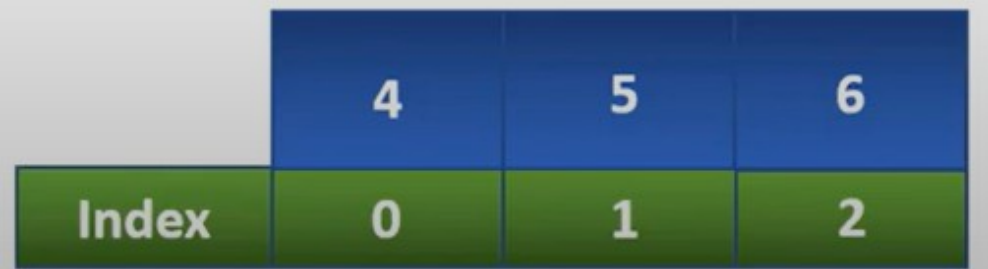
genera un ERRORE!

- nel caso si vogliono usare tipi di dato primitivi, usare le classi wrapper!

# ARRAYLIST: ADD

```
import java.util.ArrayList;

public class ArrayListExample {
    public static void main(String[] args)
    {
        ArrayList <Integer> numList = new ArrayList <Integer>();
        numList.add(4);
        numList.add(5);
        numList.add(6);
    }
}
```



	4	5	6
Index	0	1	2

gli elementi vengono aggiunti alla fine dell'ArrayList

# ARRAYLIST: GET

```
package ArrayList;
import java.util.ArrayList;

public class ArrayListExample {
    public static void main(String[] args)
    {
        ArrayList <Integer> numList = new ArrayList <Integer>();
        numList.add(4);
        numList.add(5);
        numList.add(6);
        System.out.println("The value in index 0 is"+numList.get(0));}}
```

	4	5	6
Index	0	1	2

ricerca in base all'indice



# ARRAYLIST: ADD IN BASE ALL' INDICE

```
import java.util.ArrayList;

public class ArrayListExample {
    public static void main(String[] args)
    {
        ArrayList <Integer> numList = new ArrayList <Integer>();
        numList.add(4);
        numList.add(5);
        numList.add(6);
        numList.add(1,9); //overloaded method
        numList.add(3,8);
    }
}
```

	4	9	5	8	6
Index	0	1	2	3	4

inserzione in base alla posizione

# ARRAYLIST: METODO SET

```
import java.util.ArrayList;

public class ArrayListExample {
    public static void main(String[] args)
    {
        ArrayList <Integer> numList = new ArrayList <Integer>();
        numList.add(4);
        numList.add(5);
        numList.add(6);
        numList.set(0, 1);
    }
}
```

	4	5	6
Index	0	1	2

	1	2	6
Index	0	1	2

Modifica di un elemento esistente

# ARRAYLIST: RIASSUNTO DEI METODI

<code>List.add(value);</code>	appends value at end of list
<code>List.add(index, value);</code>	inserts given value just before the given index, shifting subsequent values to the right
<code>List.clear();</code>	removes all elements of the list
<code>List.get(index)</code>	returns the value at given index
<code>List.indexOf(value)</code>	returns first index where given value is found in list (-1 if not found)
<code>List.isEmpty()</code>	returns true if the list contains no elements
<code>List.remove(index);</code>	removes/returns value at given index, shifting subsequent values to the left
<code>List.remove(value);</code>	removes the first occurrence of the value, if any
<code>List.set(index, value);</code>	replaces value at given index with given value
<code>List.size()</code>	returns the number of elements in the list
<code>List.toString()</code>	returns a string representation of the list such as "[3, 42, -7, 15]"

# JAVA GENERICS

- abbiamo usato il costrutto dei generics

```
ArrayList <String> list = new ArrayList<String>();
```

- introdotti in Java 5 per permettere controlli di tipo a compile-time
- tutte le collezioni riscritte per supportare i generics
- idee di base:
  - molti algoritmi sono logicamente identici indipendentemente dal tipo di dati a cui vengono applicati (Stack di Integer, di String, ...).
  - permettono di creare classi, interfacce e metodi che funzionano in modo sicuro che operano con vari tipi di dati.
  - permettono di definire un algoritmo una volta sola, indipendentemente da qualsiasi tipo specifico di dati.

# JAVA GENERICS

- prima dell'introduzione dei generics

```
import java.util.*;
```

```
public class GenericsExample1 {  
    public static void main (String args[])  
    { List list = new ArrayList();  
      list.add("abc");  
      list.add(new Integer(5)); //OK  
  
      for(Object obj : list){  
          //type casting leading to ClassCastException at runtime  
          String str=(String) obj;  
      }  
    }  
}
```

- il codice precedente avrebbe sollevato una `ClassCastException` a runtime
- nella slide successiva lo stesso codice dopo Java5

# JAVA GENERICS

- dopo l'introduzione dei generics

```
import java.util.*;
```

```
public class Generics2Ex {  
    public static void main (String args[])  
    {  
        List<String> list1 = new ArrayList();  
        list1.add("abc");  
        //list1.add(new Integer(5)); //compiler error  
        for(String str : list1){  
            //no type casting needed, avoids ClassCastException  
        }  
    }  
}
```

- specifico il tipo degli elementi della lista al momento della creazione della lista
- se inserisco nella lista elementi di tipo diverso, ottengo un errore a compile time

# JAVA GENERICS

- per definire il tipo generico: si usa nome placeholder (e.g. "T", "E") tra "<>" e separati da "," se più tipi
- classi generiche

```
public class Classe <T1, T2> {}
```

- metodi: prima del nome del metodo

```
private <T> void method(List<T> list) {}
```

- i generics possono essere "bounded", ossia ristretti rispetto ad altre classi usando la keyword `extends` (sia per classi che interfacce) aggiungendo la classe eventualmente presente come primo tipo

```
public class Classe <T extends Comparable<T>> { }
```

Comparable interfaccia

# JAVA GENERICS ED EREDITARIETA'

- attenzione a come i generics vengono utilizzati nelle collezioni!

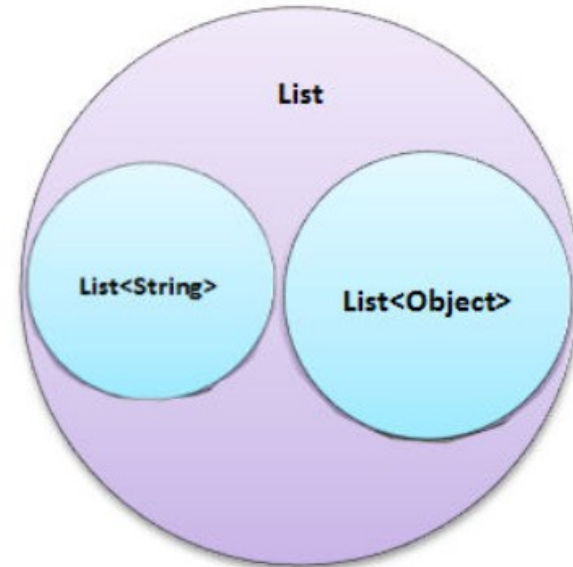
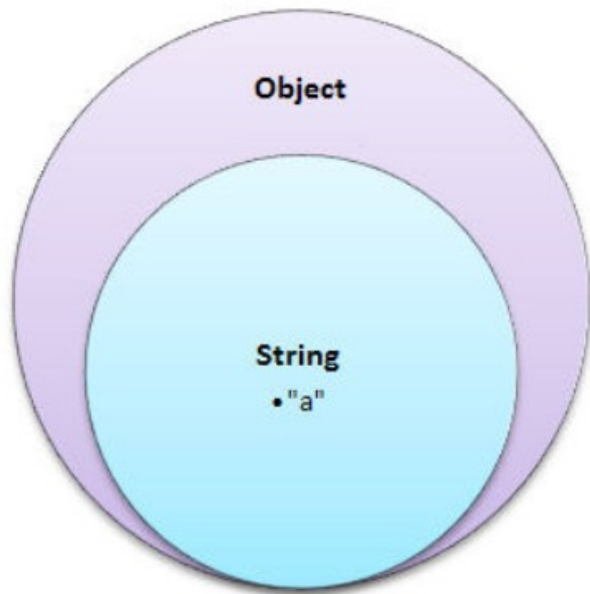
```
import java.util.*;
```

```
public class GenericInheritance {  
    public static void main (String args[])  
    { String a = "A";  
      Object b = a;  
      List<String> stringList = new ArrayList<>();  
      List<Object> objectList = stringList;  
      //error: cannot convert from List<String> to List<Object>  
    }  
}
```

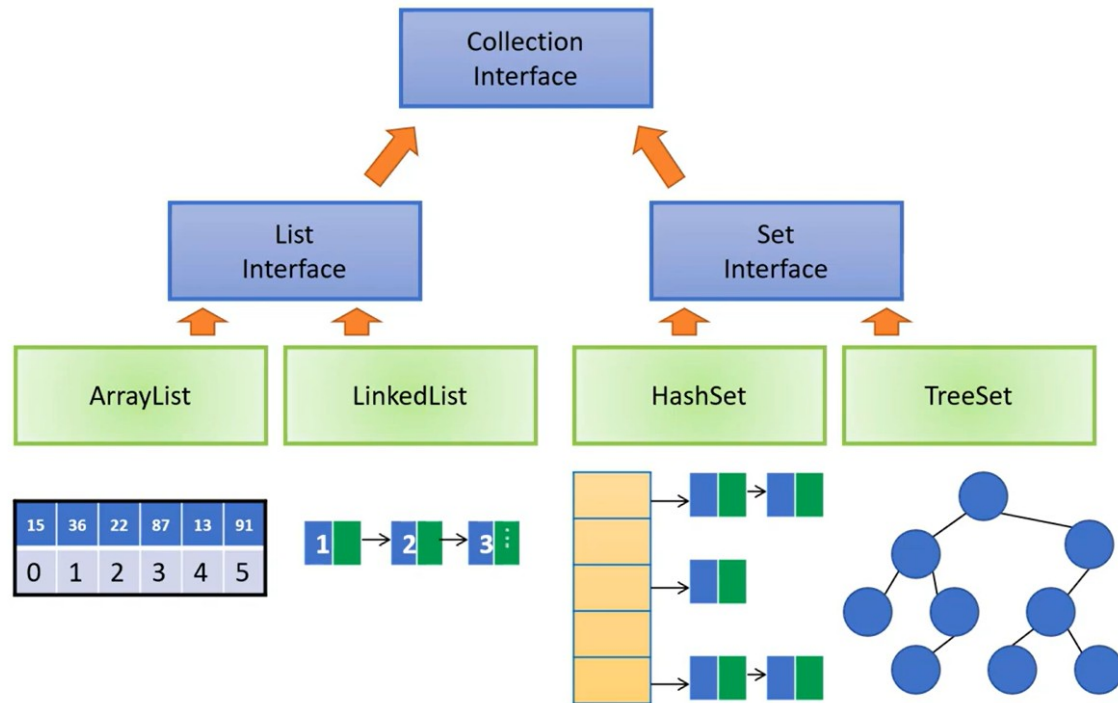
- se `List<Object>` è l'insieme di tutte le liste che contengono oggetti, allora `List<String>` dovrebbe essere un sottoinsieme.....
- questa è l'intuizione, però il compilatore segnala un errore.
- non è possibile fare upcasting!



# JAVA GENERICS ED EREDITARIETA'



# JAVA ITERATORS



- l'implementazione delle diverse collezioni è molto diversa
- ogni collezione memorizza i dati in modo diverso
- iteratori: consentono di definire un modo uniforme per scorrere gli elementi di ogni collezione, indipendentemente dalla loro implementazione

# JAVA ITERATORS

- un iteratore è un oggetto di supporto usato per accedere agli elementi di una collezione, uno alla volta e in sequenza
- è sempre associato ad un oggetto collezione
  - lavora su uno specifico insieme o lista
- per funzionare, un oggetto iteratore deve conoscere (e poter accedere) alla rappresentazione interna della classe che implementa la collezione (tabella hash, albero, array, lista puntata, ecc...)

# ARRAYLIST ITERATORS

```
import java.util.ArrayList;
import java.util.Iterator;
ArrayList <Integer> list = new ArrayList <Integer>();
list.add(15);
list.add(22);
list.add(19);
list.add(99);
Iterator here = list.iterator();
```



- notare che l'oggetto Iterator non viene creato con una new
- invece devo associarlo all'istanza della struttura dati che sto utilizzando, perché ogni struttura dati ha un diverso iteratore

# ARRAYLIST ITERATORS

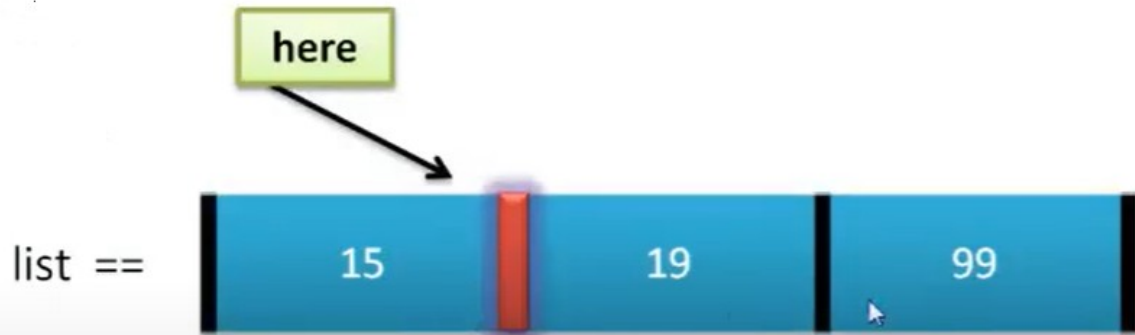
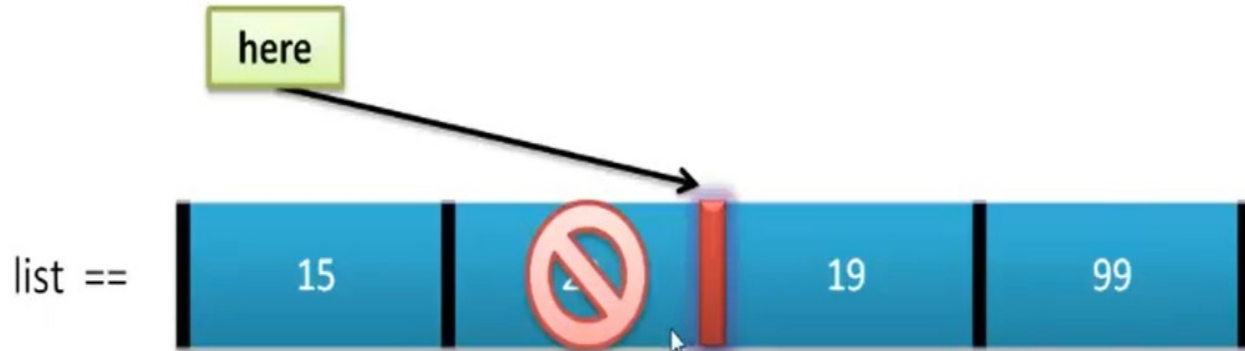
```
import java.util.ArrayList;
import java.util.Iterator;
ArrayList <Integer> list = new ArrayList <Integer>();
list.add(15);
list.add(22);
list.add(19);
list.add(99);
Iterator here = list.iterator();
System.out.println(here.next());
```



Stampa 15

# ARRAYLIST ITERATORS

```
ArrayList <Integer> list = new ArrayList <Integer>();  
list.add(15);  
list.add(22);  
list.add(19);  
list.add(99);  
Iterator here = list.iterator();  
here.next();  
here.next();  
here.remove();
```



# ARRAYLIST ITERATORS

```
ArrayList <String> list = new ArrayList <String>();  
list.add("Laura");  
list.add("Mario");  
list.add("Giovanni");  
list.add("Maria");  
Iterator here = list.iterator();  
String str= here.next();
```



- il compilatore restituisce errore
- gli iteratore memorizzano Objects
- Objects è la superclasse di tutti gli oggetti
- non posso assegnare a una sottoclasse un oggetto della superclasse!

# ARRAYLIST ITERATORS

```
ArrayList <String> list = new ArrayList <String>();  
list.add("Laura");  
list.add("Mario");  
list.add("Giovanni");  
list.add("Maria");  
Iterator <String> here = list.iterator();  
String str= here.next();  
System.out.println(str);
```



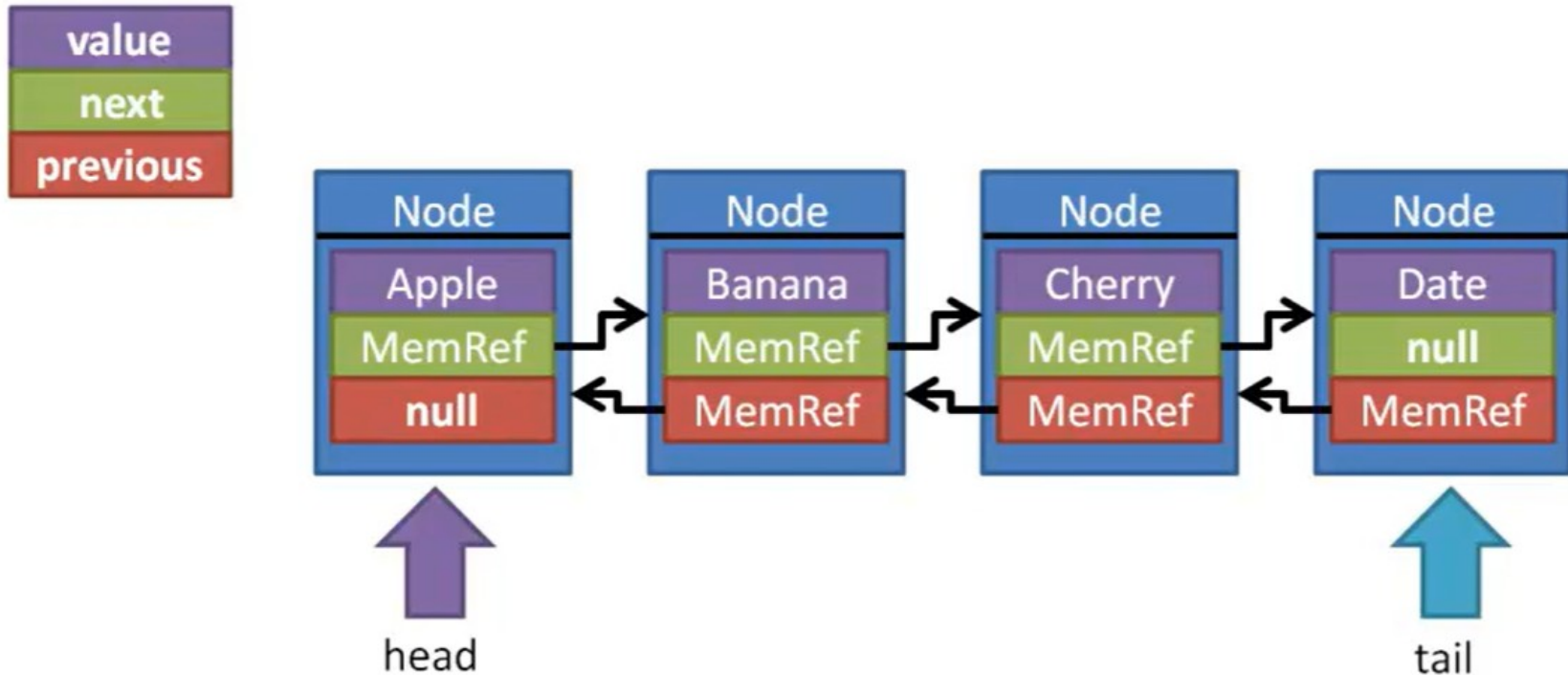
stampa "Laura"



# JAVA ITERATORS

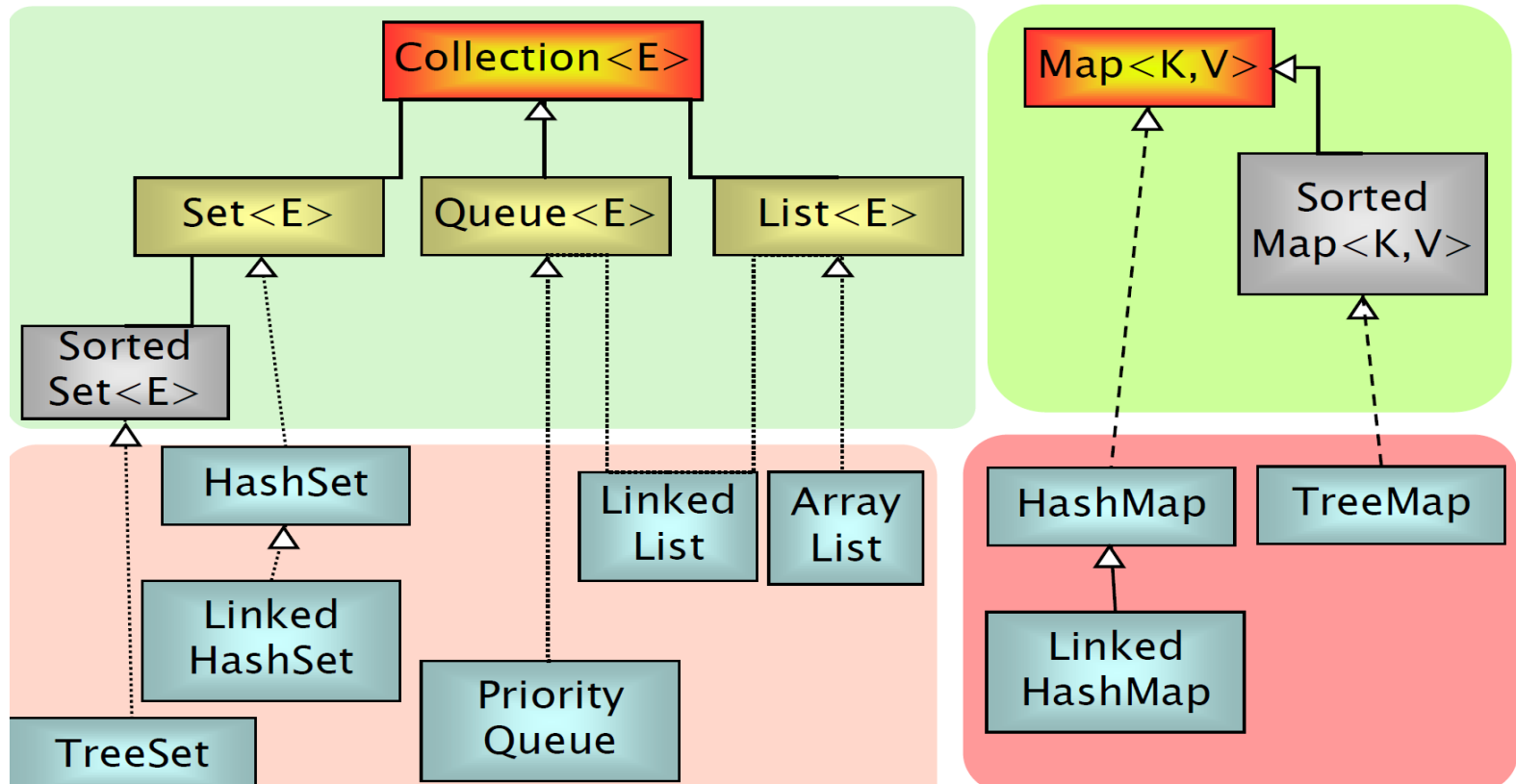
- l'interfaccia `Iterator` prevede i seguenti metodi
  - `next()` sposta il puntatore e restituisce il prossimo elemento della collezione
  - `hasNext()` che verifica se c'è un elemento successivo da fornire o se invece si è raggiunto la fine della collezione
  - `remove()` che elimina l'elemento nella posizione precedente al puntatore
- l'iteratore non ha alcuna funzione che lo “resetti”
  - una volta iniziata la scansione, non si può fare tornare indietro l'iteratore
  - una volta finita la scansione, l'iteratore non è più utilizzabile (bisogna crearne uno nuovo)

# LINKEDLIST



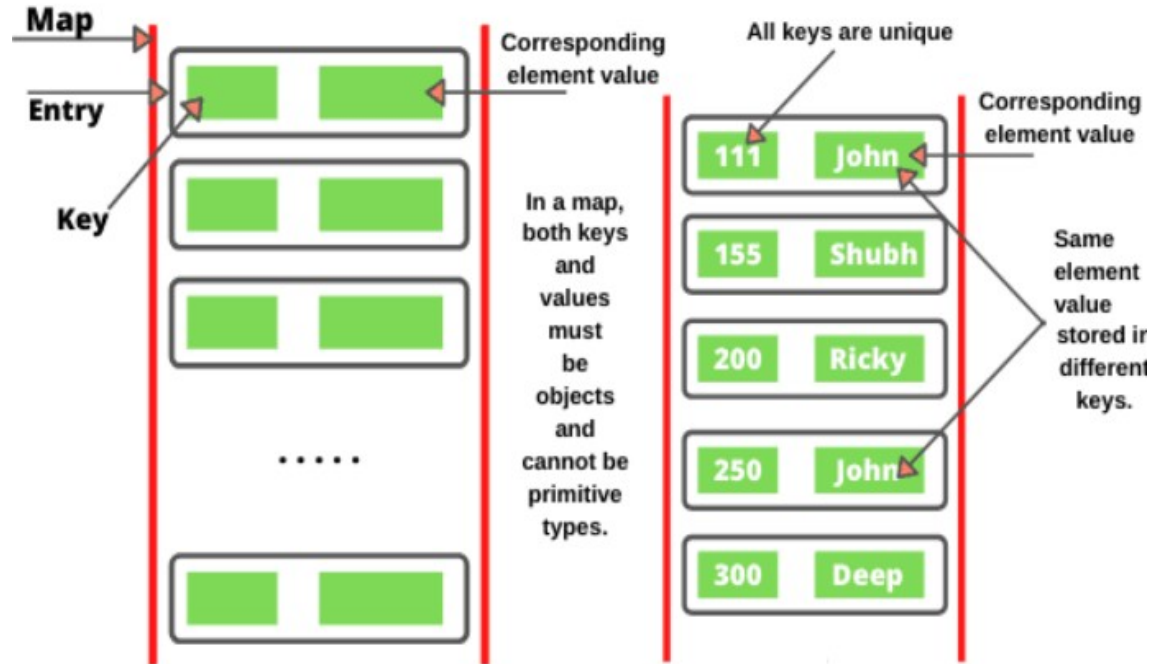
- reperimento primo elemento  $O(1)$
- reperimento ultimo elemento  $O(1)$
- reperimento in una posizione qualsiasi  $O(n)$

# ALTRE COLLECTIONS

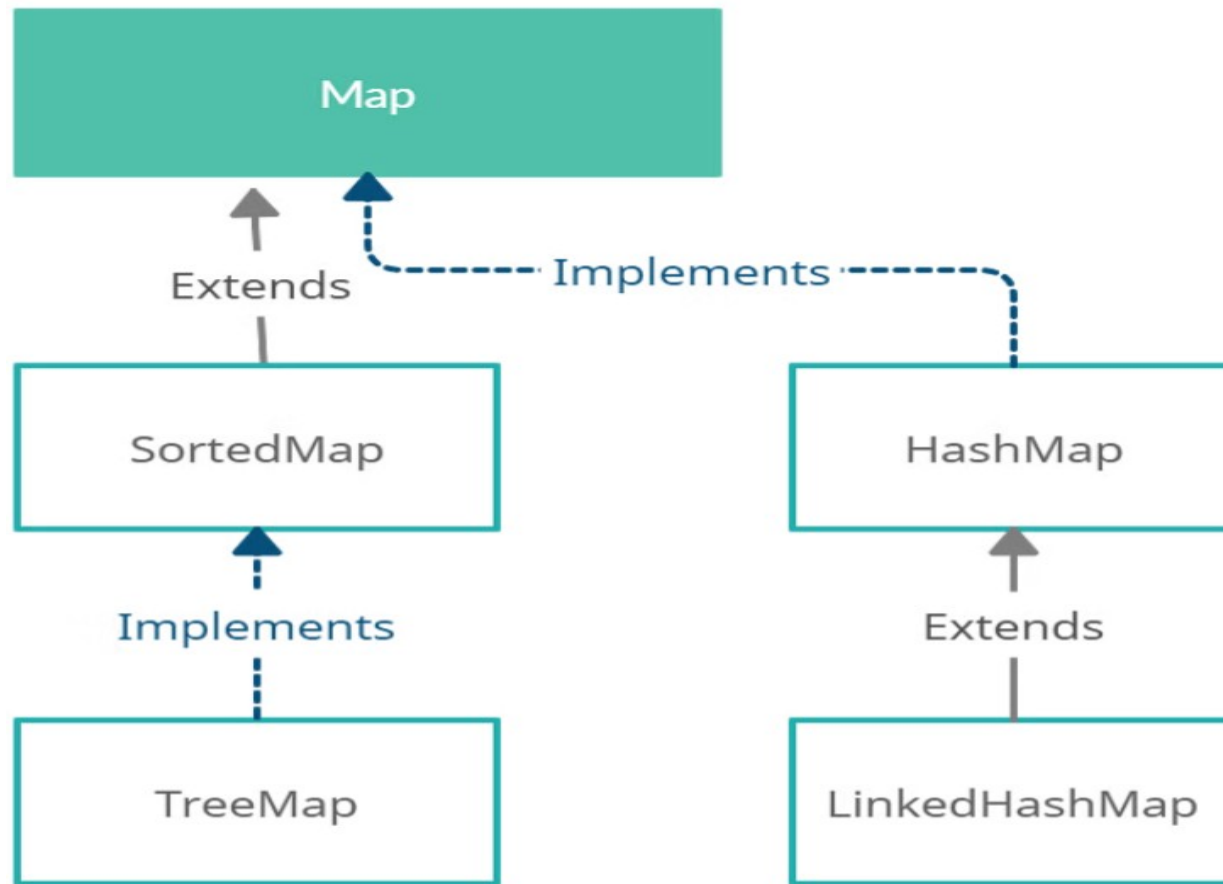


# JAVA MAP

- un container che associa chiavi a valori e fornisce la possibilità di accesso ai dati per chiave
- chiavi e valori devono essere oggetti
- non ci possono essere chiavi duplicate
- one-to-one mapping: ogni chiave mappa ad un solo valore

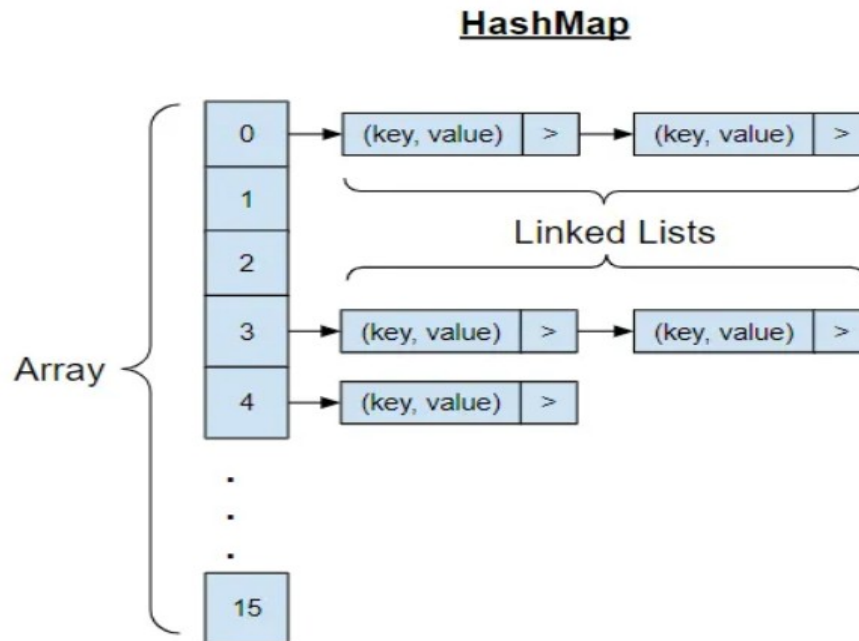


# MAP: IMPLEMENTAZIONI



# IMPLEMENTAZIONI DI MAP: HASHMAP

- l'implementazione comunemente utilizzata della interfaccia Map
- una collezione non ordinata di coppie (chiave-valore)
- fornisce **tempo costante  $O(1)$**  per le operazioni base, come put e get
- usa una funzione hash per associare un codice hash alla chiave, il codice viene quindi utilizzato per individuare il bucket che contiene l'elemento ricercato

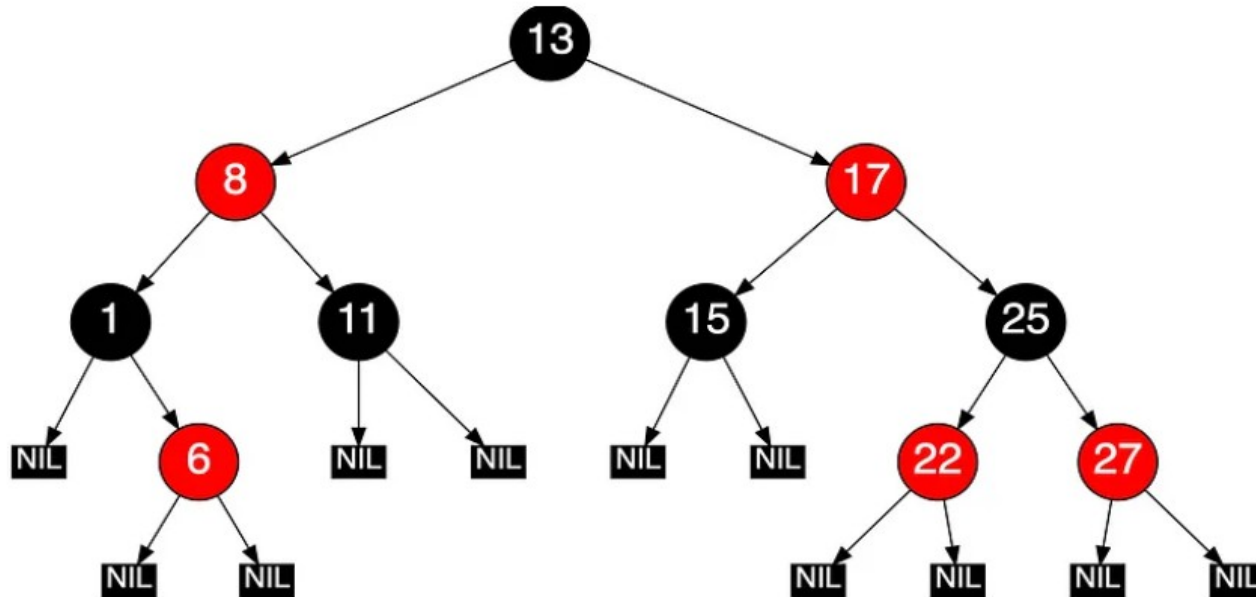


# JAVA HASHMAP

```
import java.util.*;
class Person
{String name; String Surname;
  Person(String name, String surname){
  this.name=name; this.Surname=surname;}}
public class HAsHMapExample {
public static void main(String args[])
{ Map<String,Person> people =new HashMap<>();
  people.put("XB3C", new Person("Laura", "Ricci"));
  people.put("AVBC", new Person ("Mario", "Rossi"));
  Person p = people.get("AVBC");
  if( p== null )
    System.out.println("Not found");
  else System.out.println(p.Surname);
  int populationSize = people.size();
  System.out.println(populationSize); }}
```

# IMPLEMENTAZIONI DI MAP: TREEMAP

- red-black tree based implementation
- garantisce che le chiavi possano essere restituite in ordine
- complessità  $\log(n)$  per le principali operazioni, get e put
- non possono esistere chiavi nulle





# DIFFERENZE TRA HASHMAP E TREEMAP

```
package HashTree;
import java.util.*;
public class HashMapVsTreeMapExample {
    public static void main(String[] args) {
        Map<Integer, String> hMap = new HashMap<Integer, String>();
        hMap.put(5, "A");
        hMap.put(11, "C");
        hMap.put(4, "Z");
        hMap.put(77, "Y");
        hMap.put(9, "P");
        hMap.put(66, "Q");
        hMap.put(0, "R");
        Map<Integer, String> tMap = new TreeMap<Integer, String>();
        tMap.put(5, "A");
        tMap.put(11, "C");
        tMap.put(4, "Z");
        tMap.put(77, "Y");
        tMap.put(9, "P");
        tMap.put(66, "Q");
        tMap.put(0, "R");
        // continua pagina successiva
    }
}
```

# DIFFERENZE TRA HASHMAP E TREEMAP

```
System.out.println("HashMap iteration order =====");
for (Map.Entry<Integer, String> entry : hMap.entrySet()) {
    System.out.println(entry.getKey() + " = " + entry.getValue());
}
System.out.println("\nTreeMap iteration order =====");
for (Map.Entry<Integer, String> entry : tMap.entrySet()) {
    System.out.println(entry.getKey() + " = " + entry.getValue());
} } }
```

HashMap iteration order =====

0 = R  
66 = Q  
4 = Z  
5 = A  
9 = P  
11 = C  
77 = Y

TreeMap iteration order =====

0 = R  
4 = Z  
5 = A  
9 = P  
11 = C  
66 = Q  
77 = Y

# ASSIGNMENT N. 2

- in un centro di calcolo le singole **unità di calcolo** che eseguono tasks sono inserite in slot, che vengono identificati univocamente in base a tre numeri
  - la fila
  - posizione relativa della colonna di unità nella fila
  - posizione relativa dell'unità all'interno di tale colonna (e.g. terza fila della stanza, quarta colonna della fila, dodicesima unità nella colonna).
- il coordinatore del centro deve mantenere una struttura dati efficiente per l'assegnazione e rimozione di task di ogni unità, con gestione FIFO in cui il task più vecchio viene completato e rimosso per primo.

# ASSIGNMENT N. 2

- si definisca una classe generica per la gestione di tuple ordinabili di lunghezza arbitraria e contenenti tipi qualsiasi ma che supportano un ordinamento.
- oltre a supportare l'ordinamento la classe offre getters (getters di tipo non primitivo) e setters per accedere o modificare l'elemento i-esimo.
- si utilizzi la classe delle tuple per individuare univocamente le unità.
- si rappresenti ogni task attraverso una stringa di lunghezza fissata di 16 caratteri.
- si scelga poi un'opportuna combinazione di collections per l'assegnamento e rimozione efficiente di task ad ogni unità.
- L'utente invia una richiesta (aggiunta, rimozione o visione del task corrente) specificando l'unità attraverso la propria tupla e il task stesso come stringa (solo per l'inserimento).
- si definisca un nuovo tipo di eccezione per gestire il caso in cui si tenti di creare una nuova unità con la stessa tripla identificativa di un'unità già esistente.