Tagesprogramm

Methoden von Object

toString

instanceof, class, getClass und Cast

equals und hashCode

Object

`java.lang.Object` ist die oberste Oberklasse aller Klassen

Methoden von `Object` sind in jeder Klasse vorhanden:

```
public String toString() {...}
public boolean equals(Object other) {...}
public int hashCode() {...}
public Class<?> getClass() {...}
```

`clone, finalize, wait, notifyAll, ...`

toString

```
public String toString() {...}
```

wenn `x` Objekt eines Referenztyps, dann

```
"Text" + x
```
entspricht
```
"Text" + x.toString()
```

```
System.out.print(x)
```
entspricht
```
System.out.print(x.toString())
```

Implementierung in `Object` liefert wenig Information

→ sollte abhängig von Objektverwendung überschrieben werden

Achtung:

aus `x.toString().equals(y.toString())` folgt nicht `x.equals(y)`

Aufgabe: Wozu toString in Object?

Man kann in jeder Klasse Methoden zur Erzeugung von Strings einführen.
Warum ist dennoch `toString` in `Object` und daher in jeder Klasse vorhanden?

A Um Programmierer(inne)n das Schreiben von `toString` zu ersparen.

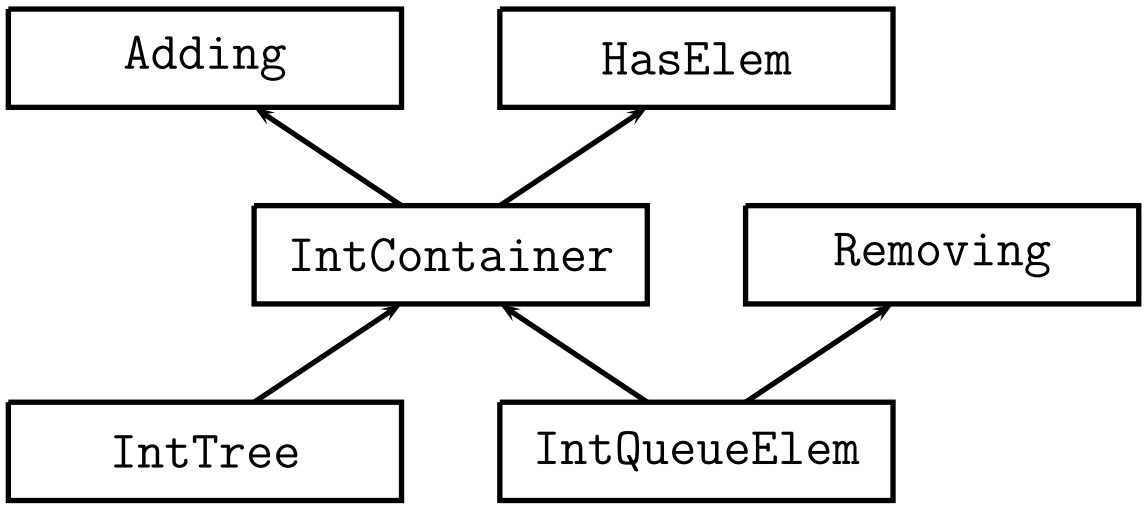
B Um die spezielle Semantik von + zu ermöglichen.

C Um eine einheitliche Namensgebung zu erreichen.

D Um Abstraktion auf die Verwendung von Strings auszudehnen.

instanceof

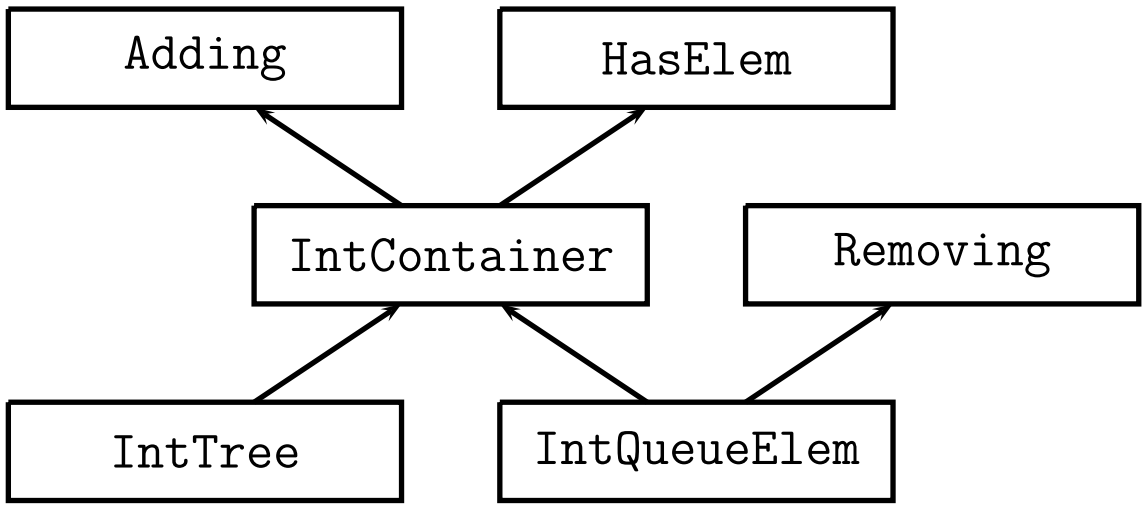
```
Adding x = new IntTree();

System.out.println(x instanceof Adding);  →  true
System.out.println(x instanceof Removing);  →  false
System.out.println(x instanceof IntContainer);  →  true
System.out.println(x instanceof IntTree);  →  true
System.out.println(null instanceof IntTree);  →  false
```



class und getClass

```
Adding x = new IntTree();
```

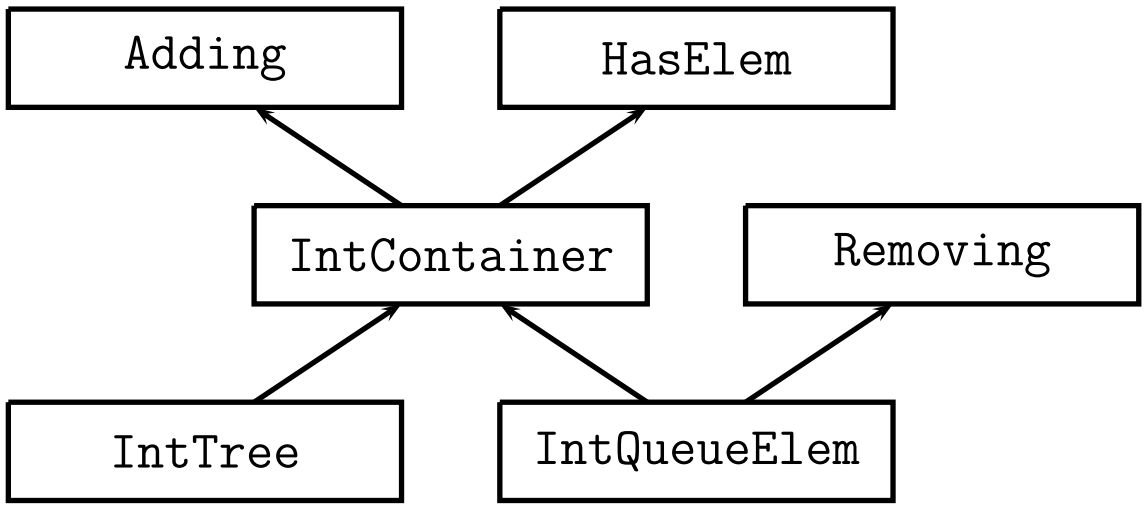
```
System.out.println(x.getClass().getName());   →   IntTree
System.out.println(IntTree.class.getName());   →   IntTree
System.out.println(Adding.class.getName());   →   Adding
System.out.println(x.getClass().equals(IntTree.class));
System.out.println(x.getClass().equals(Adding.class));
```



Cast auf Referenztypen

```
Adding x = new IntTree();
```

```
System.out.println(x.empty());          ... Compilationsfehler
System.out.println((IntTree)x).empty());   →   true
System.out.println((IntQueueElem)x).empty()); ... Laufzeitfehler
System.out.println((IntContainer)x).empty());   →   true
System.out.println((IntContainer)null);   →   null
```



Aufgabe: Verwendung von Casts

Erfahrene Programmierer(innen) meiden die Verwendung von Casts auf Referenztypen. Deren Verwendung gilt als schlechter Programmierstil.

Warum ist das so?

A Weil Casts leicht zu Fehlern im Programm führen.

B Weil Casts das Erkennen von Fehlern im Programm erschweren.

C Weil Casts Prüfungen durch den Compiler ausschalten.

D Weil es heute bessere Alternativen zu Casts gibt.

equals

```
public boolean equals(Object other) {...}
```

in `Object` definiert durch `return this == other;`

→ muss für inhaltliche Vergleiche überschrieben werden

Bedingungen:

- wenn `x != null` dann `x.equals(null) == false`
- wenn `x != null` dann `x.equals(x)`
- wenn `x != null && y != null`
  dann `x.equals(y) == y.equals(x)`
- wenn `x.equals(y) && y.equals(z)` dann `x.equals(z)`
- wenn `x` und `y` unverändert
  dann liefern wiederholte Aufrufe von `x.equals(y)` gleiche Ergebnisse

Typische Implementierung von equals

```
@Override
public boolean equals(Object other) {
    if (this == other) {    // nicht immer notwendig
        return true;
    }
    if (other == null       // immer notwendig
        || ! this.getClass().equals(other.getClass())) {
        return false;
    }
    ClassName that = (ClassName)other;
    // ClassName ist Name der aktuellen Klasse
    // ...Vergleich der Inhalte von this und that ...
}
```

Aufgabe: Wozu equals?

Wozu benötigt man `equals`, obwohl es für Vergleiche auch == gibt?

A Vergleiche mit == sind nur auf elementaren Typen sinnvoll.

B Bei einem Vergleich mit == wird im Hintergrund `equals` aufgerufen.

C Das System kann nicht wissen, wann zwei Objekte als gleich angesehen werden sollen, und Operatoren kann man nicht selbst programmieren.

D Gleichheit (`equals`) und Identität (==) sind nicht dasselbe.

hashCode

`public int hashCode() {...}`

gibt eine aus dem Objektinhalt errechnete (fast beliebige) Zahl zurück

wird von einigen Datenstrukturen (z.B. `HashMap`) benötigt

Bedingungen:

wenn `x.equals(y)` dann `x.hashCode() == y.hashCode()`

wenn `x` unverändert
dann liefern wiederholte Aufrufe von `x.hashCode()` gleiche Ergebnisse

→ wenn `equals` überschrieben dann auch `hashCode` überschrieben

Achtung:
aus `x.hashCode() == y.hashCode()` folgt nicht `x.equals(y)`

Aufgabe: Untertypen und Unterklassen

Suchen Sie in Gruppen zu 2 bis 3 Personen Antworten auf folgende Fragen:

Aus welchen Gründen verwendet man Untertypen und Unterklassen?

Kann man auf Unterklassen verzichten, wenn man Untertypen hat?

Kann man auf Untertypen verzichten, wenn man Unterklassen hat?