7.1 Polimorfismo, Classi Astratte e Generics in Java

Questo capitolo esplora in dettaglio i concetti di polimorfismo, classi astratte e la gestione dei tipi a run-time in Java. Inoltre, introduce i Generics, una funzionalità chiave per la creazione di codice riutilizzabile e sicuro.

2.1. Polimorfismo Inclusivo con le Classi

Il polimorfismo inclusivo, o *subtyping*, è un pilastro della programmazione orientata agli oggetti che permette a un oggetto di essere trattato come un'istanza di diversi tipi, purché questi siano correlati da una relazione di ereditarietà.

2.1.1. Ereditarietà e Principio di Sostituibilità

Il **principio di sostituibilità** (o principio di Liskov Substitution) è fondamentale per il polimorfismo:
"SE B è un sottotipo di A, ALLORA ogni oggetto di B può/deve poter essere utilizzabile dove ci si attende un oggetto di A."

- *Provenienza*: 12-polymorphism_slides.pdf, Pagina 5

In Java, la relazione di ereditarietà (`class B extends A { ... }`) stabilisce che la classe `B` è un sottotipo di `A`. Questo significa che `B` eredita tutti i membri (campi e metodi) di `A` e non può restringerene la visibilità. Di conseguenza, un oggetto di tipo `B` può essere utilizzato ovunque sia richiesto un oggetto di tipo `A`, senza causare problemi di tipo.

Esempio:
Se abbiamo una classe `C` e le sue sottoclassi `C1`, `C2`, ecc., in un contesto (`D`) che si aspetta un oggetto di tipo `C`, possiamo passare un oggetto di `C1` o `C2`.

```
class C {
   public void foo() { /* ... */ }
}

class D { // Rappresenta un generico "contesto"
   C c;
   public void m(C c) { c.foo(); }
}

class C1 extends C {}
class C2 extends C {}

// Uso
D d = new D();
d.c = new C1(); // OK: un C1 è un C
d.m(new C2());  // OK: un C2 è un C
```

- *Provenienza*: 12-polymorphism_slides.pdf, Pagina 7

Le sottoclassi di `C` (come `C1`, `C2`) sono compatibili con `C` perché supportano lo stesso contratto (e spesso operazioni aggiuntive), hanno lo stesso stato (campi definiti in `C`, più eventuali altri) e un comportamento compatibile.

2.1.2. Layout degli Oggetti in Memoria

Sebbene le JVM possano avere implementazioni leggermente diverse, la struttura generale di un oggetto in memoria è comune e supporta la sostituibilità:

- **Intestazione**: Ogni oggetto inizia con un'intestazione (circa 16 byte) ereditata da `Object`, che include informazioni sul tipo a run-time dell'oggetto e una tabella di puntatori ai metodi per supportare il *late-binding* (chiamata dinamica dei metodi).
- **Campi**: Seguono tutti i campi della classe, a partire da quelli delle superclassi.

Questa organizzazione rende un oggetto di una sottoclasse simile a un oggetto della sua superclasse, con informazioni aggiuntive in coda, facilitando la sostituibilità.

- *Provenienza*: 12-polymorphism_slides.pdf, Pagina 8

2.1.3. Esempio di Polimorfismo: `Person`, `Student`, `Teacher`

Consideriamo una gerarchia di classi per modellare persone, studenti e insegnanti:

- `Person`: Classe base con nome e ID.
- `Student`: Estende `Person`, aggiungendo l'anno di immatricolazione.
- `Teacher`: Estende `Person`, aggiungendo un array di corsi.

```
// Person.java
public class Person {
    private final String name;
    private final int id;

    public Person(final String name, final int id) {
        this.name = name;
        this.id = id;
    }
    public String getName() { return this.name; }
    public int getId() { return this.id; }
    public String toString() { return "P [name=" + this.name + ", id=" + this.id + "]"; }
}

// Student.java
public class Student extends Person {
    final private int matriculationYear;

    public Student(final String name, final int id, final int matriculationYear) {
        super(name, id);
        this.matriculationYear = matriculationYear;
    }
    public int getMatriculationYear() { return matriculationYear; }
    public String toString() {
        return "S [getName()=" + getName() + ", getId()=" + getId() + ", matriculationYear=" + matriculationYear + "]";
    }
}

// Teacher.java
import java.util.Arrays;
public class Teacher extends Person {
    final private String[] courses;

    public Teacher(final String name, final int id, final String[] courses) {
        super(name, id);
        this.courses = Arrays.copyOf(courses, courses.length);
    }
    public String[] getCourses() { return Arrays.copyOf(courses, courses.length); } // Copia difensiva
    public String toString() {
        return "T [getName()=" + getName() + ", getId()=" + getId() + ", courses=" + Arrays.toString(courses) + "]";
    }
}

// UsePerson.java (per dimostrare il polimorfismo)
public class UsePerson {
    public static void main(String[] args) {
        final var people = new Person[]{
            new Student("Marco Rossi", 334612, 2013),
```

```
            new Student("Gino Bianchi", 352001, 2013),
            new Student("Carlo Verdi", 354100, 2012),
            new Teacher("Mirko Viroli", 34121, new String[]{"OOP", "PPS", "PC"})
        };
        for (final var p : people) {
            System.out.println(p.getName() + ": " + p); // Chiamata polimorfica a toString()
        }
    }
}
```

- *Provenienza*: 12-polymorphism_slides.pdf, Pagina 9-13

In `UsePerson`, un array di `Person` può contenere istanze di `Student` e `Teacher`. Quando si itera sull'array e si chiama `toString()`, viene invocata la versione specifica del metodo per il tipo a run-time dell'oggetto (polimorfismo).

2.1.4. Differenze: Polimorfismo con Interfacce vs. Classi (Ereditarietà Multipla)

Il polimorfismo può essere ottenuto sia tramite ereditarietà di classi che tramite implementazione di interfacce. Esistono differenze chiave:

Caratteristica	Polimorfismo con Interfacce	Polimorfismo con Classi (Ereditarietà)
Contratto	La classe aderisce a un contratto (metodi da implementare).	La classe eredita contratto e comportamento.
Comportamento	Non si assume un comportamento specifico, solo un contratto.	Si eredita il comportamento della superclasse.
Ereditarietà Multipla	**Possibile** (una classe può implementare più interfacce).	**Non possibile** in Java (una classe può estendere una sola classe).
Problemi Comuni	Nessun problema di "Diamond Problem" o "Triangle Problem".	Problemi se più superclassi hanno proprietà comuni.
Flessibilità	Facilita l'adesione al contratto per classi esistenti.	Generalmente si aderisce per costruzione fin dall'inizio.

- *Provenienza*: 12-polymorphism_slides.pdf, Pagina 14-15

Java non supporta l'ereditarietà multipla di classi (`class C extends D1, D2, ... { ... }`) per evitare problemi di ambiguità (es. quale implementazione ereditare se `D1` e `D2` hanno lo stesso metodo) e complessità nella gestione della memoria.

2.1.5. Simulazione dell'Ereditarietà Multipla

Sebbene Java non permetta l'ereditarietà multipla di classi, è possibile simularla combinando ereditarietà singola e **delegazione** (composizione).

Esempio: Contatori
Se abbiamo `CounterImpl`, `MultiCounterImpl` (che estende `CounterImpl` e implementa `MultiCounter`), e `BiCounterImpl` (che estende `CounterImpl` e implementa `BiCounter`), e vogliamo una classe `BiAndMultiCounterImpl` che implementi sia `BiCounter` che `MultiCounter`, non possiamo estendere entrambe le classi di implementazione.

La soluzione è estendere una delle implementazioni (es. `BiCounterImpl`) e delegare le funzionalità dell'altra (es. `MultiCounterImpl`) a un oggetto composto.

```
// Interfacce
interface Counter { void increment(); int getValue(); }
interface MultiCounter extends Counter { void multiIncrement(); }
interface BiCounter extends Counter { void decrement(); }
interface BiAndMultiCounter extends MultiCounter, BiCounter { }

// Implementazioni
class CounterImpl implements Counter { /* ... */ }
class MultiCounterImpl extends CounterImpl implements MultiCounter { /* ... */ }
class BiCounterImpl extends CounterImpl implements BiCounter { /* ... */ }

// Simulazione dell'ereditarietà multipla tramite estensione e delegazione
```

```
class BiAndMultiCounterImpl extends BiCounterImpl implements BiAndMultiCounter {
    private final MultiCounterImpl multiCounter = new MultiCounterImpl(); // Delegazione

    // Costruttore che inizializza anche l'oggetto delegato
    public BiAndMultiCounterImpl() {
        super(); // Chiama il costruttore di BiCounterImpl
    }

    @Override
    public void multiIncrement() {
        multiCounter.multiIncrement(); // Delega la chiamata a multiCounter
    }

    // Altri metodi di BiCounterImpl sono ereditati
    // Altri metodi di MultiCounterImpl devono essere delegati o implementati
}
```

- *Provenienza*: 12-polymorphism_slides.pdf, Pagina 16-17

Questo approccio garantisce la corretta sostituibilità tramite le interfacce e un buon riuso delle implementazioni, anche se aumenta la complessità del codice.

2.2. Tipi a Run-time

In Java, la filosofia "Everything is an Object" (tutto è un oggetto) è centrale. Questo significa che tutte le classi, direttamente o indirettamente, ereditano dalla classe `java.lang.Object`.

2.2.1. La Radice Comune `Object` e `Object[]`

La classe `Object` funge da radice comune per tutte le classi, consentendo di fattorizzare comportamenti comuni a ogni oggetto e di costruire funzionalità che operano su qualsiasi tipo di oggetto.

Esempio: `Object[]`
Un array di tipo `Object[]` può contenere oggetti di natura diversa, grazie al polimorfismo.

```
import java.util.Arrays;
public class AObject {
    public static void main(String[] s) {
        final Object[] os = new Object[5];
        os[0] = new Object();
        os[1] = "stringa";
        os[2] = Integer.valueOf(10); // Autoboxing
        os[3] = new int[] { 10, 20, 30 };
        os[4] = new java.util.Date();
        printAll(os);
        System.out.println(Arrays.toString(os)); // Stampa la rappresentazione di default
        System.out.println(Arrays.deepToString(os)); // Stampa ricorsivamente il contenuto degli array
    }
    public static void printAll(final Object[] array) {
        for (final Object o : array) {
            System.out.println("Oggetto:" + o.toString());
        }
    }
}
```

- *Provenienza*: 12-polymorphism_slides.pdf, Pagina 19-20

2.2.2. Autoboxing e Unboxing dei Tipi Primitivi

Java fornisce un meccanismo chiamato **autoboxing** e **unboxing** per supportare l'equivalenza tra tipi primitivi e i loro Wrapper (classi come `Integer`, `Double`, `Boolean`).

- **Autoboxing**: Se un primitivo è atteso dove si attende un oggetto, Java lo "impacchetta" automaticamente in un oggetto Wrapper.
- **Unboxing**: Se un Wrapper è atteso dove si attende un primitivo, Java lo "disimpacchetta" automaticamente nel valore primitivo.

Questo meccanismo rende i tipi primitivi utilizzabili in contesti che richiedono oggetti (es. `Object[]`), simulando meglio l'idea "Everything is an Object".

Esempio:

```
public class Boxing {
    public static void main(String[] s) {
        final Object[] os = new Object[5];
        os[1] = 5; // Autoboxing: int 5 diventa new Integer(5)
        os[3] = true; // Autoboxing: boolean true diventa new Boolean(true)

        final Integer[] is = new Integer[] { 10, 20, 30, 40 };
        final int i = is[0] + is[1] + is[2] + is[3]; // Unboxing: gli Integer vengono convertiti in int per la somma
        System.out.println("Somma: " + i); // Output: Somma: 100
    }
}
```

- *Provenienza*: 12-polymorphism_slides.pdf, Pagina 41-42

2.2.3. Tipo Statico e Tipo a Run-time

Questa dualità è introdotta dal polimorfismo inclusivo:

- **Tipo Statico**: Il tipo di dato di una variabile così come dichiarata nel codice (determinato a compile-time).
- **Tipo a Run-time (o Dinamico)**: Il tipo di dato effettivo del valore o dell'oggetto presente in memoria (determinato a run-time). Se il tipo a run-time è un sottotipo del tipo statico, le chiamate ai metodi sono risolte tramite *late-binding* (dispatch dinamico).
- *Provenienza*: 12-polymorphism_slides.pdf, Pagina 21

Esempio:
Nel metodo `printAll(Object[] array)` dell'esempio `AObject`, il tipo statico di `o` all'interno del ciclo `for` è `Object`, ma il suo tipo a run-time varia (`Object`, `String`, `Integer`, ecc.) a seconda dell'elemento corrente.

2.2.4. Ispezione del Tipo a Run-time: `instanceof` e Conversioni (Cast)

In alcuni casi, è necessario ispezionare il tipo a run-time di un oggetto per eseguire operazioni specifiche di quel tipo. Questo si fa con l'operatore `instanceof` e il *cast*.

- `instanceof`: Verifica se un oggetto è un'istanza di un determinato tipo (o di una sua sottoclasse). Restituisce `true` o `false`.
- **Cast (Conversione di Tipo)**:
  - **Upcast**: Da sottoclasse a superclasse (spesso automatico e sicuro).
  - **Downcast**: Da superclasse a sottoclasse (potrebbe fallire a run-time se l'oggetto non è realmente del tipo desiderato).

Esempio:

```
public class AObject2 {
    public static void printAllAndSum(final Object[] array) {
        int sum = 0;
        for (final Object o : array) {
            System.out.println("Oggetto:" + o.toString());
            if (o instanceof Integer) { // Test a run-time
```

```
            final Integer i = (Integer) o; // Downcast
            sum = sum + i.intValue(); // Operazione specifica di Integer
        }
      }
      System.out.println("Somma degli Integer: " + sum);
   }
}
```

- *Provenienza*: 12-polymorphism_slides.pdf, Pagina 22

2.2.4.1. Errori Connessi alle Conversioni (`ClassCastException`)

Un `downcast` verso una classe incompatibile con il tipo dinamico dell'oggetto causerà una `ClassCastException` a run-time. Questo è un errore molto pericoloso che può essere evitato usando `instanceof` prima del cast.

Esempio:

```
public class ShowCCE {
   public static void main(String[] as) {
      Object o = Integer.valueOf(10);
      Integer i = (Integer) o; // OK
      String s = (String) o;   // ClassCastException: L'oggetto 'o' è un Integer, non una String
   }
}
```

- *Provenienza*: 12-polymorphism_slides.pdf, Pagina 24

2.2.5. Argomenti Variabili (`Variable Arguments`)

Java permette ai metodi di accettare un numero variabile di argomenti dello stesso tipo, utilizzando la sintassi `Type... argname`. All'interno del metodo, `argname` viene trattato come un array di `Type`.

Esempio:

```
public class VarArgs {
   // Somma un numero variabile di Integer
   public static int sum(final Integer... args) {
      int sum = 0;
      for (int i : args) {
         sum = sum + i;
      }
      return sum;
   }

   // Stampa il contenuto degli argomenti, se meno di 10
   public static void printAll(final String start, final Object... args) {
      System.out.println(start);
      if (args.length < 10) {
         for (final Object o : args) {
            System.out.println(o);
         }
      }
   }

   public static void main(String[] s) {
      System.out.println(sum(10, 20, 30, 40, 50, 60, 70, 80)); // Passa direttamente gli argomenti
      printAll("inizio", 1, 2, 3.2, true, new int[] { 10 }, new Object());
   }
```

```
}
```

- *Provenienza*: 12-polymorfism_slides.pdf, Pagina 43-44

2.3. Classi Astratte

Le classi astratte in Java rappresentano un costrutto intermedio tra le interfacce (che definiscono solo un contratto) e le classi concrete (che definiscono un comportamento completo).

2.3.1. Motivazioni e Proprietà

- **Comportamento Parziale**: Le classi astratte sono usate per descrivere classi con un comportamento parziale, dove alcuni metodi sono dichiarati ma non implementati (metodi astratti).
- **Non Istanziali**: Non possono essere istanziate direttamente (`new AbstractClass()` è un errore).
- **Estensione e Completamento**: Possono essere estese da sottoclassi che le completano implementando i metodi astratti. Solo le sottoclassi concrete (non astratte) possono essere istanziate.
- **Contenuto**: Possono definire campi, costruttori, e metodi concreti (implementati) oltre ai metodi astratti.
- **Ereditarietà e Interfacce**: Possono estendere una classe (astratta o non) e implementare interfacce. Se implementano un'interfaccia ma non tutti i suoi metodi, la classe deve rimanere astratta e i metodi non implementati diventano astratti nella classe.
- *Provenienza*: 12-polymorphism_slides.pdf, Pagina 27-28

2.3.2. Pattern Template Method

Una tipica applicazione delle classi astratte è il **Pattern Template Method** (comportamentale, su classi). Questo pattern definisce lo scheletro (template) di un algoritmo, lasciando l'implementazione di alcuni suoi aspetti alle sottoclassi.

- **Soluzione**: L'algoritmo è realizzato attraverso un "metodo template" (spesso `final`) che definisce un comportamento comune, basato su chiamate a metodi astratti/da specializzare. Le sottoclassi forniscono l'implementazione concreta dei metodi astratti.
- *Provenienza*: 12-polymorphism_slides.pdf, Pagina 46-47

Esempio: `BankAccount`
Una classe astratta `BankAccount` può avere un metodo `withdraw()` (template method) che gestisce la logica comune di prelievo, ma delega il calcolo della commissione (`operationFee()`) a un metodo astratto che sarà implementato dalle sottoclassi.

```
public abstract class BankAccount {
    private int amount;
    public BankAccount(int amount){ this.amount = amount; }
    public abstract int operationFee(); // Metodo astratto: costo bancario operazione
    public int getAmount(){ return this.amount; }
    public void withdraw(int n){ // Template method
        this.amount = this.amount - n - this.operationFee();
    }
}

class BankAccountWithConstantFee extends BankAccount {
    public BankAccountWithConstantFee(int amount) { super(amount); }
    public int operationFee(){ return 1; } // Implementazione concreta della commissione
}

public class UseBankAccount {
    public static void main(String[] args){
        final BankAccount b = new BankAccountWithConstantFee(100);
        b.withdraw(20);
        System.out.println(b.getAmount()); // Output: 79 (100 - 20 - 1)
    }
```

```
}
```

- *Provenienza*: 12-polymorphism_slides.pdf, Pagina 48

2.3.3. Esempio: `LimitedLamp` (Template Method in Azione)

Questo esempio mostra come una `SimpleLamp` possa essere estesa in una `LimitedLamp` astratta, che definisce la logica di accensione (template method `switchOn()`) basandosi su metodi astratti (`okSwitch()`, `isOver()`) che vengono implementati dalle sottoclassi per definire diverse strategie di esaurimento.

- `SimpleLamp`: Lampadina base.
- `LimitedLamp` (astratta): Estende `SimpleLamp`, aggiunge `switchOn()` (final, template method) e metodi astratti `okSwitch()` e `isOver()`.
- `UnlimitedLamp`: Non si esaurisce mai.
- `CountdownLamp`: Si esaurisce dopo `n` accensioni.
- `ExpirationTimeLamp`: Si esaurisce dopo un certo tempo dalla prima accensione.

```
// SimpleLamp.java
public class SimpleLamp {
    private boolean switchedOn;
    public SimpleLamp() { this.switchedOn = false; }
    public void switchOn() { this.switchedOn = true; }
    public void switchOff() { this.switchedOn = false; }
    public boolean isSwitchedOn() { return this.switchedOn; }
}

// LimitedLamp.java (Classe Astratta)
public abstract class LimitedLamp extends SimpleLamp {
    public LimitedLamp() { super(); }
    public final void switchOn() { // TEMPLATE METHOD
        if (!this.isSwitchedOn() && !this.isOver()) {
            super.switchOn();
            this.okSwitch();
        }
    }
    protected abstract void okSwitch(); // Cosa fare all'accensione (dipende dalla strategia)
    public abstract boolean isOver(); // Strategia per riconoscere se la lamp è esaurita
    public String toString() { return "Over: " + this.isOver() + ", switchedOn: " + this.isSwitchedOn(); }
}

// UnlimitedLamp.java
public class UnlimitedLamp extends LimitedLamp {
    public UnlimitedLamp() { super(); }
    protected void okSwitch() { } // Non fa nulla
    public boolean isOver() { return false; } // Non è mai esaurita
}

// CountdownLamp.java
public class CountdownLamp extends LimitedLamp {
    private int countdown;
    public CountdownLamp(final int countdown) { super(); this.countdown = countdown; }
    protected void okSwitch() { this.countdown--; } // Decrementa il contatore
    public boolean isOver() { return this.countdown == 0; } // Esaurita se il contatore è a zero
}

// ExpirationTimeLamp.java
import java.util.Date;
```

```
public class ExpirationTimeLamp extends LimitedLamp {
   private Date firstSwitchDate;
   private long duration;
   public ExpirationTimeLamp(final long duration) { super(); this.duration = duration; this.firstSwitchDate = null; }
   protected void okSwitch() { if (this.firstSwitchDate == null) { this.firstSwitchDate = new Date(); } }
   public boolean isOver() { return this.firstSwitchDate != null && (new Date().getTime() - this.firstSwitchDate.getTime()
>= this.duration); }
}

// UseLamps.java
public class UseLamps {
   public static void main(String[] s) throws Exception {
      LimitedLamp lamp = new UnlimitedLamp();
      lamp.switchOn(); System.out.println("ul| " + lamp); // Non si esaurisce mai

      lamp = new CountdownLamp(5);
      for (int i = 0; i < 4; i++) { lamp.switchOn(); lamp.switchOff(); }
      System.out.println("cl| " + lamp);
      lamp.switchOn(); System.out.println("cl| " + lamp); // Al quinto switch si esaurisce

      lamp = new ExpirationTimeLamp(1000); // 1 secondo
      lamp.switchOn(); System.out.println("el| " + lamp);
      Thread.sleep(3000); // Attendo 3 secondi
      System.out.println("el| " + lamp); // Dopo 1 secondo si è esaurita
   }
}
```

- *Provenienza*: 12-polymorphism_slides.pdf, Pagina 29-36

2.3.4. Classi Astratte vs. Interfacce (con Metodi di Default)

A partire da Java 8, le interfacce possono avere metodi con implementazioni di default, rendendole più simili alle classi astratte. Tuttavia, rimangono differenze cruciali:

Caratteristica	Classi Astratte	Interfacce (con metodi di default)
Stato (Campi)	Possono definire variabili d'istanza (stato).	Non possono definire variabili d'istanza (solo costanti statiche e final).
Costruttori	Possono definire costruttori.	Non possono definire costruttori.
Visibilità Membri	Possono definire membri con visibilità diverse (`public` , `protected` , `private` , `package-private`).	Tutti i metodi sono implicitamente `public` (e `abstract` se non `default` o `static`). I campi sono implicitamente `public static final` .
Overriding `Object`	Possono fare overriding di metodi da `Object` .	Non possono fare overriding di metodi da `Object` (se non attraverso un metodo di default).
`final` **metodi**	I metodi concreti possono essere `final` .	I metodi di default non possono essere `final` .
Ereditarietà	Ereditarietà singola.	Ereditarietà multipla (implementazione di più interfacce).

- *Provenienza*: 12-polymorphism_slides.pdf, Pagina 37-38

2.4. Polimorfismo Parametrico (Generics)

Prima di Java 5, la gestione delle collezioni polimorfiche in Java era complessa e soggetta a errori a run-time. I Generics hanno introdotto il **polimorfismo parametrico**, migliorando la sicurezza e la riusabilità del codice.

2.4.1. Il Problema delle Collezioni Polimorfiche (Pre-Java 5)

In passato, per creare collezioni di tipi diversi, si usava `Object` come tipo degli elementi (es. `ObjectVector` , `ObjectList`).

Esempio: `ObjectVector`

```
// Da IntVector a ObjectVector
public class ObjectVector {
    private Object[] elements; // Deposito per gli elementi
    private int size;
    // ... costruttore e metodi addElement, getElementAt, getLength, expand, toString
    public Object getElementAt(final int position) {
        return this.elements[position]; // Restituisce Object
    }
}

// Uso di ObjectVector
public class UseObjectVector {
    public static void main(String[] s) {
        final ObjectVector vobj = new ObjectVector();
        vobj.addElement(1); // Autoboxing
        vobj.addElement(1);
        for (int i = 0; i < 20; i++) {
            // Necessari downcast espliciti, soggetti a ClassCastException
            vobj.addElement((Integer) vobj.getElementAt(vobj.getLength() - 1) +
                    (Integer) vobj.getElementAt(vobj.getLength() - 2));
        }
        System.out.println(vobj);
    }
}
```

- *Provenienza*: 13-generics_slides.pdf, Pagina 11-15

Questo approccio portava a:

- **Perdita di Traccia del Contenuto**: Non era chiaro a compile-time quale tipo di oggetto fosse effettivamente contenuto nella collezione.
- **Necessità di Downcast Espliciti**: Ogni volta che si recuperava un elemento, era necessario un downcast esplicito, che poteva generare `ClassCastException` a run-time se il tipo non corrispondeva.

2.4.2. Introduzione ai Generics (Polimorfismo Parametrico)

I Generics sono un meccanismo di Java (introdotto in Java 5) che implementa il polimorfismo parametrico, permettendo di definire classi, interfacce e metodi che operano su tipi specificati come parametri.

- **Idea di Base**: Se un frammento di codice `F` può lavorare uniformemente su diversi tipi (es. `String`, `Integer`), lo si rende parametrico sostituendo il tipo specifico con una **type-variable** o **type-parameter** (es. `X`). Quando serve il codice istanziato per un tipo specifico, si usa `F<String>` o `F<Integer>`.
- **Java Generics**: Sono un meccanismo a compile-time. Il compilatore "compila a erasure", il che significa che le informazioni sui tipi generici vengono rimosse dopo la compilazione, e a run-time il codice è simile a quello che userebbe `Object`. Questo ha delle limitazioni (vedi 2.4.4).

2.4.3. Classi Generiche

Una classe generica è una classe che dichiara uno o più parametri di tipo.

Esempio: `List<X>`

```
public class List<X> { // X è la type-variable
    private final X head;
    private final List<X> tail;

    public List(final X head, final List<X> tail) {
        this.head = head;
        this.tail = tail;
```

```
    }
    public X getHead() { return this.head; }
    public List<X> getTail() { return this.tail; }
    // ... getLength() e toString()
}

// Uso di una classe generica
public class UseList {
    public static void main(String[] s) {
        final List<Integer> list = new List<Integer>(10, // Autoboxing
            new List<Integer>(20, new List<Integer>(30, new List<Integer>(40, null))));
        final int first = list.getHead(); // Nessun cast necessario
        System.out.println(first);
    }
}
```

- *Provenienza*: 13-generics_slides.pdf, Pagina 19-22

Terminologia Essenziale:

Data una classe generica `C<X, Y>`:

- `C` è il **tipo parametrico**.
- `X` e `Y` sono le **type-variable** o **parametri di tipo**.
- I clienti devono usare **versioni "istanziate"** (es. `C<String, Integer>`).
- Ogni type-variable deve essere sostituita con un **argomento di tipo** (una classe non-generica, un'altra type-variable, o un tipo generico istanziato). **NON con un tipo primitivo**.
- *Provenienza*: 13-generics_slides.pdf, Pagina 23

2.4.3.1. Inferenza dei Parametri (`<>` e `var`)

Per ridurre la verbosità del codice, Java supporta l'inferenza dei parametri di tipo. Si può usare il "diamond symbol" (`<>`) nella `new` e il compilatore cercherà di inferire i tipi dagli argomenti o dal contesto. A partire da Java 10, la `local variable type inference` (`var`) può essere usata in alternativa.

Esempi di Inferenza:

```
final Vector<Pair<String,Integer>> v = new Vector<>(); // Inferenza del tipo di Vector
v.addElement(new Pair<>("Prova",1)); // Inferenza del tipo di Pair
final var v2 = new Vector<Integer>(); // Inferenza del tipo di Vector usando 'var'
```

- *Provenienza*: 13-generics_slides.pdf, Pagina 30-31

2.4.4. Il Problema della Type-Erasure e le sue Limitazioni

A causa dell'implementazione "ad erasure" dei Generics in Java, ci sono alcune limitazioni:

- Una type-variable (`X`) **non può** essere usata per:
  - Creare nuove istanze: `new X()` , `new X[] {...}` , `new X[10]` .
  - Verificare il tipo con `instanceof` : `o instanceof X` .
- I cast a tipi generici (`(X)o` , `(C<String>)o`) generano un "unchecked warning" perché il compilatore non può garantire la sicurezza del tipo a run-time.

Queste limitazioni derivano dal fatto che il compilatore trasforma una classe generica (es. `List<X>`) in qualcosa di simile a una versione non generica (es. `ObjectList`) prima della compilazione effettiva.

2.4.5. Interfacce Generiche

Un'interfaccia generica è un'interfaccia che dichiara parametri di tipo. Questi parametri appaiono nei metodi definiti dall'interfaccia. Quando una classe implementa l'interfaccia, deve istanziare i parametri di tipo.

Esempio: `Iterator<E>`

```
public interface Iterator<E> {
   E next(); // Torna il prossimo elemento dell'iterazione
   boolean hasNext(); // Dice se vi saranno altri elementi
}
```

- *Provenienza*: 13-generics_slides.pdf, Pagina 33-35

Diverse implementazioni di `Iterator` (es. `IntRangeIterator`, `ListIterator`, `VectorIterator`) possono fornire un accesso uniforme a diverse strutture dati.

2.4.6. Metodi Generici

Un metodo generico è un metodo che lavora su argomenti e/o valori di ritorno in modo indipendente dal loro tipo effettivo. Tale tipo è astratto in una type-variable del metodo.

Sintassi: `def: <X1,..,Xn> ret-type nome-metodo(formal-args) { ... }call: receiver.<X1,..,Xn>nome-metodo(actual-args) { ... }` (spesso con inferenza, senza specificare i tipi)

- *Provenienza*: 13-generics_slides.pdf, Pagina 39-41

Esempio:

```
public class UseIterators2 {
   public static <E> void printAll(Iterator<E> iterator) { // Metodo generico
      while (iterator.hasNext()) {
         System.out.println("Elemento : " + iterator.next());
      }
   }

   public static void main(String[] s) {
      Iterator<Integer> iterator = new IntRangeIterator(5, 10);
      printAll(iterator); // Chiamata con inferenza
   }
}
```

- *Provenienza*: 13-generics_slides.pdf, Pagina 42-43

2.5. Java Wildcards

Le Java Wildcards (`?`) sono un meccanismo che fornisce nuovi tipi, simili a interfacce, usati principalmente come tipo dell'argomento di metodi per esprimere una maggiore flessibilità nei tipi accettati.

2.5.1. Tipi di Wildcard

Esistono tre tipi principali di wildcard:

- **Bounded (Covariante):** `C<? extends T>`
  - Accetta qualsiasi `C<S>` dove `S` è un sottotipo di `T` (`S <: T`).
  - Significa "può fornire ma non accettare `T`".
  - **Esempio**: `Vector<? extends Number>` può contenere `Vector<Integer>`, `Vector<Double>`, ecc. Da un tale `Vector`, si possono *leggere* solo oggetti che sono almeno `Number` (o un suo supertipo), ma *non si possono aggiungere* elementi (tranne `null`) perché non si sa quale sottotipo specifico sia.
- **Bounded (Controvariante):** `C<? super T>`
  - Accetta qualsiasi `C<S>` dove `S` è un supertipo di `T` (`S >: T`).
  - Significa "può accettare ma non fornire `T`".
  - **Esempio**: `Vector<? super Integer>` può contenere `Vector<Integer>`, `Vector<Number>`, `Vector<Object>`. A un tale `Vector`, si possono *aggiungere* elementi di tipo `Integer` (o suoi sottotipi), ma quando si *legge* un elemento, il tipo garantito è solo `Object` (o un suo supertipo), perché non si sa quale supertipo specifico sia.

- **Unbounded:** `C<?>`
  - Accetta qualsiasi `C<S>` (equivalente a `C<? extends Object>`).
  - Significa "può fornire e accettare qualsiasi tipo, ma senza garanzie sul tipo specifico". Utile quando il metodo non dipende dal tipo specifico degli elementi.
  - **Esempio**: `Vector<?>` può contenere `Vector<Integer>`, `Vector<String>`, ecc. Da un tale `Vector`, si possono leggere solo `Object`, e non si possono aggiungere elementi (tranne `null`).

2.5.2. Sostituibilità e Safety con i Generics

Una domanda comune è: `Vector<Integer>` è un sottotipo di `Vector<Object>`? La risposta è **no**.

Se `Vector<Integer>` fosse un sottotipo di `Vector<Object>`, si potrebbe passare un `Vector<Integer>` a un metodo che si aspetta un `Vector<Object>`. Questo metodo potrebbe poi aggiungere una `String` al `Vector<Object>`, compromettendo l'integrità del `Vector<Integer>` sottostante e portando a un `ClassCastException` quando si tenta di leggere un `Integer` da esso.

Esempio di insicurezza (se fosse permesso):

```
void addAString(Vector<Object> vector){
   vector.addElement("warning!"); // Aggiungo una String a un Vector<Object>
}

// ... nel main
Vector<Integer> vec = new Vector<>();
vec.addElement(Integer.valueOf(0));
addAString(vec); // Se fosse consentito, ora 'vec' conterrebbe una String!
int n = vec.getElementAt(1).intValue(); // ClassCastException a run-time
```

Per mantenere la **safety** del linguaggio (nessuna combinazione di istruzioni porta a invocare un metodo su un oggetto la cui classe non lo definisce), Java ha scelto di rendere le diverse istanziazioni di una classe generica **scollegate** (né covarianti né controvarianti per default).

2.5.3. Unsafety con gli Array di Java

Contrariamente ai Generics, gli array in Java sono trattati come **covarianti** (`Integer[]` è un sottotipo di `Object[]`). Questa decisione è stata presa nelle prime versioni di Java, prima dell'introduzione dei Generics, per permettere il riuso del codice.

Questo comporta una potenziale insicurezza:

```
Object[] o = new Integer[]{1,2,3}; // OK per covarianza
o[0] = "a"; // Lancia ArrayStoreException a run-time
```

Questo errore a run-time (`ArrayStoreException`) è la protezione di Java contro l'insicurezza derivante dalla covarianza degli array.