

# Ingegneria del Software

## *Esercitazione 3*

# Secure String

*Definire classi che permettano la stampa condizionata di una stringa*

## *Specifiche:*

- *SecureString* è una classe astratta che incapsula una stringa e che offre un metodo *securePrint(Object o)* che stampa la stringa dopo un controllo di sicurezza.
- Il controllo di sicurezza deve essere personalizzabile dalle sottoclassi
- Implementare una sottoclasse *CapabilitySecureString* di *SecureString* che offre un metodo *getCapability()* che ritorna un oggetto. Una volta chiamato, la stampa è possibile solo se l'oggetto passato a *securePrint* è quello ritornato da *getCapability()*

# Runtime Type Checking

*Cosa stampa questo programma?*

```
class Father { }
class Son extends Father { }
class Test {
    public static void main(String[] s) {
        Father f = new Son();
        Father f2 = new Father();
        if (f instanceof Father)
            System.out.println("True");
        else
            System.out.println("False");
        if (f.getClass() == f2.getClass())
            System.out.println("True");
        else
            System.out.println("False");
    }
}
```

# Runtime Type Checking

***Risposta:***

> *True*

> *False*

*instanceof* restituisce true se c'è compatibilità di assegnamento.

*getClass()* ritorna una reference alla (unica) istanza di tipo *Class* della classe dell'oggetto su cui è chiamato

```
class Father {  
    int x;  
    public Father(int x) {  
        this.x = x;  
    }  
    public int m(Father f) {  
        return (f.x - this.x);  
    }  
}  
class Son extends Father {  
    int y;  
    public Son(int x, int y) {  
        super(x); this.y = y;  
    }  
    public int m(Father f) {  
        return 100;  
    }  
    public int m(Son s) {  
        return super.m(s) + (s.y - this.y);  
    }  
}
```

```
public class MainClass {  
    public static void main(String args[]) {  
        Father f1, f2;   Son s1, s2;  
        f1 = new Father(3);  
        f2 = new Son(3,10);  
        System.out.println(f1.m(f2));           /* 1 */  
        System.out.println(f2.m(f1));           /* 2 */  
        s1 = new Son(4,21);  
        System.out.println(s1.m(f1) + s1.m(f2)); /*3*/  
        System.out.println(f1.m(s1) + f2.m(s1)); /*4*/  
        s2 = new Son(5,22);  
        System.out.println(s1.m(s2));           /* 5 */  
    }  
}
```

- Risposta: La classe Son definisce un metodo m(Father), che effettua un overriding del metodo m(Father) in Father, e un overloading di m, con signature m(Son).
- Istruzione 1:
  - Parte statica (overloading): f1 ha tipo statico Father -> il metodo viene cercato nella classe father. f2 ha tipo statico Father -> viene cercato un metodo la cui signature è compatibile con m(Father). Il metodo viene trovato, è proprio Father.m(Father), e occorre cercare tra i metodi che ne effettuano un overriding.
  - Parte dinamica (overriding): f1 ha tipo dinamico Father -> viene scelto il metodo Father.m(Father). Stampato f2.x - f1.x, ossia 0.
- Istruzione 2:
  - Parte statica (overloading): ancora, f1 e f2 hanno tipo statico Father. Quindi viene sempre scelto Father.m(Father) (o uno che ne fa overriding).
  - Parte dinamica (overriding): stavolta f2 ha tipo dinamico Son, e quindi viene scelto il metodo Son.m(Father), che effettua overriding. Viene stampato 100.

- `System.out.println(s1.m(f1) + s1.m(f2)); /*3*/`
- Istruzione 3:
  - Parte statica: le due chiamate hanno come tipo statico `Son.m(Father)`. Quindi viene scelto questo metodo, o un metodo che ne fa override...
  - Parte dinamica: ...ma nessun metodo fa override di `Son.m(Father)`, quindi per entrambe le chiamate viene eseguito questo. Notare che, nonostante `f2` abbia tipo dinamico `Son`, `s.m(f2)` NON esegue `Son.m(Son)!!!` Viene stampato 200.
- `System.out.println(f1.m(s1) + f2.m(s1)); /*4*/`
- Istruzione 4:
  - Parte statica: le due chiamate hanno come tipo statico `Father.m(Son)`. Non esiste un metodo con questa signature, ma `Father.m(Father)` è compatibile. Viene scelto quindi `Father.m(Father)`, o un metodo che ne fa override.
  - Parte dinamica: dal momento che `f1` e `f2` hanno diversi tipi dinamici, la prima chiamata usa il metodo `Father.m(Father)`, la seconda usa il metodo overridden `Son.m(Father)`.  
Il risultato è  $1 + 100 = 101$ .

- Istruzione 5:
  - Statico è `Son.m(Son)`, e non ci sono metodi che ne fanno overriding. All'interno, viene effettuata una chiamata di `super.m(s)`, con `s` parametro il cui tipo statico è `Son`; `super` significa "della superclasse statica" - quindi di `Father`. Staticamente, questo significa cercare `Father.m(Son)`, che non esiste: Però esiste `Father.m(Father)`, che è compatibile. A runtime viene invocato questo. Quindi, `super.m(s)` restituisce 1, e l'istruzione 5 stampa 2 a schermo.



# Cron

*Si progetti un package che offra un "demone temporale" simile a cron di Unix*

## *Specifiche:*

- L'utente del package deve poter creare un demone, registrare presso di lui una serie di coppie *<orario, azione da compiere>*
- Il demone temporale, una volta avviato, deve eseguire le azioni registrate all'orario prestabilito.
- Si supponga che non si possano registrare più di 10 azioni, che ogni azione debba venir eseguita una volta soltanto e che una volta eseguite tutte le azioni cron termini la sua esecuzione.
- Si può interpretare l'orario di esecuzione come "orario indicativo": viene garantito che l'azione viene eseguita *\*dopo\** l'orario specificato

# Exceptions

Ask for forgiveness

```
try{  
    set.add(new Complex(1.0,1.0));  
}catch(FullStackException e){  
    System.err.println("Stack is full");  
}
```

Ask for permission

```
if (!set.isFull()) {  
    set.add(new Complex(1.0, 1.0));  
}
```

# Stack with Exception (I)

*Eccezioni gestite a compile time (checked)*

```
public class OutOfDataException extends Exception {}
```

```
public class SafeStack extends Stack {  
    public int safePop() throws OutOfElementException {  
        if (cur > 0) {  
            cur--;  
            return data[cur];  
        }  
        else throw new OutOfElementException();  
    }  
}
```

# Stack with Exception (II)

*Eccezioni gestite solamente a runtime*

```
public class OutOfDataException extends RuntimeException {}
```

```
public class SafeStack extends Stack {  
    public int safePop() {  
        if (cur > 0) {  
            cur--;  
            return data[cur];  
        }  
        else throw new OutOfElementException();  
    }  
}
```

# Stack with Exception (III)

*Aggiungere a SafeStack un metodo safePush  
che gestisca i casi limite dell'inserimento*

# StringBuffer vs String

*Testare la differenza tra la costruzione di stringhe con la classe String e con la classe StringBuffer.*

# Input/Output

*Scrivere un programma che legga da tastiera una frase e una parola, queste devono essere passate ad un metodo statico e che restituisca il numero di occorrenze della parola nella frase.*

# Stack

Implementare la classe Stack con i Generics