La classe Object è la superclasse di tutte le classi in Java. Questo significa che ogni classe in Java, direttamente o indirettamente, eredita da Object. È una classe fondamentale nel linguaggio, poiché fornisce metodi comuni che tutte le classi possono utilizzare.

Caratteristiche principali della classe Object

1. **Superclasse di tutte le classi: La classe Object è la radice della gerarchia delle classi in Java. Qualunque classe tu definisca in Java, ereditando da essa, avrà automaticamente accesso ai metodi della classe Object.**

Esempio:

```
class MyClass extends Object {
   // MyClass è una sottoclasse di Object
}
```

2. **Metodi principali: La classe Object fornisce una serie di metodi generali che sono disponibili per tutte le classi in Java. Questi metodi possono essere sovrascritti (override) nelle sottoclassi per comportamenti personalizzati. Ecco alcuni dei metodi principali:**
    - **toString(): Questo metodo restituisce una rappresentazione in formato stringa dell'oggetto. Di default, restituisce una stringa che contiene il nome della classe e l'indirizzo di memoria dell'oggetto, ma molte classi sovrascrivono questo metodo per fornire informazioni più significative sull'oggetto.**
    - **@Override**
    - **public String toString() {**
    - **return "MyClass Object";**

}

    - **equals(Object obj): Questo metodo confronta l'oggetto corrente con un altro oggetto passato come parametro. La versione predefinita in Object confronta gli indirizzi di memoria (cioè verifica se gli oggetti sono la stessa istanza), ma le classi spesso sovrascrivono questo metodo per effettuare confronti basati sul contenuto degli oggetti.**
    - **@Override**
    - **public boolean equals(Object obj) {**
    - **if (this == obj) return true;**
    - **if (obj == null || getClass() != obj.getClass()) return false;**
    - **MyClass myClass = (MyClass) obj;**

- **return this.someField == myClass.someField;**

}

- **hashCode(): Restituisce un valore intero che rappresenta un "codice hash" dell'oggetto. Il codice hash viene utilizzato dalle strutture dati come HashMap o HashSet per identificare rapidamente gli oggetti. Se si sovrascrive equals(), è buona pratica sovrascrivere anche hashCode() per garantire la coerenza.**
- **@Override**
- **public int hashCode() {**
- **return Objects.hash(someField);**

}

- **getClass(): Restituisce un oggetto di tipo Class che rappresenta la classe dell'oggetto corrente. È utile per ottenere informazioni sulla classe di un oggetto a runtime.**
- **Class<?> clazz = obj.getClass();**

System.out.println(clazz.getName());

- **clone(): Crea e restituisce una copia dell'oggetto. La classe Object fornisce una versione predefinita di questo metodo, ma è necessario implementare l'interfaccia Cloneable per utilizzarlo correttamente. Se non implementata, il metodo clone() lancerà un'eccezione CloneNotSupportedException.**
- **@Override**
- **protected Object clone() throws CloneNotSupportedException {**
- **return super.clone();**

}

- **finalize(): Viene chiamato dal garbage collector prima che un oggetto venga eliminato dalla memoria. È una specie di "metodo di pulizia" che può essere utilizzato per liberare risorse (come file aperti o connessioni di rete) prima che l'oggetto venga distrutto. Tuttavia, questo metodo è stato deprecato nelle versioni recenti di Java.**
- **@Override**
- **protected void finalize() throws Throwable {**
- **// Pulizia delle risorse**
- **super.finalize();**

}

- **wait(), notify(), notifyAll(): Questi metodi sono utilizzati per la gestione della concorrenza e del multithreading. Vengono invocati sugli oggetti che agiscono come monitor per il sincronismo tra i thread.**
- **synchronized (obj) {**
- **obj.wait();**

}

Esempio di utilizzo della classe Object

```
public class Person extends Object {
   private String name;
   private int age;

   public Person(String name, int age) {
      this.name = name;
      this.age = age;
   }

   @Override
   public String toString() {
      return "Person{name='" + name + "', age=" + age + "}";
   }

   @Override
   public boolean equals(Object obj) {
      if (this == obj) return true;
      if (obj == null || getClass() != obj.getClass()) return false;
      Person person = (Person) obj;
      return age == person.age && name.equals(person.name);
```

```
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, age);
    }

    public static void main(String[] args) {
        Person p1 = new Person("Alice", 30);
        Person p2 = new Person("Alice", 30);

        System.out.println(p1.toString());  // Person{name='Alice', age=30}
        System.out.println(p1.equals(p2));  // true
        System.out.println(p1.hashCode()); // Un codice hash basato su name e age
    }
}
```

Perché è importante la classe Object?

1. **Ereditarietà universale: Poiché tutte le classi in Java ereditano da Object, ogni oggetto in Java ha accesso ai metodi di Object (come toString(), equals(), hashCode(), ecc.).**
2. **Polimorfismo: I metodi di Object sono fondamentali per il polimorfismo, poiché permettono di trattare oggetti di tipo diverso in modo uniforme. Puoi, ad esempio, scrivere un codice che accetta qualsiasi oggetto (di tipo Object) e utilizzare i suoi metodi, come toString(), senza sapere il tipo esatto dell'oggetto.**

Conclusione

La classe Object è centrale nel design di Java, poiché fornisce metodi di base per il confronto, la visualizzazione e la gestione della memoria degli oggetti. Sebbene molte di queste funzionalità possano essere sovrascritte dalle sottoclassi per un comportamento più specifico, tutte le classi in Java ereditano da Object e beneficiano dei suoi metodi.