Esercizi
e
soluzioni

Esercizi modulo 1

Esercizio 1.a)

HelloWorld

Digitare, salvare, compilare ed eseguire il programma `HelloWorld`. Consigliamo al lettore di eseguire questo esercizio due volte: la prima volta utilizzando il Notepad e il prompt Dos, e la seconda utilizzando EJE.

> **EJE permette di inserire parti di codice preformattate tramite il menu "Inserisci". Per eventuali problemi con EJE è possibile consultare l'Appendice C.**

Esercizio 1.b)

Caratteristiche di Java, Vero o Falso:

1. Java è il nome di una tecnologia e contemporaneamente il nome di un linguaggio di programmazione.
2. Java è un linguaggio interpretato ma non compilato.
3. Java è un linguaggio veloce ma non robusto.
4. Java è un linguaggio difficile da imparare perché in ogni caso obbliga ad imparare l'object orientation.
5. La Java Virtual Machine è un software che supervisiona il software scritto in Java.
6. La JVM gestisce la memoria automaticamente mediante la Garbage Collection.
7. L'indipendenza dalla piattaforma è una caratteristica poco importante.
8. Java non è adatto per scrivere un sistema sicuro.
9. La Garbage Collection garantisce l'indipendenza dalla piattaforma.
10. Java è un linguaggio gratuito che raccoglie le caratteristiche migliori di altri linguaggi, e ne esclude quelle ritenute peggiori e più pericolose.

Esercizio 1.c)

Codice Java, Vero o Falso:

1. La seguente dichiarazione del metodo `main()` è corretta:
   ```
   public static main(String argomenti[]) {...}
   ```

2. La seguente dichiarazione del metodo `main()` è corretta:
   `public static void Main(String args[]){...}`
3. La seguente dichiarazione del metodo main() è corretta:
   `public static void main(String argomenti[]) {...}`
4. La seguente dichiarazione del metodo main() è corretta:
   `public static void main(String Argomenti[]) {...}`
5. La seguente dichiarazione di classe è corretta:
   `public class {...}`
6. La seguente dichiarazione di classe è corretta:
   `public Class Auto {...}`
7. La seguente dichiarazione di classe è corretta:
   `public class Auto {...}`
8. È possibile dichiarare un metodo al di fuori del blocco di codice che definisce una classe.
9. Il blocco di codice che definisce un metodo è delimitato da due parentesi tonde.
10. Il blocco di codice che definisce un metodo è delimitato da due parentesi quadre.

Esercizio 1.d)

Ambiente e processo di sviluppo, Vero o Falso:

1. La JVM è un software che simula un hardware.
2. Il bytecode è contenuto in un file con suffisso ".class".
3. Lo sviluppo Java consiste nello scrivere il programma, salvarlo, mandarlo in esecuzione ed infine compilarlo.
4. Lo sviluppo Java consiste nello scrivere il programma, salvarlo, compilarlo ed infine mandarlo in esecuzione.
5. Il nome del file che contiene una classe Java deve coincidere con il nome della classe, anche se non si tiene conto delle lettere maiuscole o minuscole.
6. Una volta compilato un programma scritto in Java è possibile eseguirlo su un qualsiasi sistema operativo che abbia una JVM.
7. Per eseguire una qualsiasi applicazione Java basta avere un browser.
8. Il compilatore del JDK viene invocato tramite il comando "javac" e la JVM viene invocata tramite il comando "java".

9. Per mandare in esecuzione un file che si chiama “pippo.class”, dobbiamo lanciare il seguente comando dal prompt: “java pippo.java”.
10. Per mandare in esecuzione un file che si chiama “pippo.class”, dobbiamo lanciare il seguente comando dal prompt: “java pippo.class”.

Soluzioni esercizi modulo 1

Esercizio 1.a)

HelloWorld

Non è fornita una soluzione per questo esercizio.

Esercizio 1.b)

Caratteristiche di Java, Vero o Falso:

1. **Vero**
2. **Falso**
3. **Falso**
4. **Vero**
5. **Vero**
6. **Vero**
7. **Falso**
8. **Falso**
9. **Falso**
10. **Vero**

Per avere spiegazioni sulle soluzioni dell’esercizio 1.b sopra riportate, il lettore può rileggere il paragrafo 1.1.

Esercizio 1.c)

Codice Java, Vero o Falso:

1. **Falso** manca il tipo di ritorno (`void`).
2. **Falso** l'identificatore dovrebbe iniziare con lettera minuscola (`main`).
3. **Vero**
4. **Vero**
5. **Falso** manca l'identificatore.
6. **Falso** la parola chiave si scrive con lettera iniziale minuscola (`Class`).
7. **Vero**
8. **Falso**
9. **Falso** le parentesi sono graffe.
10. **Falso** le parentesi sono graffe.

Esercizio 1.d)

Ambiente e processo di sviluppo, Vero o Falso:

1. **Vero**
2. **Vero**
3. **Falso** bisogna prima compilarlo per poi mandarlo in esecuzione.
4. **Vero**
5. **Falso** bisogna anche tenere conto delle lettere maiuscole o minuscole.
6. **Vero**
7. **Falso** un browser basta solo per eseguire applet.
8. **Vero**
9. **Falso** il comando giusto è java pippo.
10. **Falso** il comando giusto è java pippo.

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Saper definire il linguaggio di programmazione Java e le sue caratteristiche (unità 1.1)	☐	
Interagire con l'ambiente di sviluppo: il Java Development Kit (unità 1.2, 1.5, 1.6)	☐	
Saper digitare, compilare e mandare in esecuzione una semplice applicazione (unità 1.3, 1.4, 1.5, 1.6)	☐	

Note:

Esercizi modulo 2

Esercizio 2.a)

Tipi di dati fondamentali

Viene fornita (copiare, salvare e compilare) la seguente classe:

```
public class NumeroIntero
{
    public int numeroIntero;
    public void stampaNumero()
    {
        System.out.println(numeroIntero);
    }
}
```

Questa classe definisce il concetto di numero intero come oggetto. In essa vengono dichiarati una variabile (ovviamente) intera ed un metodo che stamperà la variabile stessa.

- ❏ Scrivere, compilare ed eseguire una classe che:istanzierà almeno due oggetti dalla classe `NumeroIntero` (contenente ovviamente un metodo `main()`), cambierà il valore delle relative variabili e testerà la veridicità delle avvenute assegnazioni, sfruttando il metodo `stampaNumero()`.
- ❏ Aggiungerà un costruttore alla classe `NumeroIntero`, che inizializzi la variabile d'istanza.

Due domande ancora:

1. A che tipologia di variabili appartiene la variabile `numeroIntero` definita nella classe `NumeroIntero`?
2. Se istanziamo un oggetto della classe `NumeroIntero`, senza assegnare un nuovo valore alla variabile `numeroIntero`, quanto varrà quest'ultima?

Esercizio 2.b)

Concetti sui componenti fondamentali, Vero o Falso:

1. Una variabile d'istanza deve essere per forza inizializzata dal programmatore.
2. Una variabile locale condivide il ciclo di vita con l'oggetto in cui è definita.
3. Un parametro ha un ciclo di vita coincidente con il metodo in cui è dichiarato: nasce quando il metodo viene invocato, muore quando termina il metodo.

4. Una variabile d'istanza appartiene alla classe in cui è dichiarata.
5. Un metodo è sinonimo di azione, operazione.
6. Sia le variabili sia i metodi sono utilizzabili di solito mediante l'operatore dot, applicato ad un'istanza della classe dove sono stati dichiarati.
7. Un costruttore è un metodo che non restituisce mai niente, infatti ha come tipo di ritorno `void.`
8. Un costruttore viene detto di default, se non ha parametri.
9. Un costruttore è un metodo e quindi può essere utilizzato mediante l'operatore dot, applicato ad un'istanza della classe dove è stato dichiarato.
10. Un package è fisicamente una cartella che contiene classi, le quali hanno dichiarato esplicitamente di far parte del package stesso nei rispettivi file sorgente.

Esercizio 2.c)

Sintassi dei componenti fondamentali. Vero o Falso:

1. Nella dichiarazione di un metodo, il nome è sempre seguito dalle parentesi che circondano i parametri opzionali, ed è sempre preceduto da un tipo di ritorno. Il seguente metodo è dichiarato in maniera corretta:
```
public void metodo ()
{
  return 5;
}
```
2. Il seguente metodo è dichiarato in maniera corretta:
```
public int metodo ()
{
  System.out.println("Ciao");
}
```
3. La seguente variabile è dichiarata in maniera corretta:
```
public int a = 0;
```
4. La seguente variabile `x` è utilizzata in maniera corretta (fare riferimento alla classe `Punto` definita in questo modulo):
```
Punto p1 = new Punto();
Punto.x = 10;
```
5. La seguente variabile `x` è utilizzata in maniera corretta (fare riferimento alla classe `Punto` definita in questo modulo):
```
Punto p1 = new Punto();
Punto.p1.x = 10;
```

6. La seguente variabile `x` è utilizzata in maniera corretta (fare riferimento alla classe `Punto` definita in questo modulo):
```
Punto p1 = new Punto();
x = 10;
```
7. Il seguente costruttore è utilizzato in maniera corretta (fare riferimento alla classe `Punto` definita in questo modulo):
```
Punto p1 = new Punto();
p1.Punto();
```
8. Il seguente costruttore è dichiarato in maniera corretta:
```
public class Computer {
  public void Computer()
  {

  }
}
```
9. Il seguente costruttore è dichiarato in maniera corretta:
```
public class Computer {
  public computer(int a)
  {

  }
}
```

Soluzioni esercizi modulo 2

Esercizio 2.a)

Tipi di dati fondamentali

Di seguito viene listata una classe che aderisce ai requisiti richiesti:

```
public class ClasseRichiesta
{
  public static void main (String args [])
  {
    NumeroIntero uno = new NumeroIntero();
    NumeroIntero due = new NumeroIntero();
    uno.numeroIntero = 1;
    due.numeroIntero = 2;
    uno.stampaNumero();
    due.stampaNumero();
```

```
    }
}
```

Inoltre un costruttore per la classe `NumeroIntero` potrebbe impostare l'unica variabile d'istanza `numeroIntero`:

```
public class NumeroIntero
{
    public int numeroIntero;
    public NumeroIntero(int n)
    {
        numeroIntero = n;
    }
    public void stampaNumero()
    {
        System.out.println(numeroIntero);
    }
}
```

In tal caso, però, per istanziare oggetti dalla classe `NumeroIntero`, non sarà più possibile utilizzare il costruttore di default (che non sarà più inserito dal compilatore). Quindi la seguente istruzione produrrebbe un errore in compilazione:

```
NumeroIntero uno = new NumeroIntero();
```

Bisogna invece creare oggetti passando al costruttore direttamente il valore della variabile da impostare, per esempio:

```
NumeroIntero uno = new NumeroIntero(1);
```

Risposte alle due domande:

1. Trattasi di una variabile d'istanza, perché dichiarata all'interno di una classe, al di fuori di metodi.
2. Il valore sarà zero, ovvero il valore nullo per una variabile intera. Infatti, quando si istanzia un oggetto, le variabili d'istanza vengono inizializzate ai valori nulli, se non esplicitamente inizializzate ad altri valori.

Esercizio 2.b)

Concetti sui componenti fondamentali. Vero o Falso:

1. **Falso** una variabile locale deve essere per forza inizializzata dal programmatore.
2. **Falso** una variabile d'istanza condivide il ciclo di vita con l'oggetto in cui è definita.

3. **Vero.**
4. **Falso** una variabile d'istanza appartiene ad un oggetto istanziato dalla classe in cui è dichiarata.
5. **Vero.**
6. **Vero** .
7. **Falso** un costruttore è un metodo che non restituisce mai niente, infatti non ha tipo di ritorno.
8. **Falso** un costruttore viene detto di default, se viene inserito dal compilatore. Inoltre non ha parametri.
9. **Falso** un costruttore è un metodo speciale che ha la caratteristica di essere invocato una ed una sola volta nel momento in cui si istanzia un oggetto.
10. **Vero.**

Esercizio 2.c)

Sintassi dei componenti fondamentali. Vero o Falso:

1. **Vero.**
2. **Falso** tenta di restituire un valore intero avendo tipo di ritorno `void.`
3. **Falso** il metodo dovrebbe restituire un valore intero.
4. **Vero.**
5. **Falso** l'operatore dot deve essere applicato all'oggetto e non alla classe:
```
Punto p1 = new Punto();
p1.x = 10;
```
6. **Falso** L'operatore dot deve essere applicato all'oggetto e non alla classe, ed inoltre la classe non "contiene" l'oggetto.
7. **Falso** L'operatore dot deve essere applicato all'oggetto. Il compilatore non troverebbe infatti la dichiarazione della variabile x.
8. **Falso** un costruttore è un metodo speciale che ha la caratteristica di essere invocato una ed una sola volta nel momento in cui si istanzia un oggetto.
9. **Falso** il costruttore non dichiara tipo di ritorno e deve avere nome coincidente con la classe.
10. **Falso** il costruttore deve avere nome coincidente con la classe.

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Saper definire i concetti di classe, oggetto, variabile, metodo e costruttore (unità 2.1, 2.2, 2.3, 2.4, 2.5)	☐	
Saper dichiarare una classe (unità 2.1)	☐	
Istanziare oggetti da una classe (unità 2.2)	☐	
Utilizzare i membri pubblici di un oggetto sfruttando l'operatore dot (unità 2.2, 2.3, 2.4)	☐	
Dichiarare ed invocare un metodo (unità 2.3)	☐	
Saper dichiarare ed inizializzare una variabile (unità 2.4)	☐	
Saper definire ed utilizzare i diversi tipi di variabili (d'istanza, locali e parametri formali) (unità 2.4)	☐	
Dichiarare ed invocare un metodo costruttore (unità 2.5)	☐	
Comprendere il costruttore di default (unità 2.5)	☐	

Note:

Esercizi modulo 3

Esercizio 3.a)

Operazioni aritmetiche

Scrivere un semplice programma che svolga le seguenti operazioni aritmetiche correttamente, scegliendo accuratamente i tipi di dati da utilizzare per immagazzinare i risultati di esse.
Una divisione tra due interi `a = 5`, e `b = 3`. Immagazzinare il risultato in una variabile `r1`, scegliendone il tipo di dato appropriatamente.
Una moltiplicazione tra un `char c = 'a'`, ed uno `short s = 5000`. Immagazzinare il risultato in una variabile `r2`, scegliendone il tipo di dato appropriatamente.
Una somma tra un `int i = 6` ed un `float f = 3.14F`. Immagazzinare il risultato in una variabile `r3`, scegliendone il tipo di dato appropriatamente.
Una sottrazione tra `r1`, `r2` e `r3`. Immagazzinare il risultato in una variabile `r4`, scegliendone il tipo di dato appropriatamente.
Verificare la correttezza delle operazioni stampandone i risultati parziali ed il risultato finale. Tenere presente la promozione automatica nelle espressioni, ed utilizzare il casting appropriatamente.

Basta una classe con un `main()` che svolge le operazioni.

Esercizio 3.b)

Accesso alle proprietà

Scrivere un programma con i seguenti requisiti.
Utilizza una classe `Persona` che dichiara le variabili `nome`, `cognome`, `eta` (età). Si dichiari inoltre un metodo `dettagli()` che restituisce in una stringa le informazioni sulla persona in questione. Ricordarsi di utilizzare le convenzioni e le regole descritte in questo modulo.
Utilizza una classe `Principale` che, nel metodo `main()`, istanzia due oggetti chiamati `persona1` e `persona2` della classe `Persona`, inizializzando per ognuno di essi i relativi campi con sfruttamento dell'operatore dot.
Dichiarare un terzo reference (`persona3`) che punti ad uno degli oggetti già

istanziati. Controllare che effettivamente `persona3` punti allo oggetto voluto, stampando i campi di `persona3` sempre mediante l'operatore dot.
Commentare adeguatamente le classi realizzate e sfruttare lo strumento javadoc per produrre la relativa documentazione.

Nella documentazione standard di Java sono usate tutte le regole e le convenzioni descritte in questo capitolo. Basta osservare che `String` inizia con lettera maiuscola, essendo una classe. Si può concludere che anche `System` è una classe.

Esercizio 3.c)

Array, Vero o Falso:

1. Un array è un oggetto e quindi può essere dichiarato, istanziato ed inizializzato.
2. Un array bidimensionale è un array i cui elementi sono altri array.
3. Il metodo `length` restituisce il numero degli elementi di un array.
4. Un array non è ridimensionabile.
5. Un array è eterogeneo di default.
6. 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;
```
7. 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};
```
8. Un array di stringhe può contenere come elementi `char`, ovvero le seguenti righe di codice non producono errori in compilazione:
```
  String arr [] = {'a' , 'b'};
```
9. 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.
```
10. 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;
```

Soluzioni esercizi modulo 3

Esercizio 3.a)

Operazioni aritmetiche

```
public class Esercizio3A {
  public static void main (String args[]) {
    int a = 5, b = 3;
    double r1 = (double)a/b;
    System.out.println("r1 = " + r1);
    char c = 'a';
    short s = 5000;
    int r2 = c*s;
    System.out.println("r2 = " + r2);
    int i = 6;
    float f = 3.14F;
    float r3 = i + f;
    System.out.println("r3 = " + r3);
    double r4 = r1 – r2 – r3;
    System.out.println("r4 = " + r4);
```

```
  }
}
```

Esercizio 3.b)

Accesso alle proprietà

```
public class Persona {
  public String nome;
  public String cognome;
  public int eta;
  public String dettagli() {
    return nome + " " + cognome + " anni " + eta;
  }
}
public class Principale {
  public static void main (String args []) {
    Persona persona1 = new Persona();
    Persona persona2 = new Persona();
    persona1.nome = "Mario";
    persona1.cognome = "Rossi";
    persona1.eta = 30;
    System.out.println("persona1 "+persona1.dettagli());
    persona2.nome = "Giuseppe";
    persona2.cognome = "Verdi";
    persona2.eta = 40;
    System.out.println("persona2 "+persona2.dettagli());
    Persona persona3 = persona1;
    System.out.println("persona3 "+persona3.dettagli());
  }
}
```

Esercizio 3.c)

Array, Vero o Falso:

1. **Vero.**
2. **Vero**.
3. **Falso** la variabile `length` restituisce il numero degli elementi di un array.
4. **Vero.**
5. **Falso.**
6. **Vero** un `byte` (che occupa solo 8 bit) può essere immagazzinato in una variabile `int` (che occupa 32 bit).

7. **Vero** un `char` (che occupa 16 bit) può essere immagazzinato in una variabile `int` (che occupa 32 bit).
8. **Falso** un `char` è un tipo di dato primitivo e `String` è una classe. I due tipi di dati non sono compatibili.
9. **Falso** in Java la stringa è una oggetto istanziato dalla classe `String` e non un array di caratteri.
10. **Falso** tutte le affermazioni sono giuste tranne `arr[1][2] = 3;` perché questo elemento non esiste.

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Saper utilizzare le convenzioni per il codice Java (unità 3.1)	☐	
Conoscere e saper utilizzare tutti i tipi di dati primitivi (unità 3.2)	☐	
Saper gestire casting e promotion (unità 3.2)	☐	
Saper utilizzare i reference e capirne la filosofia (unità 3.4)	☐	
Iniziare ad esplorare la documentazione della libreria standard di Java (unità 3.4)	☐	
Saper utilizzare la classe String (unità 3.4)	☐	
Saper utilizzare gli array (unità 3.5)	☐	

Note:

Esercizi modulo 4

Esercizio 4.a)

Stampa primi cinque numeri pari

Scrivere un semplice programma, costituito da un'unica classe, che sfruttando esclusivamente un ciclo infinito, l'operatore modulo, due costrutti `if`, un `break` ed un `continue`, stampi solo i primi cinque numeri pari.

Esercizio 4.b)

Stampa alfabeto

Scrivere un'applicazione che stampi i 26 caratteri dell'alfabeto (inglese-americano) con un ciclo.

Esercizio 4.c)

Tavola pitagorica

Scrivere una semplice classe che stampi a video la tavola pitagorica.

Suggerimento 1: non sono necessari array.
Suggerimento 2: il metodo `System.out.println()` stampa l'argomento che gli viene passato e poi sposta il cursore alla riga successiva; infatti `println` sta per "print line". Esiste anche il metodo `System.out.print()`, che invece stampa solamente il parametro passatogli.
Suggerimento 3: sfruttare un doppio ciclo innestato

Esercizio 4.d)

Operatori e flusso di esecuzione, Vero o Falso:

1. Gli operatori unari di pre-incremento e post-incremento applicati ad una variabile danno lo stesso risultato, ovvero se abbiamo:
   ```
   cint i = 5;
   ```
   sia
   ```
   i++;
   ```
   sia

```
++i;
```

aggiornano il valore di `i` a `6`;

2. `d += 1` è equivalente a `d++` dove `d` è una variabile `double`.
3. Se abbiamo:

```
int i = 5;
int j = ++i;

int k = j++;
int h = k—;
boolean flag = ((i != j) && ( (j <= k ) ||  (i <= h)) );
```

`flag` avrà valore `false`.

4. L'istruzione:

```
System.out.println(1 + 2 + “3”);
```

stamperà `33`.

5. Il costrutto `switch` può in ogni caso sostituire il costrutto `if`.
6. L'operatore ternario può in ogni caso sostituire il costrutto `if`.
7. Il costrutto `for` può in ogni caso sostituire il costrutto `while`.
8. Il costrutto `do` può in ogni caso sostituire il costrutto `while`.
9. Il costrutto `switch` può in ogni caso sostituire il costrutto `while`.
10. I comandi `break` e `continue` possono essere utilizzati nei costrutti `switch`, `for`, `while` e `do` ma non nel costrutto `if`.

Soluzioni esercizi modulo 4

Esercizio 4.a)

Stampa primi cinque numeri pari

```
public class TestPari {
  public static void main(String args[]){
    int i = 0;
    while (true)
    {
      i++;
      if (i > 10)
        break;
      if ((i % 2) != 0)
        continue;
```

```
      System.out.println(i);
    }
  }
}
```

Esercizio 4.b)

Stampa alfabeto

```
public class TestArray {
  public static void main(String args[]){
    for (int i = 0; i < 26; ++i){
      char c = (char)('a' + i);
      System.out.println(c);
    }
  }
}
```

Esercizio 4.c)

Tavola pitagorica

```
public class Tabelline {
  public static void main(String args[]) {
    for (int i = 1; i <= 10; ++i){
      for (int j = 1; j <= 10; ++j){
        System.out.print(i*j + "\t");
      }
      System.out.println();
    }
  }
}
```

Esercizio 4.d)

Operatori e flusso di esecuzione, Vero o Falso:

1. **Vero.**
2. **Vero.**
3. **Falso** la variabile booleana `flag` avrà valore `true`. Le espressioni "atomiche" valgono rispettivamente `true-false-true`, sussistendo le seguenti uguaglianze : `i = 6, j = 7, k = 5, h = 6`. Infatti `(i != j)` vale `true` e inoltre `(i <= h)` vale `true`. L' espressione `( (j <= k ) || (i <= h)) )` vale `true`, sussistendo l'operatore OR. Infine l'operatore AND fa sì che la variabile `flag` valga `true.`

4. **Vero.**
5. **Falso** `switch` può testare solo una variabile intera (o compatibile) confrontandone l'uguaglianza con costanti (in realtà dalla versione 5 si possono utilizzare come variabili di test anche le enumerazioni e il tipo `Integer`…). Il costrutto `if` permette di svolgere controlli incrociati sfruttando differenze, operatori booleani etc…
6. **Falso** l'operatore ternario è sempre vincolato ad un'assegnazione del risultato ad una variabile. Questo significa che produce sempre un valore da assegnare e da utilizzare in qualche modo (per esempio passando un argomento invocando un metodo). Per esempio, se `i` e `j` sono due interi, la seguente espressione:
   ```
   i < j ? i : j;
   ```
   provocherebbe un errore in compilazione (oltre a non avere senso).
7. **Vero.**
8. **Falso** il `do` in qualsiasi caso garantisce l'esecuzione della prima iterazione sul codice. Il `while` potrebbe prescindere da questa soluzione.
9. **Falso** lo `switch` è una condizione non un ciclo.
10. **Falso** il `continue` non si può utilizzare nello `switch` ma solo nei cicli.

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Conoscere e saper utilizzare i vari operatori (unità 4.1)	☐	
Conoscere e saper utilizzare i costrutti di programmazione semplici (unità 4.2, 4.3)	☐	
Conoscere e saper utilizzare i costrutti di programmazione avanzati (unità 4.2, 4.4)	☐	

Note:

Esercizi modulo 5

Esercizio 5.a)

Object Orientation in generale (teoria), Vero o Falso:

1. L'Object Orientation esiste solo da pochi anni.
2. Java è un linguaggio object oriented non puro, SmallTalk è un linguaggio object oriented puro.
3. Tutti i linguaggi orientati agli oggetti supportano allo stesso modo i paradigmi object oriented.
4. Si può dire che un linguaggio è object oriented se supporta incapsulamento, ereditarietà e polimorfismo; infatti altri paradigmi come l'astrazione e il riuso appartengono anche alla filosofia procedurale.
5. Applicare l'astrazione significa concentrarsi solo sulle caratteristiche importanti dell'entità da astrarre.
6. La realtà che ci circonda è fonte d'ispirazione per la filosofia object oriented.
7. L'incapsulamento ci aiuta ad interagire con gli oggetti, l'astrazione ci aiuta ad interagire con le classi.
8. Il riuso è favorito dall'implementazione degli altri paradigmi object oriented.
9. L'ereditarietà permette al programmatore di gestire in maniera collettiva più classi.
10. L'incapsulamento divide gli oggetti in due parti separate: l'interfaccia pubblica e l'implementazione interna. Per l'utilizzo dell'oggetto basta conoscere l'implementazione interna e non bisogna conoscere l'interfaccia pubblica.

Esercizio 5.b)

Object Orientation in Java (teoria), Vero o Falso:

1. L'implementazione dell'ereditarietà implica scrivere sempre qualche riga in meno.
2. L'implementazione dell'incapsulamento implica scrivere sempre qualche riga in più.
3. 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.

4. Implementare l'incapsulamento non è tecnicamente obbligatorio in Java, ma indispensabile per programmare correttamente.
5. L'ereditarietà multipla non esiste in Java perché non esiste nella realtà.
6. L'interfaccia pubblica di un oggetto è costituita anche dai metodi accessor e mutator.
7. Una sottoclasse è più "grande" di una superclasse (nel senso che solitamente aggiunge caratteristiche e funzionalità nuove rispetto alla superclasse).
8. Supponiamo di sviluppare un'applicazione per gestire un torneo di calcio. Esiste ereditarietà derivata da specializzazione tra le classi `Squadra` e `Giocatore.`
9. Supponiamo di sviluppare un'applicazione per gestire un torneo di calcio. Esiste ereditarietà derivata da generalizzazione tra le classi `Squadra` e `Giocatore.`
10. In generale, se avessimo due classi `Padre` e `Figlio`, non esisterebbe ereditarietà tra queste due classi.

Esercizio 5.c)

Object Orientation in Java (pratica), Vero o Falso:

1. L'implementazione dell'ereditarietà implica l'utilizzo della parola chiave `extends.`
2. L'implementazione dell'incapsulamento implica l'utilizzo delle parole chiave `set` e `get.`
3. Per utilizzare le variabili incapsulate di una superclasse in una sottoclasse, bisogna dichiararle almeno `protected.`
4. I metodi dichiarati privati non vengono ereditati nelle sottoclassi.
5. L'ereditarietà multipla in Java non esiste ma si può solo simulare con le interfacce.
6. Una variabile privata risulta direttamente disponibile (tecnicamente come se fosse pubblica) tramite l'operatore dot, a tutte le istanze della classe in cui è dichiarata.
7. La parola chiave `this` permette di referenziare i membri di un oggetto che sarà creato solo al runtime all'interno dell'oggetto stesso
8. Se compiliamo la seguente classe:

```
public class CompilatorePensaciTu {
    private int var;
```

```
  public void setVar(int v) {
    var = v;
  }
  public int getVar()
    return var;
  }
}
```

il compilatore in realtà la trasformerà in:

```
import java.lang.*;

public class CompilatorePensaciTu extends Object {
  private int var;
  public CompilatorePensaciTu() {
  }
  public void setVar(int v) {
    this.var = v;
  }
  public int getVar()
    return this.var;
  }
}
```

9. Compilando le seguenti classi, non si otterranno errori in compilazione:

```
public class Persona {
  private String nome;
  public void setNome(String nome) {
    this.nome = nome;
  }
  public String getNome() {
    return this.nome;
  }
}

public class Impiegato extends Persona {
  private int matricola;
  public void setMatricola(int matricola) {
    this.matricola = matricola;
  }
  public int getMatricola () {
    return this.matricola;
```

```
  }
  public String getDati() {
    return getNome() + "\nnumero" + getMatricola();
  }
```

10. Alla classe `Impiegato` descritta nel punto 9) non è possibile aggiungere il seguente metodo:

```
public void setDati(String nome, int matricola) {
  setNome(nome);
  setMatricola(matricola);
}
```

perché produrrebbe un errore in compilazione.

Esercizio 5.d)

Incapsulare e completare le seguenti classi:

```
public class Pilota {
  public String nome;

  public Pilota(String nome) {
    // settare il nome
  }
}
public class Auto {
  public String scuderia;
  public Pilota pilota;

  public Auto (String scuderia, Pilota pilota) {
    // settare scuderia e pilota
  }
  public String dammiDettagli() {
    // restituire una stringa descrittiva dell'oggetto
  }

}
```

Tenere presente che le classi `Auto` e `Pilota` devono poi essere utilizzate dalle seguenti classi:

```
public class TestGara {
  public static void main(String args[]) {
```

```
        Gara imola = new Gara("GP di Imola");
        imola.corriGara();
        String risultato = imola.getRisultato();
        System.out.println(risultato);
      }
    }

    public class Gara {
      private String nome;
      private String risultato;
      private Auto griglia [];

      public Gara(String nome){
        setNome(nome);
        setRisultato("Corsa non terminata");
        creaGrigliaDiPartenza();
      }

      public void creaGrigliaDiPartenza(){
        Pilota uno = new Pilota("Pippo");
        Pilota due = new Pilota("Pluto");
        Pilota tre = new Pilota("Topolino");
        Pilota quattro = new Pilota("Paperino");
        Auto autoNumeroUno = new Auto("Ferrari", uno);
        Auto autoNumeroDue = new Auto("Renault", due);
        Auto autoNumeroTre = new Auto("BMW", tre);
        Auto autoNumeroQuattro = new Auto("Mercedes", quattro);
        griglia = new Auto[4];
        griglia[0] = autoNumeroUno;
        griglia[1] =  autoNumeroDue;
        griglia[2] = autoNumeroTre;
        griglia[3] =  autoNumeroQuattro;
      }

      private void corriGara() {
        int numeroVincente = (int)(Math.random()*4);
        Auto vincitore = griglia[numeroVincente];
        String risultato = vincitore.dammiDettagli();
        setRisultato(risultato);
      }

      public void setRisultato(String vincitore) {
        this.risultato = "Il vincitore di " + this.getNome()
        + ": " + vincitore;
      }
```

```
    public String getRisultato() {
      return risultato;
    }

    public void setNome(String nome) {
      this.nome = nome;
    }

    public String getNome() {
       return nome;
    }
  }
```

Analisi dell'esercizio

La classe `TestGara` contiene il metodo `main()` e quindi determina il flusso di esecuzione dell'applicazione. È molto leggibile: si istanzia un oggetto gara e la si chiama "`GP di Imola`", si fa correre la corsa, si richiede il risultato e lo si stampa a video.

La classe `Gara` invece contiene pochi e semplici metodi e tre variabili d'istanza: `nome` (il nome della gara), `risultato` (una stringa che contiene il nome del vincitore della gara se è stata corsa) e `griglia` (un array di oggetti `Auto` che partecipano alla gara).

Il costruttore prende in input una stringa con il nome della gara che viene opportunamente settato. Inoltre il valore della stringa `risultato` è impostata a "`Corsa non terminata`". Infine è chiamato il metodo `creaGrigliaDiPartenza()`.

Il metodo `creaGrigliaDiPartenza()` istanzia quattro oggetti `Pilota` assegnando loro dei nomi. Poi, istanzia quattro oggetti `Auto` assegnando loro i nomi delle scuderie ed i relativi piloti. Infine istanzia ed inizializza l'array `griglia` con le auto appena create.

Una gara dopo essere stata istanziata è pronta per essere corsa.

Il metodo `corriGara()` contiene codice che va analizzato con più attenzione. Nella prima riga, infatti, viene chiamato il metodo `random()` della classe `Math` (appartenente al package `java.lang` che viene importato automaticamente). La classe `Math` astrae il concetto di "matematica" e sarà descritta nel modulo 12. Essa contiene metodi che astraggono classiche funzioni matemati-

che, come la radice quadrata o il logaritmo. Tra questi metodi utilizziamo il metodo `random()` che restituisce un numero generato in maniera casuale di tipo `double`, compreso tra `0` ed `0,9999999`... (ovvero il numero `double` immediatamente più piccolo di `1`). Nell'esercizio abbiamo moltiplicato per `4` questo numero, ottenendo un numero `double` casuale compreso tra `0` e `3,99999999`... Questo poi viene "castato" ad intero e quindi vengono troncate tutte le cifre decimali. Abbiamo quindi ottenuto che la variabile `numeroVincente` immagazzini al runtime un numero generato casualmente, compreso tra `0` e `3`, ovvero i possibili indici dell'array griglia.
Il metodo `corriGara()` genera quindi un numero casuale tra `0` e `3`. Lo utilizza per individuare l'oggetto `Auto` dell'array `griglia` che vince la gara. Per poi impostare il risultato tramite il metodo `dammiDettagli()` dell'oggetto `Auto` (che scriverà il lettore).
Tutti gli altri metodi della classe sono di tipo accessor e mutator.

Esercizio 5.e)

Commento del codice

Data la seguente classe:

```
public class Persona {
    private String nome;

    public void setNome(String nome) {
      this.nome = nome;
    }

    public String getNome() {
      return nome;
    }

}
```

Commentare la seguente classe `Impiegato`, evidenziando dove sono utilizzati i paradigmi object oriented, incapsulamento, ereditarietà e riuso.

```
public class Impiegato extends Persona {

    private int matricola;
    public void setDati(String nome, int matricola) {
```

```
        setNome(nome);
        setMatricola(matricola);
    }

    public void setMatricola(int matricola) {
        this.matricola = matricola;
    }

    public int getMatricola() {
      return matricola;
    }

    public String dammiDettagli() {
      return getNome() + ", matricola: " + getMatricola();
    }
}
```

Soluzioni esercizi modulo 5

Esercizio 5.a)

Object Orientation in generale (teoria), Vero o Falso:

1. **Falso** esiste dagli anni '60.
2. **Vero**.
3. **Falso** per esempio nel C++ esiste l'ereditarietà multipla e in Java no.
4. **Vero**.
5. **Vero**.
6. **Vero**.
7. **Vero**.
8. **Vero**.
9. **Vero.**
10. **Falso** bisogna conoscere l'interfaccia pubblica e non l'implementazione interna.

Esercizio 5.b)

Object Orientation in Java (teoria), Vero o Falso:

1. **Falso** il processo di generalizzazione implica scrivere una classe in più e ciò non sempre implica scrivere qualche riga in meno.
2. **Vero**.
3. **Falso** anche se dal punto di vista della programmazione la generalizzazione può non farci sempre risparmiare codice, essa ha comunque il pregio di farci gestire le classi in maniera più naturale, favorendo l'astrazione dei dati. Inoltre apre la strada all'implementazione del polimorfismo.
4. **Vero.**
5. **Falso** l'ereditarietà multipla esiste nella realtà, ma non esiste in Java perché tecnicamente implica dei problemi di difficile risoluzione come il caso dell'ereditarietà "a rombo" in C++.
6. **Vero**.
7. **Vero**.
8. **Falso** una squadra non "è un" giocatore, né un giocatore "è una" squadra. Semmai una squadra "ha un" giocatore ma questa non è la relazione di ereditarietà. Si tratta infatti della relazione di associazione…
9. **Vero** infatti entrambe le classi potrebbero estendere una classe `Partecipante.`
10. **Falso** un `Padre` è sempre un `Figlio`, o entrambe potrebbero estendere la classe `Persona.`

Esercizio 5.c)

Object Orientation in Java (pratica), Vero o Falso:

1. **Vero**.
2. **Falso** non si tratta di parole chiave ma solo di parole utilizzate per convenzione.
3. **Falso** possono essere private ed essere utilizzate tramite i metodi accessor e mutator.
4. **Vero**.
5. **Vero**.
6. **Vero**.
7. **Vero**.
8. **Vero** anche se come vedremo più avanti, il costruttore di default non è vuoto.
9. **Vero.**
10. **Falso**.

Esercizio 5.d)

Incapsulare e completare le seguenti classi:

```
public class Pilota {
  private String nome;

  public Pilota(String nome) {
    setNome(nome);
  }
  public void setNome(String nome) {
    this.nome = nome;
  }
  public String getNome() {
    return nome;
  }
}
public class Auto {
  private String scuderia;
  private Pilota pilota;

  public Auto (String scuderia, Pilota pilota) {
    setScuderia(scuderia);
    setPilota(pilota);
  }
  public void setScuderia(String scuderia) {
    this.scuderia = scuderia;
  }
  public String getScuderia() {
    return scuderia;
  }
  public void setPilota(Pilota pilota) {
    this.pilota = pilota;
  }
  public Pilota getPilota() {
    return pilota;
  }
  public String dammiDettagli() {
    return getPilota().getNome() + " su "+ getScuderia();
  }
}
```

Esercizio 5.e)

Commento del codice:

```
public class Impiegato extends Persona { //Ereditarietà
  private int matricola;

  public void setDati(String nome, int matricola) {
    setNome(nome); //Riuso ed ereditarietà
    setMatricola(matricola); //Riuso
  }

  public void setMatricola(int matricola) {
      this.matricola = matricola; //incapsulamento
  }

  public int getMatricola() {
    return matricola; //incapsulamento
  }

  public String dammiDettagli() {
       //Riuso, incapsulamento ed ereditarietà
    return getNome() + ", matricola: " + getMatricola();
  }
}
```

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Comprendere le ragioni della nascita della programmazione ad oggetti (unità 5.1)	☐	
Saper elencare i paradigmi ed i concetti fondamentali della programmazione ad oggetti (unità 5.2)	☐	
Saper definire ed utilizzare il concetto di astrazione (unità 5.2)	☐	

Comprendere l'utilizzo e l'utilità dell'incapsulamento (unità 5.3, 5.4)	☐
Comprendere l'utilizzo e l'utilità del reference this (unità 5.4)	☐
Comprendere l'utilizzo e l'utilità dell'ereditarietà (generalizzazione e specializzazione) (unità 5.5, 5.6)	☐
Conoscere la filosofia di Java per quanto riguardo la semplicità di apprendimento (unità 5.3, 5.5)	☐

Note:

Esercizi modulo 6

Esercizio 6.a)

Polimorfismo per metodi, Vero o Falso:

1. L'overload di un metodo implica scrivere un altro metodo con lo stesso nome e diverso tipo di ritorno.
2. L'overload di un metodo implica scrivere un altro metodo con nome differente e stessa lista di parametri.
3. La segnatura (o firma) di un metodo è costituita dalla coppia identificatore – lista di parametri.
4. Per sfruttare l'override bisogna che sussista l'ereditarietà.
5. Per sfruttare l'overload bisogna che sussista l'ereditarietà.
   Supponiamo che in una classe B, la quale estende la classe A, ereditiamo il metodo:
   ```
   public int m(int a, String b) { . . . }
   ```
6. Se nella classe `B` scriviamo il metodo:
   ```
   public int m(int c, String b) { . . . }
   ```
   stiamo facendo overload e non override.
7. Se nella classe `B` scriviamo il metodo:
   ```
   public int m(String a, String b) { . . . }
   ```
   stiamo facendo overload e non override.
8. Se nella classe `B` scriviamo il metodo:
   ```
   public void m(int a, String b) { . . . }
   ```
   otterremo un errore in compilazione.
9. Se nella classe `B` scriviamo il metodo:
   ```
   protected int m(int a, String b) { . . . }
   ```
   otterremo un errore in compilazione.
10. Se nella classe `B` scriviamo il metodo:
    ```
    public int m(String a, int c) { . . . }
    ```
    otterremo un override.

Esercizio 6.b)

Polimorfismo per dati, Vero o Falso:

1. Considerando le classi introdotte in questo modulo, il seguente frammento di codice non produrrà errori in compilazione:
   ```
   Veicolo v []= {new Auto(), new Aereo(), new Veicolo()};
   ```

2. Considerando le classi introdotte in questo modulo, il seguente frammento di codice non produrrà errori in compilazione:
```
Object o []= {new Veicolo(), new Aereo(), "ciao"};
```
3. Considerando le classi introdotte in questo modulo, il seguente frammento di codice non produrrà errori in compilazione:
```
Aereo a []= {new Veicolo(), new Aereo(), new Aereo()};
```
4. Considerando le classi introdotte in questo modulo, e se il metodo della classe viaggiatore fosse questo:
```
public void viaggia(Object o) {
  o.accelera();
}
```
potremmo passargli un oggetto di tipo `Veicolo` senza avere errori in compilazione. Per esempio:
```
claudio.viaggia(new Veicolo());
```
5. Considerando le classi introdotte in questo modulo, il seguente frammento di codice non produrrà errori in compilazione:
```
PuntoTridimensionale ogg = new Punto();
```
6. Considerando le classi introdotte in questo modulo, il seguente frammento di codice non produrrà errori in compilazione:
```
PuntoTridimensionale ogg = (PuntoTridimensionale)new Punto();
```
7. Considerando le classi introdotte in questo modulo, il seguente frammento di codice non produrrà errori in compilazione:
```
Punto ogg = new PuntoTridimensionale();
```
8. Considerando le classi introdotte in questo modulo, e se la classe `Piper` estende la classe `Aereo`, il seguente frammento di codice non produrrà errori in compilazione:
```
Veicolo a = new Piper();
```
9. Considerando le classi introdotte in questo modulo, il seguente frammento di codice non produrrà errori in compilazione:
```
String stringa = fiat500.toString();
```
10. Considerando le classi introdotte in questo modulo. Il seguente frammento di codice non produrrà errori in compilazione:
```
public void pagaDipendente(Dipendente dip) {
 if (dip instanceof Dipendente) {
   dip.stipendio = 1000;
 }
 else if (dip instanceof Programmatore) {
   . . .
 }
```

Soluzioni esercizi modulo 6

Esercizio 6.a)

Polimorfismo per metodi, Vero o Falso:

1. **Falso** l'overload di un metodo implica scrivere un altro metodo con lo stesso nome e diversa lista di parametri.
2. **Falso** l'overload di un metodo implica scrivere un altro metodo con lo stesso nome e diversa lista di parametri.
3. **Vero.**
4. **Vero.**
5. **Falso** l'overload di un metodo implica scrivere un altro metodo con lo stesso nome e diversa lista di parametri.
6. **Falso** stiamo facendo override. L'unica differenza sta nel nome dell'identificatore di un parametro, che è ininfluente al fine di distinguere metodi.
7. **Vero** la lista dei parametri dei due metodi è diversa.
8. **Vero** in caso di override il tipo di ritorno non può essere differente.
9. **Vero** in caso di override il metodo riscritto non può essere meno accessibile del metodo originale.
10. **Falso** otterremo un overload. Infatti, le due liste di parametri differiscono per posizioni.

Esercizio 6.b)

Polimorfismo per dati, Vero o Falso:

1. **Vero.**
2. **Vero.**
3. **Falso** non è possibile inserire in una collezione eterogenea di aerei un `Veicolo` che è superclasse di `Aereo`.
4. **Falso** la compilazione fallirebbe già dal momento in cui provassimo a compilare il metodo `viaggia()`. Infatti, non è possibile chiamare il metodo `accelera()` con un reference di tipo `Object`.
5. **Falso** c'è bisogno di un casting, perché il compilatore non sa a priori il tipo a cui punterà il reference al runtime.
6. **Vero.**
7. **Vero.**

8. **Vero** infatti `Veicolo` è superclasse di `Piper.`
9. **Vero** il metodo `toString()` appartiene a tutte le classi perché ereditato dalla superclasse `Object.`
10. **Vero** ma tutti i dipendenti verranno pagati allo stesso modo.

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Comprendere il significato del polimorfismo (unità 6.1)	☐	
Saper utilizzare l'overload, l'override ed il polimorfismo per dati (unità 6.2 e 6.3)	☐	
Comprendere e saper utilizzare le collezioni eterogenee, i parametri polimorfi ed i metodi virtuali (unità 6.3)	☐	
Sapere utilizzare l'operatore instanceof ed il casting di oggetti (unità 6.3)	☐	

Note:

Esercizi modulo 7

Esercizio 7.a)

Riprogettare l'applicazione dell'esempio cercando di rispettare le regole dell'object orientation.

Raccomandiamo al lettore di cercare una soluzione teorica prima di "buttarsi sul codice". Avere un metodo di approcciare il problema può fare risparmiare ore di debug. Questo metodo dovrà permettere quantomeno di:
1) Individuare le astrazioni chiave del progetto (le classi più importanti).
2) Assegnare loro responsabilità avendo cura dell'astrazione.
3) Individuare le relazioni tra esse.
Utilizzare UML potrebbe essere considerato (anche da chi non l'ha mai utilizzato) un modo per non iniziare a smanettare da subito con il codice…

Non viene presentata soluzione per quest'esercizio.

Esercizio 7.b)

Realizzare un'applicazione che simuli il funzionamento di una rubrica.

Il lettore si limiti a simulare la seguente situazione:una rubrica contiene informazioni (nome, indirizzo, numero telefonico) su un certo numero di persone (per esempio 5), prestabilito (le informazioni sono preintrodotte nel metodo `main()`). L'utente dovrà fornire all'applicazione un nome da riga di comando e l'applicazione dovrà restituire le informazioni relative alla persona. Se il nome non è fornito, o se il nome immesso non corrisponde al nome di una persona preintrodotta dall'applicazione, deve essere restituito un messaggio significativo.
Il lettore non ha altri vincoli.
Non è presentata soluzione per quest'esercizio.

Se il lettore ottiene un risultato implementativo funzionante e coerente con tutto ciò che ha studiato sino ad ora, può considerarsi veramente soddisfatto. Infatti l'esercizio proposto, è presentato spesso a corsi di formazione da noi erogati, nella giornata iniziale, per testare il livello della classe. Molto spesso, anche corsisti che si dichiarano "programmatori Java", con esperienza e/o conoscenze, non riescono ad ottenere un risultato accettabile. Ricordiamo una volta di più che questo testo vuole rendere il lettore capace di programmare in Java in modo corretto e senza limiti. Non bisogna avere fretta! Con un po' di pazienza iniziale in più, si otterranno risultati sorprendenti. Una volta padroni del linguaggio, non esisteranno più ambiguità e misteri, e l'acquisizione di argomenti che oggi sembrano avanzati (Applet, Servlet etc…), risulterà semplice!

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Sviluppare un'applicazione in Java, utilizzando i paradigmi della programmazione ad oggetti (unità 7.1, 7.2, 7.3)	☐	

Note:

Esercizi modulo 8

Esercizio 8.a)

Caratteristiche avanzate del linguaggio, Vero o Falso:

1. Qualsiasi costruttore scritto da uno sviluppatore invocherà un costruttore della superclasse o un altro della stessa classe.
2. Qualsiasi costruttore di default invocherà un costruttore della superclasse o un altro della stessa classe.
3. Il reference `super` permette ad una sottoclasse di riferirsi ai membri della superclasse.
4. L'override di un costruttore non è possibile, perché i costruttori non sono ereditati. L'overload di un costruttore è invece sempre possibile.
5. Il comando `this([parametri])` permette ad un metodo di invocare un costruttore della stessa classe in cui è definito.
6. I comandi `this([parametri])` e `super([parametri])` sono mutuamente esclusivi e uno di loro deve essere per forza la prima istruzione di un costruttore.
7. Non è possibile estendere una classe con un unico costruttore dichiarato privato.
8. Una classe innestata è una classe che viene dichiarata all'interno di un'altra classe.
9. Una classe anonima è anche innestata, ma non ha nome. Inoltre, per essere dichiarata, deve per forza essere istanziata
10. Le classi innestate non sono molto importanti per programmare in Java e non sono necessarie per l'object orientation.

Soluzioni esercizi modulo 8

Esercizio 8.a)

Caratteristiche avanzate del linguaggio, Vero o Falso:

1. **Vero.**
2. **Falso** qualsiasi costruttore di default invocherà il costruttore della superclasse tramite il comando `super()` inserito dal compilatore automaticamente.

3. **Vero.**
4. **Vero.**
5. **Falso** il comando `this()` permette solo ad un costruttore di invocare un altro costruttore della stessa classe in cui è definito.
6. **Vero.**
7. **Vero.**
8. **Vero.**
9. **Vero.**
10. **Vero.**

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Saper definire ed utilizzare i costruttori sfruttando l'overload (unità 8.1)	☐	
Conoscere e saper sfruttare il rapporto tra i costruttori e il polimorfismo (unità 8.1)	☐	
Conoscere e saper sfruttare il rapporto tra i costruttori ed ereditarietà (unità 8.2)	☐	
Saper definire ed utilizzare il reference super (unità 8.3)	☐	
Saper chiamare i costruttori con i reference this e super (unità 8.3)	☐	
Conoscere le classi interne e le classi anonime (unità 8.4)	☐	

Note:

Esercizi modulo 9

Esercizio 9.a)

Modificatori e package, Vero o Falso:

1. Una classe dichiarata `private` non può essere utilizzata fuori dal package in cui è dichiarata.
2. La seguente dichiarazione di classe è scorretta:
```
public static class Classe {...}
```
3. La seguente dichiarazione di classe è scorretta:
```
public final class Classe extends AltraClasse {...}
```
4. La seguente dichiarazione di metodo è scorretta:
```
public final void metodo ();
```
5. Un metodo statico può utilizzare solo variabili statiche e, perché sia utilizzato, non bisogna per forza istanziare un oggetto dalla classe in cui è definito.
6. Se un metodo è dichiarato `final`, non si può fare overload.
7. Una classe `final` non è accessibile fuori dal package in cui è dichiarata.
8. Un metodo `protected` viene ereditato in ogni sottoclasse qualsiasi sia il suo package.
9. Una variabile `static` viene condivisa da tutte le istanze della classe a cui appartiene.
10. Se non anteponiamo modificatori ad un metodo, il metodo è accessibile solo all'interno dello stesso package.

Esercizio 9.b)

Classi astratte ed interfacce, Vero o Falso:

1. La seguente dichiarazione di classe è scorretta:
```
public abstract final class Classe {...}
```
2. La seguente dichiarazione di classe è scorretta:
```
public abstract class Classe;
```
3. La seguente dichiarazione di interfaccia è scorretta:
```
public final interface Classe {...}
```
4. Una classe astratta contiene per forza metodi astratti.
5. Un'interfaccia può essere estesa da un'altra interfaccia
6. Una classe può estendere una sola classe ma implementare più interfacce.

7. 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.
8. Il polimorfismo può essere favorito dalla definizione di interfacce.
9. Un'interfaccia può dichiarare solo costanti statiche e pubbliche.
10. Una classe astratta può implementare un'interfaccia.

Soluzioni esercizi modulo 9

Esercizio 9.a)

Modificatori e package, Vero o Falso:

1. **Falso** `private` non si può utilizzare con la dichiarazione di una classe.
2. **Vero** `static` non si può utilizzare con la dichiarazione di una classe.
3. **Falso.**
4. **Vero** manca il blocco di codice (non è un metodo `abstract`).
5. **Vero.**
6. **Falso** se un metodo è dichiarato `final`, non si può fare override.
7. **Falso** una classe `final`, non si può estendere.
8. **Vero.**
9. **Vero.**
10. **Vero.**

Esercizio 9.b)

Classi astratte ed interfacce, Vero o Falso:

1. **Vero** i modificatori `abstract` e `final` sono in contraddizione.
2. **Vero** manca il blocco di codice che definisce la classe.
3. **Vero** un'interfaccia `final` non ha senso.
4. **Falso.**
5. **Vero.**
6. **Vero.**
7. **Vero.**
8. **Vero.**
9. **Vero.**
10. **Vero.**

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Saper utilizzare tutti i modificatori d'accesso (unità 9.1, 9.2)	☐	
Saper dichiarare ed importare package (unità 9.3)	☐	
Saper utilizzare il modificatore final (unità 9.4)	☐	
Saper utilizzare il modificatore static (unità 9.5)	☐	
Saper utilizzare il modificatore abstract (unità 9.6)	☐	
Conoscere cosa sono le classi interne e le classi anonime (unità 8.4)	☐	
Comprendere l'utilità di classi astratte ed interfacce (unità 9.6, 9.7)	☐	
Comprendere e saper utilizzare l'ereditarietà multipla (unità 9.7)	☐	
Comprendere e saper utilizzare le Enumerazioni (unità 9.8)	☐	
Saper accennare alle definizione dei modificatori strictfp, volatile e native (unità 9.9)	☐	

Note:

Esercizi modulo 10

Esercizio 10.a)

Gestione delle eccezioni e degli errori, Vero o Falso:

1. Ogni eccezione che estende in qualche modo una `Arithmetic Exception` è una checked exception.
2. Un `Error` si differenzia da una `Exception` perché non può essere lanciato; infatti non estende la classe `Throwable.`
3. Il seguente frammento di codice:

```
int a = 10;
int b = 0;
try {
    int c = a/b;
    System.out.println(c);
}
catch (ArithmeticException exc) {
    System.out.println("Divisione per zero...");
}
catch (NullPointerException exc) {
    System.out.println("Reference nullo...");
}
catch (Exception exc) {
    System.out.println("Eccezione generica...");
}
finally {
    System.out.println("Finally!");
}
```

   produrrà il seguente output:

```
Divisione per zero…
Eccezione generica…
Finally!
```

4. Il seguente frammento di codice:

```
int a = 10;
int b = 0;
try {
    int c = a/b;
    System.out.println(c);
}
catch (Exception exc) {
    System.out.println("Eccezione generica...");
}
```

```
catch (ArithmeticException exc) {

    System.out.println("Divisione per zero...");
}
catch (NullPointerException exc) {
    System.out.println("Reference nullo...");
}
finally {
    System.out.println("Finally!");
}
```

produrrà un errore al runtime.

5. La parola chiave `throw` permette di lanciare "a mano" solo le sottoclassi di `Exception` che crea il programmatore.
6. La parola chiave `throw` permette di lanciare "a mano" solo le sottoclassi di `Exception`.
7. Se un metodo fa uso della parola chiave `throw`, affinché la compilazione abbia buon esito, allora nello stesso metodo deve essere gestita l'eccezione che si vuole lanciare, o il metodo stesso deve utilizzare una clausola `throws`.
8. Non è possibile estendere la classe `Error`.
9. Se un metodo `m2` fa ovverride di un altro metodo `m2` posto nella superclasse, non potrà dichiarare con la clausola `throws` eccezioni nuove che non siano sottoclassi rispetto a quelle che dichiara il metodo `m2`.
10. Dalla versione 1.4 di Java è possibile "includere" in un'eccezione un'altra eccezione.

Esercizio 10.b)

Gestione delle asserzioni, Vero o Falso:

1. Se in un'applicazione un'asserzione non viene verificata, si deve parlare di bug.
2. Un'asserzione che non viene verificata provoca il lancio da parte della JVM di un `AssertionError`.
3. Le precondizioni servono per testare la correttezza dei parametri di metodi pubblici.
4. È sconsigliato l'utilizzo di asserzioni laddove si vuole testare la correttezza di dati inseriti da un utente.

5. Una postcondizione serve per verificare che al termine di un metodo sia verificata un'asserzione
6. Un'invariante interna permette di testare la correttezza dei flussi all'interno dei metodi
7. Un'invariante di classe è una particolare invariante interna che deve essere verificata per tutte le istanze di una certa classe, in ogni momento del loro ciclo di vita, tranne che durante l'esecuzione di alcuni metodi.
8. Un'invariante sul flusso di esecuzione, è solitamente un'asserzione con una sintassi del tipo:
   ```
   assert false;
   ```
9. Non è in alcun modo possibile compilare un programma che fa uso di asserzioni con il jdk 1.3.
10. Non è in alcun modo possibile eseguire un programma che fa uso di asserzioni con il jdk 1.3.

Soluzioni esercizi modulo 10

Esercizio 10.a)

Gestione delle eccezioni e degli errori, Vero o Falso:

1. **Vero** perché `ArithmeticException` è sottoclasse di `RuntimeException`.
2. **Falso.**
3. **Falso** produrrà il seguente output:
   ```
   Divisione per zero...
   Finally!
   ```
4. **Falso** produrrà un errore in compilazione.
5. **Falso.**
6. **Falso** solo le sottoclassi di `Throwable`.
7. **Vero.**
8. **Falso.**
9. **Vero.**
10. **Vero.**

Esercizio 10.b)

Gestione delle asserzioni, Vero o Falso:

1. **Vero.**
2. **Vero.**
3. **Falso.**
4. **Vero.**
5. **Vero.**
6. **Vero.**
7. **Vero.**
8. **Vero.**
9. **Vero.**
10. **Vero.**

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Comprendere le varie tipologie di eccezioni, errori ed asserzioni (unità 10.1)	☐	
Saper gestire le varie tipologie di eccezioni con i blocchi try–catch (unità 10.2)	☐	
Saper creare tipi di eccezioni personalizzate e gestire il meccanismo di propagazione con le parole chiave throw e throws (unità 10.3)	☐	
Capire e saper utilizzare il meccanismo delle asserzioni (unità 10.4)	☐	

Note:

Esercizi modulo 11

Esercizio 11.a)

Creazione di Thread, Vero o Falso:

1. Un thread è un oggetto istanziato dalla classe `Thread` o dalla classe `Runnable`.
2. Il multithreading è solitamente una caratteristica dei sistemi operativi e non dei linguaggi di programmazione.
3. In ogni applicazione al runtime esiste almeno un thread in esecuzione.
4. 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`.
5. Il metodo `run()` deve essere chiamato dal programmatore per attivare un thread.
6. Il "thread corrente" non si identifica solitamente con il reference `this`.
7. Chiamando il metodo `start()` su di un thread, questo viene immediatamente eseguito.
8. Il metodo `sleep()` è statico e permette di far dormire per un numero specificato di millisecondi il thread che legge tale istruzione.
9. Assegnare le priorità ai thread è una attività che può produrre risultati diversi su piattaforme diverse.
10. Lo scheduler della JVM non dipende dalla piattaforma su cui viene lanciato.

Esercizio 11.b)

Gestione del multi-threading, Vero o Falso:

1. Un thread astrae un processore virtuale che esegue codice su determinati dati.
2. La parola chiave `synchronized` può essere utilizzata sia come modificatore di un metodo sia come modificatore di una variabile.
3. Il monitor di un oggetto può essere identificato con la parte sincronizzata dell'oggetto stesso.
4. Affinché due thread che eseguono lo stesso codice e condividono gli stessi dati, non abbiano problemi di concorrenza, basta sincronizzare il codice comune.

5. Si dice che un thread ha il lock di un oggetto se entra nel suo monitor.
6. I metodi `wait()`, `notify()` e `notifyAll()` rappresentano il principale strumento per far comunicare più thread.
7. I metodi `sospend()` e `resume()` sono attualmente deprecati.
8. 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. Tra questi verrà eseguito quello che era stato fatto partire per primo con il metodo `start().`
9. Il deadlock è una condizione di errore bloccante generata da due thread che stanno in reciproca dipendenza in due oggetti sincronizzati.
10. 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.

Soluzioni esercizi modulo 11

Esercizio 11.a)

Creazione di Thread, Vero o Falso:

1. **Falso** `Runnable` è un'interfaccia.
2. **Vero.**
3. **Vero** il cosiddetto thread "main".
4. **Vero.**
5. **Falso** il programmatore può invocare il metodo `start()` e lo scheduler invocherà il metodo `run().`
6. **Vero.**
7. **Falso.**
8. **Vero.**
9. **Vero.**
10. **Falso.**

Esercizio 11.b)

Gestione del multi-threading, Vero o Falso:

1. **Vero.**
2. **Falso.**
3. **Vero.**
4. **Falso.**
5. **Vero.**
6. **Vero.**
7. **Vero.**
8. **Falso** il primo thread che partirà sarà quello a priorità più alta.
9. **Vero.**
10. **Vero**

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Saper definire multithreading e multitasking (unità 11.1)	☐	
Comprendere la dimensione temporale introdotta dalla definizione dei thread in quanto oggetti (unità 11.2)	☐	
Saper creare ed utilizzare thread tramite la classe Thread e l'interfaccia Runnable (unità 11.2)	☐	
Definire che cos'è uno scheduler e i suoi comportamenti riguardo le priorità dei thread (unità 11.3)	☐	
Sincronizzare thread (unità 11.4)	☐	
Far comunicare i thread (unità 11.5)	☐	

Note:

Esercizi modulo 12

Esercizio 12.a)

Framework Collections, Vero o Falso:

1. `Collection` , `Map`, `SortedMap`, `Set`, `List` e `SortedSet` sono interfacce e non possono essere istanziate.
2. Un `Set` è una collezione ordinata di oggetti; una `List` non ammette elementi duplicati ed è ordinata.
2. Le mappe non possono contenere chiavi duplicate ed ogni chiave può essere associata ad un solo valore.
4. Esistono diverse implementazioni astratte da personalizzare nel framework come `AbstractMap.`
5. Una `HashMap` è più performante rispetto ad una `Hashtable` perché non è sincronizzata.
6. Una `HashMap` è più performante rispetto ad un `TreeMap` ma quest'ultima, essendo un'implementazione di `SortedMap`, gestisce l'ordinamento.
7. `HashSet` è più performante rispetto a `TreeSet`, ma non gestisce l'ordinamento.
8. `Iterator` ed `Enumeration` hanno lo stesso ruolo ma quest'ultima permette durante le iterazioni di rimuovere anche elementi.
9. `ArrayList` ha prestazioni migliori rispetto a `Vector` perché non è sincronizzato, ma entrambi hanno meccanismi per ottimizzare le prestazioni.
10. La classe `Collections` è un lista di `Collection.`

Esercizio 12.b)

Package java.util e java.lang, Vero o Falso:

1. La classe `Properties` estende `Hashtable` ma permette di salvare su un file le coppie chiave-valore rendendole persistenti.
2. La classe `Locale` astrae il concetto di "zona".
3. La classe `ResourceBundle` rappresenta un file di properties che permette di gestire l'internazionalizzazione. Il rapporto tra nome del file e `Locale` specificato per individuare tale file permetterà di gestire la configurazione della lingua delle nostre applicazioni.
4. L'output del seguente codice:

```
StringTokenizer st = new StringTokenizer(
         "Il linguaggio object oriented Java", "t", false);
```

```
while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

sarà:

```
Il linguaggio objec
t
orien
t
ed Java
```

5. Il seguente codice non è valido:
```
Pattern p = Pattern.compile("\bb");
Matcher m = p.matcher("blablabla…");
boolean b = m.find();
System.out.println(b);
```
6. La classe `Runtime` dipende strettamente dal sistema operativo su cui gira.
7. La classe `Class` permette di leggere i membri di una classe (ma anche le superclassi ed altre informazioni) partendo semplicemente dal nome della classe grazie al metodo `forName().`
8. Tramite la classe `Class` è possibile istanziare oggetti di una certa classe conoscendone solo il nome.
9. È possibile dalla versione 1.4 di Java sommare un tipo primitivo e un oggetto della relativa classe wrapper, come nel seguente esempio:
```
Integer a = new Integer(30);
int b = 1;
int c = a+b;
```
10. La classe `Math` non si può istanziare perché dichiarata `abstract.`

Soluzioni esercizi modulo 12

Esercizio 12.a)

Framework Collections, Vero o Falso:

1. **Vero.**
2. **Falso.**
3. **Vero.**
4. **Vero.**

5. **Vero.**
6. **Vero.**
7. **Vero.**
8. **Falso.**
9. **Vero.**
10. **Falso.**

Esercizio 12.b)

Package java.util e java.lang, Vero o Falso:

1. **Vero.**
2. **Vero.**
3. **Vero.**
4. **Falso** tutte le "t" non dovrebbero esserci.
5. **Falso** è valido ma stamperà **`false`**. Affinché stampi **`true`** l'espressione si deve modificare in "**`\\bb`**".
6. **Vero.**
7. **Vero.**
8. **Vero.**
9. **Falso** dalla versione 1.5.
10. **Falso** non si può istanziare perché ha un costruttore privato ed è dichiarata `final` per non poter essere estesa.

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Comprendere l'utilità e saper utilizzare il framework Collection (unità 12.1)	☐	
Saper implementare programmi con l'internazionalizzazione (unità 12.1)	☐	
Saper implementare programmi configurabili mediante file di properties (unità 12.1)	☐	

Saper utilizzare la classe StringTokenizer per “splittare” stringhe (unità 12.1)	☐
Saper utilizzare la Reflection per l’introspezione delle classi (unità 12.2)	☐
Saper introdurre le classi System, Math e Runtime (unità 12.1)	☐

Note:

Esercizi modulo 13

Esercizio 13.a)

Input - Output, Vero o Falso:

1. Il pattern Decorator permette di implementare una sorta di ereditarietà dinamica. Questo significa che, invece di creare tante classi quanti sono i concetti da astrarre, al runtime sarà possibile concretizzare uno di questi concetti direttamente con un oggetto.
2. Reader e writer permettono di leggere e scrivere caratteri. Per tale ragione sono detti Character Stream.
3. All'interno del package `java.io` l'interfaccia `Reader` ha il ruolo di `ConcreteComponent.`
4. All'interno del package `java.io` l'interfaccia `InputStream` ha il ruolo di `ConcreteDecorator.`
5. Un `BufferedWriter` è un `ConcreteDecorator.`
6. Gli stream che possono realizzare una comunicazione direttamente con una fonte o una destinazione vengono detti "node stream".
7. I node stream di tipo `OutputStream` possono utilizzare il metodo
   `int write(byte cbuf[])`
   per scrivere su una destinazione.
8. Il seguente oggetto `in`:
   ```
   BufferedReader in = new BufferedReader(
     new InputStreamReader(System.in));
   ```
   permette di usufruire di un metodo `readLine()` che leggerà frasi scritte con la tastiera delimitate dalla battitura del tasto Invio.
9. Il seguente codice**:**
   `File outputFile = new File("pippo.txt");`
   crea un file di chiamato "pippo.txt" nella cartella corrente.
10. Non è possibile decorare un `FileReader.`

Esercizio 13.b)

Serializzazione e networking, Vero o Falso:

1. Lo stato di un oggetto è definito dal valore delle sue variabili d'istanza (ovviamente in un certo momento).

2. L'interfaccia `Serializable` non ha metodi.
3. `transient` è un modificatore applicabile a variabili e classi. Una variabile `transient` non viene serializzata con le altre variabili; una classe `transient` non è serializzabile.
4. `transient` è un modificatore applicabile a metodi e variabili. Una variabile `transient` non viene serializzata con le altre variabili; un metodo `transient` non è serializzabile.
5. Se si prova a serializzare un oggetto che ha tra le sue variabili d'istanza una variabile di tipo `Reader` dichiarata `transient`, otterremo un `NotSerializableException` al runtime.
6. In una comunicazione di rete devono esistere almeno due socket.
7. Un client, per connettersi ad un server, deve conoscere almeno il suo indirizzo IP e la porta su cui si è posto in ascolto.
8. Un server si può mettere in ascolto anche sulla porta 80, la porta di default dell'HTTP, senza per forza utilizzare quel protocollo. È infatti possibile anche che si comunichi con il protocollo HTTP su una porta diversa dalla 80.
9. Il metodo `accept()` blocca il server in uno stato di "attesa di connessioni". Quando un client si connette, il metodo `accept()` viene eseguito per raccogliere tutte le informazioni del client in un oggetto di tipo `Socket`.
10. Un `ServerSocket` non ha bisogno di dichiarare l'indirizzo IP, ma deve solo dichiarare la porta su cui si metterà in ascolto.

Soluzioni esercizi modulo 13

Esercizio 13.a)

Input - Output, Vero o Falso:

1. **Vero.**
2. **Vero.**
3. **Falso.**
4. **Falso.**
5. **Vero.**
6. **Vero.**

7. **Vero.**
8. **Vero.**
9. **Falso.**
10. **Falso.**

Esercizio 13.b)

Serializzazione e networking, Vero o Falso:

1. **Vero.**
2. **Vero.**
3. **Falso.**
4. **Falso.**
5. **Falso.**
6. **Vero.**
7. **Vero.**
8. **Vero.**
9. **Vero.**
10. **Vero.**

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Aver compreso il pattern Decorator (unità 13.1, 13.2)	☐	
Saper riconoscere nelle classi del package java.io i ruoli definiti nel pattern Decorator (unità 13.3)	☐	
Capire le fondamentali gerarchie del package java.io (unità 13.3)	☐	
Avere confidenza con i tipici problemi che si incontrano con l'input-output, come la serializzazione degli oggetti e la gestione dei file (unità 13.4)	☐	

Avere un'idea di base del networking in Java, dei concetti di socket e del metodo accept (unità 13.5)	☐

Note:

Esercizi modulo 14

Esercizio 14.a)

JDBC, Vero o Falso:

1. L'implementazione del driver JDBC da parte del vendor, è costituita solitamente dalla implementazione delle interfacce del package `java.sql`.
2. `Connection` è solo un'interfaccia.
3. Un'applicazione JDBC è indipendente dal database solo se si parametrizzano le stringhe relative al driver, l'URL di connessione, la username e la password.
4. Se si inoltra ad un particolare database un comando non standard SQL 2, questo comando funzionerà solo su quel database. In questo modo si perde l'indipendenza dal database, a meno di controlli o parametrizzazioni.
5. Se si inoltra ad un particolare database un comando non standard SQL 2, l'implementazione JDBC lancerà un'eccezione.
6. Per cancellare un record bisogna utilizzare il metodo `executeQuery()`.
7. Per aggiornare un record bisogna utilizzare il metodo `executeUpdate()`.
8. `CallableStatement` è una sottointerfaccia di `PreparedStatement`. `PreparedStatement` è una sottointerfaccia di `Statement`.
9. Per eseguire una stored procedure bisogna utilizzare il metodo `execute()`.
10. L'autocommit è impostato a `true` per default.

Esercizio 14.b)

JAXP, Vero o Falso:

1. Per le specifiche DOM, ogni nodo è equivalente ad un altro e un commento viene visto come un oggetto di tipo `Node`.
2. Infatti l'interfaccia `Node` implementa `Text`.
3. Per poter analizzare un documento nella sua interezza con DOM, bisogna utilizzare un metodo ricorsivo.
4. Con il seguente codice:
   ```
   Node n = node.getParentNode().getFirstChild();
   ```
   si raggiunge il primo nodo "figlio" di `node`.

5. Con il seguente codice:
   ```
   Node n = node.getParentNode().getPreviousSibling();
   ```
   si raggiunge il nodo "fratello" precedente di `node`.
6. Con il seguente codice:
   ```
   NodeList list = node.getChildNodes();
     Node n = list.item(0);
   ```
   si raggiunge il primo nodo "figlio" di `node`.
7. Con il seguente codice:
   ```
   Element element = doc.createElement("nuovo");
     doc.appendChild(element);
     Text text = doc.createTextNode("prova testo");
     doc.insertBefore(text, element);
   ```
   viene creato un nodo chiamato `nuovo`, in cui viene aggiunto testo.
8. La rimozione di un nodo provoca la rimozione di tutti i suoi nodi "figli".
9. Per analizzare un documento tramite l'interfaccia SAX bisogna estendere la classe `DefaultHandler` e effettuare override dei suoi metodi.
10. Per trasformare un file XML e serializzarlo in un altro file dopo una trasformazione mediante un file XSL, è possibile utilizzare il seguente codice:

```
try {
    TransformerFactory factory = TransformerFactory.newInstance();
    Source source = new StreamSource(new File("input.xml"));
    Result result = new StreamResult(new File("output.xml"));
    Templates template = factory.newTemplates(
      new StreamSource(new FileInputStream("transformer.xsl")));
    Transformer transformer = template.newTransformer();
    transformer.transform(source, result);
} catch (Exception e) {
    e.printStackTrace();
}.
```

Soluzioni esercizi modulo 14

Esercizio 14.a)

JDBC, Vero o Falso:

1. **Vero.**
2. **Vero.**
3. **Vero.**
4. **Vero.**
5. **Falso.**
6. **Falso.**
7. **Vero.**
8. **Vero.**
9. **Vero.**
10. **Vero.**

Esercizio 14.b)

JAXP, Vero o Falso:

1. **Vero.**
2. **Falso** l'interfaccia `Text` implementa `Node`.
3. **Vero.**
4. **Falso** si raggiunge il primo nodo "fratello" di `node`.
5. **Falso**.
6. **Vero.**
7. **Falso** il testo viene aggiunto prima del tag con il metodo `insertBefore()`. Sarebbe invece opportuno utilizzare la seguente istruzione per aggiungere il testo all'interno del tag `nuovo`: `element.appendChild(text);`.
8. **Falso**.
9. **Vero.**
10. **Vero.**

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Saper scrivere codice che si connette a qualsiasi tipo di database (unità 14.2, 14.3)	☐	
Saper scrivere codice che aggiorna, interroga e gestisce i risultati qualsiasi sia il database in uso (unità 14.2, 14,3)	☐	
Avere confidenza con le tipiche caratteristiche avanzate di JDBC, come stored procedure, statement parametrizzati e transazioni (unità 13.4)	☐	
Saper gestire i concetti della libreria JAXP per la gestione dei documenti XML (unità 14.4)	☐	
Saper risolvere i problemi di utilizzo delle interfacce DOM e SAX per l'analisi dei documenti XML (unità 14.4)	☐	
Saper trasformare con XSLT i documenti XML (unità 14.4)	☐	

Note:

Esercizi modulo 15

Esercizio 15.a)

GUI, AWT e Layout Manager, Vero o Falso:

1. Nella progettazione di una GUI è preferibile scegliere soluzioni standard, per facilitare l'utilizzo all'utente.
2. Nell'MVC il Model rappresenta i dati, il controller le operazioni e la view l'interfaccia grafica.
3. Le GUI AWT, sono invisibili di default.
4. Per ridefinire l'aspetto grafico di un componente AWT è possibile estenderlo e ridefinire il metodo `paint()`.
5. AWT è basata sul pattern Decorator.
6. In un'applicazione basata su AWT è necessario sempre avere un top level container.
7. È impossibile creare GUI senza layout manager; otterremmo solo eccezioni al runtime.
8. Il `FlowLayout` cambierà la posizione dei suoi componenti in base al ridimensionamento.
9. Il `BorderLayout` cambierà la posizione dei suoi componenti in base al ridimensionamento.
10. Il `GridLayout` cambierà la posizione dei suoi componenti in base al ridimensionamento.

Esercizio 15.b)

Gestione degli eventi, Applet e Swing, Vero o Falso:

1. Il modello a delega è basato sul pattern Observer.
2. Senza la registrazione tra la sorgente dell'evento e il gestore dell'evento, l'evento non sarà gestito.
3. Le classi innestate e le classi anonime non sono adatte per implementare gestori di eventi.
4. Una classe innestata può gestire eventi se e solo se è statica.
5. Una classe anonima per essere definita si deve per forza istanziare.
6. Un `ActionListener` può gestire eventi di tipo `MouseListener`.
7. Un pulsante può chiudere una finestra.

8. È possibile (ma non consigliabile) per un gestore di eventi estendere tanti adapter per evitare di scrivere troppo codice.
9. La classe `Applet`, estendendo `Panel`, potrebbe anche essere aggiunta direttamente ad un `Frame`. In tal caso però, i metodi sottoposti a override non verranno chiamati automaticamente.
10. I componenti di Swing (`JComponent`) estendono la classe `Container` di AWT.

Soluzioni esercizi modulo 15

Esercizio 15.a)

GUI, AWT e Layout Manager, Vero o Falso:

1. **Vero.**
2. **Falso** in particolare il Model rappresenta l'intera applicazione composta da dati e funzionalità.
3. **Vero.**
4. **Vero.**
5. **Falso** è basata sul pattern Composite, che, nonostante abbia alcuni punti di contatto con il Decorator, si può tranquillamente definire completamente diverso.
6. **Vero.**
7. **Falso** ma perderemmo la robustezza a la consistenza della GUI.
8. **Vero.**
9. **Falso.**
10. **Falso.**

Esercizio 15.b)

Gestione degli eventi, Applet e Swing, Vero o Falso:

1. **Vero.**
2. **Vero.**
3. **Falso.**
4. **Falso.**

5. **Vero.**
6. **Falso.** solo di tipo `ActionListener`.
7. **Vero** può sfruttare il metodo `System.exit(0)`, ma non c'entra niente con gli eventi di tipo `WindowEvent`.
8. **Vero.**
9. **Vero.**
10. **Vero.**

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Saper elencare le principali caratteristiche che deve avere una GUI (unità 15.1)	☐	
Saper descrivere le caratteristiche della libreria AWT (unità 15.2)	☐	
Saper gestire i principali Layout Manager per costruire GUI complesse (unità 15.3)	☐	
Saper gestire gli eventi con il modello a delega (unità 15.4)	☐	
Saper creare semplici applet (unità 15.5)	☐	
Saper descrivere le caratteristiche della libreria Swing (unità 15.6)	☐	

Note:

Esercizi modulo 16

Esercizio 16.a)

Autoboxing, Autounboxing e Generics, Vero o Falso:

1. Il seguente codice compila senza errori:
```
char c = new String("Pippo");
```
2. Il seguente codice compila senza errori:
```
int c = new Integer(1) + 1 + new Character('a');
```
3. Il seguente codice compila senza errori:
```
Integer i = 0;
switch(i){
  case 0:
    System.out.println();
  break;
}
```
4. Le regole dell'overload non cambiano con l'introduzione dell'autoboxing e dell'autounboxing.
5. Per confrontare correttamente il contenuto di due `Integer` è necessario utilizzare il metodo `equals()`. L'operatore == potrebbe infatti avere un comportamento anomalo su istanze che hanno un range limitato, a causa di ottimizzazioni delle prestazioni di Java.
6. Il seguente codice:
```
List<String> strings = new ArrayList<String>( );
strings.add(new Character('A'));
```
compila senza errori.
7. Il seguente codice:
```
List<int> ints = new ArrayList<int>();
```
compila senza errori.
8. Il seguente codice:
```
List<int> ints = new ArrayList<Integer>();
```
compila senza errori.
9. Il seguente codice:
```
List<Integer> ints = new ArrayList<Integer>();
ints.add(1);
```
compila senza errori.
10. Il seguente codice:

```
List ints = new ArrayList<Integer>();
```
compila senza errori.

Esercizio 16.b)

Generics, Vero o Falso:

1. Se compiliamo file che utilizzano collection senza utilizzare generics con un JDK 1.5, otterremo errori in compilazione
2. La sintassi:
   public interface Collection<E>
   non sottintende l'esistenza di una classe `E`. Si tratta di una nuova terminologia che sta ad indicare come `Collection` supporti la parametrizzazione tramite generics.
3. Non è possibile compilare codice che utilizza Generics con l'opzione "–source 1.4".
4. Non è possibile compilare codice che utilizza Generics con l'opzione "-Xlint".
5. L'esecuzione di un file che non utilizza generics con una JVM 1.5 darà luogo a warning.
6. Il seguente codice:
```
Collection <java.awt.Component> comps =
            new   Vector<java.awt.Component>();
    comps.add(new java.awt.Button());
    Iterator i = comps.iterator();
    while (i.hasNext()){
        Button button = i.next();
        System.out.println(button.getLabel());
}
```
   compila senza errori.
7. Il seguente codice:
```
Collection <Object> objs = new Vector<String>();
```
   compila senza errori.
8. Il seguente codice:
```
public void print(ArrayList<Object> al) {
    Iterator<Object> i = al.iterator();
    while (i.hasNext()) {
      Object o = i.next();
```

```
    }
  }
```

compila senza errori.

9. Il seguente codice:

```
public class MyGeneric <Pippo extends Number> {
    private List<Pippo> list;
public MyGeneric () {
      list = new ArrayList<Pippo>();
    }
public void add(Pippo pippo) {
      list.add(pippo);
    }
public void remove(int i) {
      list.remove(i);
    }
public Pippo get(int i) {
      return list.get(i);
    }
public void copy(ArrayList<?> al) {
      Iterator<?> i = al.iterator();
      while (i.hasNext()) {
          Object o = i.next();
          add(o);
      }
    }
  }
```

compila senza errori.

10. Il seguente codice:

```
public <N extends Number> void print(List<N> list) {
    for (Iterator<A> i = list.iterator();
        i.hasNext( ); ) {
      System.out.println(i.next());
    }
  }
```

compila senza errori.

Soluzioni esercizi modulo 16

Esercizio 16.a)

Autoboxing, auto-unboxing e Generics, Vero o Falso:

1. **Falso.**
2. **Vero.**
3. **Vero.**
4. **Vero.**
5. **Vero.**
6. **Falso.**
7. **Falso.**
8. **Falso.**
9. **Falso.**
10. **Vero.**

Esercizio 16.b)

Generics, Vero o Falso:

1. **Falso** l'`Iterator` deve essere parametrizzato.
2. **Vero.**
3. **Vero.**
4. **Falso.**
5. **Falso.**
6. **Falso** solo di tipo `ActionListener`.
7. **Falso.**
8. **Vero.**
9. **Falso** otterremo il seguente errore:

```
(Pippo) in MyGeneric<Pippo> cannot be applied to
(java.lang.Object)
add(o);
^
```

10. **Vero.**

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Comprendere l'importanza delle nuove caratteristiche introdotte da Java 5 (unità 16.1)	☐	
Comprendere le semplificazioni che ci offre la nuova (doppia) feature di autoboxing e autounboxing (unità 16.2)	☐	
Conoscere le conseguenze e i problemi che genera l'introduzione dell'autoboxing e dell'autounboxing nel linguaggio Java (unità 16.2)	☐	
Capire che cos'è un tipo generic (unità 16.3)	☐	
Saper utilizzare i tipi generic (unità 16.3)	☐	
Aver presente l'impatto su Java dell'introduzione dei generics (unità 16.3)	☐	

Note:

Esercizi modulo 17

Esercizio 17.a)

Ciclo for migliorato ed enumerazioni, Vero o Falso:

1. Il ciclo `for` migliorato può in ogni caso sostituire un ciclo `for`.
2. Il ciclo `for` migliorato può essere utilizzato con gli array e con le classi che implementano `Iterable`.
3. Il ciclo `for` migliorato sostituisce l'utilizzo di `Iterator`.
4. Il ciclo `for` migliorato non può sfruttare correttamente i metodi di `Iterator`.
5. In un ciclo `for` migliorato non è possibile effettuare cicli all'indietro.
6. La classe `java.lang.Enum` implementa `Iterable`, altrimenti non sarebbe possibile utilizzare il ciclo `for` migliorato con le enumerazioni.
7. Il seguente codice viene compilato senza errori:
```
Vector <Integer> integers = new Vector<Integer>();
. . .
for (int i : integers) {
    System.out.println(i);
}
```
8. Il seguente codice viene compilato senza errori:
```
int i = new int[100];
int j = new int[100];

. . .
for (int index1, int index2 : i, j) {
  System.out.println(i+j);
}
```
9. Il seguente codice viene compilato senza errori:
```
Vector <Integer> i = new <Integer>Vector;
Vector <Integer> j = new <Integer>Vector;

. . .
for (int index1, int index2 : i, j) {
  System.out.println(i+j);
}
```
10. Facendo riferimento all'enumerazione definite in questo modulo, il seguente codice viene compilato senza errori:
```
for (TigerNewFeature t : TigerNewFeature.values()) {
  System.out.println(t);
}
```

Esercizio 17.b)

Enumerazioni, Vero o Falso:

1. Le enumerazioni non si possono istanziare se non all'interno della definizione dell'enumerazione stessa. Infatti, possono avere solamente costruttori `private`.
2. Le enumerazioni possono dichiarare metodi e possono essere estese da classi che possono sottoporne a override i metodi. Non è però possibile che un'enum estenda un'altra enum.
3. Il metodo `values()` appartiene ad ogni enumerazione ma non alla classe `java.lang.Enum`.
4. Il seguente codice viene compilato senza errori:

```
public enum MyEnum {
    public void metodo1() {

    }
    public void metodo2() {

    }
    ENUM1, ENUM2;
}
```

5. Il seguente codice viene compilato senza errori:

```
public enum MyEnum {
  ENUM1 {
    public void metodo() {

    }
  }, ENUM2;
  public void metodo2() {

  }
}
```

6. Il seguente codice viene compilato senza errori:

```
public enum MyEnum {
  ENUM1 (), ENUM2;
  private MyEnum(int i) {

  }
}
```

7. Il seguente codice viene compilato senza errori:

```
public class Volume {
    public enum Livello {
        ALTO, MEDIO, BASSO
 } ;
// implementazione della classe . . .
  public static void main(String args[]) {
    switch (getLivello()) {
      case ALTO:
      System.out.println(Livello.ALTO);
      break;
      case MEDIO:
      System.out.println(Livello.MEDIO);
      break;
      case BASSO:
      System.out.println(Livello.BASSO);
      break;
    }
  }
  public static Livello getLivello() {
    return Livello.ALTO;
  }
}
```

8. Se dichiariamo la seguente enumerazione:

```
public enum MyEnum {
  ENUM1 {
    public void metodo1() {

    }
  },
  ENUM2 {
    public void metodo2() {

    }
  }
}
```

il seguente codice potrebbe essere correttamente compilato:

```
MyEnum.ENUM1.metodo1();
```

9. Non è possibile dichiarare enumerazioni con un unico elemento.
10. Si possono innestare enumerazioni in enumerazioni in questo modo:

```
public enum MyEnum {
  ENUM1 (), ENUM2;
  public enum MyEnum2 {a,b,c}
}
```

ed il seguente codice viene compilato senza errori:

```
System.out.println(MyEnum.MyEnum2.a);
```

Soluzioni esercizi modulo 17

Esercizio 17.a)

Ciclo for migliorato ed enumerazioni, Vero o Falso:

1. **Falso.**
2. **Vero.**
3. **Vero.**
4. **Vero.**
5. **Vero.**
6. **Falso.**
7. **Vero.**
8. **Falso.**
9. **Falso.**
10. **Vero.**

Esercizio 17.b)

Enumerazioni, Vero o Falso:

1. **Vero.**
2. **Vero.**
3. **Falso.**
4. **Falso.**
5. **Vero.**
6. **Falso** non è possibile utilizzare il costruttore di default se ne viene dichiarato uno esplicitamente.
7. **Vero.**
8. **Vero.**
9. **Vero.**
10. **Vero.**

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Saper utilizzare il ciclo for-migliorato (unità 17.1)	☐	
Comprendere i limiti e quando applicare il ciclo for migliorato (unità 17.1)	☐	
Comprendere e saper utilizzare le enumerazioni (unità 17.2)	☐	
Comprendere le caratteristiche avanzate e quando utilizzare le enumerazioni (unità 17.2)	☐	

Note:

Esercizi modulo 18

Esercizio 18.a)

Varargs, Vero o Falso:

1. I varargs permettono di utilizzare i metodi come se fossero overloadati
2. La seguente dichiarazione è compilabile correttamente:
```
public void myMethod(String… s, Date d) {
  . . .
}
```
3. La seguente dichiarazione è compilabile correttamente:
```
public void myMethod(String… s, Date d…) {
  . . .
}
```
4. Considerando il seguente metodo:
```
public void myMethod(Object… o) {
  . . .
}
```
la seguente invocazione è corretta:
```
oggetto.myMethod();
```
5. La seguente dichiarazione è compilabile correttamente:
```
public void myMethod(Object o, Object os…) {
  . . .
}
```
6. Considerando il seguente metodo:
```
public void myMethod(int i, int… is) {
  . . .
}
```
La seguente invocazione è corretta:
```
oggetto.myMethod(new Integer(1));
```
7. Le regole dell'override cambiano con l'introduzione dei varargs
8. Il metodo di `java.io.PrintStream` `printf()`, è basato sul metodo `format()` della classe `java.util.Formatter`
9. Il metodo `format()` di `java.util.Formatter` non ha overload perché definito con un varargs
10. Nel caso in cui si passi un array come varargs al metodo `printf()` di `java.io.PrintStream`, questo verrò trattato non come oggetto singolo, ma come se fossero stati passati ad uno ad uno, ogni suo elemento

Esercizio 18.b)

Static import, Vero o Falso:

1. Gli static import permettono di non referenziare i membri statici importati
2. Non è possibile dopo avere importato staticamente una variabile, referenziarla all'interno del codice
3. La seguente importazione non è corretta, perché `java.lang` è sempre importato implicitamente:
```
import static java.lang.System.out;
```
4. Non è possibile importare staticamente classi innestate e/o anonime
5. In alcuni casi gli import statici, potrebbero peggiorare la leggibilità dei nostri file
6. Considerando la seguente enumerazione:
```
package mypackage;
public enum MyEnum {
  A,B,C
}
```
il seguente codice è compilabile correttamente:
```
import static mypackage.MyEnum.*;
public class MyClass {
  public MyClass(){
    out.println(A);
  }
}
```
7. Se utilizziamo gli import statici, si potrebbero importare anche due membri statici con lo stesso nome. Il loro utilizzo all'interno del codice, darebbe luogo ad errori in compilazione, se non referenziati
8. Lo shadowing è un fenomeno che potrebbe verificarsi se si utilizzano gli import statici
9. Essenzialmente l'utilità degli import statici, risiede nella possibilità di scrivere meno codice probabilmente superfluo
10. Non ha senso importare staticamente una variabile, se poi viene utilizzata una sola volta all'interno del codice

Soluzioni esercizi modulo 18

Esercizio 18.a)

Varargs, Vero o Falso:

1. **Vero**
2. **Falso**
3. **Falso**
4. **Vero**
5. **Vero**
6. **Vero**
7. **Falso**
8. **Vero**
9. **Falso**
10. **Vero**

Esercizio 18.b)

Static import, Vero o Falso:

1. **Vero**
2. **Falso**
3. **Falso**
4. **Falso**
5. **Vero**
6. **Falso** `out` non è importato staticamente
7. **Vero**
8. **Vero**
9. **Vero**
10. **Vero**

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Saper utilizzare i varargs e comprenderne le proprietà (unità 18.1)	☐	
Saper utilizzare gli static imports e comprenderne le conseguenze del loro utilizzo (unità 18.2)	☐	

Note:

Esercizi modulo 19

Esercizio 19.a)

Annotazioni, dichiarazioni ed uso, Vero o Falso:

1. Un'annotazione è un modificatore.
2. Un'annotazione è un'interfaccia.
3. I metodi di un'annotazione sembrano metodi astratti, ma in realtà sottintendono un'implementazione implicita.
4. La seguente è una dichiarazione di annotazione valida:
```
public @interface MiaAnnotazione {
    void metodo();
}
```
5. La seguente è una dichiarazione di annotazione valida:
```
public @interface MiaAnnotazione {
    int metodo(int valore) default 5;
}
```
6. La seguente è una dichiarazione di annotazione valida:
```
public @interface MiaAnnotazione {
    int metodo() default -99;
    enum MiaEnum{VERO, FALSO};
    MiaEnum miaEnum();
}
```
7. Supponiamo che l'annotazione `MiaAnnotazione` definita nel punto 6 sia corretta. Con il seguente codice essaviene utilizzata correttamente:
```
public @MiaAnnotazione (
    MiaAnnotazione.MiaEnum.VERO
)
MiaAnnotazione.MiaEnum m() {
    return MiaAnnotazione.MiaEnum.VERO;
}
```
8. Supponiamo che l'annotazione `MiaAnnotazione` definita nel punto 6 sia corretta. Con il seguente codice essa viene utilizzata correttamente:
```
public @MiaAnnotazione (
    miaEnum=MiaAnnotazione.MiaEnum.VERO
)
MiaAnnotazione.MiaEnum m() {
    return @MiaAnnotazione.miaEnum;
}
```
9. Consideriamo la seguente annotazione.

```
public @interface MiaAnnotazione {
    int valore();
}
```

Con il seguente codice essa viene utilizzata correttamente:

```
public @MiaAnnotazione (
    5
)
void  m()
    ...
}
```

10. Consideriamo la seguente annotazione:

```
public @interface MiaAnnotazione {}
```

Con il seguente codice essa, viene utilizzata correttamente:

```
public @MiaAnnotazione void  m() {
    ...
}
```

Esercizio 19.b)

Annotazioni e libreria, Vero o Falso:

1. La seguente annotazione è anche una metaannotazione:
```
public @interface MiaAnnotazione ()
```
2. La seguente annotazione è anche una metaannotazione:
```
@Target (ElementType.SOURCE)
public @interface MiaAnnotazione ()
```
3. La seguente annotazione è anche una metaannotazione:
```
@Target (ElementType.@INTERFACE)
public @interface MiaAnnotazione ()
```
4. La seguente annotazione, se applicata ad un metodo, sarà documentata nella relativa documentazione Javadoc:
```
@Documented
@Target (ElementType.ANNOTATION_TYPE)
public @interface MiaAnnotazione ()
```
5. La seguente annotazione sarà ereditata se e solo se applicata ad una classe:
```
@Inherited
@Target (ElementType.METHOD)
public @interface MiaAnnotazione ()
```
6. Per la seguente annotazione è anche possibile creare un processore di annotazioni che riconosca al runtime il tipo di annotazione, per implementare un particolare comportamento:

```
@Documented
@Target (ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface MiaAnnotazione ()
```

7. `Override` è un'annotazione standard per segnalare al runtime di Java che un metodo effettua l'override di un altro.
8. `Deprecated` in fondo può essere considerata anche una metaannotazione, perché applicabile ad altre annotazioni.
9. `SuppressWarnings` è una annotazione a valore singolo. `Deprecated` e `Override` invece sono entrambe annotazioni segnalibro.
10. Non è possibile utilizzare contemporaneamente le tre annotazioni standard su di un'unica classe.

Soluzioni esercizi modulo 18

Esercizio 19.a)

Annotazioni, dichiarazioni ed uso, Vero o Falso:

1. **Falso** è un tipo annotazione.
2. **Falso** è un tipo annotazione.
3. **Vero**
4. **Falso** un metodo di un annotazione non può avere come tipo di restituzione `void`.
5. **Falso** un metodo di un annotazione non può avere parametri in input.
6. **Vero.**
7. **Falso** infatti è legale sia il codice del metodo m(), sia il dichiarare public come primo modificatore. Non è legale però passare in input all'annotazione il valore `MiaAnnotazione.MiaEnum.VERO`, senza specificare una sintassi del tipo chiave = valore.
8. **Falso** infatti, la sintassi:
   `return @MiaAnnotazione.miaEnum;`
   non è valida. Non si può utilizzare un'annotazione come se fosse una classe con variabili statiche pubbliche…
9. **Falso** infatti l'annotazione in questione non è a valore singolo, perché il suo unico elemento non si chiama `value()`.
10. **Vero.**

Esercizio 19.b)

Annotazioni e libreria, Vero o Falso:

1. **Vero** infatti, se non si specifica con la metaannotazione `Target` quali sono gli elementi a cui è applicabile l'annotazione in questione, l'annotazione sarà di default applicabile a qualsiasi elemento.
2. **Falso** il valore `ElementType.SOURCE` non esiste.
3. **Falso** il valore `ElementType.@INTERFACE` non esiste.
4. **Falso** non è neanche applicabile a metodi per via del valore di `Target`, che è `ElementType.ANNOTATION_TYPE`.
5. **Falso** infatti non può essere applicata ad una classe se è annotata con `@Target (ElementType.METHOD)`.
6. **Vero.**
7. **Falso** al compilatore, non al runtime.
8. **Vero.**
9. **Vero**.
10. **Vero** `Override` non è applicabile a classi.

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Comprendere cosa sono i metadati e la loro relatività (unità 19.1, 19.2)	☐	
Comprendere l'utilità delle annotazioni (unità 19.1, 19.2, 19.3, 19.4)	☐	
Saper definire nuove annotazioni (unità 19.2)	☐	
Saper annotare elementi Java ed altre annotazioni (unità 19.2, 19.3)	☐	
Saper utilizzare le annotazioni definite dalla libreria: le annotazioni standard e le metaannotazioni (unità 19.3, 19.4)	☐	