Librerie predefinite, Conversioni, Programmazione generica / 04-03

Libreria Objects

Alcuni metodi:

- public static <T> T requireNonNull (T obj) — metodo generico

  vedi dopo: T variabile di tipo reference che è il tipo che restituisce

  lancia eccezione se obj è null

- public static boolean equals (Object a, Object b)

  metodo di classe
  Controlla l'uguaglianza se a != null (controllo incluso)

- public static int checkIndex (int index, int length)

  controllo che l'indice sia compreso fra zero e length
  lancia eccezione altrimenti

- public static String toString (Object o)

  se o != null ritorna o.toString() altrimenti la stringa "null"

  } fa il controllo rispetto al solito toString

poiché Java dà priorità ai metodi ereditati specifica la classe per chiamare i metodi sopra

es Objects.equals(a,b)

Nota: equals(Object o) è un metodo oggetto di Object
equals(Object a, Object b) è un metodo di classe in Objects

Metodi generici

<T> T requireNonNull(T obj)

può essere un metodo sia di classe che oggetto

<T> è un tipo polimorfo

<T> T requireNonNull(T obj)
Object nonGenRequireNonNull(Object obj)

polimorfismo parametrico
polimorfismo x sottotipo

con Object perdo delle informazioni in uscita,
ovvero utilizzo il tipo più generale possibile

Libreria StringBuilder

Maniera efficiente di manipolare stringhe

String → immutabili
StringBuilder → mutabili

Alcuni metodi:

StringBuilder append (String str) concatena str a this

StringBuilder delete (int start, int end) toglie da s a e-1

String toString () da StringBuilder a String } come in String

char charAt (index) e int length() }

Conversioni

- implicite
- espl. cile

} compatibili per stessi tipi (Reference / primitive)

Implicite

public boolean contains (CharSequence s);

"hello world".contains (<espressione>);

T tipo passato = String
S tipo atteso = CharSequence
(convertito)

String ≤ CharSequence
quindi String viene convertito al supertipo
(upcast - nota Java non fa downcast)

La medesima cosa vale per i tipi primitivi (es. int ≤ double)

Altre conversioni esplicite vengono fatte in caso di

- assegnazioni (e accesso ad array)
- String context (1 + "2") corrisponde a "12"
- operatori numerici

Nota: + associa da sx

("1" + 2 + 3) = (("1" + 2) + 3)

= "123"

2 subisce boxing a Integer

e poi ToString e concatenato a 1

stessa cosa per 3

(1 + 2 + "3") = "33"

(null + "a") = "nulla"

Esplicite

- widening / narrowing primitive conversion (nota: no cost per booleani)
  → supertipo → sottotipo
- widening reference conversion → nessuna azione, $T \in S$
- narrowing reference conversion → faccio il controllo che 'e' ha effettivamente sottotipo S

e si scrive

(S) e

↓

tipo desiderato

nota: il compilatore evita anche casT senza senso

Esempio widening

char [] c = {'a', 'b', 'c'}

String.valueOf(c);

String.valueOf((Object) c);

String.valueOf((Object) null);

valueOf(char[]) is called

valueOf(Object) is called

null è sottotipo di qualsiasi tipo reference

Esempio narrowing (old style)

```
public boolean (Object otherTimer) {
    if ( !(otherTime instanceof Timer)) return false;
    return time == ((Timer) otherTimer).getTime();
}
```

controllo che otherTime abbia tipo Timer

↓

cast non più necessario,
bisogna specificare al compilatore che si passa da Object a Timer (narrowing), non viene lanciata eccezione perché il controllo viene fatto anche prima (faccio due volte lo stesso controllo)

instanceOf

```
public boolean (Object otherTimer) {
    if ( otherTime instanceof Timer (t))
    return time == t.getTime();
    return false
}
```

var locale al then

→ permette di fare il controllo del tipo (non esiste cast di Object)

→ grazie a una hash map

public class Pair<T1, T2> {

private final T1 fst;
private final T2 snd;

⋮

}

mi disinteresso dei tipi nella classe nel main:

Pair<String, Color> p =
new Pair<String, Color>(...)

✗

Pair<Color, String>

Note: non vanno bene i tipi primitivi

es vo: Pair<int, String>

Keywords:

Metodi generici
Widening
Narrowing
instanceOf