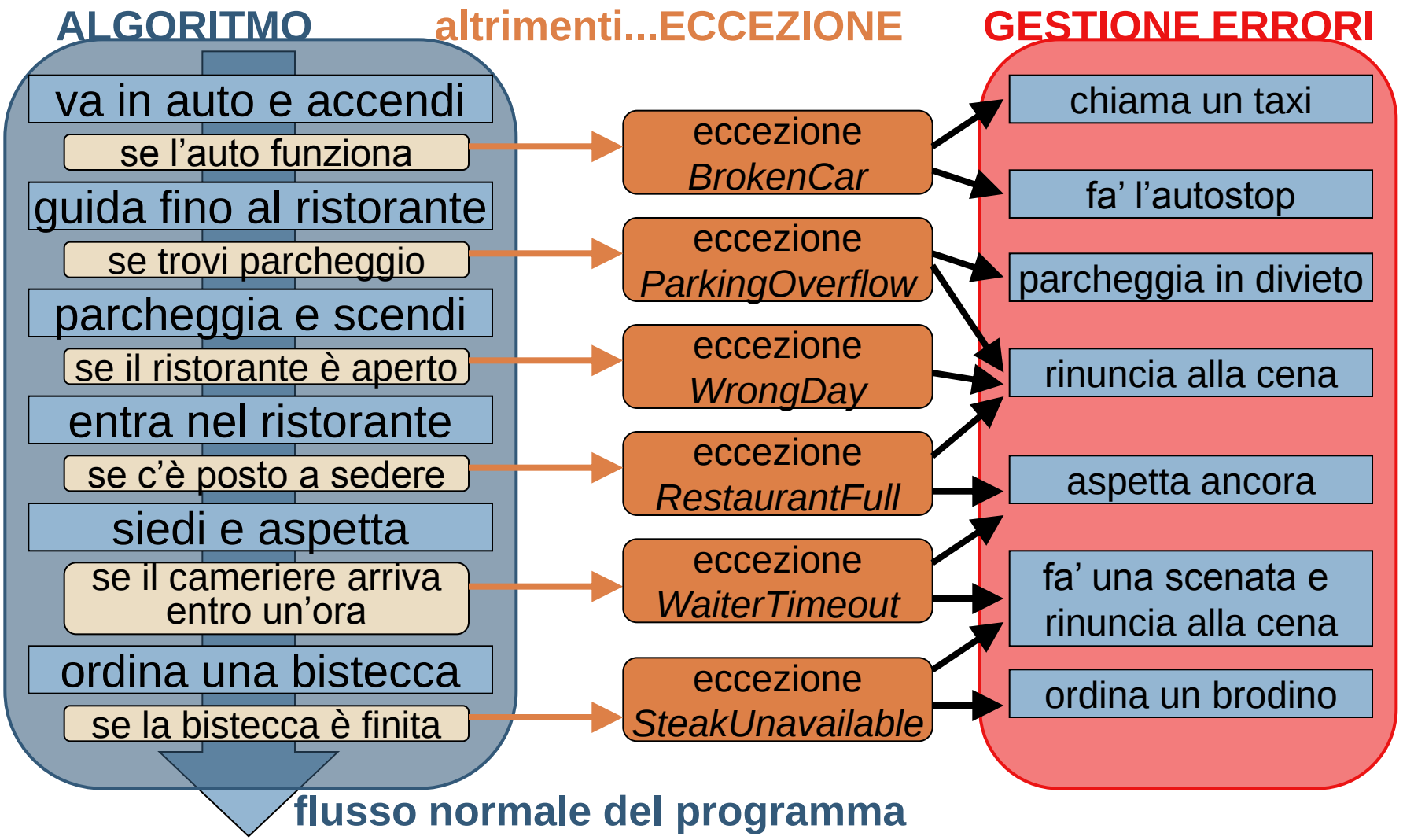
Eccezioni

- Le eccezioni sono eventi che si generano **durante l'esecuzione** di un programma e in genere corrispondono a condizioni anomale.
  - Es.: divisioni per zero, indici array fuori range, malfunzionamento dei dispositivi fisici...
- Quando si verificano, le eccezioni interrompono il normale flusso di esecuzione dell'applicazione e ritornano il controllo al chiamante

Eccezioni


ALGORITMO
altrimenti...ECCEZIONE
GESTIONE ERRORI
va in auto e accendi
se l'auto funziona
guida fino al ristorante
se trovi parcheggio
parcheggia e scendi
se il ristorante è aperto
entra nel ristorante
se c'è posto a sedere
siedi e aspetta
se il cameriere arriva entro un'ora
ordina una bistecca
se la bistecca è finita
eccezione BrokenCar
eccezione ParkingOverflow
eccezione WrongDay
eccezione RestaurantFull
eccezione WaiterTimeout
eccezione SteakUnavailable
chiama un taxi
fa' l'autostop
parcheggia in divieto
rinuncia alla cena
aspetta ancora
fa' una scenata e rinuncia alla cena
ordina un brodino
flusso normale del programma

Eccezioni

- L'uso delle eccezioni consente di separare:
  - la risoluzione del problema (l'algoritmo solutore)
  - la gestione degli errori
- È un **errore** ciò che causa malfunzionamento, ovvero comportamento effettivo ≠ atteso.
  - Un oggetto è **corretto** se nel suo comportamento **non possono verificarsi** malfunzionamenti.
  - Non basta che non capitino! I malfunzionamenti devono essere previsti ed evitati, includendo una soluzione per ognuno di essi.

Tipi di errore

- Errori sintattici: viene violata una regola della grammatica del linguaggio di programmazione

```
System.out.print(“Hi!); Classe.main(;  int y
```

- Errori semantici: espressioni ben formate ma che hanno significato diverso dalle intenzioni
  - sbagliare nomi di oggetti, metodi, tipi: possono essere inesistenti oppure un’altra cosa!

```
system.out.print(“Hi”); Classe.mayn(); doble z;
```

Tipi di errore

- Gli errori semantici possono essere:
  - **statici**: la grammatica è corretta ma il compilatore riconosce ugualmente l'errore perché vede che viene violata qualche regola di interpretazione (per es.: variabile non dichiarata, tipo incorretto)
  - **dinamici**: il compilatore non è in grado di accorgersi dell'errore perché dipende da valori di variabili, che hanno senso solo a run-time. (per es.: divisione per zero, indici array sbagliati)

Eccezioni

- Gli errori semantici dinamici sono più problematici perché:
  - accadono **in fase di esecuzione**
  - non capitano sempre!
- Le eccezioni sono malfunzionamenti causati da **errori semantici dinamici**.
- Java fornisce dei meccanismi per intercettare le eccezioni, ovvero prevedere operazioni da eseguire in caso di eccezioni.
  - Questo permette di correggere gli oggetti.

Eccezioni

- Tipico caso di eccezione: divisione per 0

```
class Divisione
{ private static int calcolaQuoto(int a, int b)
  { return a/b;
  }
  public static void main(String[] args)
  { Scanner in = new Scanner(System.in)
    System.out.print("Inserisci il dividendo: ");
    int div1 = in.nextInt();
    System.out.print("Inserisci il divisore: ");
    int div2 = in.nextInt();
    int quoto = calcolaQuoto(div1,div2);
    System.out.println("Quoziente = " + quoto);
  }
}
```

Eccezioni

- Questo codice produce errori al compilatore?
  - No: compila correttamente e può essere eseguito
- Produce errori a run-time?
  - Dipende dagli input forniti. Per esempio

```
Inserisci il dividendo: 8
Inserisci il divisore: 0

Exception in thread "main"
java.lang.ArithmeticException: / by zero at...
```

causa il verificarsi di un'**eccezione**

Eccezioni

- Il messaggio stampato a video:
  - indica che si è verificata un'eccezione dalla classe **`java.lang.ArithmeticException`**, e specifica che si tratta di una divisione per zero
  - riporta il testo del punto di codice in esecuzione al momento, e che ha causato l'eccezione
  - poi riporta la stack trace, cioè la sequenza dei metodi (eventualmente innestati) che erano stati eseguiti, dall'ultimo (che aveva il controllo) all'indietro lungo quelli in attesa

Eccezioni

tralasciamo `System.out.print()` e `in.nextInt()` per semplicità

- Flusso normale di esecuzione
  - il **`main`** invoca **`calcolaQuoto`** e gli passa il controllo
  - il metodo **`calcolaQuoto`** esegue il suo compito e ripassa il controllo al **`main`** ritornandogli un intero
  - il **`main`** lo memorizza in **`quoto`** e lo stampa a video
- Flusso di esecuzione **con l'eccezione**
  - il metodo **`calcolaQuoto`** riscontra l'eccezione, si interrompe e restituisce il controllo al chiamante (il **`main`**) chiedendogli se può risolverla
  - il **`main`** non è in grado di risolverla, si interrompe e passa il controllo alla JVM; questa termina l'esecuzione stampando i messaggi di errore

Eccezioni

- Il metodo **`charAt(int i)`** della classe **`String`** ritorna l'**`i`**-esimo carattere del parametro implicito, **se `i` è una posizione valida.**

```
class ScegliLetteraAlfabeto
{ public static void main(String[] args)
  { String alfabeto = "abcdefghijklmnopqrstuvwxyz";
    Scanner in = new Scanner(System.in)
    System.out.print("Scegli una posizione: ");
    int i = in.nextInt();
    char x = alfabeto.charAt(i);
    System.out.println("La " i + "a lettera: " + x);
  }
}
```

Eccezioni

- Se si mette un indice errato viene sollevata una **`StringIndexOutOfBoundsException`**

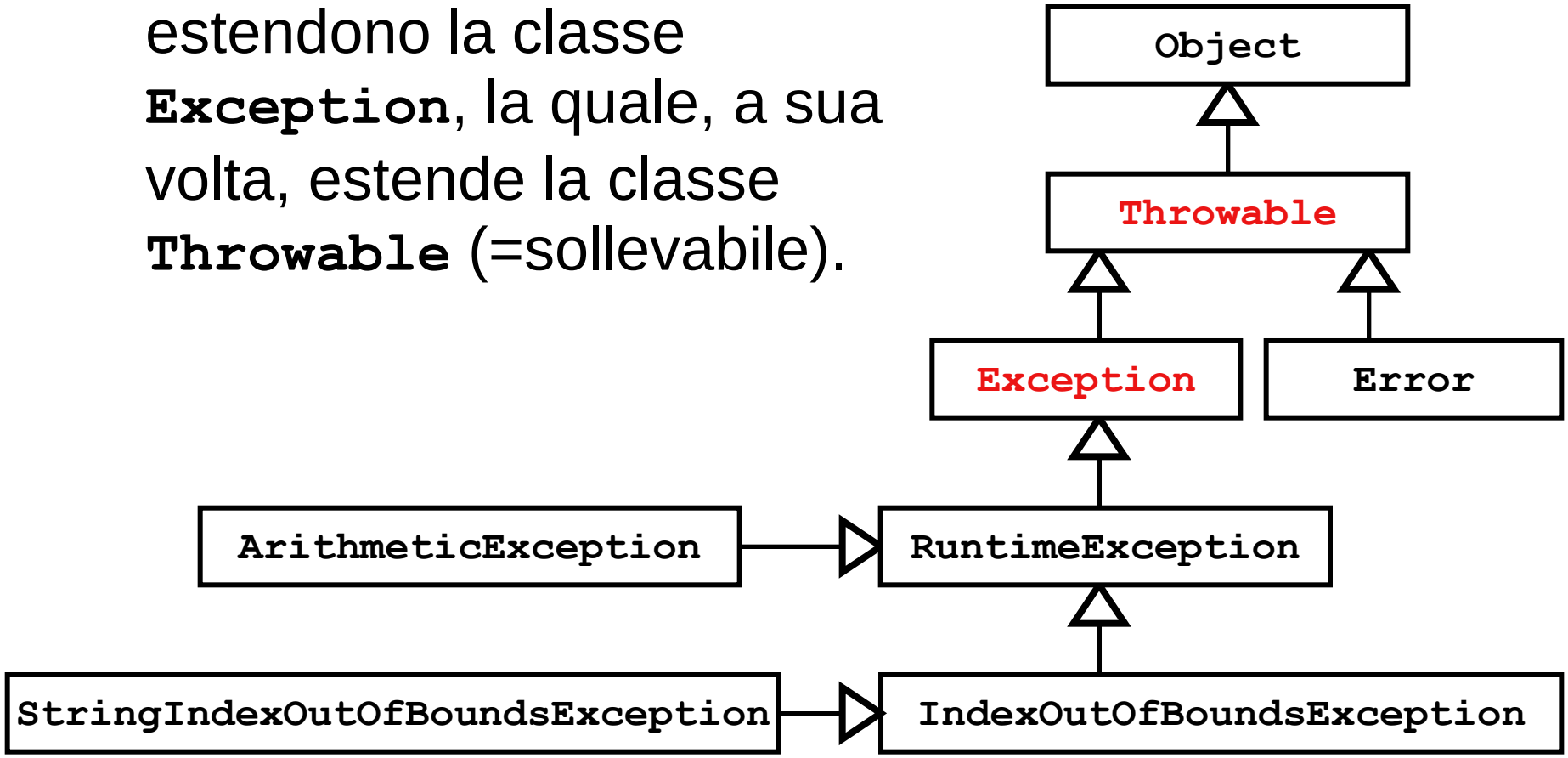
```
Scegli una posizione: 6
La 6a lettera: f

...

Scegli una posizione: 52

Exception in thread "main"
java.lang.StringOutOfBoundException: ...
```

Eccezioni

- **Le eccezioni sono oggetti**.
  - Quando si solleva un'eccezione, la JVM crea una **istanza** della classe eccezione sollevata.
  - Quello menzionato fin qua è il nome della classe (che appare anche nel messaggio d'errore), ma un'eccezione può anche avere un nome specifico
  - In questo modo si può accedere a proprietà specifiche di quella particolare eccezione.
  - Inoltre il tipo delle eccezioni partecipa a una gerarchia di classe, come per gli altri oggetti

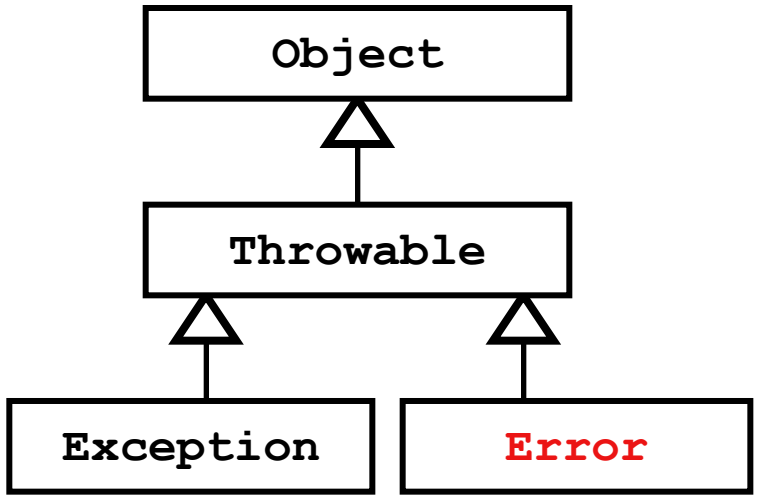
Eccezioni

- Le classi delle eccezioni estendono la classe **`Exception`**, la quale, a sua volta, estende la classe **`Throwable`** (=sollevabile).



Errori

- Nel diagramma è prevista anche la classe **`Error`**.
- Sono eventi di natura analoga alle eccezioni (ma tipicamente più gravi).
- La differenza è che per essi si consiglia di non fare nulla.
- Quindi:
  - gli errori vanno solitamente lasciati propagare
  - le eccezioni invece andrebbero gestite (intercettate)



Eccezioni

- Il punto di forza delle eccezioni sta nella loro propagazione a ritroso nello stack d'attivazione dei metodi in esecuzione.
- Ogni metodo può, alternativamente:
  - gestire l'eccezione
  - oppure propagarla al metodo che l'ha invocato
- Nel secondo caso, il metodo viene interrotto nel punto in cui si è sollevata l'eccezione.

Eccezioni

- La propagazione a ritroso comporta inoltre che
  - se risalendo lo stack si raggiunge l'inizio dell'applicazione (cioè il metodo **`main`** invocato per primo), e anche questo non gestisce l'eccezione, l'applicazione viene terminata
  - il record di attivazione (ciò che viene allocato sullo stack d'attivazione) di un metodo che non gestisce l'eccezione viene distrutto al momento in cui questo ritorna il controllo al chiamante

Eccezioni

- Supponiamo che **`main()`** → **`a()`** → **`b()`**
- Se durante l'esecuzione di **`b()`** viene sollevata un'eccezione:
  - se **`b()`** è in grado di farlo, la gestisce
  - se **`b()`** non è in grado, la sua esecuzione viene abortita (e il suo record di attivazione distrutto); il controllo passa ad **`a()`** e quindi:
    - se **`a()`** è in grado di farlo, gestisce l'eccezione
    - altrimenti, **`a()`** termina e il controllo passa a **`main()`**
      - se **`main()`** può gestire l'eccezione, lo fa
      - altrimenti l'applicazione termina

Eccezioni

```
public static void main(String[] args)
{ //istruzioni varie di main(), poi:
  a();
  //altre istruzioni di main()
}

void a()
{ //istruzioni varie di a(), poi:
  b();
  //altre istruzioni di a()
}

void b()
{ //può sollevare un'eccezione
  operazione_pericolosa;
}
```

Eccezioni

```
public static void main(String[] args)
{ //istruzioni varie di main(), poi:
  a();
  { //istruzioni varie di a(), poi:
    b();
    { //può sollevare un'eccezione
      operazione_pericolosa;
    }
  //altre istruzioni di a()
  }
//altre istruzioni di main()
}
```

Terminazione di un metodo

- Due modalità di terminazione di un metodo:
  - terminazione normale: conseguente a una istruzione **`return`** (o alla fine dell'esecuzione del corpo, se tipo di ritorno **`void`**); in tal caso il metodo fornisce un ritorno al chiamante
  - terminazione anomala: conseguente a una eccezione; in tal caso il metodo non fornisce un valore di ritorno, ma il chiamante può ugualmente fare qualcosa sulla base dell'eccezione sollevata

Intercettare le eccezioni

- Solitamente, i metodi più in basso nella gerarchia non intercettano le eccezioni.
- Le propagano solamente, lasciando che sia il metodo chiamante (o uno più in alto ancora, eventualmente **`main`**) a intercettarle.
- Quindi il costrutto tipico è del tipo:
  - fai-un-tentativo; se va bene, siamo a posto
  - se qualcosa va storto (e lo si deduce dal fatto che viene ritornata un'eccezione): adotta-soluzione

Intercettare le eccezioni

- Consideriamo il metodo **`charAt()`**.
- Non possiamo cambiare quello che fa il metodo: se solleva un'eccezione, gliela lasciamo propagare.
- Però prima di invocarlo, il metodo chiamante potrebbe avvisare la JVM che il metodo **`charAt()`** può sollevare un'eccezione; lui tenterà di eseguirlo ugualmente, candidandosi a gestirla se viene sollevata.

try... catch...

- Per questo scopo, Java mette a disposizione il costrutto try... catch... :

```
try
{ //istruzioni che potrebbero dare guai
}
catch (tipo_di_eccezione1 e1)
{ //contromisura ...
}
catch (tipo_di_eccezione2 e2)
{ //contromisura ...
}
...
```

try... catch...

```
class ScegliLetteraAlfabeto
{ public static void main(String[] args)
  { String alfabeto = "abcdefghijklmnopqrstuvwxyz";
    Scanner in = new Scanner(System.in)
    System.out.print("Scegli una posizione: ");
    int i = in.nextInt();
    try
    { char x = alfabeto.charAt(i);
      System.out.println("" + i + "a lettera: " + x);
    }
    catch(StringIndexOutOfBoundsException e)
    { System.err.println("Indice non valido!");
    }
  }
}
```

`try... catch...`

- Non c'è limite al numero di parti **`catch`** che possono seguire una **`try`**.
  - Però una **`catch`** può catturare solo le eccezioni compatibili col suo argomento e solo se generate nel blocco **`try`** a cui fa riferimento.
- Attenzione! Se una **`catch`** ha successo nel catturare un'eccezione le successive **`catch`** non vengono eseguite!
  - è analogo a una cascata di if – elseif – elseif ...

`try... catch...`

- Quindi la JVM si comporta in questo modo.
- Inizia a eseguire la parte **`try`** :
  - se durante l'esecuzione della parte **`try`** non si solleva alcuna eccezione, salta alla prima istruzione seguente a tutte le parti **`catch`**
  - se si solleva un'eccezione, scorre gli argomenti delle **`catch`** **nell'ordine in cui sono scritte:**
    - al primo compatibile, esegue il corpo di quella **`catch`** e poi va alla prima istruzione dopo tutte le parti **`catch`**
    - se nessuno è compatibile, si comporta come se non ci fosse alcun **`try-catch`** e quindi inoltra l'eccezione

Intercettazioni ed ereditarietà

```
try
{ char x = alfabeto.charAt(i);
}
catch( StringIndexOutOfBoundsException e)
{ System.err.println("Indice non valido!");
}
```

confrontiamolo con

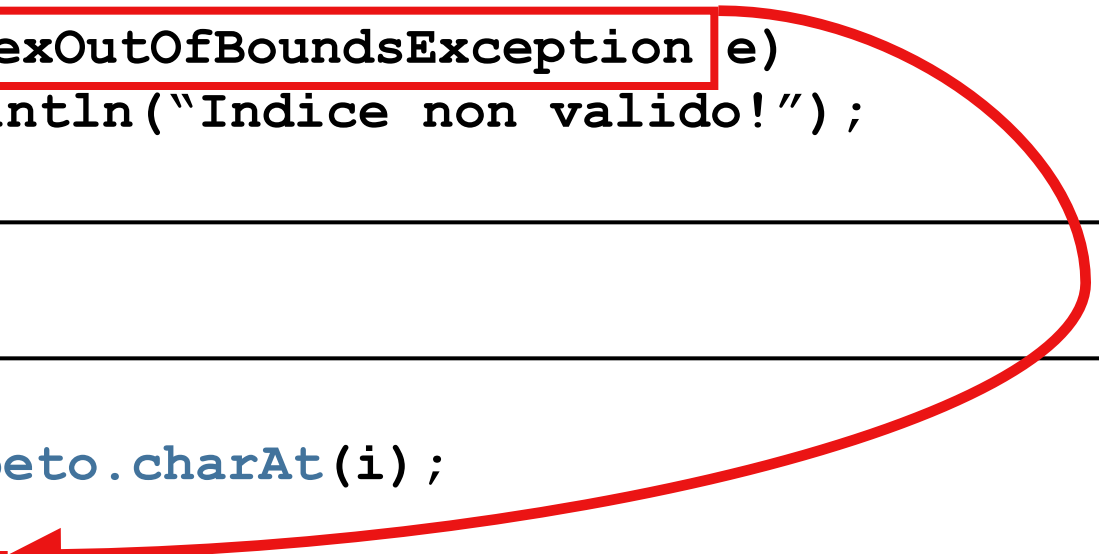
```
try
{ char x = alfabeto.charAt(i);
}
catch( RuntimeException e)
{ System.err.println("Indice non valido!");
}
```

Intercettazioni ed ereditarietà

```
try
{ char x = alfabeto.charAt(i);
}
catch( StringIndexOutOfBoundsException e)
{ System.err.println("Indice non valido!");
}
```

oppure con

```
try
{ char x = alfabeto.charAt(i);
}
catch( Exception e)
{ System.err.println("Indice non valido!");
}
```



Intercettazioni ed ereditarietà

- Le due versioni hanno lo stesso effetto perché ogni istanza della (sotto)classe **`StringIndexOutOfBoundsException`** lo è anche della classe (base) **`Exception`**.
  - Però la seconda versione è più generale e può intercettare anche eccezioni di altra natura.
- La relazione deve essere di sottoclasse.
  - Se avessimo messo **`ArithmeticException`** non avremmo intercettato l'eccezione perché le due classi non sono in relazione diretta.

Intercettazioni ed ereditarietà

- I blocchi **`catch`** vanno usati, da un lato, in modo da differenziare il più possibile le casistiche di eccezione per coprirle tutte.
- Dall'altro lato, grazie all'ereditarietà è possibile evitare di dover scrivere tanto codice per essere sicuri di trattare con tutte le eccezioni.
  - In particolare, si può considerare la classe **`Exception`** come un caso di default. Se cerchiamo di intercettare questo tipo, qualunque eccezione andrà bene.

Intercettazioni ed ereditarietà

- Attenzione all'ereditarietà! Non ha senso mettere prima una **`catch`** di un'eccezione superclasse e dopo una di sottoclasse.
  - Quella di sottoclasse non verrà mai eseguita: le sue istanze sono anche istanze della superclasse
- Il fatto che tutte le eccezioni siano anche istanze di **`Exception`** può causare errori.
  - Mettere in fondo una catch di **`Exception`** significa mettere un blocco di default
  - Metterla per prima: **errore** (lo rileva il compilatore)

Intercettazioni ed ereditarietà

```
try
{ f1(); //un'eccezione qua fa saltare f2() e f3()
  f2(); //un'eccezione qua fa saltare f3()
  f3();
}
catch(Exception e)
{ //cattura una qualsiasi eccezione di sopra
}
catch(StringIndexOutOfBoundsException e)
{ //blocco che non verrà MAI ESEGUITO!
}
catch(NullPointerException e)
{ //blocco che non verrà MAI ESEGUITO!
}
```

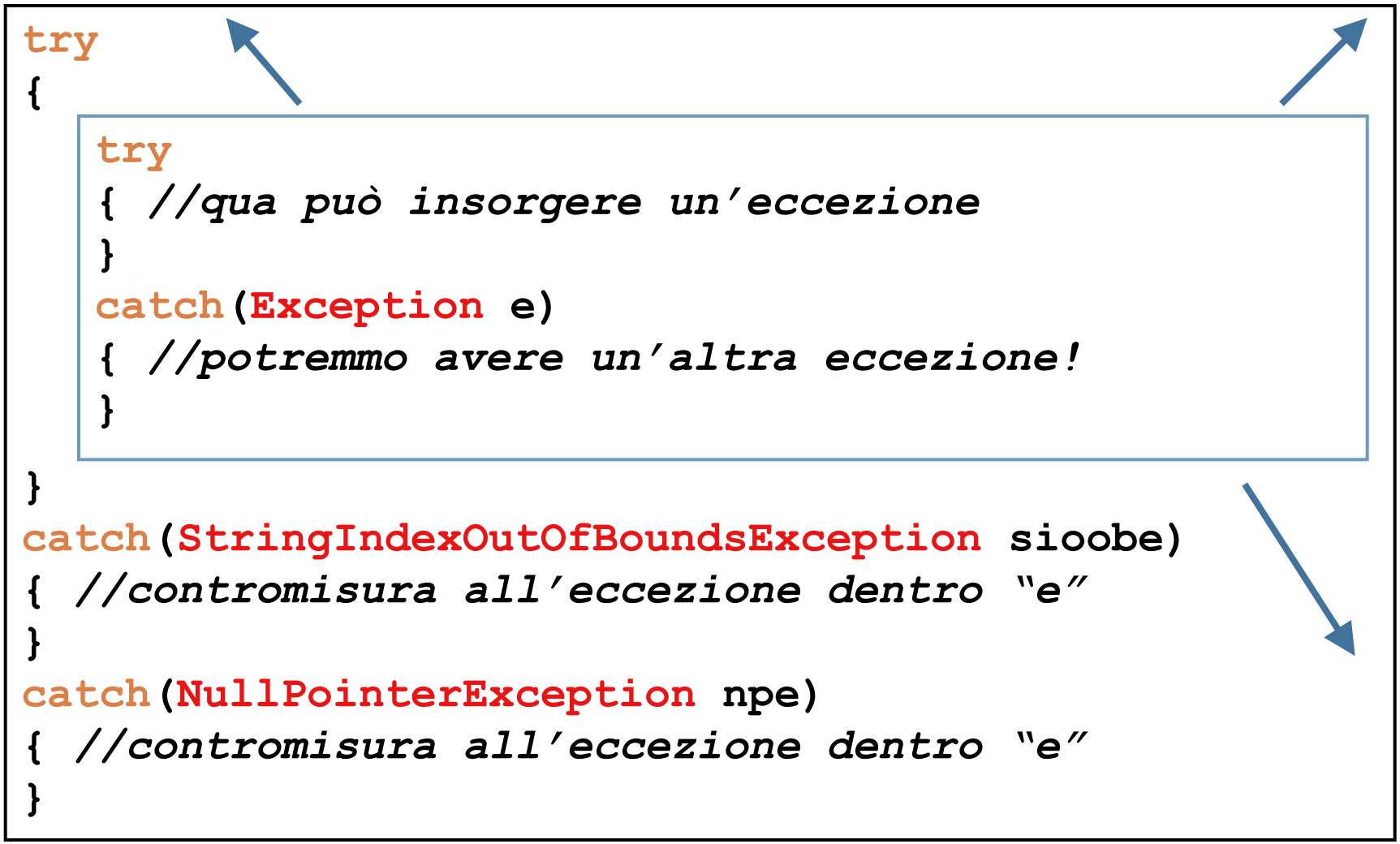
Intercettazioni ed ereditarietà

```
try
{ f1(); //un'eccezione qua fa saltare f2() e f3()
  f2(); //un'eccezione qua fa saltare f3()
  f3();
}
catch(StringIndexOutOfBoundsException e)
{ //cattura eccezioni di indici fuori range
}
catch(NullPointerException e)
{ //cattura eccezioni di riferimento a null
}
catch(Exception e)
{ //cattura tutte le eccezioni residue (default)
}
```

Annidamento

- Il costrutto **`try..catch..`** è in grado di gestire una sola eccezione per volta.
  - Quindi, se la catch causa un'altra eccezione oltre a quella che lei ha intercettato (dello stesso tipo, o diverso) le **`catch`** non vengono riesaminate.
- Se si vogliono intercettare anche le eccezioni lanciate dentro la catch, bisogna mettere un costrutto **`try..catch..`** più esterno.

Annidamento



Variabile eccezione

- Nell'identificatore di eccezione della **`catch`** all'atto di esecuzione viene creata una variabile eccezione. L'identificativo di tale variabile può essere usato dove fa comodo.
  - In particolare, di questo oggetto che viene creato è possibile invocare i metodi.
  - Tipicamente esiste un metodo **`toString()`** che consente di impostare (e poi magari stampare a video) una stringa col perché di un errore.

Variabile eccezione

```
try
{ f1(); //un'eccezione qua fa saltare f2() e f3()
  f2(); //un'eccezione qua fa saltare f3()
  f3();
}
catch(StringIndexOutOfBoundsException e)
{ //blocco per risolvere l'eccezione!
  System.out.println(e.toString());
}
```

- I messaggi delle eccezioni sono utili al programmatore per capire cos'è successo.
- Non dovrebbero rimanere nella versione finita

La clausola `finally`

- Cosa deve fare una **`catch`**?
  - Non dev'essere troppo generica
  - Può anche lasciare propagare l'eccezione intenzionalmente, ma comunque approfittando dell'eccezione per fare qualcosa
  - In questo caso, l'eccezione si propaga, e il controllo passa a un altro metodo.
- Si può prevedere di eseguire qualcosa sempre e comunque dopo la try-catch.

La clausola `finally`

- Questo “qualcosa” che viene eseguito in ogni caso viene introdotto dalla clausola **`finally`**.
- La parte di **`finally`** viene sempre eseguita:
  - sia che il try termini senza sollevare eccezioni
  - sia che l’eccezione venga catturata in una catch
- Uno scopo di finally è garantire la portabilità del codice, evidenziando quali sono le parti che vogliamo che vengano sempre eseguite.

La clausola `finally`

- Riprendiamo l'esempio della divisione

```
int c = 0;
try
{ System.out.println("Try. Prima.");
  c = calcolaQuoto(a, b); // (se b==0: eccezione)
  System.out.println("Try. Dopo.");
  return c;
}
catch (ArithmeticException e)
{ System.out.println("Catch.");
}
finally
{ System.out.println("Finally.");
}
System.out.println("Quoziente = " + c);
```

La clausola `finally`

- Esempi di esecuzione:

```
Inserisci il dividendo: 10
Inserisci il divisore: 4
Try. Prima.
Try. Dopo.
Finally.
Quoziente = 2
```

```
Inserisci il dividendo: 10
Inserisci il divisore: 0
Try. Prima.
Catch.
Finally.
Quoziente = 0
```

La clausola `finally`

- Il blocco **`finally`** è importante quando si recupera da un'eccezione.
  - Per esempio, quando si chiude un file o si recuperano risorse, è bene farlo dentro **`finally`** per essere sicuri che l'esecuzione non venga saltata per colpa di quel che fa la **`catch`**.
  - **`finally`** resiste anche a salti dovuti a **`break`**, **`continue`**, **`return`**.
- L'unica cosa davanti alla quale **`finally`** si arresta è una chiamata a **`System.exit()`**