Gestione di contenuti condivisi in una classe

Torniamo alle classi

- Vediamo ora due meccanismi che possono aiutarci a strutturare meglio il codice all'interno delle classi
  - Metodi ausiliari
    - Statici e non statici
  - Variabili statiche

Metodi ausiliari

I metodi ausiliari aiutano a **strutturare meglio il codice** svolgendo un sottoinsieme di istruzioni che possono venire invocate da altri metodi della classe, inclusi i metodi di esemplare

NB: in Java tutti i metodi devono essere dichiarati dentro una classe

Metodi statici o "di classe"

- Sono metodi che non vengono invocati con un oggetto come parametro implicito
- Es: i metodi della classe Math
  - I numeri non sono oggetti e non posso utilizzarli per invocare metodi
  - Math.sqrt(2);
    - Math non crea in oggetto ma dice solo dove si trova il metodo

Invocazione di metodi ausiliari statici

- Per invocare un **metodo statico** presente **nella stessa classe** non serve specificare il nome della classe
- Se il metodo statico si trova **in un'altra classe** devo far precedere il nome del metodo dal nome della classe:

```
public class A{

   public static void main(String args[]){
      B.faiQualcosa();
      stampa(); // metodo della classe A
   }
   public static void stampa(){…}
}
```

```
public class B{
   public static void faiQualcosa(){…}
}
```

Invocazione di metodi ausiliari non statici

- Anche ***un metodo di esemplare può invocare un altro metodo di esemplare della stessa classe*** senza specificare una variabile oggetto
- Esempio: associo ad ogni deposito una commissione (che di fatto è un prelievo!)

```
public void deposit(double amount)
{
   withdraw(2); //evito di duplicare il codice di withdraw
   balance += amount;
}
```

Invocazione di metodi ausiliari non statici

- Come per le variabili, viene usato implicitamente **this**, come se fosse

```
public void deposit(double amount)
{   this.withdraw(2);
    this.balance += amount;
}
```

- Il parametro implicito con cui è stato invocato **deposit** diventa "automaticamente" il parametro implicito con cui viene invocato **withdraw**

Metodi di esemplare ausiliari

- Aggiungiamo un contatore di operazioni effettuate

```
public class BankAccount
{  private double balance;
   private int numberOfOperations;

   // qui c'è il costruttore...
   public void deposit(double amount)
   {  balance = balance + amount;
      numberOfOperations++;
   }
   public void withdraw(double amount)
   {  balance = balance - amount;
      numberOfOperations++;
   }
   public double getBalance()
   {  return balance;
   }
   public int getNumberOfOperations()
   {  return numberOfOperations;
   }
}
```

Evitare codice duplicato

Se possibile, è sempre meglio evitare di scrivere blocchi di codice duplicati

```
public void deposit(double amount)
{   balance = balance + amount;
    numberOfOperations++;
}
public void withdraw(double amount)
{   balance = balance - amount;
    numberOfOperations++;
}
```

```
public void deposit(double amount)
{   balance = balance + amount;
    numberOfOperations++;
    balance -= 2.5;
}
public void withdraw(double amount)
{   balance = balance - amount;
    numberOfOperations++;
    balance -= 2.5;
}
// e se ci sono 20 tipi di operazioni?
```

Immaginiamo, infatti, di voler applicare una **commissione** a **ogni** operazione effettuata…

Meglio un metodo ausiliario

```
public void deposit(double amount)
{   balance = balance + amount;
    recordOperation();
}
public void withdraw(double
amount)
{   balance = balance - amount;
    recordOperation();
}

private void recordOperation()
{   numberOfOperations++;
    balance -= 2.5;
    ... // future modifiche

}
```

I metodi di esemplare **ausiliari** sono solitamente **privati**

```
// codice ESTERNO alla classe BankAccount
BankAccount account = new BankAccount();
...
account.recordOperation(); // insensato...
                           // e proibito!
```

Errori comuni e suggerimenti

- I metodi statici possono essere scritti anche in classi che generano oggetti
  - Errore: accedere da un metodo statico ad una variabile di esemplare
- Prima dell'OOP i programmi erano sostanzialmente collezioni di metodi statici
  - Si può fare anche adesso, e anche in Java ma…
  - Il risultato può essere un programma che difficilmente può evolvere senza dover riscrivere tutto in caso di aggiornamenti

Variabili statiche

Problema

- Vogliamo modificare **BankAccount** in modo che
  - il suo stato contenga anche un *numero di conto*

```
public class BankAccount
{  private int accountNumber;
   ...
   public int getAccountNumber()
   {  return accountNumber; //ispezione
   }
}
```

  - il numero di conto sia assegnato automaticamente, senza poter essere scelto da chi costruisce gli esemplari
    - ogni conto deve avere un numero diverso
    - i numeri assegnati devono essere progressivi, iniziando da 1

Soluzione

- **Prima idea** (che non funziona…)

  usiamo una ulteriore variabile per memorizzare l'ultimo numero di conto assegnato

```
public class BankAccount
{   ...
    private int accountNumber;
    private int lastAssignedNumber;
    ...
    public BankAccount()
    {   lastAssignedNumber++;
        accountNumber = lastAssignedNumber;
        balance = 0;
    }
}
```

Soluzione

```
public class BankAccount
{   ...
    private int accountNumber;
    private int lastAssignedNumber;
    ...
    public BankAccount()
    {   lastAssignedNumber++;
        accountNumber = lastAssignedNumber;
        balance = 0;
    }
}
```

- Questo costruttore non funziona perché la variabile **lastAssignedNumber** è una ***variabile di esemplare*** e, quindi, ne esiste una copia per ogni oggetto: il risultato è che tutti i conti creati hanno il numero di conto uguale a 1 !!

Variabili statiche

- Ci servirebbe una *variabile condivisa da tutti gli oggetti della classe*
  - una variabile con questa semantica si ottiene con la dichiarazione **static**

```
public class BankAccount
{   ...
    private static int lastAssignedNumber;
}
```

Variabili statiche

- Una variabile **static** (detta ***variabile di classe***) è **condivisa** da tutti gli oggetti della classe e **ne esiste un'unica copia**
  - Non si trova all'interno degli esemplari della classe, ma in una zona di memoria riservata ***alla classe***

Variabili statiche

- Ora il costruttore funziona

```
public class BankAccount
{   ...
    private int accountNumber;
    private static int lastAssignedNumber = 0;
    ...
    public BankAccount()
    {   lastAssignedNumber++;
        accountNumber = lastAssignedNumber;
        balance = 0;
    }
}
```

- Ogni metodo (o costruttore) di una classe può accedere alle variabili statiche della classe e modificarle

Variabili statiche

- **Osserviamo che le variabili statiche non dovrebbero (da un punto di vista logico) essere inizializzate nei costruttori**
  - il loro valore verrebbe inizializzato di nuovo ogni volta che si costruisce un oggetto, perdendo il vantaggio di avere una variabile condivisa!
- Bisogna inizializzarle quando si dichiarano

```
private static int lastAssignedNumber = 0;
```

  - Sappiamo che questa sintassi si può usare anche per le variabili di esemplare, anziché usare un costruttore, ma **non** è una buona pratica di programmazione

Variabili statiche

- Nella programmazione a oggetti, ***l'utilizzo di variabili statiche deve essere limitato***, perché
  - metodi che leggono variabili statiche e agiscono di conseguenza hanno un ***comportamento che non dipende soltanto dai loro parametri*** (implicito ed espliciti), quindi sono più esposti ai cosiddetti "effetti collaterali", cioè effetti più difficili da prevedere correttamente

Variabili statiche

- In ogni caso, le variabili statiche devono essere **private** come quelle di esemplare, per evitare accessi indesiderati
  - Se **lastAssignedNumber** fosse **public**, vi si potrebbe accedere (**in lettura o in scrittura**) anche da un metodo esterno alla classe usando la sintassi **BankAccount.lastAssignedNumber**, cioè usando il nome della classe

Variabili statiche

- È invece pratica comune (senza controindicazioni) usare ***costanti*** statiche, come nella classe **Math**

```
public class Math
{   ...
    public static final double PI =
                3.14159265358979323846;
}
```

- Tali costanti sono di norma **public** e per accedere al loro valore si usa il nome della classe seguito dal punto e dal nome della costante, **Math.PI**

Consigli per la progettazione di una Classe

Progettare la classe

- Stabilire quali sono **le caratteristiche essenziali** degli oggetti della classe, e fare **un elenco delle operazioni** che sarà possibile compiere su di essi: **processo di *astrazione***

- **Definire e scrivere l'*interfaccia pubblica*.** Ovvero, scrivere l'intestazione della classe, definire i costruttori ed i metodi pubblici da realizzare, e scrivere la *firma* (specificatore di accesso, tipo di valore restituito, nome del metodo, eventuali parametri espliciti) di ciascuno di essi.

Progettare la classe

- **Definire le *variabili di esemplare* ed eventuali *variabili statiche*.** E` necessario individuare tutte le variabili necessarie (quante? quali? dipende dalla classe!). Per ciascuna di esse si deve, poi, definire tipo e nome.

Consiglio

- Se un metodo non restituisce valori (ovvero il tipo del valore restituito è void), scrivete inizialmente un corpo *vuoto*, ovvero **{}**.
- Se un metodo restituisce valori non void, scrivete inizialmente un corpo fittizio contenente solo un enunciato di return:
  - **{return 0;}** per metodi che restituiscono valori numerici o char
  - **{return false;}** per metodi che restituiscono valori booleani
  - **{return null;}** per metodi che restituiscono riferimenti ad oggetti.

Consiglio

- Con questi accorgimenti il codice compilerà correttamente fin dall'inizio del vostro lavoro. Quando poi scriverete i metodi, modificherete le istruzioni di return secondo quanto richiesto da ciascun metodo.

Realizzare la classe

- Verificate le impostazioni del vostro editor di testi, al fine rendere più efficiente il vostro lavoro di programmazione. In particolare, verificate ed eventualmente modificate le impostazioni per i rientri di tabulazione (valori tipici sono di 3 o 4 caratteri), e visualizzate i numeri di riga.

- Scrivere il codice dei metodi.

Consiglio

- *Non* appena si è realizzato un metodo, si deve compilare e correggere gli errori di compilazione (se il corpo del metodo è particolarmente lungo e complicato, compilare anche prima di terminare il metodo).
- Non aspettate di aver codificato tutta la classe per compilare!
  - Altrimenti vi troverete, molto probabilmente, a dover affrontare un numero elevato di errori, con il rischio di non riuscire a venirne a capo in un tempo ragionevole (come quello a disposizione per la prova d'esame...).

Collaudare la classe

- Ovvero scrivere una *classe di collaudo* contenente un metodo main, all'interno del quale vengono definiti e manipolati oggetti appartenenti alla classe da collaudare.
  - E` possibile scrivere ciascuna classe in un file diverso. In tal caso, ciascun file avrà il nome della rispettiva classe ed avrà l'estensione .java. Tutti i file vanno tenuti nella stessa cartella, tutti i file vanno compilati separatamente, solo la classe di collaudo (contenente il metodo main) va eseguita.

Collaudare la classe

- E` possibile scrivere tutte le classi in un unico file. In tal caso, il file .java deve contenere una sola classe public. In particolare, la classe contenente il metodo main deve essere public mentre la classe (o le classi) da collaudare non deve essere public (non serve scrivere private, semplicemente non si indica l'attributo public). Il file .java deve avere il nome della classe public

22/11?

La gestione delle eccezioni

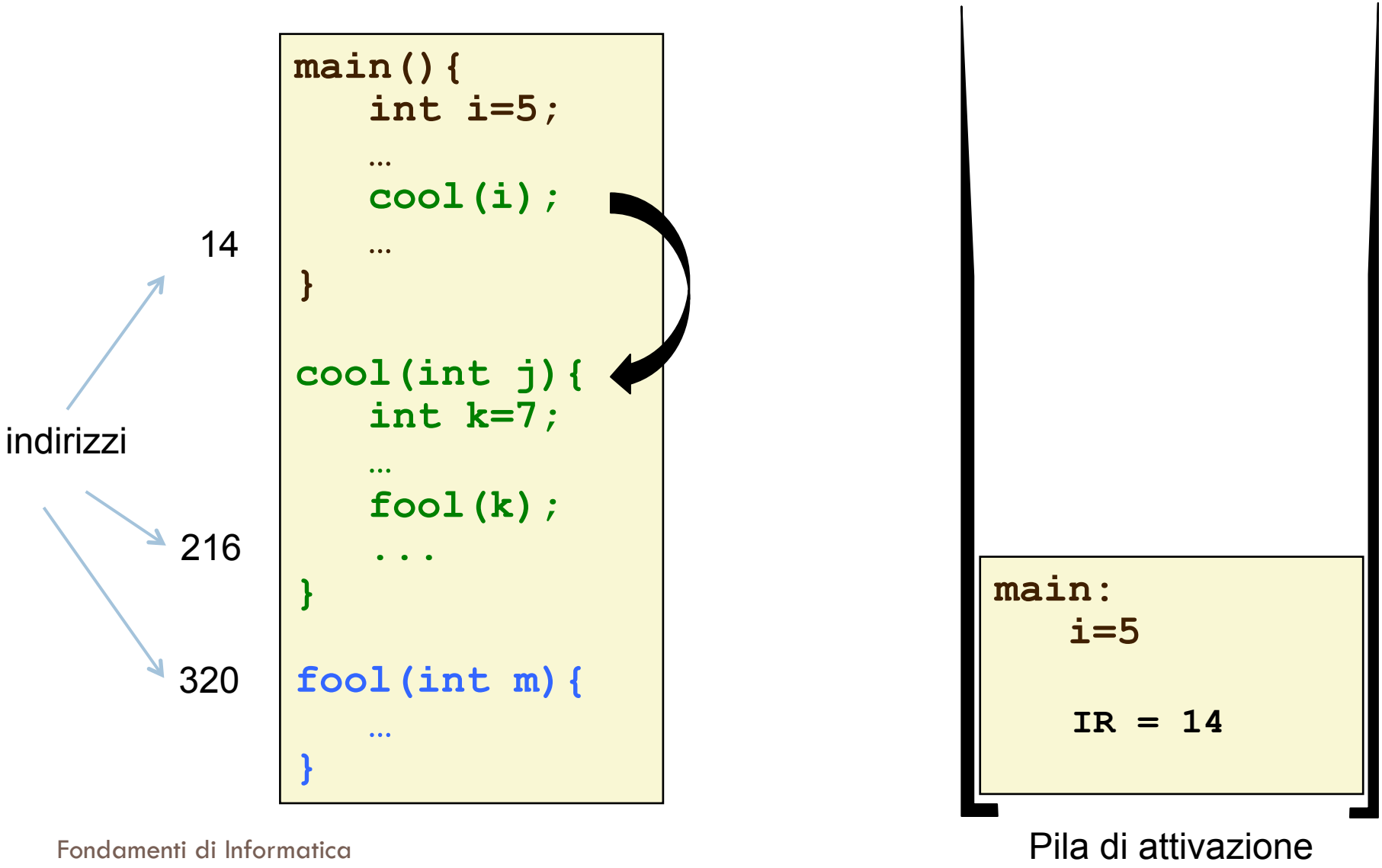
Record di attivazione

- Quando un metodo entra in esecuzione viene creata nella memoria Stack una zona riservata, chiamata record di attivazione, che contiene le informazioni di quel metodo:
  - Parametri formali
  - Variabili locali
  - Punto di ritorno

Pila di attivazione

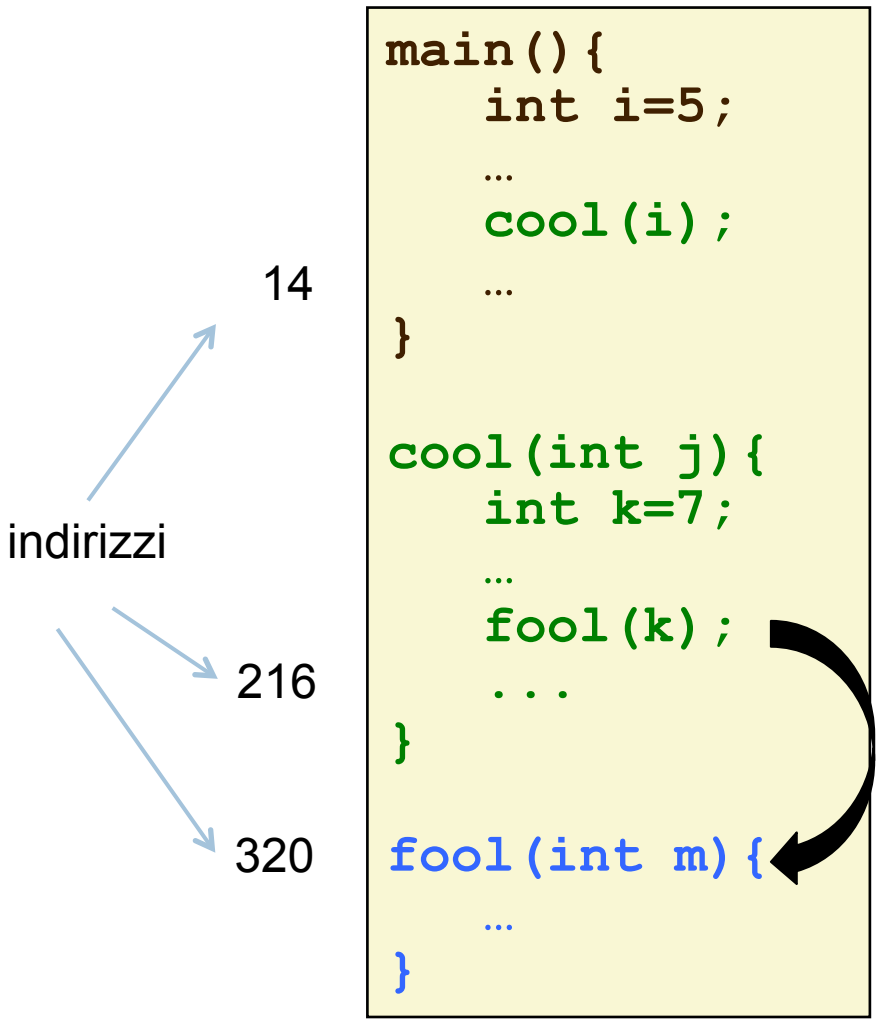
- La zona di memoria che contiene i record di attivazione dei metodi invocati si chiama Stack perché si comporta come una pila di piatti o di libri…
  - Quando un metodo in esecuzione invoca un altro metodo viene aggiunto in cima alla pila il record di attivazione del metodo invocato
  - Quando un metodo termina la sua esecuzione viene tolto dalla pila e si torna ad eseguire il metodo che l'aveva invocato (quindi quello immediatamente "sotto") dal punto in cui eravamo rimasti

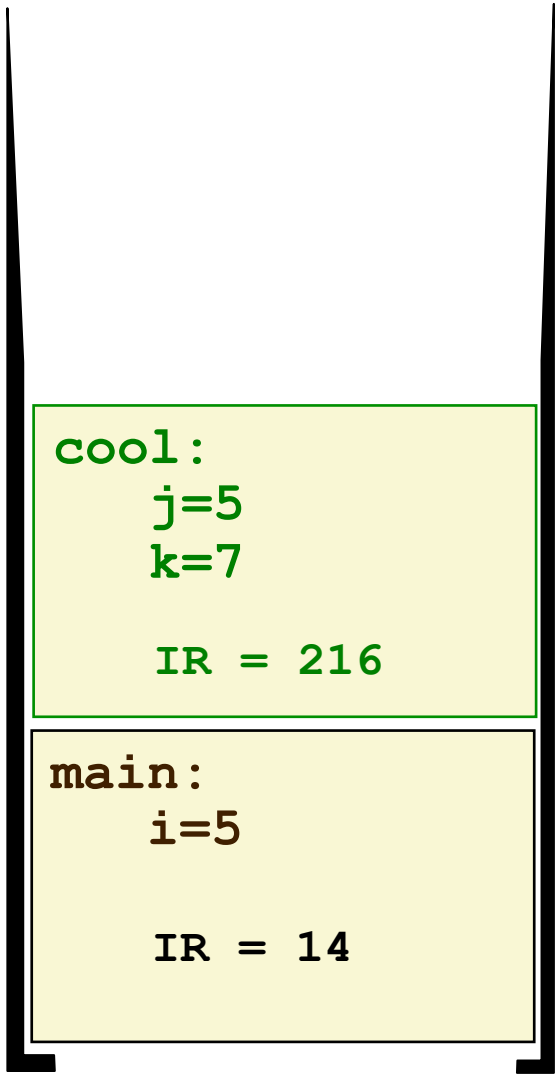
Eseguo main, invoco cool



Pila di attivazione

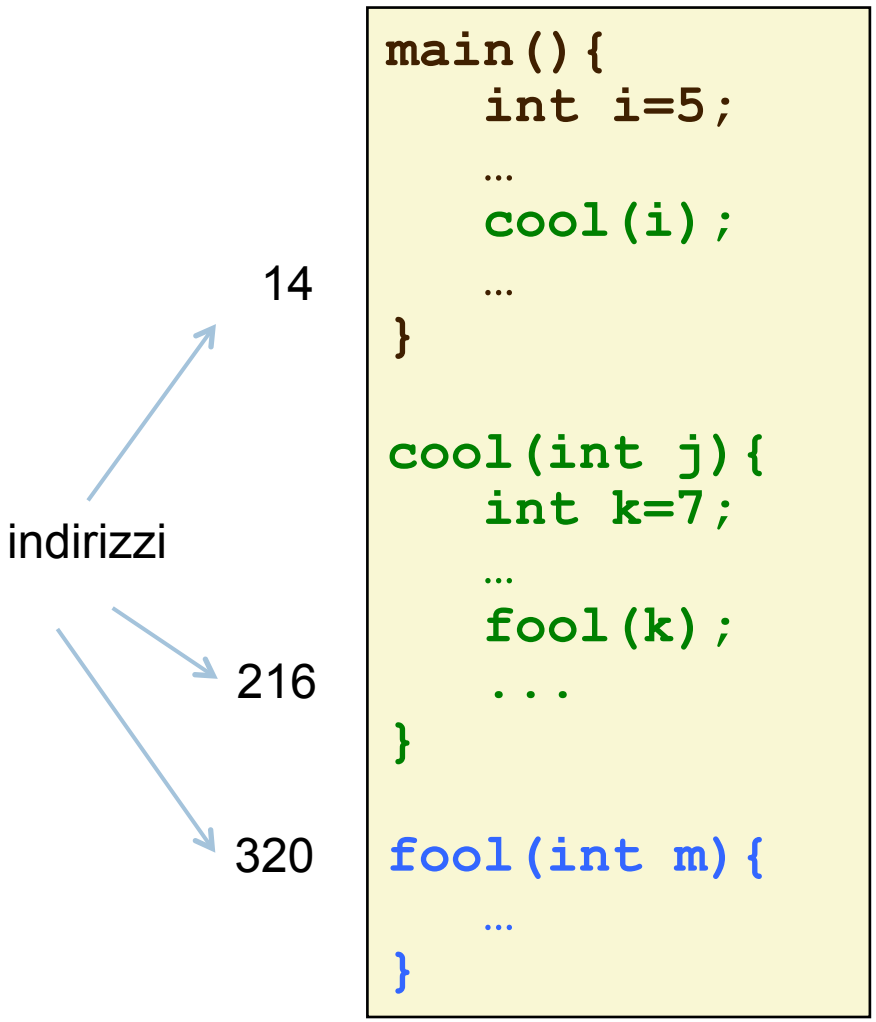
Eseguo cool, invoco fool

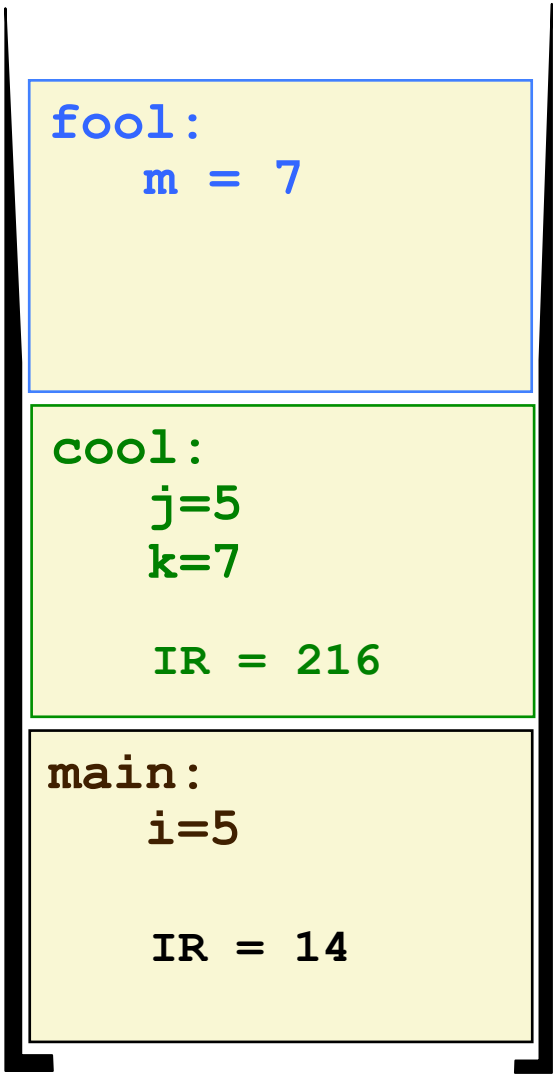




Pila di attivazione

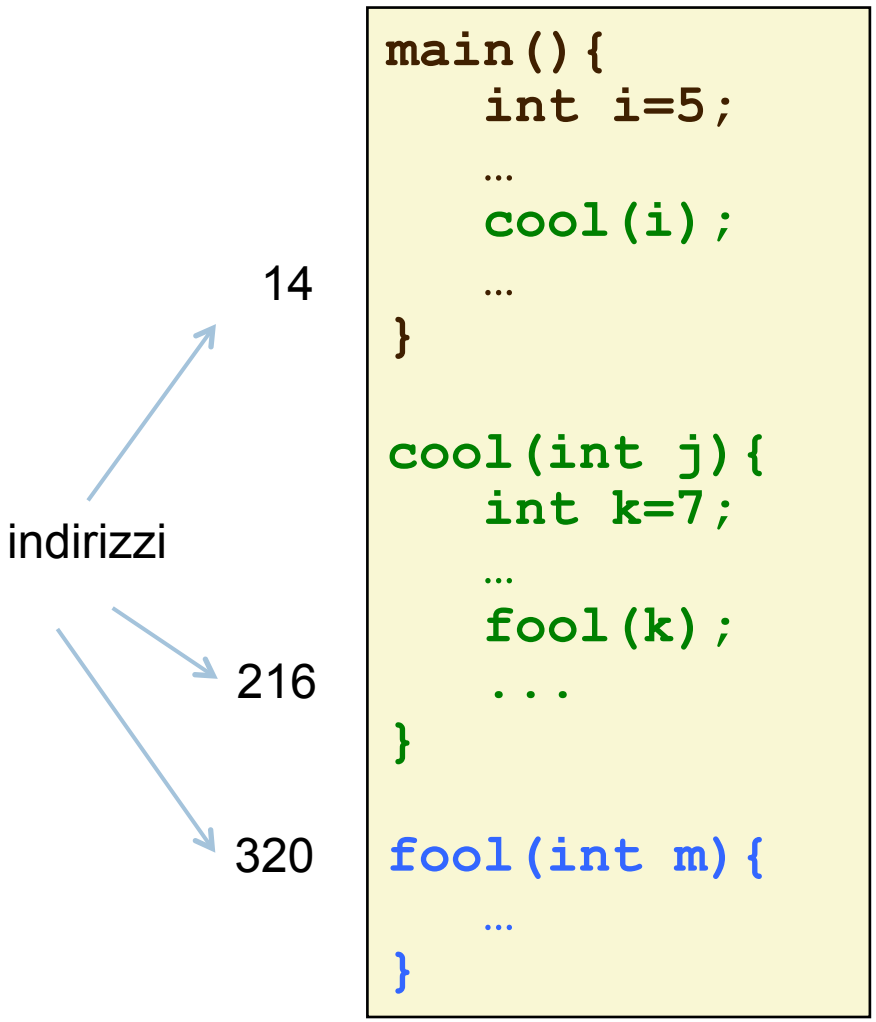
Eseguo fool

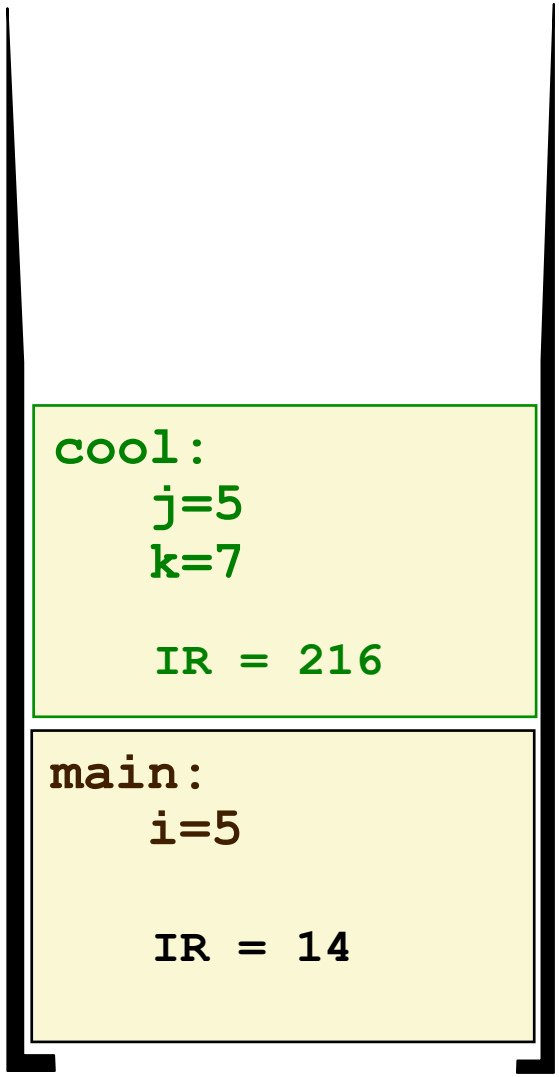




Pila di attivazione

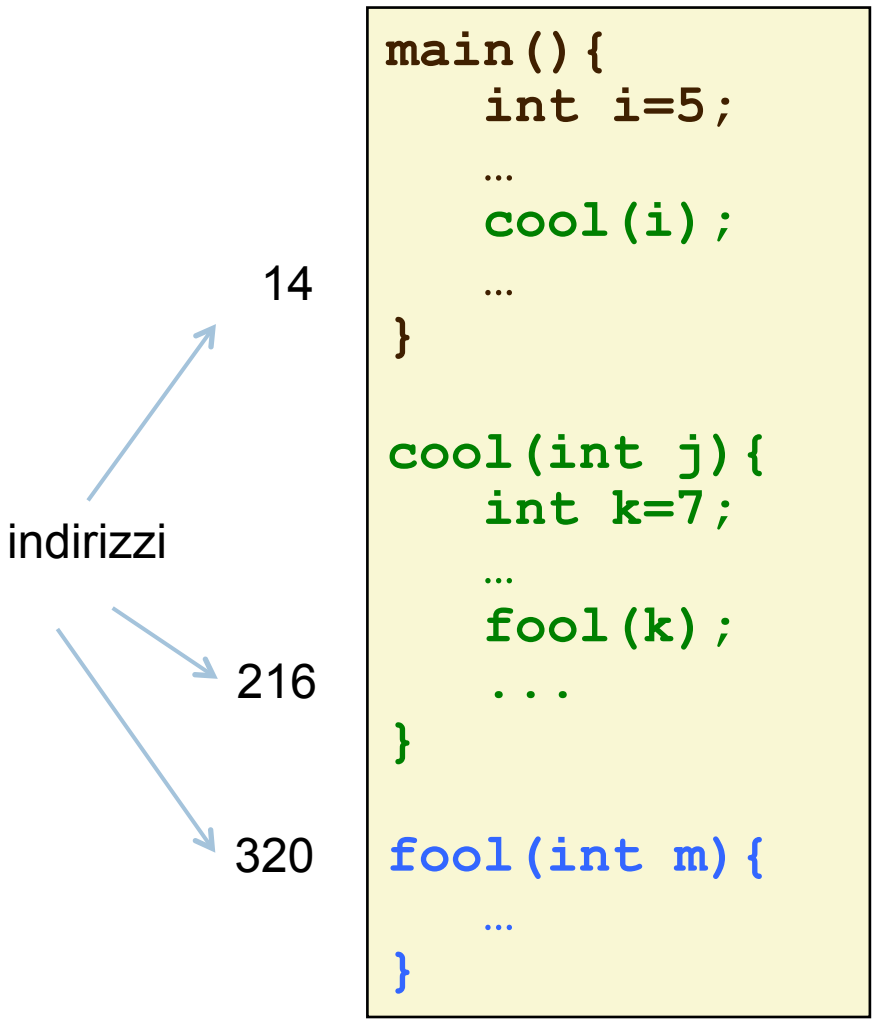
Termina fool, torno a cool

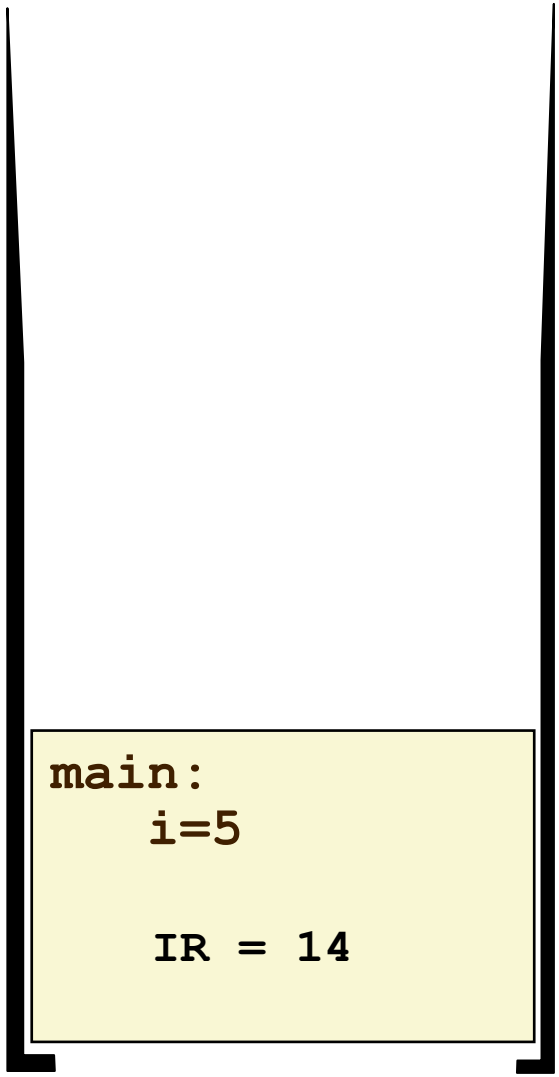




Pila di attivazione

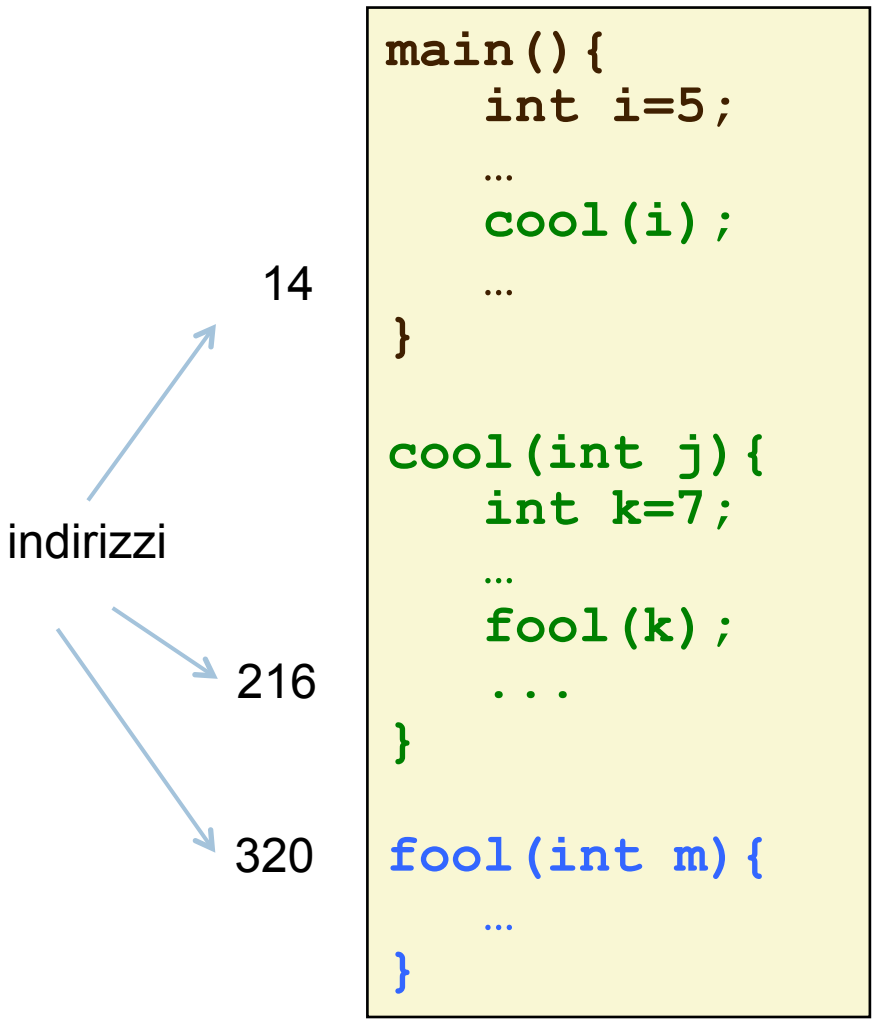
Termina cool, torno a main

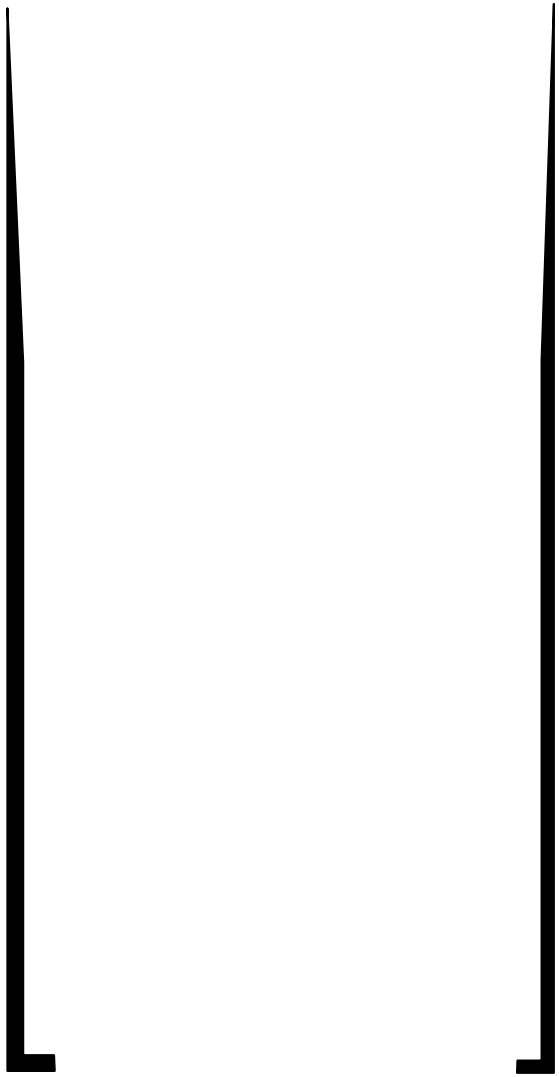




Pila di attivazione

Termina main, termina il programma





Pila di attivazione

Lancio delle eccezioni

- Il meccanismo generale di segnalazione di errori (o di condizioni di funzionamento anomale) in Java consiste nel "far lanciare" (throw) un'**eccezione** al metodo durante la cui esecuzione si è verificato il malfunzionamento
- si dice anche che il metodo solleva o genera un'eccezione

Le eccezioni in Java

- Quando un metodo ***lancia*** un'eccezione…
  - l'esecuzione del metodo viene immediatamente interrotta
  - **l'eccezione viene "propagata" al metodo chiamante, che si "risveglia" (come se il metodo chiamato fosse terminato in modo "normale") ma viene a sua volta subito interrotto per la presenza dell'eccezione**

Le eccezioni in Java

- Quando un metodo ***lancia*** un'eccezione…
  - l'eccezione viene via via propagata fino al metodo **main**, che si "risveglia" ma viene a sua volta subito interrotto
    - L'interruzione del metodo **main** provoca l'arresto **anormale** del programma con la segnalazione, da parte dell'interprete, dell'eccezione che è stata la causa di tale terminazione prematura

Le eccezioni in Java

- Il lancio di un'eccezione è quindi un modo per terminare un programma in caso di errore
  - ***non sempre, però, gli errori sono così gravi...***

Gestire le eccezioni di input

```
String line = console.nextLine();
int n = Integer.parseInt(line);
```

- In questo esempio di conversione di stringhe in numeri, supponiamo che la stringa sia stata introdotta dall'utente
  - Se la stringa **non** contiene un numero valido, il metodo `parseInt` lancia un'eccezione di tipo **NumberFormatException**

Gestire le eccezioni di input

```
String line = console.nextLine();
int n = Integer.parseInt(line);
```

- Sarebbe interessante poter ***gestire*** tale eccezione, segnalando l'errore all'utente e chiedendo di inserire nuovamente il dato numerico, ***anziché terminare*** prematuramente il programma
  - Possiamo ***intercettare*** l'eccezione e ***gestirla*** mediante il costrutto sintattico **try/catch**

Eccezioni nei formati numerici

```
int n = 0;
boolean done = false;
do
{   try
    {   System.out.println("Inserire un valore intero");
        String line = console.nextLine();
        n = Integer.parseInt(line);
        // l'assegnazione seguente viene
        // eseguita soltanto se NON viene
        // lanciata l'eccezione da parseInt
        done = true;
    }
    catch(NumberFormatException e)
    {   System.out.println("Riprova");
        // done rimane false
    }
} while (!done);
```

Attenzione: in questo esempio il blocco **try** è all'interno di un ciclo, ma è solo un esempio!
Il blocco **try/catch** è un enunciato (composto), quindi può stare… ovunque.

Gestire le eccezioni

- L'enunciato che contiene l'invocazione del metodo che **può** generare l'eccezione deve essere racchiuso tra parentesi graffe e preceduto dalla parola chiave **try**

```
try
{   ...
    n = Integer.parseInt(line);
    ...
}
```

- Bisogna poi sapere ***di che tipo* è** l'eccezione eventualmente generata dal metodo
  - nel nostro caso **NumberFormatException**

Gestire le eccezioni

- Il ***blocco try*** è seguito da una ***clausola catch***, seguita da una coppia di parentesi tonde contenenti il ***tipo dell'eccezione*** che si vuole gestire e una variabile (di solito **e** o **ex** o **exc**) che conterrà l'eccezione stessa

```
catch (NumberFormatException e)
{   System.out.println("Riprova");
}
```

- Nel blocco di enunciati che segue ***catch*** si trova ***il codice che deve essere eseguito nel caso in cui si verifichi l'eccezione***
  - l'esecuzione del blocco **try** viene interrotta nel punto in cui si verifica l'eccezione e non viene più ripresa

Blocco catch

- Il blocco **catch** può anche essere vuoto, dipende dalla logica del programma: in questo caso potrebbe esserlo

```
int n = 0;
boolean done = false;
do
{   try
    {   System.out.print("Un numero intero: ");
        String line = console.nextLine();
        n = Integer.parseInt(line);
        done = true;
    }
    catch(NumberFormatException e)
    {   // il blocco catch può anche essere vuoto
    }
} while (!done);
```

Blocco catch

- Per rendere evidente che il blocco catch è stato lasciato intenzionalmente vuoto, di solito:
  - Vi si inserisce un'istruzione nulla (il solo punto e virgola)
  - Oppure, si mette un commento, che spesso è
    // **intentionally left blank**

Eccezioni fuori dai cicli

```
int n;
try
{   System.out.print("Un numero intero: ");
    n = Integer.parseInt(console.nextLine());
}
catch(NumberFormatException e)
{   n = 5;
}
```

oppure

```
int n = 5;
try
{   System.out.print("Un numero intero: ");
    n = Integer.parseInt(console.nextLine());
}
catch(NumberFormatException e)
{   // intentionally left blank
}
```

Blocco try

- Sintassi:

```
try
{   enunciatiCheForseGeneranoUnaEccezione
}
catch (TipoEccezione1 oggettoEccez1)
{   enunciatiEseguitiInCasoDiEccezione1
}
catch (TipoEccezione2 oggettoEccez2)
{   enunciatiEseguitiInCasoDiEccezione2
}
```

- Scopo:

  eseguire enunciati che **possono** generare una eccezione

  - ***se si verifica l'eccezione*** di tipo ***TipoEccezione***, eseguire gli enunciati contenuti nella ***clausola*** **catch** **corrispondente** a ***TipoEccezione***
    - ***altrimenti***, ignorare la ***clausola*** **catch**

Eccezioni numeriche e Scanner

- Se invece di nextLine() uso nextInt(), le eccezioni ci sono lo stesso (le lancia nextInt invece di parseInt)!

```
int n = 0;
boolean done = false;
do
{   try
    {   n = console.nextInt();
        done = true;
    }
    catch(java.util.InputMismatchException e)
    {   System.out.println("Riprova");
        console.next(); // elimina dal flusso
    }                   // la parola sbagliata
} while (!done);         // altrimenti rimane lì e
                        // viene letta di nuovo!!!
                // oppure si può usare nextLine() che
                // elimina più parole, tutta una riga
```

L'eccezione viene lanciata dal metodo **nextInt**, che al suo interno usa **parseInt**

Eccezioni numeriche e Scanner

- Potrebbe sorgere un dubbio: l'invocazione **`console.next()`** restituisce una stringa, che non viene memorizzata in una variabile… dove va a finire? Si può fare?

```
catch(java.util.InputMismatchException e)
{   System.out.println("Riprova");
    console.next(); // elimina dal flusso
}                   // la parola sbagliata
```

- È perfettamente lecito, la stringa restituita viene "abbandonata" e non sarà più disponibile all'interno del programma in esecuzione (ma, in effetti, non serve)

Precisazione

- In generale, **è sempre lecito invocare un metodo e ignorare il valore che restituisce.**
- Naturalmente, perché ciò **abbia senso**, bisogna che il metodo provochi qualche conseguenza (in questo caso, estrae una parola dal flusso di input).
- Non avrebbe senso invocare **Math.sqrt** e ignorare il valore restituito!

Eccezioni numeriche e Scanner

- Usando **`nextLine()`** invece di **`next()`** si pone rimedio a errori più gravi da parte dell'utente (che magari ha scritto **`"pippo pluto"`**, con uno spazio, invece di un numero....)

```
catch(java.util.InputMismatchException e)
{   System.out.println("Riprova");
    console.nextLine();
}
```

- Se avessi usato next() "eliminavo" pippo ma non pluto

Eccezioni numeriche e Scanner

- Si potrebbe anche pensare di "svuotare completamente il flusso", nell'ipotesi che l'errore dell'utente si possa estendere su più righe, in questo modo

```
catch(java.util.InputMismatchException e)
{   System.out.println("Riprova");
    while (console.hasNextLine())
        console.nextLine();
}
```

- Ma questo è un errore logico: se viene eseguito questo blocco **`catch`**, il programma rimane bloccato nell'esecuzione del ciclo **`while`** finché l'utente non chiude esplicitamente il flusso di input (cosa che, ovviamente, non ha senso, visto che poi dovrebbe inserire un numero…)

Eccezioni numeriche e Scanner

- Allora, come si fa a "pulire bene il flusso" dopo un errore?

```
catch(java.util.InputMismatchException e)
{   System.out.println("Riprova");
    console.nextLine();
}
```

- Con questo codice, togliamo tutta la riga scritta per errore dall'utente, quando avrebbe dovuto scrivere un numero (e premere Invio)

Eccezioni numeriche e Scanner

```
catch(java.util.InputMismatchException e)
{   System.out.println("Riprova");
    console.nextLine();
}
```

- **Questo è sufficiente:** infatti, è veramente difficile (se non impossibile) che l'utente sia riuscito a scrivere qualcos'altro dopo aver premuto Invio e prima che venga eseguito di nuovo il metodo **nextLine** del blocco **try** (dopo la pulizia nel **catch**), quindi è fortemente probabile che veda prima il "Riprova". Se si sbaglia di nuovo, semplicemente verrà eseguito di nuovo il ciclo

Scanner: comportamenti anomali

- A volte **nextInt** e **nextDouble** si comportano in modo "strano"…
  in realtà sempre in modo ben documentato, cerchiamo di capire…
  - Per prima cosa, leggono e "consumano" (cioè eliminano dall'estremità iniziale del flusso su cui opera lo **Scanner**) eventuali ***caratteri di spaziatura*** (che sono spazi, caratteri di tabulazione e caratteri di "andata a capo", chiamati *newline*) che possono precedere il numero che stanno cercando, ignorandoli; **anche in caso di lancio di eccezione, questi caratteri saranno definitivamente consumati**
  - Poi, leggono caratteri provenienti dal flusso finché sono idonei a costituire un numero (intero o, rispettivamente, frazionario)

Scanner: comportamenti anomali

- Appena "vedono" un carattere non idoneo (ad esempio, una lettera o un carattere di spaziatura) interrompono la propria azione, lasciando all'inizio del flusso, senza "consumarlo", il carattere appena "visto"
  - Se hanno visto caratteri sufficienti a costituire un numero e SOLTANTO caratteri che costituiscono un numero, li "consumano" e lo restituiscono
  - Altrimenti si verifica l'eccezione **InputMismatchException** e i caratteri NON vengono "consumati", rimangono nel flusso, **così come se il flusso viene chiuso prima che si sia visto un carattere non idoneo**

Scanner: comportamenti "strani"

- Un altro comportamento "strano" (ma sempre ben documentato) si ha quando un'invocazione di **next**/**nextInt**/**nextDouble** è seguita da un'invocazione di **nextLine**
- Ad esempio, si vuole leggere un numero intero e un nome, su due righe consecutive, in questo modo

```
int n = console.nextInt();
String name = console.nextLine();
System.out.println(name);
```

- Se l'utente scrive 46 e va a capo, non fa nemmeno in tempo a scrivere, ad esempio, Marco, perché il programma termina, scrivendo una riga vuota, esattamente come se avesse letto, in **name**, una stringa vuota

Scanner: comportamenti "strani"

- Perché?
  - **nextInt** legge i due caratteri che compongono il numero 46 e, quando "vede" il carattere *newline*, si ferma e restituisce il numero 46, lasciando all'inizio del flusso proprio il carattere *newline*.
  - A questo punto viene invocato **nextLine**, che ha il compito di leggere caratteri finché non legge un carattere *newline*, restituendo una stringa costituita dai caratteri letti (il carattere *newline* NON viene accodato alla stringa restituita ma viene "consumato" dal flusso): in questo caso, **nextLine** vede subito il carattere *newline*, quindi termina e restituisce una stringa vuota, che viene visualizzata da **println**

Un altro uso di Scanner

- La classe **Scanner** ha anche un altro costruttore molto utile, oltre a quello che già conosciamo
- E' possibile creare un oggetto della classe **Scanner** fornendo una stringa come parametro al costruttore

```
String x = "pippo pluto paperino";
Scanner sc = new Scanner(x);
```

- Scanner considera come delimitatori predefiniti gli spazi, i caratteri di tabulazione e i caratteri di "andata a capo". Questi e altri caratteri sono detti **whitespaces** e sono riconosciuti dal metodo predicativo: Character.isWhitespace(char c)

Un altro uso di Scanner

- Se come parametro di costruzione viene fornita una stringa, lo scanner esaminerà i caratteri di quella stringa (invece di esaminare i caratteri di un flusso)
  - Può essere utile per scomporre una stringa in "parole"
  - Attenzione: non ha NIENTE a che vedere con la redirezione del flusso di input (in questo esempio, il flusso di input NON è coinvolto… infatti NON si usa **System.in**, non compare nel codice!)

Un altro uso della classe Scanner

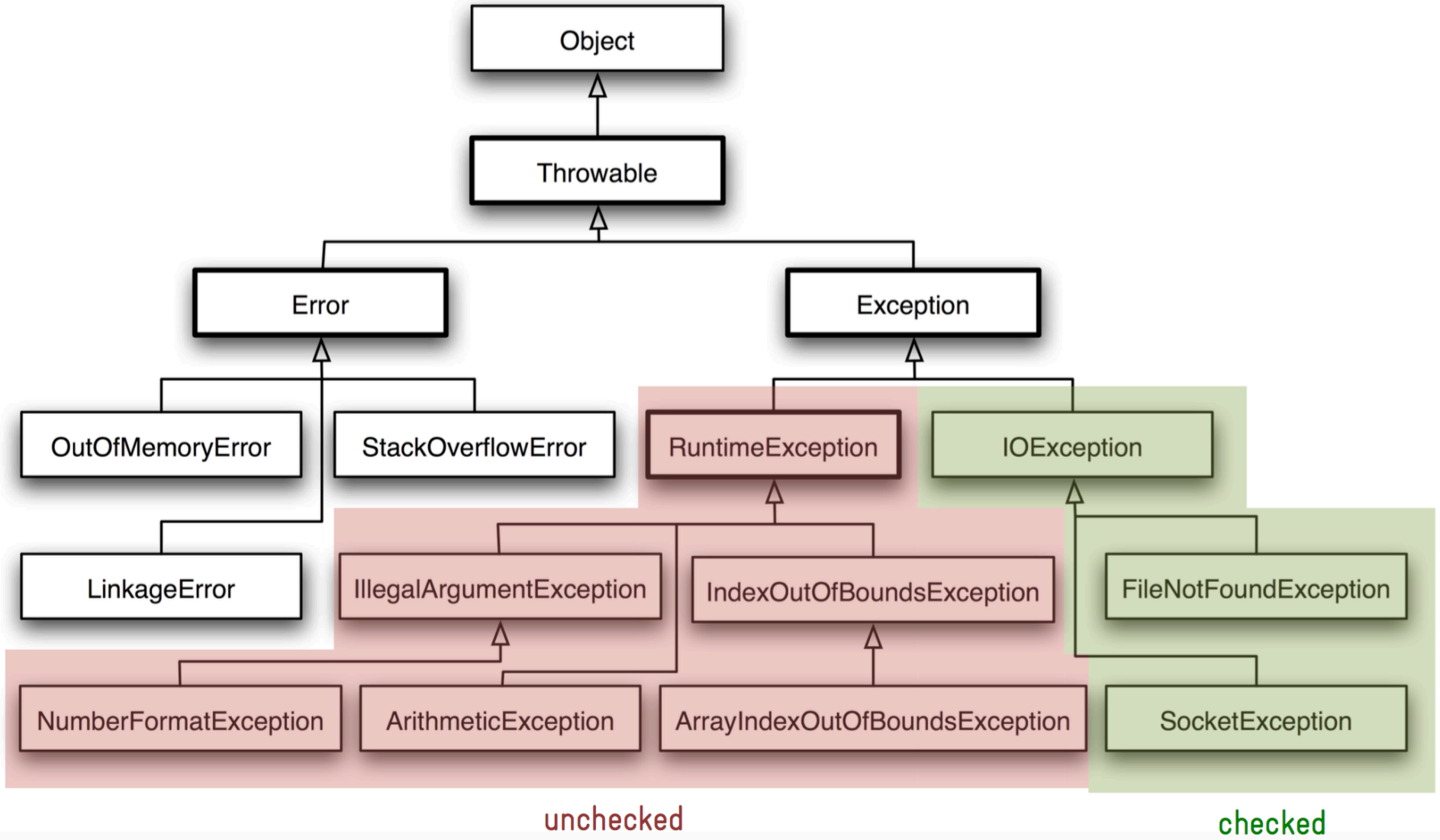
- Per accedere al token successivo si usa next() della classe Scanner
- Se l'elemento successivo non c'è next() lancia l'eccezione java.util.NoSuchElementException
- Poiché non è noto a priori il numero di "token" (parole ben delimitate da whitespaces) si utilizza "hasNext()"

```
String x = "pippo pluto paperino";
Scanner sc = new Scanner(x);
while (sc.hasNext())
   System.out.println(sc.next());
```

Diversi tipi di eccezioni

- In Java esistono diversi tipi di eccezioni (cioè diverse classi di cui le eccezioni sono esemplari)
  - eccezioni di tipo Error
  - eccezioni di tipo Exception
    - un sottoinsieme sono di tipo RuntimeException
      - ArithmeticException
      - IndexOutOfBoundException
      - NullPointerException
      - …

Gerarchia delle eccezioni



Diversi tipi di eccezioni

- La gestione delle eccezioni di **tipo Error** e di **tipo RuntimeException è facoltativa**
  - se non vengono gestite e vengono lanciate, provocano la terminazione del programma

- La gestione delle **altre eccezioni è obbligatoria** (se non c'è, si ha un errore in compilazione)
  - Si dice che sono "controllate"

Eccezioni controllate …

- Le eccezioni **controllate**
  - Descrivono problemi che possono verificarsi prima o poi, indipendentemente dalla bravura del programmatore
- Per questo motivo le eccezioni di tipo **IOException** sono controllate
- Se si invoca un metodo che può lanciare un'eccezione controllata, è **obbligatorio** gestirla con try/catch
- Altrimenti viene segnalato un **errore in compilazione**

...e non

- Le eccezioni **non controllate**
  - Descrivono problemi dovuti a errori del programmatore e che quindi non dovrebbero verificarsi (in teoria...)
- Per questo motivo le eccezioni di tipo **RuntimeException** sono non controllate
- Non è obbligatorio catturarle tramite try/catch

Realizzazione dei metodi: argomenti inattesi (pre-condizioni)

Argomenti inattesi

- Finora abbiamo visto come gestire il lancio di eccezioni da parte di metodi di classi della libreria standard, ma qualsiasi metodo può lanciare eccezioni
- Spesso un metodo richiede che i suoi argomenti
  - siano di un tipo ben definito
    - questo viene garantito dal compilatore
  - abbiano un valore che rispetti certi vincoli, ad esempio sia un numero positivo
    - in questo caso il compilatore non aiuta…

Argomenti inattesi

Come deve reagire il metodo se riceve un parametro che non rispetta i requisiti richiesti (chiamati precondizioni)?

Argomenti inattesi

- Ci sono tre modi per reagire ad argomenti inattesi
  - non fare niente: il metodo semplicemente termina la sua esecuzione senza alcuna segnalazione d'errore
    - questo però si può fare solo per metodi con valore di ritorno void, altrimenti che cosa restituisce il metodo?
    - se restituisce un valore casuale senza segnalare un errore, chi ha invocato il metodo probabilmente andrà incontro ad un errore logico
  - terminare il programma con System.exit(1)
    - System.exit(0) : è andato tutto bene
  - lanciare un'eccezione

Argomenti inattesi

- Lanciare un'eccezione in risposta ad un parametro che non rispetta una precondizione è la soluzione più corretta in ambito professionale
  - la libreria standard mette a disposizione tale eccezione
    - ***IllegalArgumentException***

```
public void deposit(double amount)
{   if (amount <= 0)
       throw new IllegalArgumentException();
    balance = balance + amount;
}
```

Enunciato *throw*

- Sintassi:

```
throw oggettoEccezione;
```

- Scopo: lanciare un'eccezione
- Nota: di solito l'oggettoEccezione viene creato con new ClasseEccezione( )

Gestione dei casi degeneri

- Spesso (**ma non sempre**) i casi degeneri di un algoritmo vanno gestiti separatamente
  - Qui è **inutile** gestire separatamente il caso in cui l'array ha lunghezza zero, perché il codice "normale" che segue fa esattamente la stessa cosa!

```
public static double sum(double[] values)
{  if (values.length == 0) return 0; // inutile...
   double sum = 0;
   for (int i = 0; i < values.length; i++)
      sum += values[i];
   return sum;
}
```

Gestione dei casi degeneri

```
public static double sum(double[] values)
{  if (values.length == 0) return 0; // inutile...
   double sum = 0;
   for (int i = 0; i < values.length; i++)
      sum += values[i];
   return sum;
}
```

- L'obiettivo del programmatore era quello di rendere più veloce l'elaborazione nel caso di array di lunghezza zero
  - L'effetto ottenuto è quello di rendere più lenta l'elaborazione nel caso (probabilmente assai più frequente) di array con lunghezza diversa da zero! Perché la **condizione** va valutata sempre

Gestione dei casi degeneri

- In un metodo che calcola, invece, il valore medio dei dati presenti in un array, il caso di array di lunghezza zero va gestito separatamente, per evitare divisioni per zero

```
public static double average(double[] values)
{   if (values.length == 0)
       throw new IllegalArgumentException();
    double sum = 0;
    for (int i = 0; i < values.length; i++)
       sum = sum + values[i];
    // il controllo su values.length == 0 si
    // può mettere anche qui, ma è più logico
    // metterlo all'inizio; basta che sia prima
    // di fare la divisione!
    return sum / values.length;
}
```

Take home message

- Eccezioni: meccanismo per segnalare situazioni di errore o inconsistenti con quanto atteso
  - Gestione obbligatoria
  - Gestione non obbligatoria (Error e RuntimeException)

- Per gestire un'eccezione: blocco try-catch
  - try { istruzioni che possono lanciare un'eccezione}
  - catch(tipoEccezione e) { istruzioni da svolgere in caso di verifichi tipoEccezione }
- Per lanciare un'eccezione: enunciato throw
  - if (condizione) throw new costruttore_tipoEccezione

Esercizio

- Realizzare una classe **FactorGenerator** che ***effettui la scomposizione di un numero intero positivo nei suoi fattori primi*** con un comportamento simile a quello della classe **Scanner** con i suoi metodi **nextInt** e **hasNextInt**

La classe deve avere

- un costruttore che:
  - riceve come unico parametro un numero intero
  - verifica che sia positivo e maggiore di uno
  - lancia **IllegalArgumentException** in caso contrario

La classe deve avere

- un metodo **non statico nextFactor**() che:
  - non riceve parametri espliciti
  - ad ogni successiva invocazione, restituisce uno dei fattori primi in cui viene scomposto il numero fornito nel costruttore
  - invocando un numero sufficiente di volte tale metodo si ottengono tutti e soli i fattori primi del numero (eventualmente ripetuti), in modo che moltiplicandoli si ottenga il numero originario
  - se il metodo viene invocato dopo aver ottenuto tutti i fattori primi, esso lancia **IllegalStateException**

La classe deve avere:

- un metodo **non statico hasMoreFactors**() che:
  - non riceve parametri espliciti
  - restituisce il valore booleano true se e solo se esistono fattori primi non ancora restituiti da nextFactor

Svolgimento

- La classe **FactorGenerator** è una classe che descrive le caratteristiche e il funzionamento di un oggetto

```
public class FactorGenerator
{
  //costruttore secondo specifiche

  //metodo nextFactor secondo specifiche

  //metodo hasMoreFactors secondo specifiche

  //variabili di esemplare
}
```

Scelta delle variabili di esemplare

- Sicuramente una caratteristica peculiare di un oggetto di tipo FactorGenerator è il numero intero positivo di cui si vuole effettuare la scomposizione in fattori primi

```
public class FactorGenerator
{
  //costruttore secondo specifiche

  //metodo nextFactor secondo specifiche

  //metodo hasMoreFactors secondo specifiche

  private int numero;
}
```

Realizzazione del costruttore

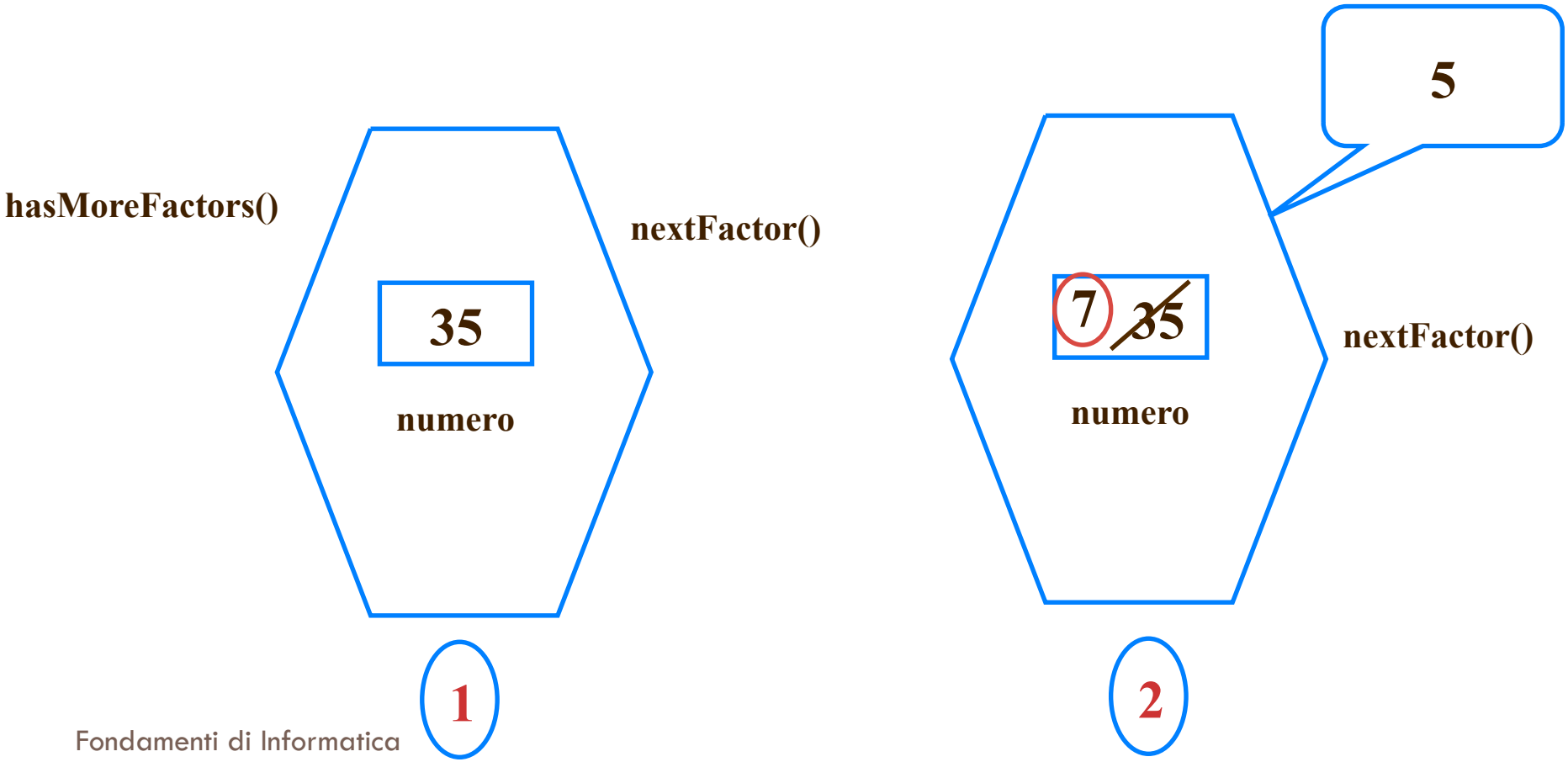
- il costruttore
  - riceve come unico parametro un numero intero
  - verifica che sia positivo e maggiore di uno
  - lanciando **IllegalArgumentException** in caso contrario

```
public class FactorGenerator
{ public FactorGenerator(int valore)
  { if(valore<=1)
      throw new IllegalArgumentException();
    numero = valore;
  }

  //metodo nextFactor secondo specifiche
  //metodo hasMoreFactors secondo specifiche
  private int numero;
}
```

Svolgimento

- Il metodo **nextFactor()** restituisce uno dei fattori primi del numero (finché ce ne sono)



Svolgimento

- Sarà sufficiente eseguire le divisioni fra il numero e tutti i numeri interi a partire da 2 fino a N, restituendo il divisore quando il resto della divisione risulta 0 e aggiornando la variabile numero con il quoziente

Realizzazione di nextFactor()

- Il metodo nextFactor deve restituire ad ogni invocazione un fattore primo del numero

```
public class FactorGenerator
{ public FactorGenerator(int valore)
  { if(valore<=1)
      throw new IllegalArgumentException();
    numero = valore;
  }
  public int nextFactor()
  { for (int i = 2; i <= numero; i++)
        if (numero % i == 0)
        { numero /= i;    //numero = numero/i;
          return i;
        }
    throw new IllegalStateException();
  }
  //metodo hasMoreFactors secondo specifiche
  private int numero;
}
```

Svolgimento

- Il metodo hasMoreFactors() restituisce true se e solo se esistono fattori primi non ancora restituiti da nextFactor

- Sarà sufficiente controllare il valore di numero:
  - se è diverso da 1=> ci sono ancora fattori primi
  - se è uguale a 1=> non ci sono più fattori primi

Realizzazione di hasMoreFactors()

```
public class FactorGenerator
{ public FactorGenerator(int valore)
  { if(valore<=1)
      throw new IllegalArgumentException();
    numero = valore;
  }
  public int nextFactor()
  { for (int i = 2; i <= numero; i++)
        if (numero % i == 0)
        { numero /= i;
          return i;
        }
    throw new IllegalArgumentException();
  }
  public boolean hasMoreFactors()
  { return (numero != 1);
  }
private int numero;
}
```

Collaudo

- Scrivere un programma **FactorGeneratorTest** che:
  - legge dallo standard input un numero intero maggiore di uno
  - crea un esemplare di FactorGenerator fornendo come parametro il numero ricevuto
  - visualizza sull'uscita standard i fattori primi del numero

FactorGeneratorTester

```
import java.util.*;
public class FactorGeneratorTester
{ public static void main(String[] args)
  {Scanner console = new Scanner(System.in);
   int n;
   do
   { System.out.println("Inserire un numero intero > 1");
     try
     { n = console.nextInt();
     }
     catch (InputMismatchException e)
     { n = 0; // un qualsiasi valore minore di 2
       console.nextLine();
     }
   } while (n < 2);
   FactorGenerator f = new FactorGenerator(n);
   System.out.println("Fattori primi del numero " + n);
   while (f.hasMoreFactors())
       System.out.println(f.nextFactor());
  }
}
```