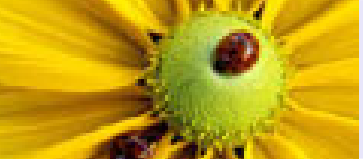
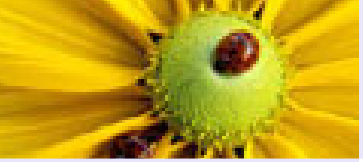
Linguaggi di Programmazione I – Lezione 15

Prof. Marco Faella
mailto://m.faella@unina.it
http://wpage.unina.it/mfaella
Materiale didattico elaborato con i Proff. Sette e Bonatti

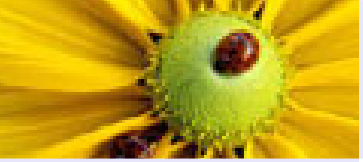
3 aprile 2023



Eccezioni (Gestione degli errori)

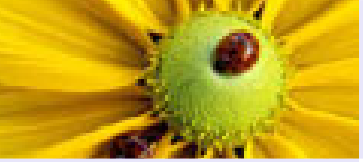


Eccezioni: il meccanismo



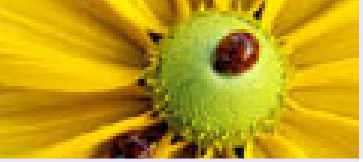
Introduzione

- Le `eccezioni` denotano "eventi eccezionali" la cui occorrenza altera il flusso normale delle istruzioni



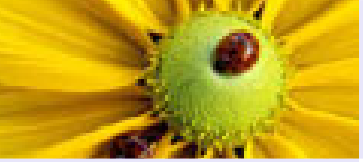
Introduzione

- Le `eccezioni` denotano "eventi eccezionali" la cui occorrenza altera il flusso normale delle istruzioni
- Es.: risorse hardware indisponibili, hardware malfunzionante, bachi nel software ...



Introduzione

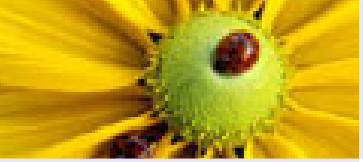
- Le `eccezioni` denotano "eventi eccezionali" la cui occorrenza altera il flusso normale delle istruzioni
- Es.: risorse hardware indisponibili, hardware malfunzionante, bachi nel software ...
- Quando capita un tale evento, si dice che viene "lanciata una eccezione"



Meccanismi linguistici

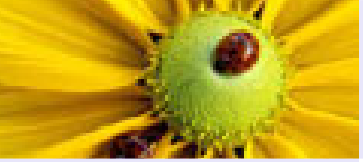
Il linguaggio supporta i seguenti meccanismi relativi alle eccezioni:

- **Lanciare** un'eccezione (istruzione `throw`)
- **Dichiarare** che un metodo lancia un'eccezione (dichiarazione `throws`)
- **Catturare** un'eccezione (blocco `try-catch`)



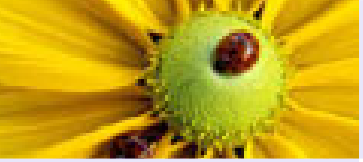
Lanciare un'eccezione

- In Java tutto ciò che non è primitivo è un oggetto.



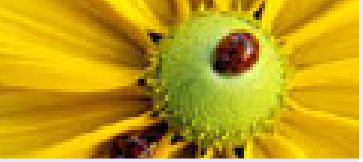
Lanciare un'eccezione

- In Java tutto ciò che non è primitivo è un oggetto. Le eccezioni non fanno "eccezione" a questa regola.



Lanciare un'eccezione

- In Java tutto ciò che non è primitivo è un oggetto. Le eccezioni non fanno "eccezione" a questa regola.
- Ogni eccezione è una istanza di una sottoclasse della classe `Throwable`



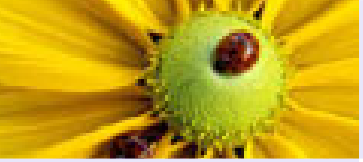
Lanciare un'eccezione

- In Java tutto ciò che non è primitivo è un oggetto. Le eccezioni non fanno "eccezione" a questa regola.
- Ogni eccezione è una istanza di una sottoclasse della classe `Throwable`
- Una eccezione viene lanciata usando la parola riservata `throw`:

```
throw <exp>;
```

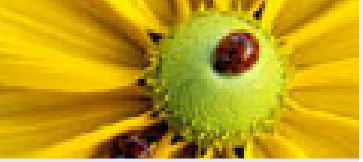
dove <exp> è un'espressione di tipo dichiarato Throwable o suo sottotipo.
Esempio:

```
throw new IllegalArgumentException();
```

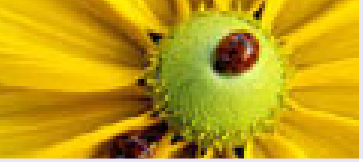


Ciclo di vita di un'eccezione

- Lanciare un'eccezione interrompe il normale flusso di esecuzione
- Se non viene *catturata* localmente, l'eccezione termina il metodo corrente e passa al chiamante, che ha la possibilità di catturarla
- Se neanche il metodo chiamante la cattura, l'eccezione continua a risalire lo stack di attivazione, fino a raggiungere il main
- Se neanche il main la cattura, l'eccezione termina il programma e la JVM stampa il contenuto dell'eccezione *(stack trace)*

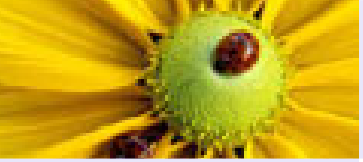


Catturare le eccezioni



try e catch

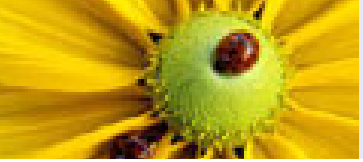
- Per catturare le eccezioni, il codice che potrebbe lanciare eccezioni viene inglobato in un blocco marcato `try`
- Il codice che assume la responsabilità di gestire un'eccezione va inglobato in una clausola `catch`



try e catch

- Per catturare le eccezioni, il codice che potrebbe lanciare eccezioni viene inglobato in un blocco marcato `try`
- Il codice che assume la responsabilità di gestire un'eccezione va inglobato in una clausola `catch`

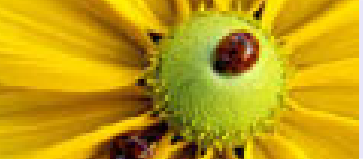
```
try {
  // Codice "rischioso"
} catch (Eccezione1 e) {
  // Codice che gestisce una Eccezione1
} catch (Eccezione2 e) {
  // Codice che gestisce una Eccezione2
}
// Codice non rischioso
```



finally

- Un blocco (opzionale) marcato `finally` verrà SEMPRE eseguito, anche dopo il lancio e la gestione (eventuale) dell'eccezione.

```
try {
  // Codice "rischioso"
} catch (Eccezione1 e) {
  // Codice che gestisce una Eccezione1
} catch (Eccezione2 e) {
  // Codice che gestisce una Eccezione2
} finally {
  // Codice da eseguire in ogni caso
}
// Codice non rischioso
```

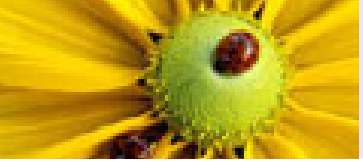


finally

- Un blocco (opzionale) marcato `finally` verrà SEMPRE eseguito, anche dopo il lancio e la gestione (eventuale) dell'eccezione.

```
try {
  // Codice "rischioso"
} catch (Eccezione1 e) {
  // Codice che gestisce una Eccezione1
} catch (Eccezione2 e) {
  // Codice che gestisce una Eccezione2
} finally {
  // Codice da eseguire in ogni caso
}
// Codice non rischioso
```

- Il blocco `finally` viene eseguito perfino dopo una eventuale istruzione `return` presente nei blocchi `try` o `catch`

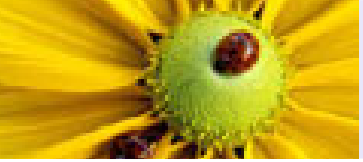


finally

- Un blocco (opzionale) marcato `finally` verrà SEMPRE eseguito, anche dopo il lancio e la gestione (eventuale) dell'eccezione.

```
try {
  // Codice "rischioso"
} catch (Eccezione1 e) {
  // Codice che gestisce una Eccezione1
} catch (Eccezione2 e) {
  // Codice che gestisce una Eccezione2
} finally {
  // Codice da eseguire in ogni caso
}
// Codice non rischioso
```

- Il blocco `finally` viene eseguito perfino dopo una eventuale istruzione `return` presente nei blocchi `try` o `catch`
- IL BLOCCO `finally` VIENE ESEGUITO SEMPRE

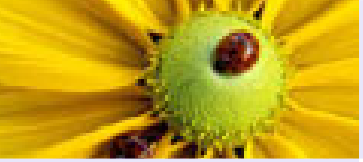


finally

- Un blocco (opzionale) marcato `finally` verrà SEMPRE eseguito, anche dopo il lancio e la gestione (eventuale) dell'eccezione.

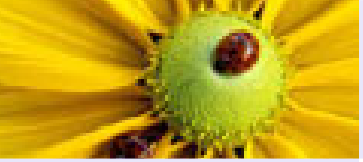
```
try {
  // Codice "rischioso"
} catch (Eccezione1 e) {
  // Codice che gestisce una Eccezione1
} catch (Eccezione2 e) {
  // Codice che gestisce una Eccezione2
} finally {
  // Codice da eseguire in ogni caso
}
// Codice non rischioso
```

- Il blocco `finally` viene eseguito perfino dopo una eventuale istruzione `return` presente nei blocchi `try` o `catch`
- IL BLOCCO `finally` VIENE ESEGUITO SEMPRE
- Il blocco `finally` potrebbe non essere eseguito o potrebbe non completare l'esecuzione solo in conseguenza di un crash totale del sistema oppure tramite una invocazione di `System.exit(int status)`



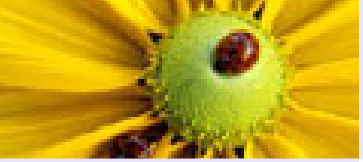
Vincoli sintattici

- Le clausole `catch` ed il blocco `finally` sono opzionali, ma deve essere presente **almeno uno dei due**
- Un blocco `try` solitario causa un errore di compilazione



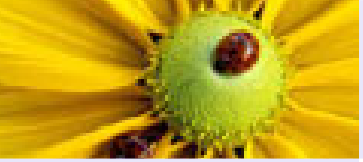
Vincoli sintattici

- Le clausole `catch` ed il blocco `finally` sono opzionali, ma deve essere presente **almeno uno dei due**
- Un blocco `try` solitario causa un errore di compilazione
- Se esistono una o più clausole `catch`, esse devono seguire immediatamente il blocco `try`



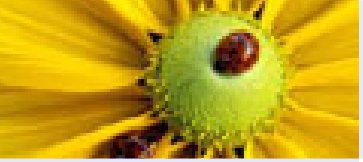
Vincoli sintattici

- Le clausole `catch` ed il blocco `finally` sono opzionali, ma deve essere presente **almeno uno dei due**
- Un blocco `try` solitario causa un errore di compilazione
- Se esistono una o più clausole `catch`, esse devono seguire immediatamente il blocco `try`
- Se esiste il blocco `finally`, esso deve comparire per ultimo

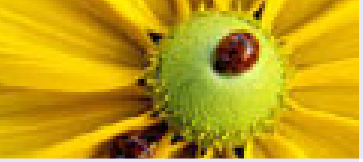


Vincoli sintattici

- Le clausole `catch` ed il blocco `finally` sono opzionali, ma deve essere presente **almeno uno dei due**
- Un blocco `try` solitario causa un errore di compilazione
- Se esistono una o più clausole `catch`, esse devono seguire immediatamente il blocco `try`
- Se esiste il blocco `finally`, esso deve comparire per ultimo
- È significativo l'ordine delle clausole `catch` (vedremo tra poco)

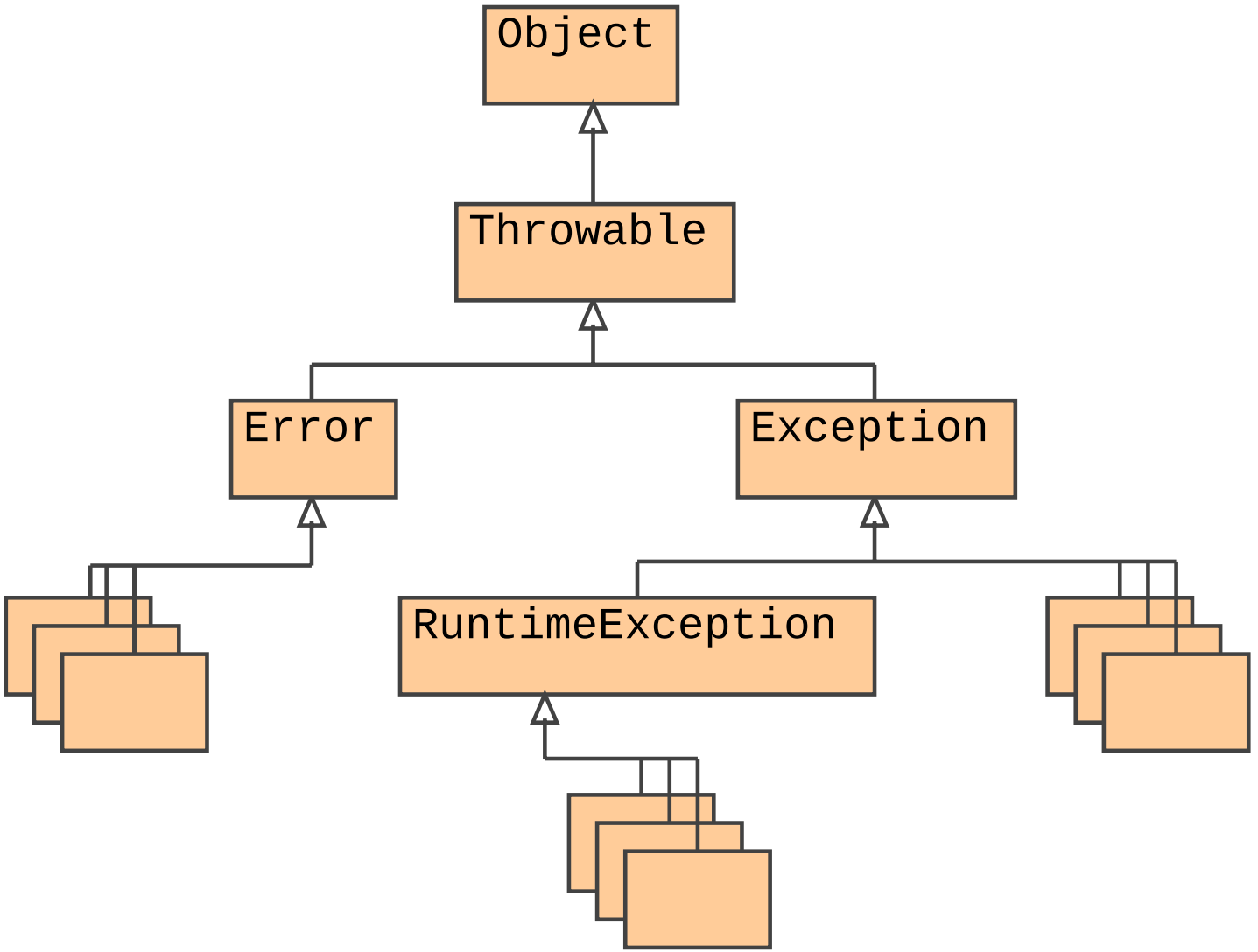


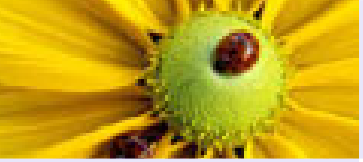
Eccezioni: i dettagli



Gerarchia (1)

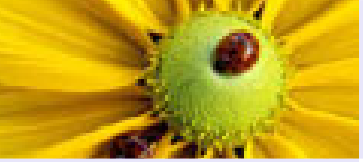
Ecco le principali classi di eccezioni:





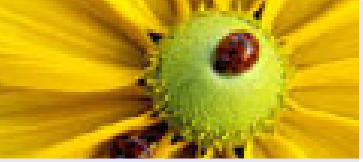
Gerarchia (2)

- La classe `Throwable` rappresenta tutti gli oggetti che possono essere lanciati



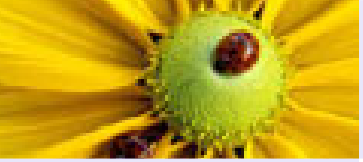
Gerarchia (2)

- La classe `Throwable` rappresenta tutti gli oggetti che possono essere lanciati Essa contiene il metodo `printStackTrace`
- La classe `Error` e le sue sottoclassi rappresentano situazioni insolite che non sono causate da errori di programmazione o da ciò che normalmente succede durante l'esecuzione del programma. Per esempio, la JVM ha esaurito la memoria oppure qualche altra risorsa non è disponibile.



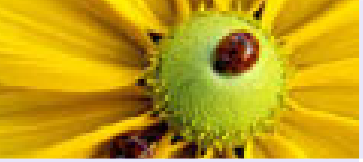
Gerarchia (2)

- La classe `Throwable` rappresenta tutti gli oggetti che possono essere lanciati Essa contiene il metodo `printStackTrace`
- La classe `Error` e le sue sottoclassi rappresentano situazioni insolite che non sono causate da errori di programmazione o da ciò che normalmente succede durante l'esecuzione del programma. Per esempio, la JVM ha esaurito la memoria oppure qualche altra risorsa non è disponibile.
  In genere, una applicazione non è capace di riprendersi da una situazione di errore. Pertanto, queste eccezioni solitamente non vengono catturate



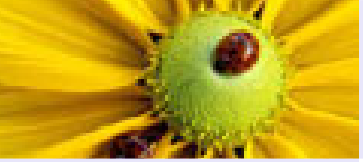
Gerarchia (2)

- La classe `Throwable` rappresenta tutti gli oggetti che possono essere lanciati Essa contiene il metodo `printStackTrace`
- La classe `Error` e le sue sottoclassi rappresentano situazioni insolite che non sono causate da errori di programmazione o da ciò che normalmente succede durante l'esecuzione del programma. Per esempio, la JVM ha esaurito la memoria oppure qualche altra risorsa non è disponibile.
  In genere, una applicazione non è capace di riprendersi da una situazione di errore. Pertanto, queste eccezioni solitamente non vengono catturate
- La classe `RuntimeException` rappresenta pure eventi eccezionali, ma dovuti al programma (errori di programmazione, bachi). Il programmatore che si accorge di un baco dovuto ad un suo errore deve correggerlo, non gestirlo! Pertanto, anche queste eccezioni solitamente non vengono catturate



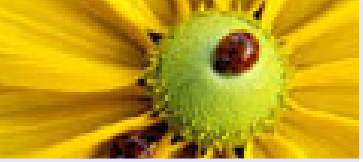
Eccezioni catturate (1)

- Una clausola `catch (E e)` cattura ogni oggetto-eccezione il cui tipo effettivo è sottotipo di E



Eccezioni catturate (1)

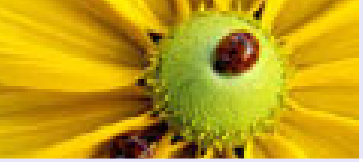
- Una clausola `catch (E e)` cattura ogni oggetto-eccezione il cui tipo effettivo è sottotipo di E
- Un'eccezione di tipo effettivo E verrà catturata dal *primo* blocco catch in grado di catturarla
- Esempio: la classe `IndexOutOfBoundsException` ha due sottoclassi, `ArrayIndexOutOfBoundsException` e `StringIndexOutOfBoundsException`; si può scrivere una unica clausola che catturi una qualunque di queste eccezioni:



Eccezioni catturate (1)

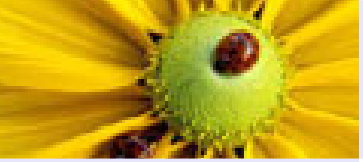
- Una clausola `catch (E e)` cattura ogni oggetto-eccezione il cui tipo effettivo è sottotipo di E
- Un'eccezione di tipo effettivo E verrà catturata dal *primo* blocco catch in grado di catturarla
- Esempio: la classe `IndexOutOfBoundsException` ha due sottoclassi, `ArrayIndexOutOfBoundsException` e `StringIndexOutOfBoundsException`; si può scrivere una unica clausola che catturi una qualunque di queste eccezioni:

```
try {
  // Codice che potrebbe lanciare una eccezione
  // IndexOutOfBoundsException oppure
  // ArrayIndexOutOfBoundsException oppure
  // StringIndexOutOfBoundsException
}
catch (IndexOutOfBoundsException e) {
  e.printStackTrace();
}
```



Casi particolari

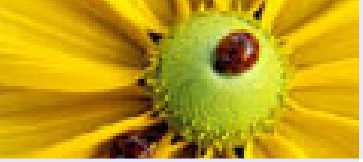
- Eventuali eccezioni lanciate dall'interno dei blocchi catch e finally non vengono catturate dagli altri blocchi catch dello stesso costrutto
- Se proprio necessario, i try-catch possono essere annidati



Eccezioni catturate (2)

- Resistere alla tentazione di scrivere una unica clausola catch-all:

```
try {
  // codice rischioso
} catch (Exception e) {
}
```

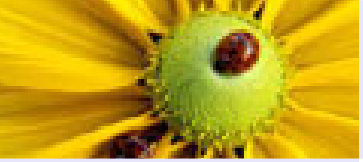


Eccezioni catturate (2)

- Resistere alla tentazione di scrivere una unica clausola catch-all:

```
try {
  // codice rischioso
} catch (Exception e) {
}
```

- L'ordine delle clausole `catch` è importante.



Eccezioni catturate (2)

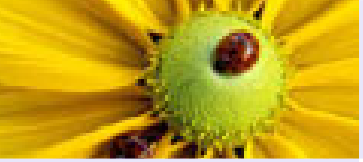
- Resistere alla tentazione di scrivere una unica clausola catch-all:

```
try {
  // codice rischioso
} catch (Exception e) {
}
```

- L'ordine delle clausole `catch` è importante.
- Nell'esempio precedente, se avessimo scritto:

```
try {
  // Codice che potrebbe lanciare una eccezione
  //    IndexOutOfBoundsException, oppure
  //    ArrayIndexOutOfBoundsException
} catch (IndexOutOfBoundsException e) {
  // Gestisce l'eccezione
} catch (ArrayIndexOutOfBoundsException e) {
  // Gestisce l'eccezione
}
```

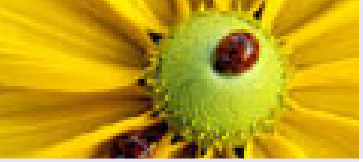
il codice non sarebbe stato compilato, perché il secondo catch è ridondante



Eccezioni catturate (3)

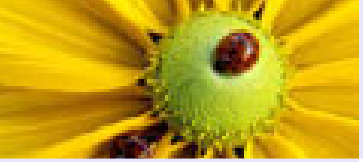
- È corretto, invece, scrivere:

```
try {
  // Codice che potrebbe lanciare una eccezione
  //    IndexOutOfBoundsException, oppure
  //    ArrayIndexOutOfBoundsException
} catch (ArrayIndexOutOfBoundsException e) {
  // Gestisce l'eccezione
} catch (IndexOutOfBoundsException e) {
  // Gestisce l'eccezione
}
```



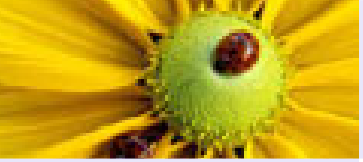
. . . e non catturate

- Come facciamo a sapere che un metodo può lanciare una eccezione che dobbiamo catturare?



. . . e non catturate

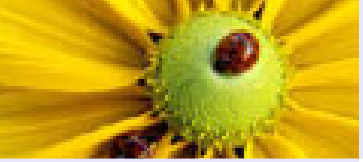
- Come facciamo a sapere che un metodo può lanciare una eccezione che dobbiamo catturare?
- Così come la dichiarazione del metodo deve specificare il numero e il tipo dei parametri, il tipo di ritorno, anche le eccezioni che un metodo può lanciare DEVONO essere dichiarate (a meno che non siano sottoclassi di `Error` o di `RuntimeException`).



. . . e non catturate

- Come facciamo a sapere che un metodo può lanciare una eccezione che dobbiamo catturare?
- Così come la dichiarazione del metodo deve specificare il numero e il tipo dei parametri, il tipo di ritorno, anche le eccezioni che un metodo può lanciare DEVONO essere dichiarate (a meno che non siano sottoclassi di `Error` o di `RuntimeException`).
- La parola chiave `throws` viene usata per elencare le eccezioni che possono fuoriuscire da un metodo:

```
void miaFunzione() throws MiaEccezione1, MiaEccezione2 {
  // qui il codice per il metodo
}
```

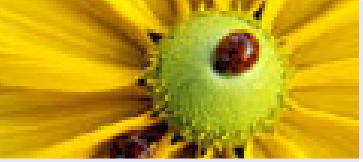


. . . e non catturate

- Come facciamo a sapere che un metodo può lanciare una eccezione che dobbiamo catturare?
- Così come la dichiarazione del metodo deve specificare il numero e il tipo dei parametri, il tipo di ritorno, anche le eccezioni che un metodo può lanciare DEVONO essere dichiarate (a meno che non siano sottoclassi di `Error` o di `RuntimeException`).
- La parola chiave `throws` viene usata per elencare le eccezioni che possono fuoriuscire da un metodo:

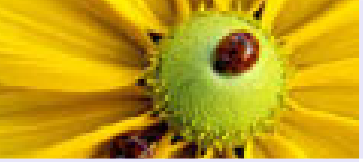
```
void miaFunzione() throws MiaEccezione1, MiaEccezione2 {
  // qui il codice per il metodo
}
```

- Il fatto che un metodo dichiara l'eccezione non significa che esso la lancerà sempre, ma avverte l'utilizzatore che esso potrebbe lanciarla.



Handle or Declare

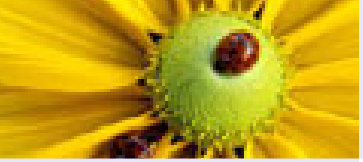
- Se un metodo non lancia direttamente una eccezione, ma richiama un altro metodo che può farlo, allora si deve scegliere almeno una di queste opzioni (regola *handle or declare*):



Handle or Declare

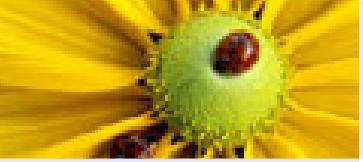
- Se un metodo non lancia direttamente una eccezione, ma richiama un altro metodo che può farlo, allora si deve scegliere almeno una di queste opzioni (regola *handle or declare*):

  1. Gestire l'eccezione fornendo le opportune sezioni `try/catch`



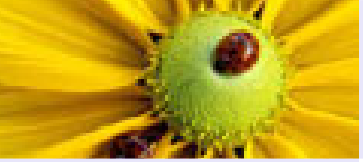
Handle or Declare

- Se un metodo non lancia direttamente una eccezione, ma richiama un altro metodo che può farlo, allora si deve scegliere almeno una di queste opzioni (regola *handle or declare*):
  1. Gestire l'eccezione fornendo le opportune sezioni `try/catch`
  2. Dichiarare l'eccezione nell'intestazione del metodo



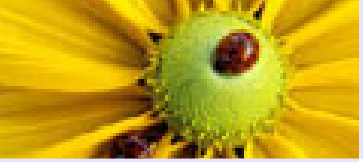
Handle or Declare

- Se un metodo non lancia direttamente una eccezione, ma richiama un altro metodo che può farlo, allora si deve scegliere almeno una di queste opzioni (regola *handle or declare*):
  1. Gestire l'eccezione fornendo le opportune sezioni `try/catch`
  2. Dichiarare l'eccezione nell'intestazione del metodo
- Una "eccezione" a questa regola: `Error` e `RuntimeException` sono esenti dall'obbligo di dichiarazione



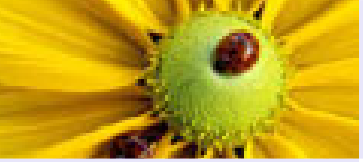
Handle or Declare

- Se un metodo non lancia direttamente una eccezione, ma richiama un altro metodo che può farlo, allora si deve scegliere almeno una di queste opzioni (regola *handle or declare*):
  1. Gestire l'eccezione fornendo le opportune sezioni `try/catch`
  2. Dichiarare l'eccezione nell'intestazione del metodo
- Una "eccezione" a questa regola: `Error` e `RuntimeException` sono esenti dall'obbligo di dichiarazione
  Esse sono *unchecked* (non verificate) dal compilatore, mentre le rimanenti eccezioni sono dette *checked* (verificate)



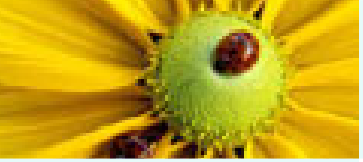
Handle or Declare

- Se un metodo non lancia direttamente una eccezione, ma richiama un altro metodo che può farlo, allora si deve scegliere almeno una di queste opzioni (regola *handle or declare*):
  1. Gestire l'eccezione fornendo le opportune sezioni `try/catch`
  2. Dichiarare l'eccezione nell'intestazione del metodo
- Una "eccezione" a questa regola: `Error` e `RuntimeException` sono esenti dall'obbligo di dichiarazione
  Esse sono *unchecked* (non verificate) dal compilatore, mentre le rimanenti eccezioni sono dette *checked* (verificate)
  Ora, forse, si capisce il motivo della gerarchia precedentemente esposta.



Handle or Declare

- Se un metodo non lancia direttamente una eccezione, ma richiama un altro metodo che può farlo, allora si deve scegliere almeno una di queste opzioni (regola *handle or declare*):
  1. Gestire l'eccezione fornendo le opportune sezioni `try/catch`
  2. Dichiarare l'eccezione nell'intestazione del metodo
- Una "eccezione" a questa regola: `Error` e `RuntimeException` sono esenti dall'obbligo di dichiarazione
  Esse sono *unchecked* (non verificate) dal compilatore, mentre le rimanenti eccezioni sono dette *checked* (verificate)
  Ora, forse, si capisce il motivo della gerarchia precedentemente esposta.
- Nota: l'*or* della regola non è esclusivo, nel senso che si può decidere di fare entrambe le cose.

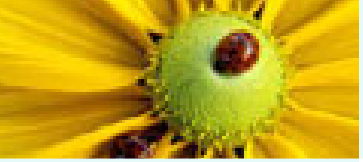


Esempio (1)

Quali problemi ci sono in questo codice?

```
void f1() {
  f2();
}

void f2() {
  throw new IOException();
}
```



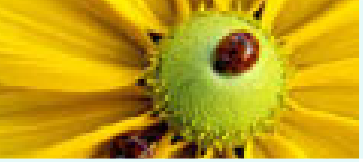
Esempio (1)

Quali problemi ci sono in questo codice?

```
void f1() {
  f2();
}

void f2() {
  throw new IOException();
}
```

- Il metodo `f2` lancia una eccezione `checked` ma non la dichiara; questo è un errore di compilazione



Esempio (1)

Quali problemi ci sono in questo codice?

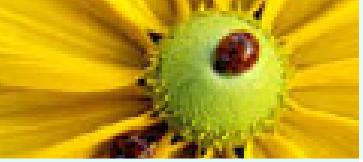
```
void f1() {
  f2();
}

void f2() {
  throw new IOException();
}
```

- Il metodo `f2` lancia una eccezione `checked` ma non la dichiara; questo è un errore di compilazione
- Se esso l'avesse dichiarata, come in:

```
void f2() throws IOException {...}
```

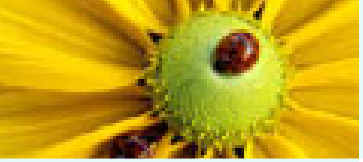
  il problema l'avrebbe `f1` che dovrebbe ora dichiararla o catturarla.



Esempio (2)

Quali problemi ci sono in questo codice?

```
import java.io.*;
class Test {
  public int f1() throws EOFException {
    return f2();
  }
  public int f2() throws EOFException {
    // qui il codice che lancia effettivamente l'eccezione
    return 1;
  }
}
```

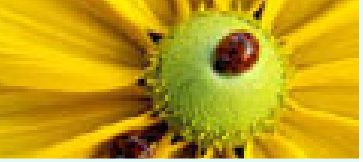


Esempio (2)

Quali problemi ci sono in questo codice?

```
import java.io.*;
class Test {
  public int f1() throws EOFException {
    return f2();
  }
  public int f2() throws EOFException {
    // qui il codice che lancia effettivamente l'eccezione
    return 1;
  }
}
```

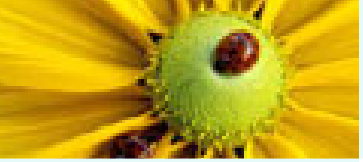
- Nessun problema
- Poiché `EOFException` è sottoclasse di `IOException`, che è sottoclasse di `Exception`, essa è una eccezione *checked*. Essa viene regolarmente dichiarata ed il codice regolarmente compilato.



Esempio (3)

Quali problemi ci sono in questo codice?

```
public void  f1() {
  // qui codice che puo' lanciare NullPointerException
}
```

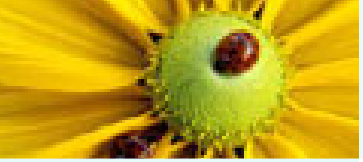


Esempio (3)

Quali problemi ci sono in questo codice?

```
public void  f1() {
  // qui codice che puo' lanciare NullPointerException
}
```

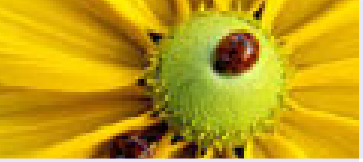
- Nessun problema
- Poiché `NullPointerException` è sottoclasse di `RuntimeException`, essa è una eccezione *unchecked*. Non è necessario nè dichiararla, nè catturarla, ed il codice viene regolarmente compilato.



Esempio (4)

Analogamente, questo codice compila correttamente:

```
class TestEx {
  public static void main (String [] args) {
    mioMetodo();
  }
  static void mioMetodo() { // Non c'e' bisogno di
                            // dichiarare un Error
    fai();
  }
  static void fai() { // Non c'e' bisogno di dichiarare un Error
    try {
      throw new Error();
    }
    catch(Error me) {
      throw me; // Ti ho preso, ma ora di rilancio
    }
  }
}
```

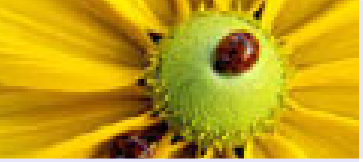


Nuove eccezioni

- È possibile usare tipi di eccezioni già presenti nelle Java API, oppure crearne di propri in questo modo:

```
class MiaEccezione extends Exception { }
```

oppure estendendo una qualunque sottoclasse di `Exception`.

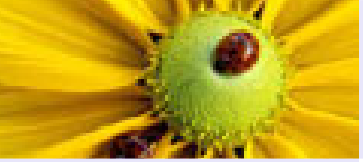


Nuove eccezioni

- È possibile usare tipi di eccezioni già presenti nelle Java API, oppure crearne di propri in questo modo:

```
class MiaEccezione extends Exception { }
```

  oppure estendendo una qualunque sottoclasse di `Exception`.
- Da questo momento in poi si può lanciare un oggetto del tipo (checked) `MiaEccezione`.



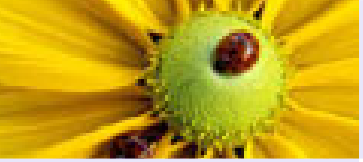
Nuove eccezioni

- È possibile usare tipi di eccezioni già presenti nelle Java API, oppure crearne di propri in questo modo:

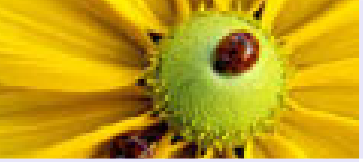
```
class MiaEccezione extends Exception { }
```

  oppure estendendo una qualunque sottoclasse di `Exception`.
- Da questo momento in poi si può lanciare un oggetto del tipo (checked) `MiaEccezione`.
- Pertanto, il codice seguente non compila:

```
class TestEx {
  void f() {
    throw new MiaEccezione();
  }
}
```

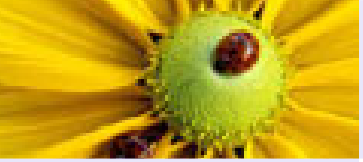


Regole di overriding



Overriding

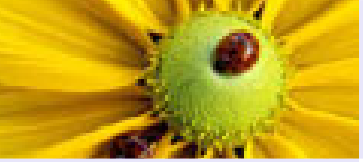
Posto che sono sovrapponibili solo i metodi visibili della superclasse, ecco finalmente le **regole complete per l'overriding di metodi**:



Overriding

Posto che sono sovrapponibili solo i metodi visibili della superclasse, ecco finalmente le **regole complete per l'overriding di metodi**:

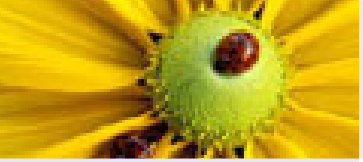
- I due metodi devono avere identica segnatura (nome del metodo e tipo dei parametri formali)



Overriding

Posto che sono sovrapponibili solo i metodi visibili della superclasse, ecco finalmente le **regole complete per l'overriding di metodi**:

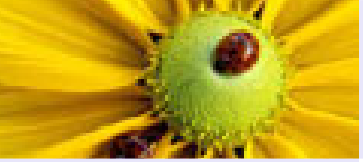
- I due metodi devono avere identica segnatura (nome del metodo e tipo dei parametri formali)
- Il tipo di ritorno nella sottoclasse può essere uguale o un sottotipo del tipo di ritorno originario



Overriding

Posto che sono sovrapponibili solo i metodi visibili della superclasse, ecco finalmente le **regole complete per l'overriding di metodi**:

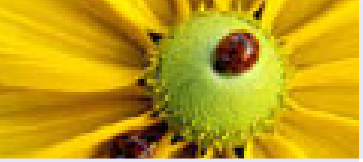
- I due metodi devono avere identica segnatura (nome del metodo e tipo dei parametri formali)
- Il tipo di ritorno nella sottoclasse può essere uguale o un sottotipo del tipo di ritorno originario
- Non si può marcare `static` uno solo dei metodi



Overriding

Posto che sono sovrapponibili solo i metodi visibili della superclasse, ecco finalmente le **regole complete per l'overriding di metodi**:

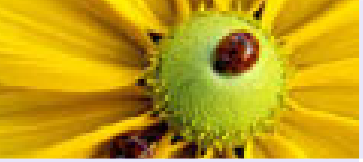
- I due metodi devono avere identica segnatura (nome del metodo e tipo dei parametri formali)
- Il tipo di ritorno nella sottoclasse può essere uguale o un sottotipo del tipo di ritorno originario
- Non si può marcare `static` uno solo dei metodi
- Il metodo nella superclasse non può essere marcato `final`



Overriding

Posto che sono sovrapponibili solo i metodi visibili della superclasse, ecco finalmente le **regole complete per l'overriding di metodi**:

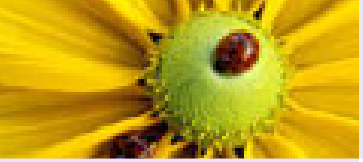
- I due metodi devono avere identica segnatura (nome del metodo e tipo dei parametri formali)
- Il tipo di ritorno nella sottoclasse può essere uguale o un sottotipo del tipo di ritorno originario
- Non si può marcare `static` uno solo dei metodi
- Il metodo nella superclasse non può essere marcato `final`
- Il metodo nella sottoclasse deve avere visibilità non inferiore a quello della superclasse



Overriding

Posto che sono sovrapponibili solo i metodi visibili della superclasse, ecco finalmente le **regole complete per l'overriding di metodi**:

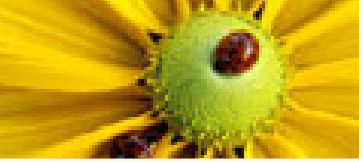
- I due metodi devono avere identica segnatura (nome del metodo e tipo dei parametri formali)
- Il tipo di ritorno nella sottoclasse può essere uguale o un sottotipo del tipo di ritorno originario
- Non si può marcare `static` uno solo dei metodi
- Il metodo nella superclasse non può essere marcato `final`
- Il metodo nella sottoclasse deve avere visibilità non inferiore a quello della superclasse
- Se il metodo nella sottoclasse dichiara di lanciare un tipo di eccezione *checked*, tale tipo deve essere un sottotipo di una delle eccezioni dichiarate dal metodo della superclasse



Overriding

Posto che sono sovrapponibili solo i metodi visibili della superclasse, ecco finalmente le **regole complete per l'overriding di metodi**:

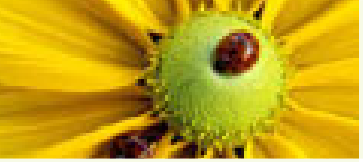
- I due metodi devono avere identica segnatura (nome del metodo e tipo dei parametri formali)
- Il tipo di ritorno nella sottoclasse può essere uguale o un sottotipo del tipo di ritorno originario
- Non si può marcare `static` uno solo dei metodi
- Il metodo nella superclasse non può essere marcato `final`
- Il metodo nella sottoclasse deve avere visibilità non inferiore a quello della superclasse
- Se il metodo nella sottoclasse dichiara di lanciare un tipo di eccezione *checked*, tale tipo deve essere un sottotipo di una delle eccezioni dichiarate dal metodo della superclasse
  Cioè, le EVENTUALI eccezioni *checked* dichiarate dal metodo nella sottoclasse, DEVONO essere tipi posti al di sotto nella gerarchia delle eccezioni dichiarate dal metodo nella superclasse.



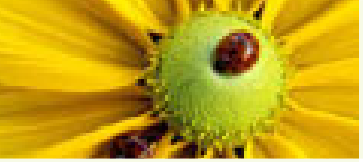
Esercizi

Sono proposti in allegato due esercizi:

1. In questo esercizio si dovranno usare i blocchi `try-catch` per gestire una semplice `RuntimeException`.
2. In questo esercizio si dovrà creare il nuovo tipo `OverdraftException` di oggetti lanciabili dal metodo `preleva` nella classe `Conto`.



Questionario



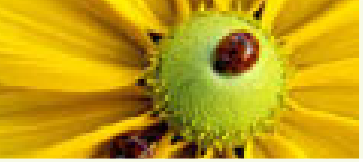
D 1

Dato il codice seguente:

```
 1. try {
 2.   // codice rischioso; hp:
 3.   // Exception
 4.   //   +--- EccA
 5.   //          +--- EccB
 6    //          +--- EccC
 7.   System.out.print(1);
 8. }
 9. catch (EccB e) {
10.   System.out.println(2);
11. }
12. catch (EccC e) {
13.   System.out.println(3);
14. }
15. catch (Exception e) {
16.   System.out.println(4);
17. }
18. finally {
19.   System.out.println(5);
20. }
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo EccB?

A. 1
B. 2
C. 3
D. 4
E. 5
F. 6



D 1

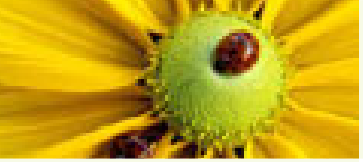
Dato il codice seguente:

```
 1. try {
 2.   // codice rischioso; hp:
 3.   // Exception
 4.   //   +--- EccA
 5.   //          +--- EccB
 6    //          +--- EccC
 7.   System.out.print(1);
 8. }
 9. catch (EccB e) {
10.   System.out.println(2);
11. }
12. catch (EccC e) {
13.   System.out.println(3);
14. }
15. catch (Exception e) {
16.   System.out.println(4);
17. }
18. finally {
19.   System.out.println(5);
20. }
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo `EccB`?

A. 1
B. 2
C. 3
D. 4
E. 5
F. 6

B. E. F. – Viene catturata l'eccezione, eseguito il blocco `finally`, infine l'esecuzione continua normalmente.



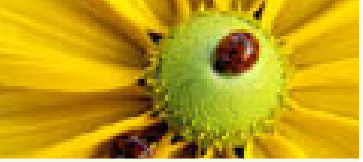
D 2

Dato il codice seguente:

```
 1. try {
 2.   // codice rischioso; hp:
 3.   // Exception
 4.   //   +--- EccA
 5.   //          +--- EccB
 6    //          +--- EccC
 7.   System.out.print(1);
 8. }
 9. catch (EccB e) {
10.   System.out.println(2);
11. }
12. catch (EccC e) {
13.   System.out.println(3);
14. }
15. catch (Exception e) {
16.   System.out.println(4);
17. }
18. finally {
19.   System.out.println(5);
20. }
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 non venga lanciata alcuna eccezione?

A. 1
B. 2
C. 3
D. 4
E. 5
F. 6



D 2

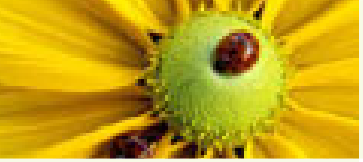
Dato il codice seguente:

```
 1. try {
 2.   // codice rischioso; hp:
 3.   // Exception
 4.   //   +--- EccA
 5.   //          +--- EccB
 6    //          +--- EccC
 7.   System.out.print(1);
 8. }
 9. catch (EccB e) {
10.   System.out.println(2);
11. }
12. catch (EccC e) {
13.   System.out.println(3);
14. }
15. catch (Exception e) {
16.   System.out.println(4);
17. }
18. finally {
19.   System.out.println(5);
20. }
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 non venga lanciata alcuna eccezione?

A. 1
B. 2
C. 3
D. 4
E. 5
F. 6

A. E. F. – Viene completato il blocco `try`, eseguito il blocco `finally`, infine l'esecuzione continua normalmente.



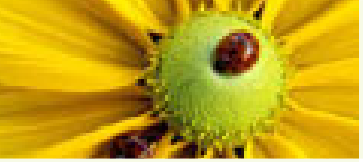
D 3

Dato il codice seguente:

```
 1. try {
 2.   // codice rischioso; hp:
 3.   // Exception
 4.   //   +--- EccA
 5.   //          +--- EccB
 6    //          +--- EccC
 7.   System.out.print(1);
 8. }
 9. catch (EccB e) {
10.   System.out.println(2);
11. }
12. catch (EccC e) {
13.   System.out.println(3);
14. }
15. catch (Exception e) {
16.   System.out.println(4);
17. }
18. finally {
19.   System.out.println(5);
20. }
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo EccC?

A. 1
B. 2
C. 3
D. 4
E. 5
F. 6



D 3

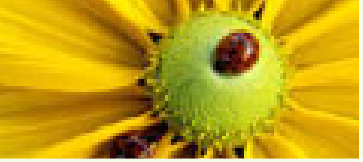
Dato il codice seguente:

```
 1. try {
 2.   // codice rischioso; hp:
 3.   // Exception
 4.   //   +--- EccA
 5.   //          +--- EccB
 6    //          +--- EccC
 7.   System.out.print(1);
 8. }
 9. catch (EccB e) {
10.   System.out.println(2);
11. }
12. catch (EccC e) {
13.   System.out.println(3);
14. }
15. catch (Exception e) {
16.   System.out.println(4);
17. }
18. finally {
19.   System.out.println(5);
20. }
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo `EccC`?

A. 1
B. 2
C. 3
D. 4
E. 5
F. 6

C. E. F. – Viene catturata l'eccezione, eseguito il blocco `finally`, infine l'esecuzione continua normalmente.



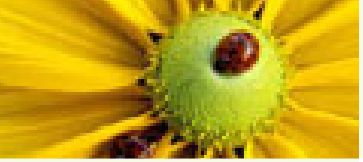
D 4

Dato il codice seguente:

```
 1. try {
 2.   // codice rischioso; hp:
 3.   // Exception
 4.   //   +--- EccA
 5.   //          +--- EccB
 6    //          +--- EccC
 7.   System.out.print(1);
 8. }
 9. catch (EccB e) {
10.   System.out.println(2);
11. }
12. catch (EccC e) {
13.   System.out.println(3);
14. }
15. catch (Exception e) {
16.   System.out.println(4);
17. }
18. finally {
19.   System.out.println(5);
20. }
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo `EccA`?

A. 1
B. 2
C. 3
D. 4
E. 5
F. 6



D 4

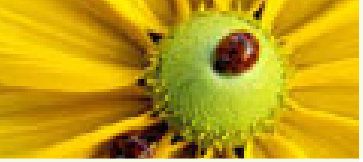
Dato il codice seguente:

```
 1. try {
 2.   // codice rischioso; hp:
 3.   // Exception
 4.   //   +--- EccA
 5.   //          +--- EccB
 6    //          +--- EccC
 7.   System.out.print(1);
 8. }
 9. catch (EccB e) {
10.   System.out.println(2);
11. }
12. catch (EccC e) {
13.   System.out.println(3);
14. }
15. catch (Exception e) {
16.   System.out.println(4);
17. }
18. finally {
19.   System.out.println(5);
20. }
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo `EccA`?

A. 1
B. 2
C. 3
D. 4
E. 5
F. 6

D. E. F. – Viene catturata l'eccezione, eseguito il blocco `finally`, infine l'esecuzione continua normalmente.



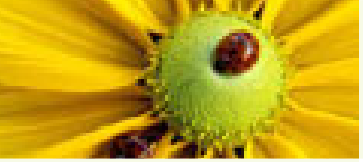
D 5

Dato il codice seguente:

```
 1. try {
 2.   // codice rischioso; hp:
 3.   // Exception
 4.   //   +--- EccA
 5.   //          +--- EccB
 6    //          +--- EccC
 7.   System.out.print(1);
 8. }
 9. catch (EccB e) {
10.   System.out.println(2);
11. }
12. catch (EccC e) {
13.   System.out.println(3);
14. }
15. catch (Exception e) {
16.   System.out.println(4);
17. }
18. finally {
19.   System.out.println(5);
20. }
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo `Exception`?

A. 1
B. 2
C. 3
D. 4
E. 5
F. 6



D 5

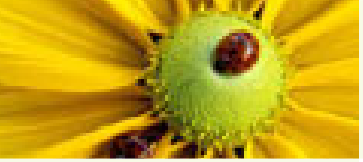
Dato il codice seguente:

```
 1. try {
 2.   // codice rischioso; hp:
 3.   // Exception
 4.   //   +--- EccA
 5.   //          +--- EccB
 6    //          +--- EccC
 7.   System.out.print(1);
 8. }
 9. catch (EccB e) {
10.   System.out.println(2);
11. }
12. catch (EccC e) {
13.   System.out.println(3);
14. }
15. catch (Exception e) {
16.   System.out.println(4);
17. }
18. finally {
19.   System.out.println(5);
20. }
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo `Exception`?

A. 1
B. 2
C. 3
D. 4
E. 5
F. 6

D. E. F. – Viene catturata l'eccezione, eseguito il blocco `finally`, infine l'esecuzione continua normalmente.



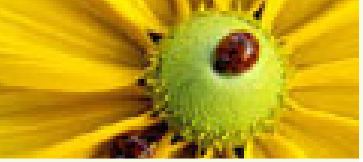
D 6

Dato il codice seguente:

```
 1. try {
 2.   // codice rischioso; hp:
 3.   // Exception
 4.   //   +--- EccA
 5.   //          +--- EccB
 6    //          +--- EccC
 7.   System.out.print(1);
 8. }
 9. catch (EccB e) {
10.   System.out.println(2);
11. }
12. catch (EccC e) {
13.   System.out.println(3);
14. }
15. catch (Exception e) {
16.   System.out.println(4);
17. }
18. finally {
19.   System.out.println(5);
20. }
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo `RuntimeException`?

A. 1
B. 2
C. 3
D. 4
E. 5
F. 6



D 6

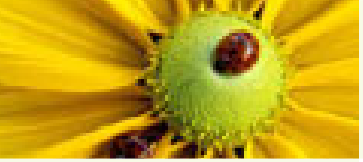
Dato il codice seguente:

```
 1. try {
 2.   // codice rischioso; hp:
 3.   // Exception
 4.   //   +--- EccA
 5.   //          +--- EccB
 6    //          +--- EccC
 7.   System.out.print(1);
 8. }
 9. catch (EccB e) {
10.   System.out.println(2);
11. }
12. catch (EccC e) {
13.   System.out.println(3);
14. }
15. catch (Exception e) {
16.   System.out.println(4);
17. }
18. finally {
19.   System.out.println(5);
20. }
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo `RuntimeException`?

A. 1
B. 2
C. 3
D. 4
E. 5
F. 6

D. E. F. – Viene catturata l'eccezione, eseguito il blocco `finally`, infine l'esecuzione continua normalmente.



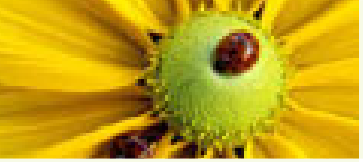
D 7

Dato il codice seguente:

```
 1. try {
 2.   // codice rischioso; hp:
 3.   // Exception
 4.   //   +--- EccA
 5.   //          +--- EccB
 6    //          +--- EccC
 7.   System.out.print(1);
 8. }
 9. catch (EccB e) {
10.   System.out.println(2);
11. }
12. catch (EccC e) {
13.   System.out.println(3);
14. }
15. catch (Exception e) {
16.   System.out.println(4);
17. }
18. finally {
19.   System.out.println(5);
20. }
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo `Error`?

A. 1
B. 2
C. 3
D. 4
E. 5
F. 6



D 7

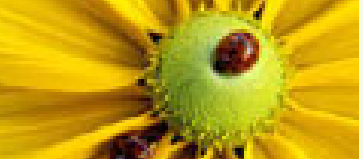
Dato il codice seguente:

```
 1. try {
 2.   // codice rischioso; hp:
 3.   // Exception
 4.   //   +--- EccA
 5.   //          +--- EccB
 6    //          +--- EccC
 7.   System.out.print(1);
 8. }
 9. catch (EccB e) {
10.   System.out.println(2);
11. }
12. catch (EccC e) {
13.   System.out.println(3);
14. }
15. catch (Exception e) {
16.   System.out.println(4);
17. }
18. finally {
19.   System.out.println(5);
20. }
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo `Error`?

A. 1
B. 2
C. 3
D. 4
E. 5
F. 6

E. – Non viene catturata l'eccezione, viene eseguito il blocco `finally`, l'eccezione propagata.

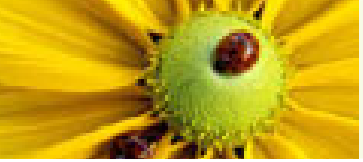


D 8

```
public class M {
  public static void
      main(String[] args) {
    int k=0;
    try {
      int i=5/k;
    }
    catch (ArithmeticException e) {
      System.out.print(1);
    }
    catch (RuntimeException e) {
      System.out.print(2);
      return;
    }
    catch (Exception e) {
      System.out.print(3);
    }
    finally {
      System.out.print(4);
    }
    System.out.print(5);
  }
}
```

Qual è l'output del programma precedente?

A. 5
B. 14
C. 124
D. 145
E. 1245
F. 35



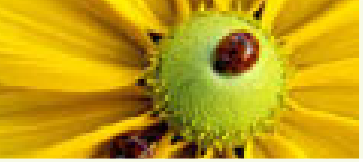
D 8

```
public class M {
  public static void
      main(String[] args) {
    int k=0;
    try {
      int i=5/k;
    }
    catch (ArithmeticException e) {
      System.out.print(1);
    }
    catch (RuntimeException e) {
      System.out.print(2);
      return;
    }
    catch (Exception e) {
      System.out.print(3);
    }
    finally {
      System.out.print(4);
    }
    System.out.print(5);
  }
}
```

Qual è l'output del programma precedente?

A. 5
B. 14
C. 124
D. 145
E. 1245
F. 35

D.



D 9


```
public class Eccezioni {
  public static void main(String[] args) {
    try {
      if (args.length == 0) return;
      System.out.println(args[0]);
    }
    finally {
      System.out.println("Fine");
    }
  }
}
```

Quali affermazioni riguardanti il precedente programma sono vere?

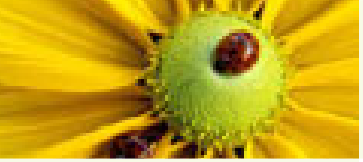
A. Se eseguito senza argomenti, il programma non produce output.

B. Se eseguito senza argomenti, il programma stampa `Fine`.

C. Il programma lancia un `ArrayIndexOutOfBoundsException`.

D. Se eseguito con un argomento, il programma stampa solo l'argomento dato.

E. Se eseguito con un argomento, il programma stampa l'argomento dato seguito da `Fine`.



D 9


```
public class Eccezioni {
  public static void main(String[] args) {
    try {
      if (args.length == 0) return;
      System.out.println(args[0]);
    }
    finally {
      System.out.println("Fine");
    }
  }
}
```

Quali affermazioni riguardanti il precedente programma sono vere?

A. Se eseguito senza argomenti, il programma non produce output.

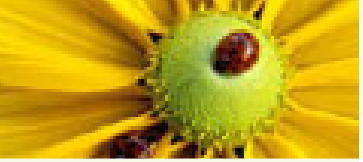
B. Se eseguito senza argomenti, il programma stampa `Fine`.

C. Il programma lancia un `ArrayIndexOutOfBoundsException`.

D. Se eseguito con un argomento, il programma stampa solo l'argomento dato.

E. Se eseguito con un argomento, il programma stampa l'argomento dato seguito da `Fine`.

B. E.



D 10

Qual è l'output del seguente programma?

```
public class MyClass {
  public static void main(String[] args) {
    RuntimeException re = null;
    throw re;
  }
}
```

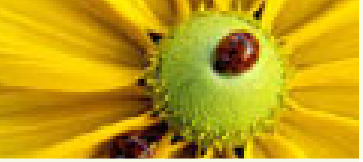
A. Il codice non viene compilato, poiché il `main` non dichiara che lancia una `RuntimeException`.

B. Il codice non viene compilato, poiché non può rilanciare `re`.

C. Il programma viene compilato e lancia `java.lang.RuntimeException` in esecuzione.

D. Il programma viene compilato e lancia `java.lang.NullPointerException` in esecuzione.

E. Il programma viene compilato, eseguito e termina senza produrre alcun output.



D 10

Qual è l'output del seguente programma?

```
public class MyClass {
  public static void main(String[] args) {
    RuntimeException re = null;
    throw re;
  }
}
```

A. Il codice non viene compilato, poiché il `main` non dichiara che lancia una `RuntimeException`.

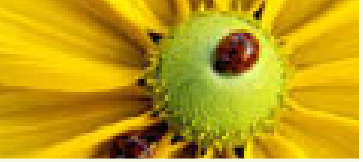
B. Il codice non viene compilato, poiché non può rilanciare `re`.

C. Il programma viene compilato e lancia `java.lang.RuntimeException` in esecuzione.

D. Il programma viene compilato e lancia `java.lang.NullPointerException` in esecuzione.

E. Il programma viene compilato, eseguito e termina senza produrre alcun output.

D.



D 11

Quali di queste affermazioni sono vere?

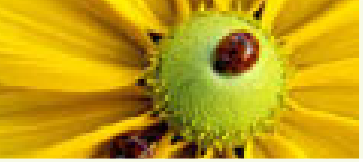
A. Se una eccezione non è catturata in un metodo, il metodo termina e viene ripresa la successiva normale esecuzione.

B. Un metodo sovrapposto in una sottoclasse deve dichiarare che lancia lo stesso tipo di eccezione del metodo che sovrappone.

C. Il `main` può dichiarare che lancia eccezioni `checked`.

D. Un metodo che dichiara di lanciare una certo tipo di eccezione, può lanciare una istanza di una qualunque sottoclasse di quel tipo.

E. Il blocco `finally` è eseguito se e solo se viene lanciata una eccezione all'interno del corrispondente blocco `try`.



D 11

Quali di queste affermazioni sono vere?

A. Se una eccezione non è catturata in un metodo, il metodo termina e viene ripresa la successiva normale esecuzione.

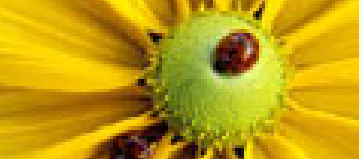
B. Un metodo sovrapposto in una sottoclasse deve dichiarare che lancia lo stesso tipo di eccezione del metodo che sovrappone.

C. Il `main` può dichiarare che lancia eccezioni `checked`.

D. Un metodo che dichiara di lanciare una certo tipo di eccezione, può lanciare una istanza di una qualunque sottoclasse di quel tipo.

E. Il blocco `finally` è eseguito se e solo se viene lanciata una eccezione all'interno del corrispondente blocco `try`.

C. D.

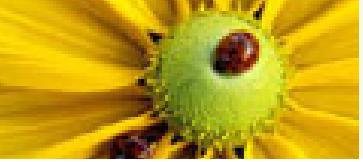


D 12

```
public class MiaClasse {
  public static void main
           (String[] args} {
    try {
      f();
    }
    catch (MiaEcc e) {
      System.out.print(1);
      throw new RuntimeException();
    }
    catch (RuntimeException e) {
      System.out.print(2);
      return;
    }
    catch (Exception e) {
      System.out.print(3);
    }
    finally {
      System.out.print(4);
    }
    System.out.print(5);
  }
  // MiaEcc e'
  // sottoclasse di Exception
  static void f() throws MiaEcc {
    throw new MiaEcc();
  }
}
```

Qual è l'output del programma precedente?

A. 5
B. 14
C. 124
D. 145
E. 1245
F. 35



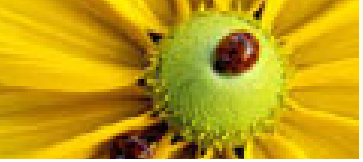
D 12

```
public class MiaClasse {
  public static void main
           (String[] args} {
    try {
      f();
    }
    catch (MiaEcc e) {
      System.out.print(1);
      throw new RuntimeException();
    }
    catch (RuntimeException e) {
      System.out.print(2);
      return;
    }
    catch (Exception e) {
      System.out.print(3);
    }
    finally {
      System.out.print(4);
    }
    System.out.print(5);
  }
  // MiaEcc e'
  // sottoclasse di Exception
  static void f() throws MiaEcc {
    throw new MiaEcc();
  }
}
```

Qual è l'output del programma precedente?

A. 5
B. 14
C. 124
D. 145
E. 1245
F. 35

B.



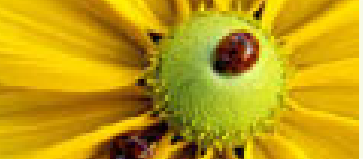
D 13

```
public class MiaClasse {
  public static void main
              (String[] args)
                throws MiaEcc {
    try {
      f();
      System.out.print(1);
    }
    finally {
      System.out.print(2);
    }
    System.out.print(3);
  }

  // MiaEcc e'
  // sottoclasse di Exception
  static void f() throws MiaEcc {
    throw new MiaEcc();
  }
}
```

Qual è l'output del programma precedente?

A. 2 e lancia `MiaEcc`.
B. 12
C. 123
D. 23
E. 32
F. 13



D 13

```
public class MiaClasse {
  public static void main
              (String[] args)
                throws MiaEcc {
    try {
      f();
      System.out.print(1);
    }
    finally {
      System.out.print(2);
    }
    System.out.print(3);
  }

  // MiaEcc e'
  // sottoclasse di Exception
  static void f() throws MiaEcc {
    throw new MiaEcc();
  }
}
```

Qual è l'output del programma precedente?

A. 2 e lancia `MiaEcc`.
B. 12
C. 123
D. 23
E. 32
F. 13

A.