Università di Roma "La Sapienza", Facoltà di Ingegneria

Corso di

Progettazione del Software

Corso di Laurea in Ingegneria Gestionale

Prof. Toni Mancini & Prof. Monica Scannapieco

AUTOV.Java.4
Nozioni Preliminari di Programmazione e Java

Versione del 3 gennaio 2007

Librerie di Classi

- Java mette a disposizione moltissime classi predefinite, alcune delle quali sono strettamente integrate con il linguaggio stesso.

- Alcune classi, dette *Classi Wrapper*, forniscono un corrispondente ad oggetti dei tipi di base.

- Vediamo poi (alcuni dettagli) delle classi String, System, Math

La classe Wrapper Integer

- Corrisponde al tipo primitivo int

- metodi di conversione:
  - int $\rightarrow$ Integer: **Integer(int value)** [costruttore]
  - Integer $\rightarrow$ int: **int intValue()**

- parsing di Stringhe che rappresentano interi:
  - **Integer(String s) [costruttore]**, **static Integer valueOf(String s)**
  - **static int parseInt(String s)**

La classe String

- Le istanze rappresentano stringhe "immutabili".

- Java offre un supporto speciale per le stringhe:
  - Allocazione nella forma String s="stringa";
  - Pooling: Utilizzando più volte l'allocazione tramite "stringa" si ottiene un riferimento allo STESSO oggetto.
  - new String("stringa") invece costruisce UNA COPIA.
  - Operatore "+" overloaded: permette di effettuare concatenazione di stringhe. Se uno dei due operandi non è di tipo stringa, viene convertito implicitamente a stringa (usando la funzione toString(), nel caso degli oggetti).

Metodi importanti

- Poichè le istanze della classe rappresentano stringhe costanti, i metodi hanno in genere un carattere "funzionale". Restituiscono una nuova stringa.

```
public static void main(String args[]){
    String s1="sono una stringa";
//trova l'indice della sottostringa specificata, -1 se non trova nulla.
    int i=s.indexOf("must");
    //estrae una sottostringa
    String s2=s.substring(5);
    String s3=s.substring(3, 7);
}
```

System

- è una classe che ha solo membri static (e non può essere istanziata);
- Abbiamo già incontrato System.out, che chiaramente è un campo variabile pubblico (è un oggetto di tipo PrintStream che rappresenta il terminale video). Ci sono anche System.in e System.err
- static void System.exit(int status); abbandona istantaneamente il programma.
- static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length) copia un array.

Eccezioni

- Una delle qualità del software è la Robustezza. Un programma dovrebbe essere in grado di gestire situazioni impreviste di vario genere, reagendo adeguatamente (eventualmente dando informazioni sul problema, se non è possibile risolverlo ed il programma deve essere interrotto). Esempi di situazione imprevista:

  - Bug nel codice (es. cerco di accedere all'indice 6 di un array di dimensione 6)

  - Cambiamenti dell'ambiente al di fuori del controllo del programma. Es: cerco di aprire un file che è stato cancellato.

- il meccanismo delle *Eccezioni* serve a gestire (in modo elegante) gli imprevisti, mantenendo il programma leggibile e permettendo al programmatore di concentrarsi sulla logica dell'applicazione.

Idea...

- Ricordiamo che in qualsiasi istante dell'esecuzione di un programma java ci si trova nel corpo di un metodo, il cui record di attivazione è in cima allo stack.

- Se si verifica un evento imprevisto durante l'esecuzione del metodo (ad esempio l'accesso ad una posizione non valida di un array)

  - Il metodo "lancia" (throws) un'eccezione. Cioè viene creato un oggetto di tipo "eccezione" e viene passato alla virtual machine. L'eccezione ha tipo correlato al genere di problema verificatosi e contiene informazioni diagnostiche.

  - La virtual machine cerca un metodo che può "catturare" (catch) l'eccezione, cioè contiene codice per gestire il problema.

- Come avviene la ricerca? Si parte dal metodo in cui si è verificato il problema. Se questo può gestire l'eccezione, si riprende l'esecuzione dal blocco di gestione. Altrimenti si rimuove il record di attivazione e si passa al successivo nello stack (cioè al metodo chiamante), e così via...

- Se nessuno può gestire l'eccezione, si arriverà a rimuovere il record di attivazione del main(), e il programma terminerà. In questo caso viene stampata l'informazione diagnostica contenuta nell'eccezione, in particolare una rappresentazione dello stack dei record di attivazione.

Tipi di Eccezione

- Le eccezioni sono suddivise in famiglie di problemi tramite l'ereditarietà: sono una gerarchia che ha la classe Exception come capostipite

- La maggiorparte delle eccezioni è legata a problemi riconducibili a porzioni di codice localizzate (es. apertura di un file). Sono inoltre problemi a cui si può rimediare senza che sia compromessa la correttezza del programma. Queste eccezioni vengono dette *Checked Exceptions*

- Una particolare sottofamiglia di eccezioni, dette *Runtime Exception*, riguarda invece errori tipicamente dovuti a bug del programma (es. uso di un puntatore *null*, accesso ad un indice fuori range). Tali errori sono realmente imprevedibili, possono avvenire in qualsiasi porzione di codice, e spesso non possono essere risolti. Per questo motivo, il meccanismo di gestione differisce da quello delle altre.

Lancio esplicito di Eccezioni

- Per lanciare un'eccezione di un certo tipo bisogna creare un oggetto del tipo corrispondente e poi usare l'istruzione *throw* (lancia).

```
MyException e=new MyException("Messaggio");
throw e;
```

- Quando si incontra la clausola throw, l'esecuzione del metodo in cui si trova l'istruzione si arresta, e la JVM parte alla ricerca di un gestore.

- E' possibile lanciare esplicitamente sia eccezioni Checked che Runtime. Le eccezioni Runtime, tuttavia, possono verificarsi anche in assenza di un lancio esplicito (vengono lanciate dalla virtual machine).

Il Meccanismo di gestione delle Eccezioni

- Se nel corpo di un metodo si può verificare una Checked Exception, il metodo può alternativamente:
  - gestirla, tramite un blocco di codice appropriato, oppure
  - indicare esplicitamente che non la gestisce (cioè la "rilancia")
- Il fatto che si possa verificare un'eccezione può dipendere da:
  - un lancio esplicito
  - un'invocazione di un metodo che rilancia l'eccezione
- Inoltre un'eccezione Runtime può verificarsi in qualsiasi punto del codice.

Differenze fra eccezioni Checked e Runtime

- La gestione descritta differisce per Runtime e Checked Exceptions.

- Nel caso di Checked Exception il metodo DEVE gestire oppure rilanciare l'eccezione. Per questo motivo si chiamano "Checked": il compilatore "verifica" che una di queste due condizioni sia soddisfatta, e in caso contrario genera un errore in compilazione. Il compilatore verifica anche che non si laci o gestisca un'eccezione dove non può avvenire.

- Per le Runtime Exceptions questo vincolo non esiste. E' possibile dichiarare il rilancio oppure gestirle, ma non è obbligatorio (perchè vista la loro ubiquità, si appesantirebbe il codice).

Gestire le eccezioni: try..catch..

- Un metodo dichiara di poter gestire un'eccezione che si può verificare in una determinata porzione di codice, e fornisce un blocco di gestione, attraverso un'istruzione della forma:

```
try{
   //Blocco in cui può verificarsi un'eccezione
}
catch(MyException e){ //tipo dell'eccezione e parametro formale
del catch.
  //blocco di gestione dell'eccezione
}
catch(YourException e1){
  //altro blocco per un altro tipo di eccezione...
}
...
```

Rilanciare le eccezioni: throws

- Un metodo dichiara che nel suo corpo può verificarsi un'eccezione non gestita usando nell'intestazione la clausola *throw***s** TipoEccezione1, TipoEccezione2,...:

```
public void mioMetodo() throws MyException{
    //codice in cui può verificarsi myException
}
```

- Notare che se si invoca un metodo che rilancia una eccezione, allora in quel punto del codice potrebbe verificarsi tale eccezione. Quindi un metodo che contiene una chiamata ad un altro metodo che rilancia MyException dovrà gestire l'eccezione (con try..catch) oppure rilanciarla a sua volta.

Esempio

```
class ClasseMain{
    public static void main(String Args[]){
        FileInputStream f=new FileInputStream("miofile.txt");
    }
}
//No!! unreported exception java.io.FileNotFoundException; must be caught or declared
//to be thrown

//Le versioni corrette sono:
public static void main(String Args[]) throws FileNotFoundException{
    FileInputStream f=new FileInputStream("miofile.txt");
}

//oppure:
public static void main(String Args[]){
    try{
        FileInputStream f=new FileInputStream("miofile.txt");
    }
    catch(FileNotFoundException e){
        System.out.println("Non riesco ad aprire il file!");
    }
}
```

try..finally

- Oltre che da blocchi catch, un blocco try può essere seguito da un blocco finally (deve essere seguito da almeno un catch o un finally).

- Il codice in un blocco finally viene eseguito comunque, anche se è avvenuta un'eccezione nel blocco try.

- In genere si usa per fare operazioni di "pulizia" (es: chiudere file o connessioni aperti), che non verrebbero effettuati nel caso di eccezioni che non vengono catturate nel metodo.

```
try{
    ...
}
finally{
    //questa porzione di codice viene eseguita comunque
}
```

try..catch..catch..(etc.)..finally

- La forma completa ha un blocco try, uno o più blocchi catch ed un finally. Notare che anche se si catturano tutte le eccezioni, la presenza del finally evita la necessità di duplicare il codice "di pulizia" nei vari blocchi catch.

```
try{...
}
catch(ThisException e){...
}
catch(ThatException e1){...
}
...
finally{...
}
```

Eccezioni definite dall'utente

- Il meccanismo delle eccezioni permette di far fronte a molti problemi legati alle classi di libreria già esistenti.

- Quando si implementano nuove classi, si può avere l'esigenza di definire eccezioni personalizzate legate a particolari funzionalità della classe

- Basta creare una classe MiaEccezione che estende Exception (o sua sottoclasse) e definire un costruttore appropriato (in molti casi si tratta di delegare al costruttore di Exception).

- Poi si deve lanciare l'eccezione al momento giusto (con throw) quando serve.

- Notare: estendiamo Exception per condizioni di errore "prevedibili", estendiamo RunTimeException per errori "imprevedibili".

Vantaggi del meccanismo delle eccezioni

- Codice per la gestione degli eventi imprevisti separato logicamente dal codice applicativo;

- Gli errori si propagano attraverso lo stack dei record di attivazione in modo trasparente per il programmatore (quindi le eccezioni raggiungono il metodo preposto a gestirle senza bisogno di restituire codici d'errore ai metodi chiamanti);

- funzione diagnostica durante la fase di debugging del programma.