**Test Valutazione finale**

1. **Che funzione ha il Garbage Collector?**

* Tiene traccia degli oggetti utilizzati da un’applicazione Java
* Colleziona classi
* Colleziona oggetti

1. **Cosa vuol dire &&?**

|  |
| --- |
| * No |
| * Yes logico |
| 🗵 And logico |
|  |

**Cosa vuol dire ||?**

|  |
| --- |
| 🗵OR logico |
| * O logico |
| * No logico |

1. **Che cos'è una classe?**

|  |
| --- |
| 🗵Un insieme di oggetti |
| * Un insieme di alunni |
| * Un insieme di funzioni |

1. **Cosa indica Public?**

|  |
| --- |
| * Che l'intera applicazione è aperta ad ogni modifica |
| 🗵 L'intera classe è liberamente accessibile da ogni classe utilizzata nella applicazione |
|  |

1. **Cosa fa il metodo main()?**

|  |
| --- |
| * Viene eseguito per ultimo quando si lancia l'applicazione |
| 🗵Viene eseguito per primo quando si lancia l'applicazione |
| * Blocca l'applicazione |

1. **Quale simbolo si utilizza per concatenare le stringhe?**

|  |
| --- |
| 🗵 + |
| * & |
| * - |

1. **Che funzione ha il size()?**

|  |
| --- |
| * Non ha nessuna funzione |
| 🗵 Restituisce il numero degli elementi |
| * Definisce la misura degli oggetti |

1. **A che cosa serve l'istruzione IF?**

|  |
| --- |
| * A porre delle domande all'utente |
| * Ad eseguire un Interrupt File |
| 🗵 A controllare il flusso del programma |

1. **A cosa serve l'istruzione Break?**

|  |
| --- |
| * A prendere una pausa caffè |
| * A chiudere forzatamente il programma |
| 🗵 A forzare l'uscita da un ciclo |

1. **A cosa serve l'istruzione RETURN?**

|  |
| --- |
| * A tornare ad un preciso punto del programma |
| 🗵 A restituire un valore al termine dell'esecuzione di un metodo |
| * Ad andare a capo nell'istruzione |

1. **A cosa serve l'operatore ! ?**

|  |
| --- |
| 🗵 NOT logico |
| * Pericolo logico |
| * Alt |

1. **Il codice contenuto in un blocco finally ?**

|  |
| --- |
| 🗵 Viene sempre eseguito |
| * Viene eseguito solo se si verifica un’eccezione |
| * Non è mai eseguito |

1. **Affinchè un metodo non restituisca alcun valore si usa la parola chiave void?**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **La chiave "synchronized" acquisisce il lock dell'oggetto per renderlo inaccessibile da più processi contemporaneamente**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **I metodi definiti "private"**

|  |
| --- |
| * Sono liberamente accessibile da ogni classe utilizzata nella applicazione |
| 🗵Sono accessibili solo da altri metodi della stessa classe |
| * Nessuna delle precedenti |

1. **Nel linguaggio Java, il cast serve a:**

|  |
| --- |
| * Il cast non è possibile in Java |
| 🗵Ottenere una attribuzione forzata di tipo |
| * Convertire una variabile double in una variabile int |

1. **Il codice contenuto nei blocchi "try" e "catch" serve a gestire le eccezioni**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Il nome del file .java deve coincidere con:**

|  |
| --- |
| * Il nome del package che lo contiene |
| 🗵Il nome della classe pubblica, se esiste |
| * Con un metodo contenuto nella classe |

1. **l'istruzione "int[] a = new int[100];" :   Crea un array che può contenere 100 numeri interi**

|  |
| --- |
| 🗵Crea un array che può contenere 100 numeri interi |
| * Crea un array con un solo elemento intero del valore 100 |
| * E' sintatticamente errata |

1. **per utilizzare la classe esistente "Date" è necessario usare l'istruzione:**

|  |
| --- |
| * import java.util.Calendar.\*; |
| 🗵import java.util.Date; |
| * Non è necessario nessun import |

1. **una sottoclasse di una classe già esistente eredita tutto della superclasse ma accede solo a ciò che ha visibilità pubblica, protetta o package**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Che cosa è l'overloading delle funzioni in Java?**

|  |
| --- |
| * Ridefinendo un metodo ereditato da una superclasse, si specificano i compiti |
| * E' l'assegnamento di un nome unico a funzioni diverse |
| 🗵Nessuna delle precedenti |

1. **Che cosa è l'overriding delle funzioni in Java**

|  |
| --- |
| 🗵Ridefinendo un metodo ereditato da una superclasse, si specificano i compiti |
| * E' l'assegnamento di un nome unico a funzioni diverse |
| * Nessuna delle precedenti |

1. **Con quale parola chiave di Java si ottiene l'ereditarietà?**

|  |
| --- |
| * derived |
| * import |
| 🗵extends |

1. **E' possibile istanziare una interfaccia in Java?**

|  |
| --- |
| * Sempre |
| 🗵Mai |

1. **I file compilati Java hanno estensione:**

|  |
| --- |
| 🗵.class |
| * .java |
| * .js |

1. **I metodi di una interfaccia in Java sono sempre**

|  |
| --- |
| * final |
| * private |
| 🗵 public |
|  |

1. **L' operazione di garbage collection in Java è  Automatica?**

|  |
| --- |
| * Falso |
| 🗵 Vero |

1. **La superclasse alla radice della gerarchia delle classi in Java è** **Object**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Quale delle seguenti istruzioni Java dichiara una costante?**

|  |
| --- |
| * public double pGreco = 3.14; |
| * Private double pGreco = 3.14; |
| 🗵final double pGreco = 3.14; |

1. **Quali sono i tipi di numeri interi previsti in Java?**

|  |
| --- |
| * int, short, long, byte |
| 🗵float, double, int |
| * integer, boolean, float |

1. **Come si crea un Thread:**

|  |
| --- |
| * estendere la classe java.lang.Thread |
| * implementare l'interfaccia java.lang.Runnable. |
| 🗵Sia la A che la B |
| * Nessuna delle precedenti |

1. **Perché alcune eccezioni non mi generano errore durante la compilazione?**

|  |
| --- |
| 🗵Perché sono eccezioni unchecked |
| * Perché sono eccezioni checked |
| * Le eccezioni generano sempre errore in fasi di compilazione |

1. **il polimorfismo è la capacità di un oggetto di poter assumere diverse forme**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Nella collection List I duplicati non sono permessi?**

|  |
| --- |
| 🗵Falso |
| * Vero |

1. **Nella collection Set I duplicati non sono permessi?**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **La collection Map Non contiene chiavi duplicate, ogni chiave ha al massimo un valore.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Quali delle seguenti affermazioni, relative ai costruttori Java, è vera?**

|  |
| --- |
| 🗵Una classe può avere uno o più costruttori che non restituiscono mai alcun valore |
| * I costruttori sono solo quelli predefiniti dal linguaggio |
| * Un costruttore è sempre chiamato con 'new' e restituisce sempre un valore |

1. **Quale delle seguenti affermazioni è vera per i costruttori di default?**

|  |
| --- |
| 🗵Sono costruttori senza parametri |
| * Sono costruttori con almeno un parametro |
| * Non esistono |

1. **Quale delle affermazioni seguenti, relative alla gestione della memoria, è corretta?**

|  |
| --- |
| * La ripulitura di spazio in memoria è sempre compito del programmatore |
| 🗵La memoria è sottoposta automaticamente a garbage collection |
| * Non esiste garbage collection in Java |

1. **Che cosa fa, in Java, il metodo "finalize"?**

|  |
| --- |
| 🗵E' il "distruttore", il cui compito è liberare la memoria occupata dall'oggetto |
| * Libera eventuali risorse che il garbage collector non è in grado di liberare |
| * E' un metodo che viene richiamato quando l'handle esce dallo scope in cui è stato dichiarato |

1. **Se si tenta di dividere un numero di tipo "integer" per zero:**

|  |
| --- |
| 🗵Viene lanciata una eccezione |
| * Viene eseguita la divisione regolarmente |
| * Non viene lanciata una eccezione, ma la divisione non restituisce alcun risultato | |

1. **Il seguente metodo è thread-Safe ? perchè ?**

**class** MyCounter {

**private** **static** **int** counter = 0;

**public** **static** **int** getCount() {

**return** counter++;

}

}

No, non lo è perché non è presente alcun elemento che la garantisce (una chiamata al metodo wait(), la keyword synchronized all’interno della signature del metodo)

1. **come si può rendere thread-safe il metodo al punto precedente ?**

implementando uno degli accorgimenti scritti nella domanda precedente

1. **Qual'è la differenza tra una query LEFT JOIN e una RIGHT JOIN?**

In SQL il join viene utilizzato per combinare i dati di due o più tabelle. La differenza principale tra left e right join è che il primo ritorna tutti i dati della prima tabella che si sta considerando i dati della seconda che matchano la condizione del join. Il right esegue esattamente l’opposto: prende tutti i dati della seconda tabella e i dati della prima che matchano

1. **A cosa serve una query di JOIN?**

Serve per poter combinare i dati di due o più tabelle rispetto a una o più condizioni e restituisce una tabella unica con tutti i dati che rispettano la condizione.

1. **Cos'è una sequence?**

Si tratta di un’oggetto associato a uno schema che genera una sequenza di valori numerici in base alla specifica con la quale è stata creata la sequenza.

1. **Data il seguente estratto di codice, cosa si ottiene come risultato?**

public class Cruiser implements Runnable {  
    public static void main(String[] args) throws InterruptedException {  
        Thread a = new Thread(new Cruiser());  
        a.start();  
          
        System.out.print("Begin");  
        a.join();  
        System.out.print("End");  
    }  
      
    public void run() {  
        System.out.print("Run");  
    }  
}

|  |
| --- |
| * Non compila |
| * Viene lanciata un’eccezione a runtime. |
| 🗵Viene stampato "BeginRunEnd". |
| * Viene stampato "BeginEndRun". |
| * Viene stampato "BeginEnd". |

1. **Cosa è necessario per correggere gli errori di compilazione? (risposta multipla)**

public class Creature {  
    private int legCount;  
    private int wingCount;  
      
    public Creature(int legCount) {  
        this.legCount = this.legCount;  
        this.wingCount = 0;  
    }  
      
    public String toString() {  
        return "legs=" + this.legCount + " wings=" + wingCount;  
    }  
}  
  
public class Animal extends Creature {  
    public Animal(int legCount) {  
        this.wingCount = 0;  
    }  
  
}

|  |
| --- |
| * inserire una chiamata a super() nel costruttore di Creature. |
| 🗵inserire una chiamata a super() nel costruttore di Animal. |
| * inserire una chiamata a this()nel costruttore di Animal. |
| * inserire una chiamata a super(legCount) nel costruttore di Animal. |
| 🗵cambiare la visibilità della variabile wingCount nella classe Creature a protected. |
| * cambiare la stringa "this.wingCount = 0" nella classe Animal a "super.wingCount = 0" |

1. **Data il seguente estratto di codice, cosa si ottiene come risultato?**

public static void main(String args[]) {  
      Object myObj = new String[]{"one", "two", "three"} {  
          for (String s : (String[])myObj) System.out.print(s + ".");  
      }  
  }

|  |
| --- |
| * one.two.three. |
| * Non compila per un errore alla linea 2 |
| 🗵Non compila per un errore alla linea 3 |
| * Viene lanciata un’eccezione a runtime. |

1. **Data il seguente estratto di codice, quale affermazione è vera?**

    public void waitForSomething() {  
        SomeClass o = new SomeClass();  
        synchronized (o) {  
            o.wait();  
            o.notify();           
        }  
    }

|  |
| --- |
| 🗵Questo pezzo di codice potrebbe lanciare una InterruptedException |
| * Questo pezzo di codice potrebbe lanciare una IllegalStateException |
| * Questo pezzo di codice potrebbe lanciare una TimeOutException |
| * Invertendo l’ordine di o.wait() e o.notify() permette a questo metodo di essere eseguito senza eccezioni/errori. |

1. **Data il seguente estratto di codice, cosa si ottiene come risultato?**

import java.io.\*;  
  
public class Hotel implements Serializable {  
    private Room room = new Room();  
      
    public static void main(String[] args) {  
        Hotel h = new Hotel();  
        try {  
            FileOutputStream fos = new FileOutputStream("Hotel.dat");  
            ObjectOutputStream oos = new ObjectOutputStream(fos);  
            oos.writeObject(h);  
            oos.close();  
        } catch(Exception ex) {  
            ex.printStackTrace();  
        }  
    }  
}  
  
class Room implements Serializable {  
}

|  |  |
| --- | --- |
| * Non compila. | |
| * Viene lanciata un’eccezione a runtime. | |
| * Un’istanza della classe Hotel viene serializzata. | |
| 🗵Un’istanza della classe Hotel e Un’istanza della classe Room vengono entrambe serializzate. | |
| |  | | --- | |  | |

1. **Data il seguente estratto di codice, cosa si ottiene come risultato?**

public class TrickyNum<X extends Object> {  
      
    private X x;  
      
    public TrickyNum(X x) {  
        this.x = x;  
    }  
      
    private double getDouble() {  
        return x.doubleValue();  
    }  
      
    public static void main(String args[]) {  
        TrickyNum<Integer> a = new TrickyNum<Integer>(new Integer(1));  
        System.out.print(a.getDouble());  
    }  
}

|  |  |
| --- | --- |
| 🗵Non compila. | |
| * Viene lanciata un’eccezione a runtime. | |
| □Viene stampato "1.0". | |
| * Viene stampato "1". | |
| |  | | --- | |  | |

|  |
| --- |
|  |

1. **Data il seguente estratto di codice scegli le due affermazioni vere.**

public class Cruiser {  
    private int a = 0;  
      
    public void foo() {  
        Runnable r = new LittleCruiser();  
        new Thread(r).start();  
        new Thread(r).start();  
    }  
      
    public static void main(String arg[]) {  
        Cruiser c = new Cruiser();  
        c.foo();  
    }  
      
    public class LittleCruiser implements Runnable {  
        public void run() {  
            int current = 0;  
            for (int i = 0; i < 4; i++) {  
                current = a;  
                System.out.print(current + ", ");  
                a = current + 2;  
            }  
        }  
    }  
}

|  |
| --- |
| 🗵0, 2, 4, 0, 2, 4, 6, 6, |
| * 0, 2, 4, 6, 8, 10, 12, 14, |
| * 0, 2, 4, 6, 8, 10, 2, 4, |
| * 0, 0, 2, 2, 4, 4, 6, 6, 8, 8, 10, 10, 12, 12, 14, 14, |
| * 0, 2, 4, 6, 8, 10, 12, 14, 0, 2, 4, 6, 8, 10, 12, 14, |

1. **Un array è un oggetto e quindi può essere dichiarato, istanziato ed inizializzato.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Un array bidimensionale è un array i cui elementi sono altri array.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Il metodo length restituisce il numero degli elementi di un array.**

|  |  |
| --- | --- |
| 🗵Falso |  |
| * Vero |  |

1. **Un array non è ridimensionabile.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Un array è eterogeneo di default.**

|  |
| --- |
| * Falso |
| 🗵eVero |

1. **Un array di interi può contenere come elementi byte, ovvero le seguenti righe di codice non producono errori in compilazione:**

int arr [] = new int[2];

byte a = 1, b=2;

arr [0] = a;arr [1] = b;

|  |
| --- |
| 🗵Falso |
| * Vero |

1. **Un array di interi può contenere come elementi char, ovvero le seguenti righe di codice non producono errori in compilazione:**

char a = 'a', b = 'b';

int arr [] = {a,b};

|  |
| --- |
| 🗵Falso |
| * Vero |

1. **Un array di stringhe può contenere come elementi char, ovvero le seguenti righe di codice non producono errori in compilazione:**

String arr [] = {'a' , 'b'};

|  |
| --- |
| 🗵Falso |
| * Vero |

1. **Un array di stringhe è un array bidimensionale, perché le stringhe non sono altro che array di caratteri. Per esempio:**

String arr [] = {"a" , "b"};

**è un array bidimensionale.**

|  |
| --- |
| 🗵Falso |
| * Vero |

1. **Se abbiamo il seguente array bidimensionale:**

int arr [][]= {

{1, 2, 3},

{1,2},

{1,2,3,4,5}

};

**risulterà che:**

arr.length = 3;

arr[0].length = 3;

arr[1].length = 2;

arr[2].length = 5;

arr[0][0] = 1;

arr[0][1] = 2;

arr[0][2] = 3;

arr[1][0] = 1;

arr[1][1] = 2;

arr[1][2] = 3;

arr[2][0] = 1;

arr[2][1] = 2;

arr[2][2] = 3;

arr[2][3] = 4;

arr[2][4] = 5;

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **L’implementazione dell’ereditarietà implica scrivere sempre qualche riga in meno.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **La seguente dichiarazione di classe è scorretta:**

public final class Classe extends AltraClasse {...}

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **L’ereditarietà è utile solo se si utilizza la specializzazione. Infatti, specializzando ereditiamo nella sottoclasse (o sottoclassi) membri della superclasse che non bisogna riscrivere. Con la generalizzazione invece creiamo una classe in più, e quindi scriviamo più codice.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **La parola chiave super permette di chiamare metodi e costruttori di superclassi. La parola chiave this consente di chiamare metodi e costruttori della stessa classe in cui ci si trova.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **L’ereditarietà multipla non esiste in Java perché non esiste nella realtà.**

|  |
| --- |
| 🗵Falso |
| * Vero |

1. **Un’interfaccia funzionale è un’interfaccia che dichiara un unico metodo di default.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Una sottoclasse è più “grande” di una superclasse (nel senso che solitamente aggiunge caratteristiche e funzionalità nuove rispetto alla superclasse).**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Collection , Map, SortedMap, Set, List e SortedSet sono interfacce e non possono essere istanziate.**

|  |
| --- |
| 🗵Falso |
| * Vero |

1. **Un Set è una collezione ordinata di oggetti; una List non ammette elementi duplicati ed è ordinata.**

|  |
| --- |
| 🗵Falso |
| * Vero |

1. **Le mappe non possono contenere chiavi duplicate ed ogni chiave può essere associata ad un solo valore.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Esistono diverse implementazioni astratte da personalizzare nel framework come AbstractMap.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Una HashMap è più performante rispetto ad una Hashtable perché non è sincronizzata.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Una HashMap è più performante rispetto ad un TreeMap ma quest’ultima, essendo un’implementazione di SortedMap, gestisce l’ordinamento.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **HashSet è più performante rispetto a TreeSet ma non gestisce l’ordinamento.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Iterator ed Enumeration hanno lo stesso ruolo ma quest’ultima permette durante le iterazioni di rimuovere anche elementi.**

|  |
| --- |
| 🗵Falso |
| * Vero |

1. **ArrayList ha prestazioni migliori rispetto a Vector perché non è sincronizzato, ma entrambi hanno meccanismi per ottimizzare le prestazioni.**

**La classe Collections è una lista di Collection.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Quali delle seguenti affermazioni sono corrette?**

* L’interfaccia Iterable dichiara il metodo forEach.
* Iterator estende Iterable.
* Iterator definisce il metodo forEachRemaining().

🗵Collection implementa l’interfaccia Iterable.

1. **Quali delle seguenti affermazioni sono corrette?**

🗵Collection è una superclasse di List.

* Una Collection può essere trasformata in un array invocando il metodo

toArray() definito nella classe Arrays.

* Un array può essere trasformato in una Collection mediante il metodo

toCollection() della classe Arrays.

🗵Collection definisce il metodo add().

1. **La seguente dichiarazione di classe è scorretta:**

public abstract final class Classe {...}

|  |
| --- |
| 🗵Falso |
| * Vero |

1. **La seguente dichiarazione di classe è scorretta:**

public abstract class Classe;

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **La seguente dichiarazione di interfaccia è scorretta:**

public final interface Classe {...}

|  |
| --- |
| 🗵Falso |
| * Vero |

1. **Una classe astratta contiene per forza metodi astratti.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Un’interfaccia può essere estesa da un’altra interfaccia.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Una classe può estendere una sola classe ma implementare più interfacce.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Il pregio delle classi astratte e delle interfacce è che obbligano le sottoclassi ad implementare i metodi astratti ereditati. Quindi rappresentano un ottimo strumento per la progettazione object oriented.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Un’interfaccia può dichiarare più costruttori.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Un’interfaccia non può dichiarare variabili ma costanti statiche e pubbliche.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Una classe astratta può implementare un’interfaccia.**

|  |
| --- |
| 🗵Falso |
| * Vero |

1. **Quali di queste affermazioni è vera (potrebbero essere anche tutte vere)?**

🗵L’ereditarietà permette di mettere in relazione di aggregazione più classi.

🗵L’ereditarietà permette di mettere in relazione di aggregazione più interfacce.

* L’ereditarietà permette di mettere in relazione di estensione più classi ed interfacce.
* L’ereditarietà permette di mettere in relazione di aggregazione più classi ed interfacce.

1. **L’overload di un metodo implica scrivere un altro metodo con lo stesso nome e diverso tipo di ritorno.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **L’overload di un metodo implica scrivere un altro metodo con nome differente e stessa lista di parametri.**

|  |
| --- |
| 🗵Falso |
| * Vero |

1. **La segnatura (o firma) di un metodo è costituita dalla coppia identificatore - lista di parametri.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Per sfruttare l’override bisogna che sussista l’ereditarietà.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Per sfruttare l’overload bisogna che sussista l’ereditarietà.**

|  |
| --- |
| 🗵Falso |
| * Vero |

1. **Non tenendo conto del costrutto try with resources, il blocco finally è obbligatorio**

(scegliere tutte le affermazioni valide):

* Quando non ci sono blocchi catch dopo un blocco try.
* Quando non ci sono blocchi try prima di un blocco catch.
* Quando ci sono almeno due blocchi catch dopo un blocco try.

🗵Mai.

1. **Quali delle seguenti affermazioni sono corrette?**

🗵È possibile dichiarare solo checked exception.

* Se dichiariamo una sottoclasse di NullPointerException, questa verrà

lanciata insieme alla NullPointerException.

🗵Se dichiariamo una sottoclasse di NullPointerException, questa verrà

lanciata al posto della NullPointerException.

* Se dichiariamo una sottoclasse di ArithmeticException, questa verrà lanciata nel caso ci sia un problema in un’operazione aritmetica.

1. **Quali delle seguenti affermazioni sono corrette?**

🗵Le RuntimeException sono equivalenti alle unchecked exception.

* ArithmeticException è una checked exception.
* ClassCastException è una unchecked exception.

🗵NullPointerException è una checked exception.

1. **Quali delle seguenti affermazioni sono corrette?**

🗵Nella clausola throws è possibile dichiarare solo le checked exception.

* Nella clausola throws è possibile dichiarare solo le unchecked exception.

🗵Nella clausola throws è possibile dichiarare una NullPointerException.

* Con la clausola throw è possibile lanciare solo checked exception.
* Con la clausola throw è possibile lanciare solo unchecked exception.
* La clausola throws è obbligatoria se nel nostro metodo potrebbe essere lanciata una checked exception.
* Un metodo che dichiara una clausola throws può essere invocato solo se si

gestisce all’interno di un blocco try catch.

1. **Un thread è un oggetto istanziato dalla classe Thread o dalla classe Runnable.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Il multithreading è solitamente una caratteristica dei sistemi operativi e non dei linguaggi di programmazione.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **In ogni applicazione al runtime esiste almeno un thread in esecuzione.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **A parte il thread principale, un thread ha bisogno di eseguire codice all’interno di un oggetto la cui classe estende Runnable o estende Thread.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Il metodo run() deve essere chiamato dal programmatore per attivare un thread.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Il thread corrente non si identifica solitamente con il reference this.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Chiamando il metodo start() su di un thread, questo viene immediatamente eseguito.**

|  |
| --- |
| 🗵Falso |
| * Vero |

1. **Il metodo sleep() è statico e permette di far dormire per un numero specificato di millisecondi il thread che legge tale istruzione.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Assegnare le priorità ai thread è una attività che può produrre risultati diversi**

**su piattaforme diverse.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Lo scheduler della JVM non dipende dalla piattaforma su cui viene eseguito.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Un thread astrae un processore virtuale che esegue codice su determinati dati.**

|  |
| --- |
| 🗵Falso |
| * Vero |

1. **La parola chiave synchronized può essere utilizzata sia come modificatore di un metodo sia come modificatore di una variabile.**

|  |
| --- |
| 🗵Falso |
| * Vero |

1. **Il monitor di un oggetto può essere identificato con la parte sincronizzata dell’oggetto stesso.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Affinché due thread che eseguono lo stesso codice e condividono gli stessi dati non abbiano problemi di concorrenza, è necessario sincronizzare il codice comune.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Si dice che un thread ha il lock di un oggetto se entra nel suo monitor.**

|  |
| --- |
| 🗵Falso |
| * Vero |

1. **I metodi wait(), notify() e notifyAll() rappresentano il principale strumento per far comunicare più thread.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **I metodi suspend() e resume() sono attualmente deprecati.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Il metodo notityAll(), invocato su di un certo oggetto o1, risveglia dallo stato di pausa tutti i thread che hanno invocato wait() sullo stesso oggetto.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Tra questi verrà eseguito quello che era stato fatto partire per primo con il metodo start().**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Il deadlock è una condizione di errore bloccante generata da due thread che stanno in reciproca dipendenza in due oggetti sincronizzati.**

|  |
| --- |
| * Falso |
| 🗵Vero |

1. **Se un thread t1 esegue il metodo run() nell’oggetto o1 della classe C1, e un thread t2 esegue il metodo run() nell’oggetto o2 della stessa classe C1, la parola chiave synchronized non serve a niente.**

|  |
| --- |
| 🗵Falso |
| * Vero |