Riassunto per me
Programmazione orientata ad oggetti
Java utilizza una programmazione orientata ad oggetti, uno stile programmativo che è stato creato per far si che la programmazione si avvicini al mondo reale come pensiero
Ogni oggetto è una unità indipendente con un'identità UNICA, esattamente come un oggetto del mondo reale
Gli oggetti hanno anche caratteristiche che sono usate per descriverli, un oggetto può essere rosso o blu etc etc, queste caratteristiche sono anche chiamate Attributi. Gli attributi descrivono lo stato attuale dell'oggetto
Ovviamente c'è anche il behavior (comportamento)

Classi
Una classe descrive cosa un oggetto sarà, ma è separato dall'oggetto stesso
In altre parole si può pensare alle classi come blueprints, descrizioni o definizioni per un oggetto. Puoi usare la stessa classe come un blueprint per creare multipli oggetti. Il primo passo è di definire una classe, che allora diverrà un blueprint per la creazione dell'oggetto
Ogni classe ha un nome, ed ognuna è usata per definire attributi e comportamenti
In altre parole: un oggetto è l'istanza di una classe

Metodi
I metodi definiscono il behavior (comportamento). Un metodo è una collezione di dichiarazioni che sono raggruppate insieme per eseguire una operazione.
System.out.println() è un esempio di un metodo
Si possono definire i nostri propri metodi per eseguire un desiderato compito
Esempio

```
class MyClass {

    static void sayHello() {
        System.out.println("Hello World!");
    }

    public static void main(String[ ] args) {
        sayHello();
    }
```

```
}
```
Il codice precedente dichiara un metodo chiamato “sayHello”, che stamperà un testo, che successivamente verrà chiamato nel Main
Per chiamare un metodo bisogna scrivere il suo nome e, successivamente, un set di parentesi

Chiamare Metodi
Tu puoi chiamare un metodo quante volte sono necessarie
Quando un metodo viene eseguito, il codice salta sotto, dove il metodo è definito, eseguendo il codice dentro ad esso, per poi ritornare indietro e procede alla riga successiva
Esempio:
```
class MyClass {

    static void sayHello() {
        System.out.println("Hello World!");
    }

    public static void main(String[ ] args) {
        sayHello();
        sayHello();
        sayHello();
    }
}
```
Bhe potremmo usare anche i cicli for o while

Parametri del metodo
Puoi anche creare un metodo che ottenga dei dati, chiamati parametri, quando viene chiamato. Scrivi i parametri dentro le parentesi del metodo
Esempio:
```
class MyClass {

    static void sayHello() {
        System.out.println("Hello " + name);
    }

    public static void main(String[ ] args) {
        sayHello(“David”);
        sayHello(“Amy);
    }
}
```

I metodi qua sopra ottengono una stringa chiamata name come parametro che è usato nel corpo del metodo
I vantaggi sono il riuso del codice ed a seconda del parametro data i metodi possono avere comportamente diversi

Il tipo Return
La keyword return può essere usata per ritornare un valore

Esempio

```
static int sum (int val1, int val2){
        return val1+val2;
}
```

Notare come noi definiamo un return type prima di definire il nome del metodo
Dello static si parlerà in futuro

Esempio:

```
class Myclass {

        static int sum (int val1, int val2){
                return val1 + val2;
        }

        public static void main(String[] args){
                int x = sum(2, 5);
                System.out.println(x);
        }
}
```

Se non vogliamo mettere la keyword return possiamo mettere la keyword void che non ritornerà nulla
Osserviamo anche che il main ha il void

Creare classi
In modo di creare i tuo personalizzati oggetti, bisogna prima creare la classe corrispondente. Questo è possibile usando il tasto destro del mouse nella cartella src e poi premere “new class”, dargli un nome ed è fatta
Esempio con Animal.java

```
public class Animal {
        void bark(){
                System.out.println(“Woof-Woof”);
        }
}
```

Noi dichiariamo il metodo “bark” nella nostra classe Animale

Ora andiamo nel nostro main ed mettiamo un nuovo oggetto nella nostra classe MyClass.java

```
class MyClass {
        public static void main (String[] args){
                Animal dog = new Animal ();
                dog.bark();
        }
}
```

Ora dog è un oggetto del tipo Animal. Possiamo ovviamente usare bark mettendo il nome dell'oggetto, poi un punto, ed infine il metodo
Il punto è usato per accedere ad gli attributi ed ai metodi dell'oggetto
Abbiamo appena creato il nostro primo oggetto!

Definire attributi
Una classe ha attributi e metodi, Gli attributi sono, di base, variabili con una classe
Creiamo una classe Veicoli, con i corrispondenti attributi e metodi

```
public class Veicolo {
	int maxSpeed;
	int wheels;
	String color;
	double fuelCapacity;

	void horn(){
		System.out.println("Beep!");
	}
}
```

maxSpeed, wheels, color and fuelCapacity sono attributi del nostro veicolo classe, e horn() è il solo metodo

Puoi definire così tutti i metodi e gli attributi necessari

Creando oggetti
ora creiamo multipli veicoli ed accediamo ai loro attributi con il punto

```
class MyClass {
	public static void main (String[] args){
		Veicolo v1 = new Veicolo();
		Veicolo v2 = new Veicolo();
		v1.color = "red";
		v2.horn();
	}
}
```

Accedere ai Modificatori
Ora discutiamo della keyword public che risiede nel metodo main
public static void main (String[] args)

public è un modificatore di accesso, il che significa che è usato per settare un certo livello di accesso

Questi modificatori sono usati per Classi, attributi e metodi
Per le classi:
Public: La classe è accessibile da qualunque altra classe
Default: La classe è accessibile solo dalle classi nello stesso package

Le seguenti scelte sono invece per gli attributi e per i metodi
default: Una variabile, o metodo, dichiarato con nessun access control modifier è disponibile per tutte le altre classi
public: Accessibile da tutte le altre classi
protected: Provvede lo stesso accesso come il default, con l'addizione che le sottoclassi possono accedere ai metodi e variabili delle superclassi (ne parleremo dopo)
private: accessibile solo dentro la classe stessa

```
public class Veicolo {
	private int maxSpeed;
	private int wheels;
	private String color;
	private double fuelCapacity;

	public void horn(){
		System.out.println("Beep!");
	}
}
```

é buon costume tenere le variabili private. Le variabili sono accessibili e modificabili usando i Getters ed i Setters

Getters & Setters
Sono usati per proteggere i tuoi dati, particolarmente quando creiamo classi. Per ogni variabile, il metodo get ritorna il suo valore, mentre il metodo set, "setta" il valore
Esempio pratico (Buona pratica dopo il get/set mettere il nome della variabile)

```
public class Veicolo{
	private String color;

	//Getter
	public String getColor(){
		return color;
	}

	//Setter
	public void setColor(String c){
		this.color = c;
	}
}
```

Il metodo Getter ritorna il valore di tutti gli attributi
Il metodo Setter ottiene un parametro e lo assegna all'attributo

La keyword this è usata per riferirsi all'oggetto attuale. Di base, this.color è l'attributo color dell'attuale oggetto

Ovviamente i Getter ed i Setters si usano, dopo essere definiti nel nostro main

```
public static void main (String[] args) {
        Veicolo v1 = new Veicolo();
        v1.setColor(“Red”);
        System.out.println(v1.getColor());
}
```

I Getter ed i Setter sono blocchi fondamentali per l’incapsulamento

Costrutti
I Costrutti sono metodi speciali invocati quando un oggetto è creato e sono usati per inizializzarli (gli oggetti)
Un costruttore può essere usato per provvedere un valore iniziale per gli attributi degli oggetti
Il nome del costruttore deve essere lo stesso della sua classe
Un costruttore deve non avere un return esplicito

```
public class Veicolo {
        private String color;
        Veicolo () {
                color = “Red”;
        }
}
```

Il veicolo() metodo è il costruttore della nostra classe, quindi qualunque oggetto della classe creato, l’attributo colore sarà “Red”
Un costruttore può anche prendere un parametro per inizializzare attributi

```
public class Veicolo {
        private String color;
        Veicolo(String c){
                color = c;
        }
}
```

Tu puoi pensare ad un costruttore come metodi che verranno settati sulla tua classe di default, quindi non devi ripetere lo stesso codice ogni volta

Usando i Costruttori
Il costruttore è chiamato quando crei un oggetto usando una nuova keyword
Esempio:

```
public class MyClass {
        public static void main (String[] args){
                Veicolo v = new Veicolo(“Blue”);
        }
}
```

Questo chiamerà il costruttore, che setterà il colore dell’attributo in “Blue”

Costruttori
Una singola class può avere multipli costruttori con differenti numeri di parametri
Il metodo setter dentro il costruttore può essere usato per settare il valore dell'attributo

```
public class Veicolo {
	private String color;

	Veicolo(){
		this.setColor(c);
	}
	Veicolo(String c){
		this.setColor(c);
	}

	//Setter
	public void setColor(String c){
		this.color = c;
	}
}
```

La classe sopra ha due 2 costrutti, uno senza qualunque parametro, setta il colore dell'attributo ad un valore di default "Rosso", e un'altro costrutto che accetta un parametro e lo assegna al suo attributo

Usiamo i costruttori per creare gli oggetti

```
//color will be "Red"
Veicolo v1 = new Veicolo();

//color will be "green"
Veicolo v2 = new Veicolo("Green");
```

java provvede in automatico un costruttore di default, così tutte le classi hanno un costruttore, in ogni caso, definito o meno

Tipi di valore
I value types (tipi di valore) sono i tipi base, ed includono byte, short, long, float, double, boolean and char
Questi data types tengono il valore assegnato a loro in corrispondenza della locazione di memoria. Quindi quando gli passi un metodo, tu operi sul valore di una variabile, e non sulla variabile stessa

Esempio

```
public class MyClass {
	public static void main (String[] args){
		int x = 5;
		addOneTo(x);
		System.out.println(x);
	}
```

```
	static void addOneTo(int num) {
		num = num + 1;
	}
}
```

Il metodo dell'esempio sopra prende un valore come suo parametro, che è il motivo per il quale la variabile originale non è affetta e 5 rimane il suo valore
per volere il 6 dovremmo fare:
int y = addOneTo(x);

Reference Types
un reference type memorizza una reference (o un indirizzo) alla locazione di memoria dove il dato corrispondente è memorizzato
Quando crei un oggetto usando un costrutto, tu crei una variabile reference. Immaginiamo di avere una class Person

```
public class MyClass {
	public static void main (String[] args){
		Person j;
		j = new Person("John");
		j.setAge(20);
		celebrateBirthday(j);
		System.out.println(j.getAge());
	}
	static void celebrateBirthday(Person p){
		p.setAge(p.getAge() + 1);
	}
}
```

Il metodo celebrateBirthday prende un oggetto Person come suo parametro, e incrementa il suo attributo
Perchè j è un reference type, il metodo influenza l'oggetto stesso, è in grado di cambiare il valore attuale dell'attributo

Arrays e Stringhe sono anche reference data types

La Classe Math
The JDK definisce un numero di classi utili, una di quelle essendo il Math class, che provvede metodi predefiniti per operazioni matematiche
Esempi dove usiamo Math

Math.abs() ritorna il valore assoluto del parametro passato
int a = Math.abs(-20); // 20

Math.ceil() arrotonda il numero per eccesso
double c = Math.ceil(7.342) // 8.0

Math.floor() arrotonda il numero per difetto

```
double f = Math.floor(7.343);  // 7.0
```

Math.max() ritorna il valore più grande tra i parametri passati

```
int m = Math.max(10, 20);      // 20
```

Math.min() ritorna il valore più piccolo tra i parametri passati

Math.pow() prende 2 parametri e ritorna il primo parametro e ne fa la potenza del secondo parametro

c'è anche sqrt etc etc

Static
Quando dichiariamo una variabile o un metodo come statico, ciò significa che appartiene alla classe, piuttosto che di una specifica istanza. Ciò significa che solo una istanza di un membro statico esiste, anche se crei multipli oggetti della classe, o se non ne crei nessuno. verranno tutti condivisi da tutti gli oggetti
Esempio

```
public class Counter {
	public static int COUNT = 0;
	Counter() {
		COUNT++;
	}
}
```

La variabile COUNT sarà condivisa con tutti gli oggetti di quella classe.
Ora creiamo oggetti della nostra classe Counter in main, e accediamo la variabile statica

```
public class MyClass é
	public static void main(String[] args){
		Counter c1 = new Counter();
		Counter c2 = new Counter();
		System.out.println(Counter.COUNT);
	}
}
```

l'output è 2, perchè la variabile COUNT è statica ed ottiene un incremento ogni volta che un nuovo oggetto della classe Counter è creata. Nel codice sopra, creiamo 2 oggetti. si può anche accedere la variabile statica usando qualunque oggetto di quella classe, come per esempio: c1.COUNT.
è pratica comune mettere le variabile statiche (non sempre necessario)

Lo stesso concetto si applica ai metodi statici
Esempio:

```
public class Veicolo {
	public static void horn(){
		System.out.println("Beep");
	}
```

```
}
```

Ora il metodo horn può essere chiamato senza creare l'oggetto

```
public class MyClass {
	public static void main (String[] args) {
		Veicolo.horn();
	}
}
```

Un altro esempio di metodo statico sono della classe math, che è perchè puoi chiamarli senza creare l'oggetto math

NB: il metodo main deve sempre essere statico

Final
quando usiamo la keyword final per marcare una variabile come costante, quindi puoi assegnare solo una volta
Esempio:

```
class MyClass {
	public static final double PI = 3.14;
	public static void main(String[] args){
		System.out.println(PI);
	}
}
```

PI è ora costante. Ogni tentativo di assegnare un altro valore causerà un errore

Anche i metodi e le classi possono essere final, i metodi non possono essere overriddati e le classi non possono essere subsunti (subclassati)

Packages
I packages sono usati per evitare conflitti con i nomi e per controllare gli accessi delle classi
Un package può essere definito come un gruppo fatto da simili tipi di classi, insieme con i sub-package
tasto destro su src poi New -> Package e dai il nome
nella codice compare

```
package samples;
```

esempio:

```
import samples.Veicolo;

class MyClass {
	public static void main(String[] args){
		Veicolo v1 = new Veicolo();
		v1.horn();
	}
```

}

Encapsulation
Ci sono 4 concetti centrali della programmazione ad oggetti: Incapsulamento, Ereditarietà, polimorfismo ed Astrazione

L'idea dietro incapsulamento è di assicurarsi che i dettagli dell'implementazione siano non visibili agli utenti. Le variabili della classe saranno nascoste dalle altre classi, accessibili solo tramite i metodi della classe corrente. ciò è chiamato data hiding
Per fare l'incapsulazione, dichiara le variabili della classe come privata e provvede un setter ed un getter methods per modificare e vedere i valori delle variabili

Esempio:
```
class BankAccount {
	private double balance = 0;
	public void deposit (double x){
		inf(x > 0) {
			balance += x;
		}
	}
}
```

L'implementazione nasconde la variabile balance, permettendo l'accesso solo tramite il metodo deposito, che valida l'ammontare per essere depositato dopo la modifica della variabile
Vantaggi incapsulamento insomma: Controlla l'accesso e la modifica dei dati, è molto flessibile/semplice da cambiare nel codice, ha l'abilità di cambiare una parte del codice senza influenzare altre parti

Ereditarietà
l'ereditarietà è un processo che permette ad una classe di acquisire le proprietà (I metodi e le variabili) di un'altra. Con l'ereditarietà, l'informazione è posizionata in un ordine più maneggevole ed gerarchico
La classe che eredita le proprietà di un'altra classe è la subclass (o classe figlio); quella classe, che ha le proprietà che sono ereditate, è chiamata superclasse (o classe padre)

Per ereditare da una classe si usa la keyword extends
Esempio:
```
class Dog extends Animal {
	// Some code
}
```
Superclasse: Animal
Sottoclasse: Dog

quando si eredita, si ereditano solo le variabili ed i metodi NON privati della superclasse

```
class Animal {
	protected int legs;
```

```
	public void eat() {
		System.out.println("Animal eats");
	}
}

class Dog extends Animal {
	Dog() {
	legs = 4;
	}
}
```

Come si può vedere, la classe Dog eredita la variabile legs dalla classe animal, ora possiamo dichiarare un oggetto cane e chiamare il metodo eat di una sua superclasse

```
class MyClass {
	public static void main (String[] args){
		Dog d = new Dog ();
		d.eat();
	}
}
```

Ricorda: protected prende visibile solo alle sottoclassi

puoi accedere dalla sottoclasse alla superclasse usando la keyword super
per esempio: super.var

Polimorfismo: si riferisce all'idea di avere "molte forme", questo succede quando c'è una gerarchia delle classi collegate tra loro tramite l'ereditarietà
esempio implementando makeSound()

```
class Animal {
	public void makeSound() {
	System.out.println("Grr...");
	}
}
class Cat extends Animal {
	public void makeSound() {
	System.out.println("Meow");
	}
}
class Dog extends Animal {
	public void makeSound() {
	System.out.println("Woof");
	}
}

public static void main(String[ ] args) {
	Animal a = new Dog();
	Animal b = new Cat();
```

```
        a.makeSound();
        b.makeSound();
}
```

Questo dimostra che possiamo usare la variabile Animal sempre senza sapere che contiene e che il makeSound di a e di b sono diversi
Il polimorfismo è un metodo con implementazioni diverse

Metodo Overriding
Come abbiamo visto nella precedente lezione, una subclass può definire un comportamento che è specifico al tipo della subclass, ovvero che una subclass può implementare un metodo della classe padre basato sulle necessità
Questa possibilità è nota come overriding

```
class Animal {
        public void makeSound() {
        System.out.println("Grr...");
        }
}
class Cat extends Animal {
        public void makeSound() {
        System.out.println("Meow");
        }
}
```

quindi qua Cat overreidiamo il makeSound() method della sua superclasse method
Regole del metodo Overriding
Deve avere lo stesso tipo e gli argomenti
Il livello di accesso non può essere cambiato (quindi se è pubblico, rimane pubblico e via dicendo)
Un metodo Final o Static non può essere overraidato
Se un metodo non può essere ereditato, non può essere overridato
I Costruttori non possono essere Overridati

Il metodo overriding è anche conosciuto come polimorfismo a runtime

Overloading
Quando il nome è lo stesso ma si sono parametri diversi, il metodo viene definito come overloading

per esempio:

```
int max (int a, int b) {
        // Codice
}

double max (double a, double b){
        // Codice
}
```

L’overload avviene in tempo di compilazione (compile-time polymorphism)

Astrazione
L’astrazione dei dati permette al mondo esterno di solo le informazioni essenziali, in un processo di rappresentazione delle features essenziali senza includere i dettagli delle implementazioni
Un esempio può essere un libro, sappiamo che è ma non sappiamo le pagine, il contenuto etc etc
Il concetto di astrazione è che ci concentriamo sulle qualità essenziali e non sullo specifico

in java si raggiunge l’astrazione usando classi astratte ed interfacce
Una classe astratta è definita usando la keyword abstract
Se una classe è dichiarata astratta, allora non può essere istanziata (non puoi creare oggetti di quel tipo)
per usare una classe astratta, devi ereditarla da un’altra classe
Ogni classe che contiene un metodo astratto deve essere definito come astratto
Un metodo astratto è un metodo che è dichiarato senza una implementazione:
tipo: abstract void walk();

Esempio:
```
abstract class Animal {
        int legs = 0;
        abstract void makeSound();
}
```

Il metodo makeSound è anch’esso astratto
La puoi ereditare dalla classe Animal ed implementare

```
class Cat extends Animal {
        public void makeSound(){
                System.out.println(“Meow”);
        }
}
```

Questo esempio è calzante perchè effettivamente viene dimostrato come tutti gli animali hanno un verso, e si implementa per ogni animale un verso.

Interfacce
Un'interfaccia è una completamente classe astratta che contiene solo metodi astratti
Specifiche per interfacce:
Definite usando la keyword interface
Può contenere solo variabili statiche
Può contenere solo variabili static final
Non può contenere costruttori perchè le interfacce non possono essere istanziate
Le interfacce possono essere estese ad altre interfacce
Una classe può implementare qualunque numero di interfacce

Esempio

```
interface Animal {
	public void eat ();
	public void makeSound();
}
```

Un’interfaccia ha le seguenti proprietà:
Un’interfaccia è implicitamente astratta. Non hai bisogno di usare una keyword abstract mentre dichiari una interfaccia
Ogni metodo in una interfaccia è anche implicitamente astratta, quindi la keyword non è necessaria
Metodi in una interfaccia è implicitamente pubblica

Una classe può ereditare da solo una superclasse, ma può implementare multiple interfacce
NB: quando implementiamo una interfaccia, bisogna overraidare tutti i suoi metodi
Ultimo esempio astrazioni:

```
interface Animal {
	public void eat();
	public void makeSound();
}

class Cat implements Animal {
	public void makeSound() {
		System.out.println(“Meow”);
	}
	public void eat() {
		System.out.println(“omnomnom”);
	}
}
```

Type Casting
Assegnare un valore di un tipo di una variabile di un’altro tipo è conosciuto come Type Casting
Esempio pratico di com’è la cosa:

```
int a = (int) 3.14;
System.out.println(a);
```

Insomma verrà trasformato in un int quindi la stampa, stamperà un 3, c’è ovviamente perdita di informazione, anche 3.6 sarebbe 3

Type Casting
Per le classi ci sono 2 tipi di casting
Upcasting
Si può castare una istanza di una sottoclasse nella sua superclasse
Esempio
```
Animal a = new Cat();
```

Java upcasta semplicemente le variabili di tipo di Cat a quelle di Animal

Downcasting
Castare un oggetto di una superclasse ad una sottoclasse è chiamato downcasting
Esempio:
Animal a = new Cat();
((Cat)a).makeSound();

Questo proverà a castare una variabile di tipo Cat e chiamerà makeSound()

Upcasting non può mai fallire è anche per questo che è automatica, al contrario della downcasting

Classi Anonime
Sono una maniera per estendere le classi esistenti al volo
Esempio con la class Machine

```
class Machine {
	public void start() {
		System.out.println(“Starting…”);
	}
}
```

Quando si crea l’oggetto Machine, possiamo cambiare il metodo al volo

```
public static void main(String[] args) {
	Machine m = new Machine() {
		@Override
		public void start(){
			System.out.println(“Woooo”);
		}
	};
	m.start();
}
```

@Override è una annotazione per rendere il tuo codice più veloce da scrivere

Possiamo solo modificare un oggetto per volta, se ne creiamo un altro seguirà l’originale implementazione

Inner Classes
Java support le classis Nesting; Una classe può essere un membro di un’altra classe
Crearla è facile, ah, e deve essere privata. quando la dichiari una inner class private, non può essere acceduta da un oggetto fuori dalla classe

```
class Robot {
	int id;
	Robot (int i) {
```

```
                id = i;
                Brain b =  new Brain();
                b.think();
        }

        private class Brain {
                public void think() {
                        System.out.println (id + " is thinking");
                }
        }
}
```

Brain è una inner class di Robor. Può essere acceduta dentro lo scope

Comparare Oggetti
Quando testi tramite operatore (==), guarda che compare le references e non i valori degli oggetti

```
class Animal {
        String name;
        Animal (String n) {
                name=n;
        }
}

class MyClass {
        public static void main (String[] args){
                Animal a1 = new Animal ("Robby");
                Animal a2 = new Animal ("Robby");
                System.out.println (a1==a2);
        }
}
```

Ritornerà falsa perchè abbiamo 2 oggetti diversi (Che usano diverse reference o locazioni di memoria)

Equals()
Esiste ed è implementabile facendo il tasto destro sulla tua classe e poi: Source -> Generate hashCode() and equals()...
ciò creerà automaticamente i metodi necessari

```
class Animal {
  String name;
  Animal(String n) {
    name = n;
  }
  @Override
  public int hashCode() {
    final int prime = 31;
```

```
    int result = 1;
    result = prime * result + ((name == null) ? 0 : name.hashCode());
    return result;
  }
  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    Animal other = (Animal) obj;
    if (name == null) {
      if (other.name != null)
        return false;
    } else if (!name.equals(other.name))
      return false;
    return true;
  }
}
```

Quando implementi equals, devi anche implementare hashcode
Esempio con equals method

```
public static void main (String[] args) {
        Animal a1 = new Animal("Robby");
        Animal a2 = new Animal("Robby");
        System.out.println(a1.equals(a2));
}
```

Puoi usare lo stesso metodo per i getters ed i settes

Enums
Tipo speciale usato per definire collezioni di costanti
Esempio:

```
enum Rank {
        SOLDIER,
        SERGEANT,
        CAPTAIN
}
```

Essenzialmente sono struct che chiamano i valori interni con il punto, esempio:
Rank a = Rank.SOLDIER;
Si può anche usare il costrutto switch-case

Java API
è una collezione di classi ed interfacce che sono scritte per usarle

Se vuoi vedere la documentazione etc etc:
Allocate qua: **http://docs.oracle.com/javase/7/docs/api/**
I package possono essere importati grazie alla keyword import
Esempio: import java.awt.*;
Il package awt contiene tutte le classi per creare interfacce utente e per disegnare grafiche ed immagini

il carattere * è usato per importare tutte le classi di quel package

Nota bene, le variabili degli oggetti sono memorizzate nelle references

Eccezioni
Una eccezione è un problema che accade durante l'esecuzione del programma. Le eccezione causano la terminazione anormale del programma
Gestire le eccezioni è un meccanismo potente che gestisce gli errori in runtime per mantenere il normale andamento dell'applicazione

Un eccezione può accadere per diverse ragioni:
- Un utente è immesso un dato non valido
- Un file che bisogna aprire non è stato trovato
- Una connessione network è stata persa mentre si comunica
- Memoria insufficiente

Un programma ben scritto deve essere in grado di gestire tutte le possibili eccezioni

Gestire le eccezioni
Le eccezioni possono essere catturate usando una combinazione delle keyword try & catch
Sono piazzate dove potrebbe generarsi l'eccezione
Sintassi:

```
try{
        //some code
} catch (Exception e) {
        //some code to handle errors
}
```

Un catch statement involve dichiara il tipo di eccezione che stai provando di prendere
Se una eccezione accade nel block try, il catch block che segue il try controllato. Se il tipo di eccezione che occorre è nei casi previsti in un catch block, l'eccezione è passata al catch block
Il tipo eccezione può essere usato per catturare tutte le possibili eccezioni

L'esempio sotto dimostra una gestione di eccezione quando provi ad entrare in un indice di un array che non esiste

```
public class MyClass {
        public static void main (String[] args){
                try{
```

```
                int a[] = new int[2]
                System.out.println(a[5]);
        } catch (Exception e) {
                System.out.println (“An error occurred”);
        }
    }
}
```

Senza il try/catch il programma si terminerebbe perchè a[5] non esiste

Se scrivi (Exception e) nel catch block: lo usi per coprire tutte le possibili eccezioni

Throw
La keyword throw permette di generare manualmente eccezioni sui tuoi metodi
exception type famosi: IndexOutOfBoundsException, IllegalArgumentException, ArithmeticException e cosi via
Possiamo provare a lanciare un ArithmeticException nel nostro metodo quando il parametro è 0

```
int div(int a, int b) throws ArithmeticException {
    if (b == 0){
        throw new ArithmeticException (“Division by Zero”){
    }
    else {
        return a/b;
    }
}
```

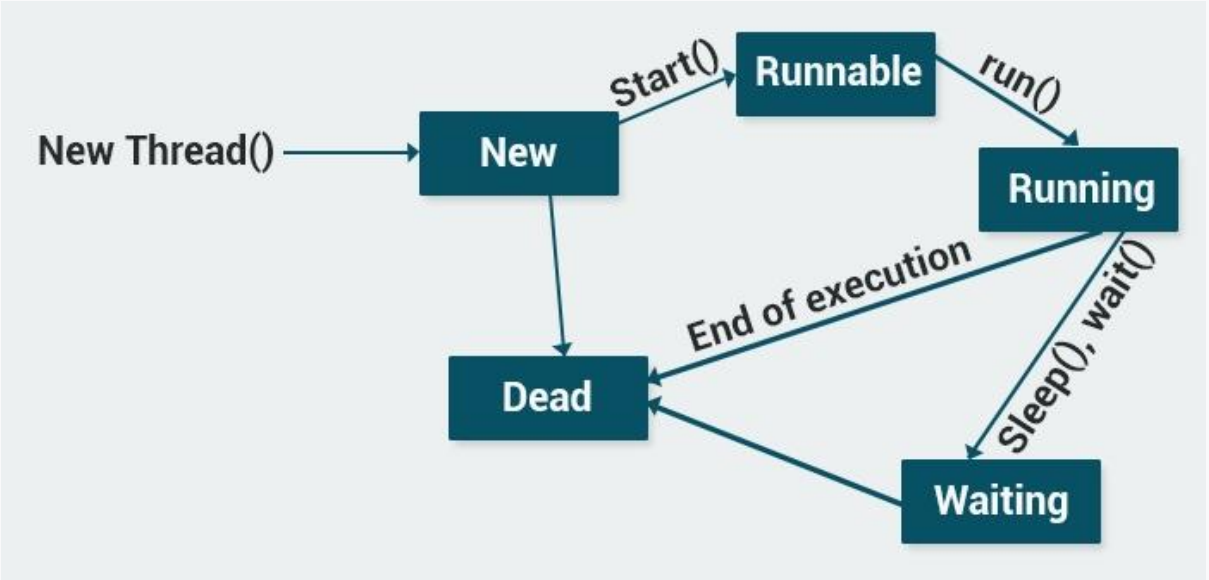
Lo statement throws nella definizione definito il tipo di exception il metodo può essere lanciato
Se chiamo il metodo div con il parametro 0, allora manderemo un ArithmeticException
Eccezione multiple possono essere definiti nello statement throws usando un comma-separated list

Un singolo try block può contenere multipli catch block che gestiscono diverse eccezioni separatamente
Esempio:

```
try {
    //some code
} catch (ExceptionType1 e1){
    //catch block
} catch (ExceptionType2 e2){
    //catch block
} etc etc
```

I blocchi dovrebbero essere ordinati dal più specifico al più generale
Seguendo le specifiche eccezioni, puoi usare il tipo exception per gestire tutte le altre eccezioni come il last catch

Threads
Java è un linguaggio di programmazione multi-thread. Questo significa che il programma può fare un uso ottimale delle risorse disponibili eseguendo due o più componenti contemporaneamente, con ogni componente che gestisce un'attività diversa.
È possibile suddividere operazioni specifiche all'interno di una singola applicazione in singoli thread che vengono eseguiti tutti in parallelo
Il diagramma seguente mostra il ciclo di vita di un thread



Ci sono 2 maniere per creare un thread

1. Estendere la classe Thread

Ereditare dalla classe thread, overridare il suo run method, e scrivere la funzionalità di un thread nel metodo run ()
Dopo crei un nuovo oggetto della tua classe e chiamate il metodo start per runnare un thread
Esempio:

```
class Loader extends Thread {
        public void run() {
                System.out.println(“Hello”);
        }
}

class MyClass {
        public static void main (String[] args) {
                Loader obj = new Loader();
                obj.start();
        }
}
```

Come puoi vedere la nostra classe Loader estende il la classe Thread ed overraida il metodo run()

Quando creiamo l'oggetto obj e chiamiamo il metodo start(), il metodo run() statement esegue in un diverso thread

Ogni thread di java è prioritizzato per aiutare il sistema operativo a determinare l'ordine con il quale si schedulano i thread
Puoi usare il setPriority() method per modificare la priorità

L'altra maniera per creare i thread è implementando l'interfaccia Runnable
Implementa il metodo run(). Dopo crea un nuovo oggetto thread, passata la classe runnable al suo costruttore, ed inizia il thread chiamando il metodo start()

Esempio:

```
class Loader implements Runnable {
        public void run() {
                System.out.println("Hello");
        }
}

class MyClass {
        public static void main (String[] args){
                Thread t = new Thread (new Loader());
                t.start();
        }
}
```

Il metodo Thread.sleep() mette il pausa un thread per uno specifico lasso di tempo. Per esempio, chiamando Thread.sleep(1000); Mette in pausa il thread per un secondo. Tenendo in mente che Thread.sleep() lancia un InterruptedException, quindi bisogna assicurarsi di metterlo in mezzo ad un try/catch

Può essere più complesso questo metodo ma è preferibile questa maniera per ti permette di estende anche un'altra classe

Tipi di eccezioni
Ci sono 2 tipi di eccezioni, checked ed unchecked (anche chiamata runtime)
La differenza principale è che la checked exception sono controllate quando compilate, mentre le unchecked exception sono controllate a runtime
Come detto prima, Thread.sleep() lancia una InterruptedException. Questo è un esempio di una exception checked. Il tuo codice non compilerà finché non avrai gestito l'eccezione

Esempio:

```
public class MyClass {
        public static void main (String[] args){
                try{
                        Thread.sleep(1000);
                } catch (interruptedException e) {
                        //some code
                }
```

```
        }
}
```

Noi abbiamo già visto esempi di eccezioni unchecked, che sono controllate a runtime
Esempio (quando provi a dividere per 0):

```
public class MyClass {
        public static void main (String[] args){
                int value = 7;
                value = value / 0;
        }
}
/*
Exception in thread “main”
java.lang.ArithmeticException: /by zero
        at MyClass.main(MyClass.java:4)
*/
```

è una buona cosa sapere i tipi di eccezione così si può debuggare il codice più velocemente

ArrayList
L’API di java provvede a speciali classi per memorizzare e manipolare gruppi di oggetti
Una di queste classi è l’ArrayList. Gli array standard di Java hanno una lunghezza fissa, il che significa che dopo che sono creati non possono essere espansi o resi più piccoli
D’altro canto, gli ArrayLists sono creati con una dimensione iniziale, ma quando la dimensione viene aumentata, allora la collezione si aumenta automaticamente
Quando un oggetto è rimosso, l’ArrayList diminuirà di dimensione
è da notare che la classe ArrayList è nel package java.util, quindi è necessario importare prima di usarlo
Crea una ArrayList come faresti con un qualsiasi oggetto
Ovvero:

```
import java.util.ArrayList;
//….
ArrayList colors = new ArrayList();
```

Puoi opzionalmente specificare una capacità ed un tipo di oggetti che l’arraylist dovrà contenere

```
ArrayList<String> colors = new ArrayList<string> (10);
```

Il codice sopra definisce un ArrayList di String con 10 come sua dimensione iniziale

Un ArrayList contiene oggetti. Il tipo specifico deve essere di tipo classe. Non puoi passare, per esempio, int come tipo dell’oggetto. Invece, usi speciali tipi di classe che corrispondono al desiderato valore tipo, come Integer for int o Double for double

La classe ArrayList provvede ad un numero di utili metodi per manipolare i suoi oggetti
Il metodo add() aggiunge nuovi oggetti all’ArrayList. Al contrario, remove() rimuove oggetti dall’ArrayList

```
import java.util.ArrayList;

public class MyClass {
	public static void main (String[] args){
		ArrayList<String> colors = new ArrayList<String>();
		colors.add(“Red”);
		colors.add(“Blue”);
		colors.add(“Green”);
		colors.add(“Orange”);
		colors.remove(“Green”);

		System.out.println(colors);
	}
}
```

Altri metodi utili sono i:

- contains(): ritorna true se nella lista è contenuto il valore specificato
- get(int index): ritorna l’elemento di una specifica posizione nella lista
- size(): Ritorna il numero di elementi nella lista
- clear(): Rimuove tutti gli elementi dalla lista

Nota: come gli array, l’indice parte da 0

LinkedList
è molto simile, come sintassi, agli ArrayList
Puoi cambiare facilmente un ArrayList in una LinkedList cambiando il tipo dell’oggetto
Esempio

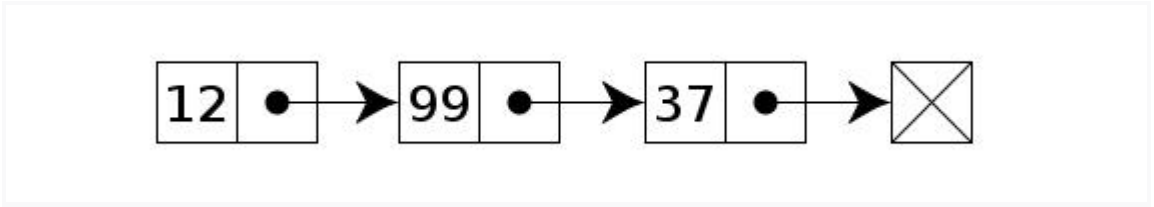
```
import java.util.LinkedList;

public class MyClass {
	public static void main(String[] args){
		LinkedList<String> c = new LinkedList<String>();
		c.add(“Red”);
		c.add(“Blue”);
		c.add(“Green”);
		c.add(“Orange”);
		c.remove(“Green”);
		System.out.println(c);
	}
}
```

Non si può specificare una iniziale capacità per le linkedlist

LinkedList vs ArrayList
La differenza tra LinkedList ed ArrayList è la maniera con la quale vengono memorizzati gli oggetti
L’ArrayList è migliore per memorizzare ed accedere ai dati, è molto simile ad un normale array

La LinkedList è migliore per manipolare i dati, come ad esempio facendo numerosi inserimenti e cancellazioni

In Addizione per memorizzare l'oggetto, la linkedlist memorizza l'indirizzo di memoria (o link) di un elemento che la segue. è chiamata una LinkedList perchè ogni elemento contiene un link all'elemento vicino



Puoi usare il ciclo for per iterare i suoi elementi

```
LinkedList<String> c = new LinkedList<String>();
c.add("Red");
c.add("Blue");
c.add("Green");
c.add("Orange");
c.remove("Green");

for (String s: c) {
        System.out.println(s);
}
```

Riassumendo:

- Usa un ArrayList quando devi avere un accesso rapido ai tuoi dati
- Usa una LinkedList quando devi fare un grande numero di inserimenti e/o cancellazioni

HashMap
Array e Liste memorizzano elementi in una collezione ordinata con ogni elemento che ha un indice
HashMap è usato per memorizzare collezioni di dati come coppia di "chiave" e "valore". Un oggetto è usato come "chiave" (index) mentre un altro oggetto è usato come "valore" (value)
La put, remove e get sono usati per aggiungere, eliminare ed accedere ai valori nella HashMap

Esempio:

```
import java.util.HashMap;
public class MyClass {
        public static void main (String[] args) {
                HashMap<String, Integer> points = new HashMap<String, Integer> ();
                points.put("Amy", 154);
                points.put("Dave", 42);
                points.put("Rob", 733);
```

```
                System.out.println(points.get8”Dave”));
        }
}
```

Abbiamo creato una HashMap con Stringhe come chiavi ed integers come suoi valori
Usiamo il metodo get e la corrispondente key per accedere agli elementi della HashMap

Una HashMap non può contenere chiavi duplicate. Aggiungendo un nuovo oggetto con una chiave che esiste già sovrascrive il vecchio elemento
La classe HashMap provvede i metodi containsKey e containsValue che determinano la presenza di una chiave o valore specifica/o
Se provi a prendere una valore che non è presente nella tua map, ti ritornerà il valore NULL

NULL è un tipo speciale che rappresenta l’assenza di valore

Sets
Un Set è una collezione che non può contenere elementi duplicati. Modella l’astrazione dell’insieme matematico
Una delle implementazioni del Set è la classe HashSet
Esempio

```
import java.util.HashSet;

public class MyClass {
        public static void main(String[] args){
                HashSet<String> set = new HashSet<String>();
                set.add(“A”);
                set.add(“A”);
                set.add(“A”);
                System.out.println(set);
        }
}
```

Puoi usare il metodo size() per prendere il numero di elementi nella HashSet

LinkedHashSet
La classe HashSet non conserva automaticamente l’ordine degli elementi come sono stati aggiunti
Per ordinarli usiamo una LinkedHashSet, che mantiene una linked list degli elementi di set nell’ordine nel quale sono stati inseriti

Cos’è l’hashing?
Una Hash table memorizza informazioni tramite un meccanismo chiamato hashing, nel quale un contenuto informazionale della chiave è usata per determinare un unico valore chiamato un hash code
Quindi, di base, ogni elemento in un HashSet è associato con un unico hash code

Sorting Lists

Per la manipolazione di dati in diversi tipi collection, la java API provvede una classe Collections, che è incluso nel java.util package
Ci sta anche il metodo sort(), che ordina gli elementi della tua collezione. I metodi nella Collections sono statici, non hai bisogno di un oggetto Collections per chiamarli

Esempio

```
import java.util.ArrayList;
import java.util.Collections;

public class MyClass {
        public static void main(String[] args) {
                ArrayList<String> animals = new ArrayList<String>();
                animals.add(“tiger”);
                animals.add(“cat”);
                animals.add(“snake”);
                animals.add(“dog”);

                Collections.sort(animals);

                System.out.println(animals);
        }
}
```

Gli elementi verranno ordinati alfabeticamente
Puoi usarlo anche per una ArrayList di numeri
Altri metodi utili:

- Max (Collection c)
- Min(Collection c)
- reverse (List list)
- shuffle (List list)

Iterators
Un Iterator è un oggetto che permette di ciclare attraverso ad una collezione, ottenere o rimuovere elementi
Prima bisogna accedere alla collezione tramite un iteratore, quindi bisogna ottenerne uno ogni classe collections ha un metodo iterator() che ritorna un iterator all’inizio della collezione. Usando questo oggetto iterator, puoi accedere ad ogni elemento della collection, un elemento alla volte

La classe iterator ti da i seguenti metodi:

- hasNext(): ritorna true se ci sta almeno un elemento dopo, sennò false
- next(): ritorna il prossimo oggetto ed avanza l’iteratore
- remove(): Rimuove l’ultimo oggetto che è stato ritornato da next dalla collezione

La classe Iterator deve essere importata dal java.util package
Esempio

```
import java.util.Iterator;
import java.util.LinkedList;
```

```
public class MyClass {
	public static void main(String[] args){
		LinkedList<String> animals = new LinkedList<String> ();
		animals.add(“fox”);
		animals.add(“cat”);
		animals.add(“dog”);
		animals.add(“rabbit”);

		Iterator<String> it = animals.iterator();
		String value = it.next();
		System.out.println(value);
	}
}
```

it.next() ritorna il primo elemento nella lista ed allora muove l’iteratore sul prossimo elemento
Ogni volta che chiami it.next(), l’iteratore si muove nel prossimo elemento della lista

Tipicamente gli iterators sono usati in loop

Esempio pratico

```
import java.util.Iterator;
import java.util.LinkedList;

public class MyClass {
	public static void main(String[] args){
		LinkedList<String> animals = new LinkedList<String> ();
		animals.add(“fox”);
		animals.add(“cat”);
		animals.add(“dog”);
		animals.add(“rabbit”);

		Iterator<String> it = animals.iterator();
		while(it.hasNext()){
			String value = it.next();
			System.out.println(value);
		}
	}
}
/*
fox
cat
dog
rabbit
*/
```

Qui, il loop while va dal primo elemento alla fine, scorrendo ogni elemento