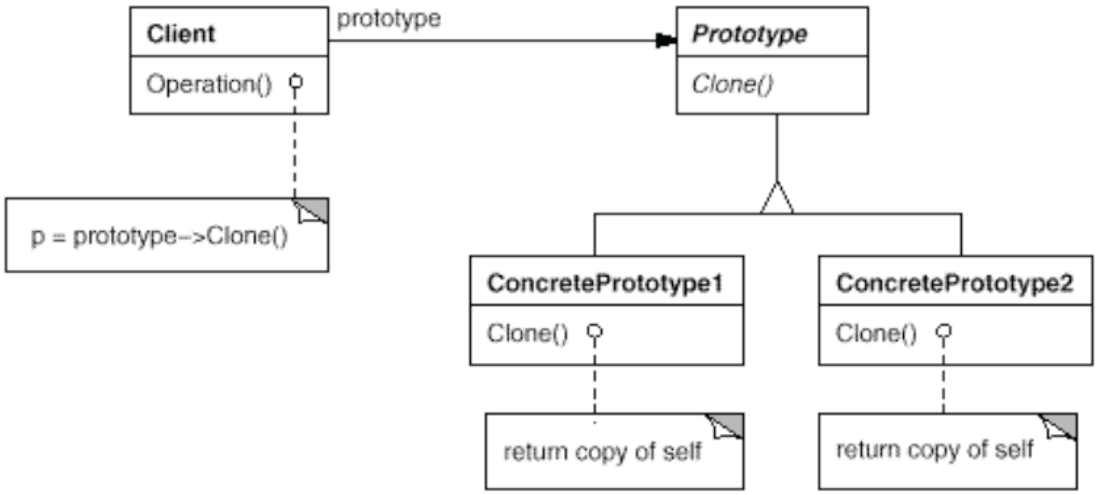
Prototype / Cloning

Idee:
Neue Objekte werden durch das Kopieren (Klonen) von Vorlagen (Prototypen) erzeugt.

Motivation:
Im Grafikeditor sollen existierende Figuren kopiert werden können (Copy & Paste).

Es wäre auch möglich, konstruierte Figuren in der Tool-Palette abzulegen. So liessen sich einfach neue Instanzen erzeugen indem diese Vorlagen geklont werden.

Struktur:



Implementierung in Java

Definition in Klasse `java.lang.Object`:

```
class Object {
   protected Object clone() throws CloneNotSupportedException {
      // ...
   }
}
```

Wie sollen Unterklassen `clone()` implementieren?

```
class Point {
   private int x, y;
   public Point (int x, int y) {
      this.x = x ; this.y = y;
   }
   public Point clone() {
      ...
   }
}
```

Wie sähe eine `class ColorPoint extends Point` mit einer clone-Methode aus?

Wie funktioniert Object.clone()?

`Object.clone` erzeugt eine neue Instanz mit identischem dynamischem Typ, d.h. es gilt[1] für Object.clone:

- `x.clone() != x`
- `x.clone().getClass() == x.getClass()`
- `x.clone().equals(x)`

Vorgehen:

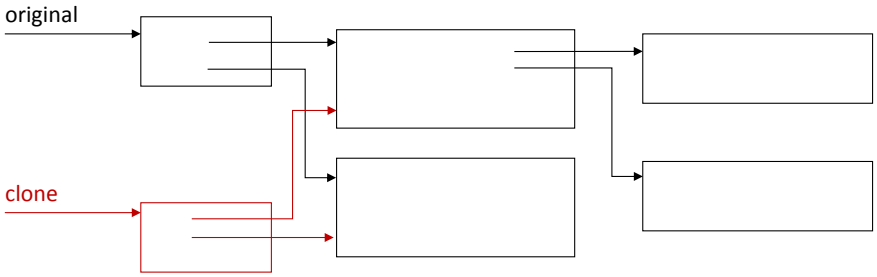
1. Es wird geprüft, ob die Klasse das Interface `Cloneable` implementiert.
   `x instanceof Cloneable`
   Falls nein ⇒ `CloneNotSupportedException` wird geworfen.

2. Neue Instanz wird erzeugt, d.h. es wird so viel Speicher allokiert, wie das Originalobjekt benötigt. Dabei wird *kein* Konstruktor aufgerufen!

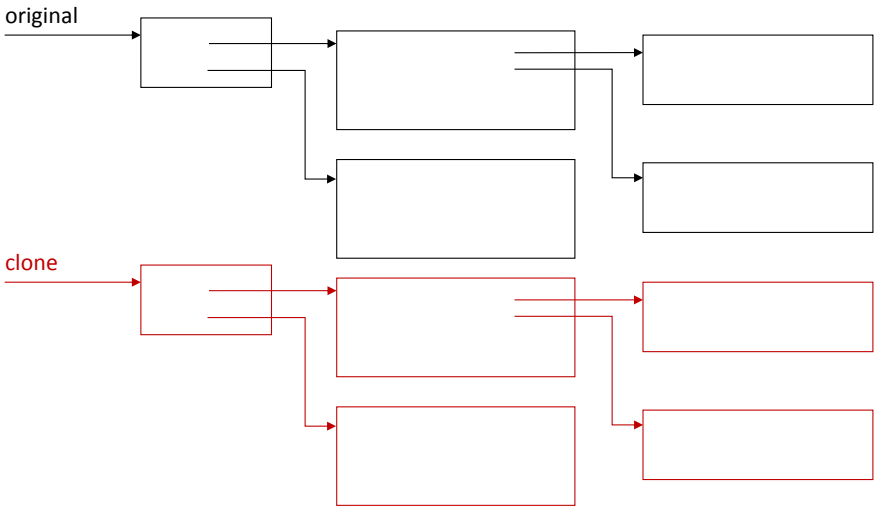
3. Stattdessen wird der Speicher des Originalobjektes Byte für Byte in die neue Instanz kopiert (memcpy). D.h. es werden alle Attribute kopiert ⇒ shallow copy

Kopierarten:

- ***shallow copy***
  nur das Objekt wird geklont, nicht aber weitere Objekte auf die verwiesen wird



- ***deep copy***
  Objekt und alle referenzierten Objekte werden kopiert.
  Problem: Alias-Referenzen (wie z.B. Zyklen)



[1] Gemäss API-Dokumentation zu java.lang.Object *sollten* diese Eigenschaften erfüllt sein, sie *müssen* es aber *nicht*.

Bemerkungen:

- `Object.clone` führt nur shallow-copy aus!

  Falls das Objekt Referenzen auf andere Objekte enthält, so werden nur die Referenzen darauf kopiert. Ist ein anderes Verhalten nötig so muss die Methode `clone()` überschrieben werden und die Referenzen müssen „von Hand“ kopiert werden.

- Signatur der `clone`-Methode

```
public Object clone() {
   ...
}
```

```
public Point clone() {
  ...
}
```

  Seit Java 5 kann der Resultattyp in der überschreibenden Methode verstärkt werden. Damit fallen auf Klientenseite die Typ-Casts weg.

- Die `clone`-Implementationen der Collection Framework Klassen erzeugen nur shallow-Kopien. Ein deep-copy ist nicht möglich da

  1. die Methode `clone` in der Klasse `Object` nicht public ist
     Aus diesem Grund kann auf dem Typ Object keine clone-Methode aufgerufen werden.

  2. das `Cloneable` Interface leer ist
     Es hilft auch nicht, wenn bei einer Collection-Klasse der generische Typparameter von `Cloneable` abgeleitet wird (`T extends Cloneable`), denn das Interface `Cloneable` ist nur ein Marker-Interface, d.h. ein Aufruf der `clone`-Methode über dieses Interface ist nicht möglich.

- Problem: finale Felder
  Felder die final deklariert sind können in der `clone`-Methode nicht neu gesetzt werden!
  Das Java-Cloning übernimmt diese Werte aus der Vorlage (shallow-copy).

  Beispiel:

```
class Car {
     private static int nextId = 1;

     private final int id = nextId++;    // jeder Wagen erhaelt eine
                                         // eigene Seriennr. die
                                         // nicht mehr geaendert
                                         // werden kann.
     private int color;
     ...
}
```

  Beim Klonen wird die Seriennummer übernommen und kann nicht mehr geändert werden.

  Lösung: clone-Methode welche Konstruktor aufruft.

Cloning via Copy-Konstruktoren

Viele Programmiersprachen haben keine "eingebaute" Unterstützung des Clonings. Sie behelfen sich meist mit sogenannten Copy-Konstruktoren. Dies sind Konstruktoren, die als Parameter ein Objekt der eigenen Klasse erhalten. Daraus initialisieren sie ein neues Objekt mit dem gleichen Inhalt:

```
class A extends O {
    private int a;
    private B b;

    A() { … } // Default constructor
    A(int a, B b) { … }// Another constructor

    A(A original) {  // Copy constructor
        super(original);  // initialize attributes of superclass
        this.a = original.a;  // REMEMBER: original.a is accessible!
        this.b = original.b.clone(); // call B's clone method
    }

    … // Rest of A's implementation
}
```

Frage: wenn das Klassendesign einer Applikation durchgehend Copy-Konstruktoren vorsieht, soll dann

1. die Methode `clone` überhaupt noch implementiert werden?
2. `implements Cloneable` für die Klassen trotzdem gelten?
3. Wie könnte so eine `clone`-Methode aussehen?

Cloning via Serialisierung

Die `clone`-Methode kann mit Hilfe von Objektserialisierung wie folgt implementiert werden:

```
Object clone {
    final int size = 100; // initial size, is increased if necessary
    ByteArrayOutputStream baos = new ByteArrayOutputStream(size);
    ObjectOutputStream     oos = new ObjectOutputStream(baos);

    oos.writeObject(this);
    oos.close();

    byte buf[] = baos.toByteArray();

    ByteArrayInputStream bais = new ByteArrayInputStream(buf);
    ObjectInputStream     ois = new ObjectInputStream(bais);

    Object c = ois.readObject();

    return c;
}
```

Diese Lösung bedingt, dass alle Klassen als `Serializable` markiert sind (d.h. müssen das `Serializable`-Interface „implementieren“ – `Serializable` ist ein Marker-Interface). Objekte, welche nicht `Serializable` implementieren und transient markierte Referenzen werden nicht kopiert!