Java

tutto è un oggetto

G. Prencipe
prencipe@di.unipi.it

Manipolare oggetti

- Ogni linguaggio di programmazione fornisce mezzi per manipolare dati
- Spesso i programmatori devono costantemente sapere che tipo di manipolazione stanno usando
  - La manipolazione dei dati è diretta o indiretta (es. puntatori), e quindi una speciale sintassi va utilizzata?
- In Java questo aspetto è semplificato
  - Tutto è un oggetto
  - Sintassi uniforme

Manipolare oggetti

- Sebbene tutto è trattato come un oggetto, l'identificatore che si manipola è in realtà una "maniglia" (handle) a un oggetto
- Es. televisione (oggetto) e telecomando (handle)
  - Quello che utilizziamo per utilizzare la tv è il telecomando
  - Quello che ci portiamo dietro quando ci muoviamo nella stanza è il telecomando

Manipolare oggetti

- Il telecomando può esistere senza tv
  - Se abbiamo una maniglia non necessariamente esiste un oggetto ad essa connesso
- Es. se vogliamo mantenere una parola, bisogna creare una maniglia a una **String**

  String s;
  - In questo modo abbiamo creato *solo* la maniglia, non l'oggetto che conterrà la parola

  String s = "pippo";

Creare oggetti

- La stringa è un caso particolare
- In genere, per collegare una maniglia a un nuovo oggetto, si utilizza **new**

  String s = **new** String("pippo");
- Quindi
  1. Creare maniglia
  2. Collegarla a un oggetto
- **String** non è il solo tipo che esiste
  - Java fornisce una serie di tipi predefiniti
  - È possibile creare nuovi tipi

Tipi primitivi

- **new** crea un oggetto inserendolo sull'heap
- Per una serie di oggetti piccoli, questo sarebbe poco efficiente
- Per essi, invece di creare l'oggetto usando **new**, viene creata una variabile automaticamente che *non è* una maniglia
- Questa variabile è messa sullo stack, ed è quindi più efficiente

Tipi primitivi

I tipi primitivi hanno anche una classe "wrap" (se si vuole creare un oggetto non primitivo da tenere sull'heap)

Tipo prim	size	min	max	Wrap type
boolean	1-bit	-	-	**Boolean**
char	16-bit	0	2^{16}-1	**Character**
byte	8-bit	-128	+127	**Byte**
short	16-bit	-2^{15}	$+2^{15}$-1	**Short**
int	32-bit	-2^{31}	$+2^{31}$-1	**Integer**
long	64-bit	-2^{63}	$+2^{63}$-1	**Long**
float	32-bit	IEEE754	IEEE754	**Float**
Double	64-bit	IEEE754	IEEE754	**Double**
void	-	-	-	**Void**

Tipi primitivi

char c = 'x';

oppure

Character C = **new** Character('x');

Scope delle variabili

- In ogni linguaggio di programmazione, è importante conoscere la "durata" delle variabili
- La maggior parte dei linguaggi procedurali utilizza il concetto di *scope* delle variabili
  - Indica la visibilità e la durata dei nomi delle variabili
- In Java (come in C e C++) lo scope è definito da

{
….
}

Scope delle variabili

```
{
   int x = 12;
   /* solo x disponibile */
    {
        int q = 96;
        /*sia x che q disponibili */
   }
    /* solo x disponibile */
    /* q fuori scope */
}
```

Una variabile è disponibile solo fino alla fine del suo scope

L'indentazione rende il programma più leggibile

Scope delle variabili

```
{
   int x = 12;
        {
        int x = 96;
        }
}
```

ILLEGALE!!

Il compilatore dirà che x è già definita

Scope degli oggetti

- Per gli oggetti è diverso
- Quando si crea un oggetto con **new**, esso rimane anche dopo la fine del suo scope
- Es.

{
String s = new String ("pippo");
} /* fine dello scope */

- La maniglia s sparisce alla fine dello scope, ma l'oggetto **String** rimane, e occupa ancora memoria
  - Nell'esempio, non c'è modo di accedere all'oggetto, perché l'unica maniglia è fuori dallo scope

Scope oggetti

- Domanda: se in Java gli oggetti restano, cosa evita che le memoria si riempia?
- Risposta: il *garbage collector*
  - Controlla gli oggetti e determina quali non sono più riferiti, e rilascia la loro memoria
- Vengono eliminati tutti i problemi relativi al programmatore che dimentica di rilasciare memoria

Le classi

- Se tutto è un oggetto, come si stabilisce il *tipo* di un oggetto?
- Tradizionalmente, nei linguaggi OO la parola chiave per determinare il tipo è **class**,seguita dal nome del nuovo tipo

  class NomeTipo {….}
- A questo punto è possibile creare un oggetto di questo tipo

  NomeTipo a = **new** NomeTipo();

Campi e metodi

- Quando si definisce una classe (e in Java tutto è definire classi, creare oggetti si esse, e mandare messaggi ad essi), in essa ci si può mettere due tipi di *membri*
  - Dati (*variabili* o *campi* o *fields*)
  - Funzioni (*metodi*)

Dati

- Un dato è un oggetto di qualsiasi tipo, con cui si può comunicare tramite la sua maniglia
- Può essere anche uno dei tipi primitivi (che non hanno maniglia)
- Se è una maniglia a un oggetto, bisogna inizializzarla per collegarla a un oggetto (con **new**) tramite un metodo speciale detto *costruttore*
- Se è un tipo primitivo, può essere inizializzato direttamente nel punto in cui viene definito
  - Anche le maniglie in realtà (come vedremo) possono essere inizializzate nel punto in cui sono definite

Dati

- Ogni oggetto ha la propria memoria per i suoi dati
  - Cioè, i dati non sono condivisi tra gli oggetti
- Esempio di classe con dati

```
Class Dati {
  int i;
  float f;
  boolean b;
}
```

Dati

```
Class Dati {
        int i;
        float f;
        boolean b;
}
```

- La classe Dati da sola non fa niente, ma possiamo creare un oggetto

  Dati d = new Dati();
- Per riferire i membri di un oggetto (ad es. per assegnare valori ai dati):
  - nome maniglia . nome membro

  d.i = 47;

  d.f = 1.1f;

  d.b = false;

Valori di default

- Quando un tipo primitivo è membro di una classe, gli viene assegnato un valore di default (non è dunque necessario inizializzarlo)
  - Questo è garantito *solo* se la variabile è utilizzata come membro di una classe (un campo)

Tipo Primitivo	Default
boolean	**false**
char	**null**
byte	**(byte)0**
short	**(short)0**
int	**0**
long	**0L**
float	**0.0f**
double	**0.0d**

Valori di default

- Questa garanzia non si applica a variabili “locali”
  - Variabili che non sono campi di una classe
  - Ad es., variabili locali a una funzione
  - Se non sono inizializzate, si ottiene errore in compilazione

Tipo Primitivo	Default
boolean	**false**
char	**null**
byte	**(byte)0**
short	**(short)0**
int	**0**
long	**0L**
float	**0.0f**
double	**0.0d**

Metodi

- Il termine *metodo* in Java sostituisce quello classico di funzione
  - Sono intercambiabili
- I metodi determinano i messaggi che un oggetto può ricevere
- Le parti fondamentali di un metodo sono
  - Il nome
  - Gli argomenti
  - Il tipo restituito (tipo di ritorno)
  - Il corpo

Metodi

```
returnType nomeMetodo (/* lista arg */) {
        /* corpo */
}
```

- Il tipo di ritorno è il tipo del valore restituito dal metodo
- Il nome identifica il metodo
- Gli argomenti definiscono il tipo e il nome delle informazioni da passare al metodo

Metodi

- I metodi possono essere creati solo come parte di una classe
- Un metodo può essere invocato solo per un oggetto
  - Una eccezione è costituita dai metodi **static**
  - Chiaramente il metodo deve essere in grado di eseguire il metodo invocato, altrimenti si ottiene errore in compilazione

Metodi

- Un metodo di un oggetto viene invocato:
  - nome oggetto . nome metodo (lista arg)

    int x = a.f(arg1, arg2);
  - L'invocazione di metodi è spesso riferita anche come *invio di messaggio a un oggetto*
  - In questo esempio, il messaggio f(arg1, arg2) è stato inviato all'oggetto a

Lista argomenti

- Le informazioni passate nella lista degli argomenti sono oggetti
- Quindi, in questa lista vanno specificati i nomi e i tipi degli oggetti da passare
  - In realtà, come già detto, vengono passate maniglie a oggetti, tranne che per i tipi primitivi
- I tipi delle maniglie passate devono essere compatibili con i tipi definiti nella lista

Lista argomenti

- Consideriamo il seguente metodo (in una classe)

```
int storage (String s) {
        return s.length() * 2;
}
```

- Il metodo length() (uno di quelli definiti per le **String**he) restituisce il numero di caratteri in una stringa
- return dichiara che il metodo è terminato, e restituisce un valore (se previsto)
  - Nell'esempio, il valore è un intero

Return

- È possibile restituire qualsiasi tipo
- Se non si vuole restituire nulla, si indica **void** come returnType del metodo

```
boolean flag() {return true;}
float logBaseNat {return 2.718;}
void nothing () {return;}
void nothing2() {}
```

- Se il tipo di ritorno è **void**, **return** è usato solo per terminare il metodo, ed è quindi non necessario
- Se il tipo di ritorno **non è void**, **return** è necessario, e deve ritornare un valore del tipo giusto
  - Altrimenti, errore in compilazione

Visibilità dei nomi

- Un problema in ogni linguaggio di programmazione è legato al controllo dei nomi
- Se utilizziamo un nome in un certo modulo, e lo stesso nome viene usato anche in un altro modulo da un altro programmatore, come vengono distinti i due nomi ed evitati conflitti?

Visibilità dei nomi

- L'approccio utilizzato in Java non è molto diverso da quello utilizzato per i domini Internet
- In Java è come se si utilizzano i domini Internet al contrario
  - Se un dominio è di.unipi.it, in Java diventa it.unipi.di
  - I punti rappresentano sottodirectory
  - Quindi, il file system locale diviene una sorta di Internet locale

Visibilità dei nomi

- Quindi ogni classe definita all'interno di un file è rappresentata da un identificatore unico all'interno del file system
  - Chiaramente, nomi di classi nello stesso file devono essere unici

Altre componenti

- Se una classe è definita nello stesso file del codice che stiamo scrivendo, allora la possiamo semplicemente riferire e utilizzare
  - Questo anche se la classe è definita dopo il suo utilizzo
- Come riferire classi presenti in altri file?
  - Problemi possono essere legati al fatto che ci possono essere più classi con lo stesso nome
- Bisogna dire al compilatore Java quali classi vogliamo usare
  - Parola chiave: **import**

```
import java.util.Vector;
import java.util.*;
```

static

- Normalmente, con la definizione di una classe definiamo come gli oggetti di quella classe sono e come si comportano
- Nulla accade fino a quando non creiamo un oggetto con **new**
- A volte vogliamo qualcosa di diverso
  - Avere un pezzo di memoria per un particolare dato, indipendentemente dal numero di oggetti creati (o anche se nessun oggetto è vreato)
  - Aver bisogno di un metodo che non è associato a nessun particolare oggetto della classe
    - Avere un metodo che può essere invocato anche se nessun oggetto è creato

static

- Questi effetti si ottengono con la parola chiave **static**
- Quando si definisce qualcosa come statico, vuol dire che dati o metodi non sono legati a nessun particolare oggetto
- Quindi, anche se non si crea alcun oggetto di una classe, è possibile invocare un metodo o un dato **static** di quella classe

static

```
class TestStastico {
  static int i = 47;
}
```

- Esiste una sola area di memoria per TestStatico.i

```
TestStatico ts1 = new TestStatico();
TestStatico ts2 = new TestStatico();
```

- ts1 e ts2 condividono lo stesso i (ts1.i e ts2.i valgono entrambi 47)

static

```
class TestStastico {
  static int i = 47;
}
```

- Ci sono due modi per riferire una variabile **static**
  - Tramite un oggetto (ts1 e ts2)
  - Direttamente tramite il nome della classe
    - Metodo preferito (enfatizza la natura **static**)
    - Non si può fare con variabili non **static**

```
TestStatico.i++;
```

- A questo punto, quanto vale ts1.i? **48**
- E ts2.i? **48**

static

- La stessa logica si applica ai metodi **static**

```
class MetodoStatico {
        static void incr() {TestStatico.i++;}
}
```

- Per invocare incr()

```
MetodoStatico ms = new MetodoStatico();
ms.incr();

Oppure

MetodoStatico.incr();
```

static

- Con i dati **static**, viene creato un dato per ogni classe (e non per ogni oggeto creato)
  - Questo può influire sulle prestazioni
- Nel caso dei metodi la cosa non è così critica
  - Un importante uso di **static** nel caso dei metodi è quello di poterli invocare senza creare oggetti

Primo programma Java

```
// Questo commenta fino alla fine della riga
import java.util.*;
public class HelloDate {
  public static void main(String[] args) {
    System.out.println("Hello, it's: ");
    System.out.println(new Date());
  }
}
```

NOTA: una delle classi nel file deve avere lo stesso nome del file

Cioè, il nome del file sarà *HelloDate.java*

Primo programma Java

```
// Questo commenta fino alla fine della riga
import java.util.*;
public class HelloDate {
  public static void main(String[] args) {
    System.out.println("Hello, it's: ");
    System.out.println(new Date());
  }
}
```

Alcune librerie sono importate automaticamente: **java.lang**

Primo programma Java

```
// Questo commenta fino alla fine della riga
import java.util.*;
public class HelloDate {
  public static void main(String[] args) {
    System.out.println("Hello, it's: ");
    System.out.println(new Date());
  }
}
```

System è una classe in **java.lang**

Documentazione da *java.sun.com*

Per scoprire cosa fanno, imparare a spulciare la documentazione

Primo programma Java

```
// Questo commenta fino alla fine della riga
import java.util.*;
public class HelloDate {
  public static void main(String[] args) {
    System.out.println("Hello, it's: ");
    System.out.println(new Date());
  }
}
```

Date non è in **java.lang**….

provare per credere!!

Primo programma Java

```
// Questo commenta fino alla fine della riga
import java.util.*;
public class HelloDate {
  public static void main(String[] args) {
    System.out.println("Hello, it's: ");
    System.out.println(new Date());
  }
}
```

Date è in **java.util**….

provare per credere!!

Primo programma Java

```
// Questo commenta fino alla fine della riga
import java.util.*;
public class HelloDate {
  public static void main(String[] args) {
    System.out.println("Hello, it's: ");
    System.out.println(new Date());
  }
}
```

Cercare **System.out** nei doc

Primo programma Java

```
// Questo commenta fino alla fine della riga
import java.util.*;
public class HelloDate {
  public static void main(String[] args) {
    System.out.println("Hello, it's: ");
    System.out.println(new Date());
  }
}
```

Cercare **System.out** nei doc

è un oggetto **static PrintStream**

Essendo **static**, per usarlo non è necessario creare oggetti

Primo programma Java

```
// Questo commenta fino alla fine della riga
import java.util.*;
public class HelloDate {
  public static void main(String[] args) {
    System.out.println("Hello, it's: ");
    System.out.println(new Date());
  }
}
```

System.out è un oggetto **static PrintStream**

Quali metodi è possibile invocare per un **PrintStream**?

Spulciare doc….

Primo programma Java

```
// Questo commenta fino alla fine della riga
import java.util.*;
public class HelloDate {
  public static void main(String[] args) {
    System.out.println("Hello, it's: ");
    System.out.println(new Date());
  }
}
```

System.out è un oggetto **static PrintStream**

Quali metodi è possibile invocare per un **PrintStream**?

Tra quelli disponibili vi è **println()**….Cosa fa?

Primo programma Java

```
// Questo commenta fino alla fine della riga
import java.util.*;
public class HelloDate {
  public static void main(String[] args) {
    System.out.println("Hello, it's: ");
    System.out.println(new Date());
  }
}
```

Stampa sulla console quello che gli viene passato, e termina con un newline

System.out è un oggetto **static PrintStream**

Quali metodi è possibile invocare per un **PrintStream**?

Tra quelli disponibili vi è **println()**….Cosa fa?

Primo programma Java

```
// Questo commenta fino alla fine della riga
import java.util.*;
public class HelloDate {
  public static void main(String[] args) {
    System.out.println("Hello, it's: ");
    System.out.println(new Date());
  }
}
```

Quindi, con

System.out.println(“stampa”)

possiamo stampare quello che vogliamo a console

Primo programma Java

```
// Questo commenta fino alla fine della riga
import java.util.*;
public class HelloDate {
  public static void main(String[] args) {
    System.out.println("Hello, it's: ");
    System.out.println(new Date());
  }
}
```

In ogni programma Java esiste una classe che contiene il metodo **main()**

Questa classe ha lo stesso nome del file che la contiene

Primo programma Java

```
// Questo commenta fino alla fine della riga
import java.util.*;
public class HelloDate {
  public static void main(String[] args) {
    System.out.println("Hello, it's: ");
    System.out.println(new Date());
  }
}
```

Il metodo **main()** deve avere questa segnatura

Primo programma Java

```
// Questo commenta fino alla fine della riga
import java.util.*;
public class HelloDate {
  public static void main(String[] args) {
    System.out.println("Hello, it's: ");
    System.out.println(new Date());
  }
}
```

public significa che il metodo è visibile al mondo esterno

Primo programma Java

```
// Questo commenta fino alla fine della riga
import java.util.*;
public class HelloDate {
  public static void main(String[] args) {
    System.out.println("Hello, it's: ");
    System.out.println(new Date());
  }
}
```

L'argomento di **main()** è un array di oggetti **String**

Esso contiene gli argomenti passati da linea di comando (non usato in questo esempio)

Primo programma Java

```
// Questo commenta fino alla fine della riga
import java.util.*;
public class HelloDate {
  public static void main(String[] args) {
    System.out.println("Hello, it's: ");
    System.out.println(new Date());
  }
}
```

La linea che stampa la data crea un oggetto di tipo **Date** e lo passa come argomento a **println()**

Verificate la compatibilità dei tipi….nei doc!!

Primo programma Java

```
// Questo commenta fino alla fine della riga
import java.util.*;
public class HelloDate {
  public static void main(String[] args) {
    System.out.println("Hello, it's: ");
    System.out.println(new Date());
  }
}
```

La linea che stampa la data crea un oggetto di tipo **Date** e lo passa come argomento a **println()**

Dopo l'esecuzione di questa linea, **Date** non è più necessario, e il garbage collector può eliminarlo

Verificate la compatibilità dei tipi….nei doc!!

Commenti

- Ci sono due tipi di commenti in Java
  - Il primo ereditato dal C (usato anche in C++)

    ```
    /* commenta tutto
      quello compreso
      fino a ora */
    ```
  - Il secondo dal C++

    ```
    // commenta tutta la linea
    ```

Javadoc

- Una delle buone idee in Java è legata all'uso dei commenti per creare automaticamente la documentazione del codice che si scrive
- Lo strumento che estrae dai commenti la documentazione è il *javadoc*
- Cerca speciali *tag* nella documentazione
- Utilizza anche i nomi di classi e metodi
- L'output di javadoc è un file html
  - Del tutto simile a quello che navigate per la documentazione di Java

Javadoc -- sintassi

- Tutti i comandi javadoc iniziano con /**
  - Terminano normalmente con */
  - Ci sono due modi per usare javadoc
    - Html
    - Doc tags, che sono di due tipi
      - Standalone: comandi che iniziano con @ e sono messi all'inizio di una linea di commento
      - Inline: possono apparire ovunque in un commento javadoc, iniziano con @ ma sono racchiusi da parentesi graffe

Javadoc -- sintassi

- Ci sono tre tipi di commenti legati alla documentazione
- Corrispondono agli elementi che precedono
  - Commenti di classe
  - Commenti di variabile (dato)
  - Commenti di metodo (funzione)
- Javadoc processa solo i commenti legati a membri **public** e **protected**
  - Per quelli **private** bisogna usare il flag **-private**

Javadoc -- sintassi

- Ecco un esempio

```
/** Commento di classe */
public class docTest {
    /** Commento di variabile */
    public int i;
    /** Commento di metodo */
    public void f() {}
}
```

- Viene generato un file html….provare

Javadoc -- html

- Il codice html all'interno di un commento javadoc viene utilizzato per formattare il testo inserito a vostro piacimento
- Non usare headings come **<h1>** o **<hr>**, dato che javadoc inserisce i suoi, e si creerebbero delle interferenze

Javadoc -- tags

- @see
  - Riferisce la documentazione presente in altre classi
    @see nome-classe
    @see nome-classe#nome-metodo
  - Aggiunge “See Also” nella documentazione
  - Javadoc non controlla la validità dei link passati
- {@link *package.class#member* label}
  - Simile al @see
  - Può essere usato in linea, e utilizza label come testo per l’hyperlink (invece di “See Also”)

Javadoc -- tags

- {@docRoot}
  - Genera il path relativo alla directory radice della documentazione
- @version *informazioni*
  - Quando è usato il tag -version, inserisce le *informazioni*
- @author *informazioni*
  - Quando è usato il tag -author, inserisce le *informazioni*
  - Si possono avere più autori, ma i tag devono essere consecutivi

Javadoc -- tags

- @param nome-parametro descrizione
  - Utilizzato per la documentazione dei metodi
  - nome-parametro è l’identificatore di un parametro nella lista dei parametri del metodo
  - descrizione è il testo che lo descrive
    - Può continuare su più righe
  - La descrizione si considera terminata quando si incontra un nuovo tag

Javadoc -- tags

- @return descrizione
  - Descrive il significato del valore di ritorno di un metodo
- @exception nome-eccezione descrizione
  - Descrive le eccezioni generate
- Quelli presentati sono solo alcuni dei tag utilizzabili
  - Controllare la javadoc reference

Esempio -- HelloDate

```
//: c02:HelloDate.java
  import java.util.*;
/** The first Thinking in Java example program.
* Displays a string and today's date.
* @author Bruce Eckel
* @author www.BruceEckel.com
* @version 2.0  */
public class HelloDate {
/** Sole entry point to class & application
* @param args array of string arguments
* @return No return value
* @exception exceptions No exceptions thrown    */
    public static void main(String[] args) {  codice visto prima  }
}///:~
```

Stile di scrittura del codice

- Per rendere le cose più uniformi, sono state introdotte delle convenzioni per la scrittura di codice Java
  - *Code Conventions for the Java Programming Language*
  - http://java.sun.com/docs/codeconv/index.html

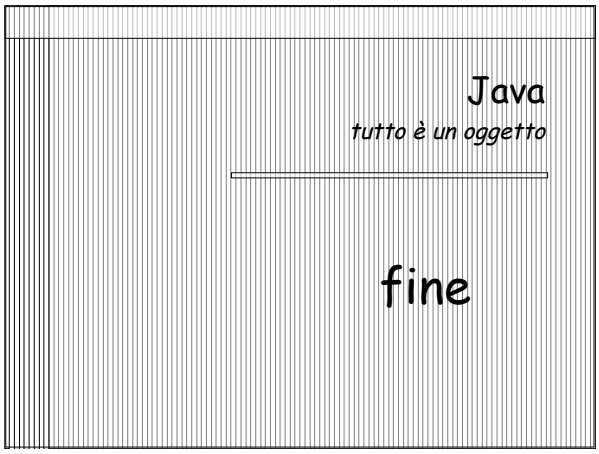
Stile di scrittura del codice

- Rendere maiuscola la prima lettera di un identificatore di classe
  - Se l'identificatore consiste di più parole, esse sono attaccate
    - Cioè, non si usano simboli (come _) per unirle
  - La prima lettera di ogni parola è maiuscola
    - Es. **class** NomeClasseConTanteParole
- Per tutto il resto (metodi, variabili), lo stile accettato è lo stesso di quello usato per le classi, *eccetto* per la prima lettera degli identificatori che è minuscola

```
int unaVariabileIntera;
void unMetodoInutile() {}
```

Esercizi

1. Scrivere un programma che stampi a console "Hello World!!"
2. Dalla classe Dati (Lucido 17) ottenere un programma funzionante
3. Modificare l'esercizio 2 in modo da stampare i valori modificati
4. Scrivere un programma che utilizzi il metodo storage() (Lucido 26)
5. Commentare (javadoc) l'esercizio 1


Java
tutto è un oggetto
fine