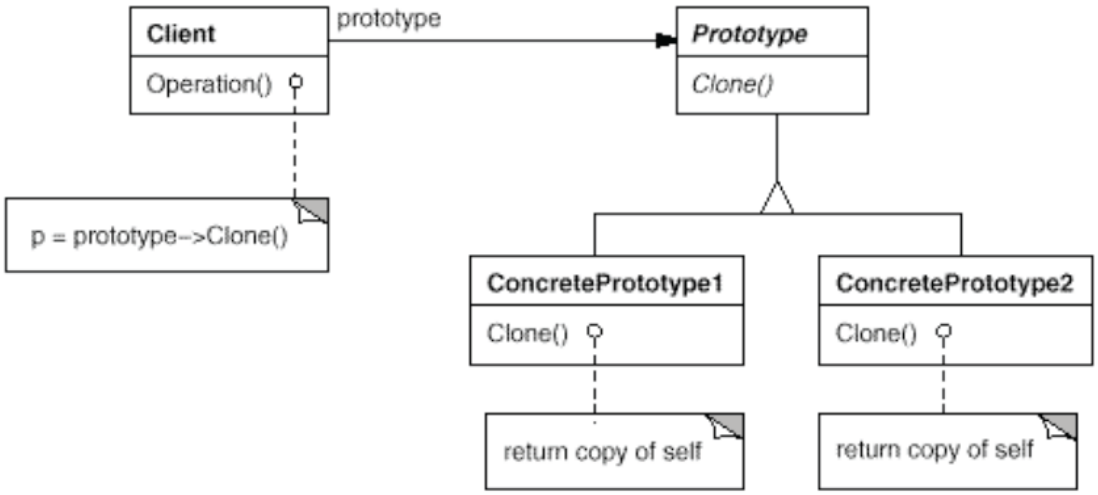
Prototype / Cloning

Idea:
New objects are created by cloning prototype objects.

Motivation:
Support of “Copy & Paste” on figures in a graphic created using JDraw.

It would also be imaginable to move constructed figures into the tool palette and to create new instances by cloning these prototypes.

Structure:



Implementation in Java

Definition of method `clone` in class `java.lang.Object`:

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

How should method `clone` be implemented in subclasses?

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

Given is the above class Point.
How would you implement method `clone` in a class `ColorPoint` extending class `Point`?

How does method Object.clone() work?

`Object.clone` creates a new instance with the same dynamic type, i.e. the following requirements[1] are defined for method `Object.clone`:

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

Procedure:

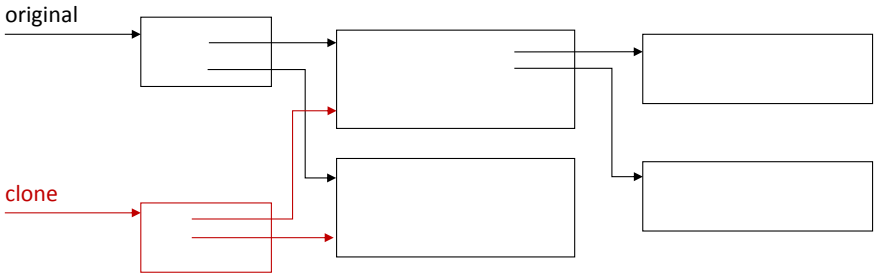
1. It is checked whether the class implements interface `Cloneable`:
   `x instanceof Cloneable`
   If this is not the case, a `CloneNotSupportedException` is thrown.

2. A new instance is created, i.e. as much memory as used by the original object is allocated.
   No constructor is invoked!

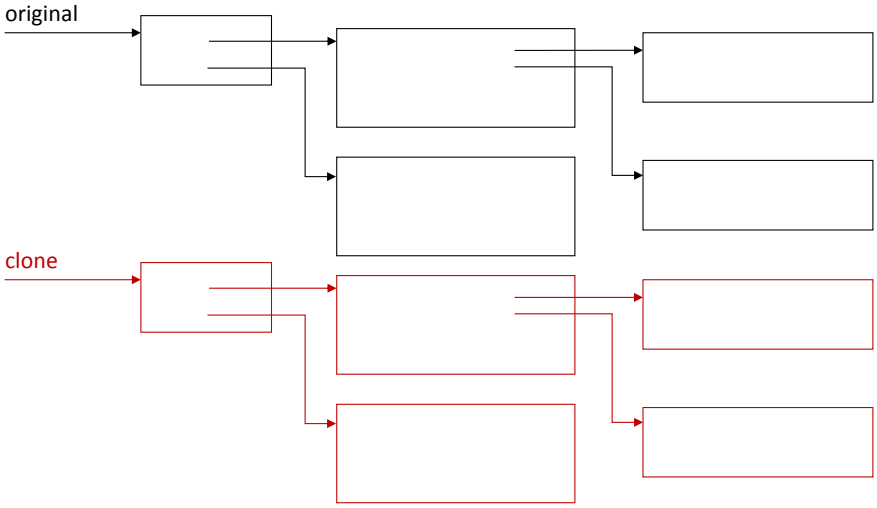
3. Instead, the memory of the original object is copied byte for byte into the new instance (=> memcpy), i.e. all attributes are copied over into the new instance (=> shallow copy).

Copy modes:

- ***shallow copy***
  only the object is cloned, but no other object referenced by the cloned object.



- ***deep copy***
  The object and all the referenced objects (i.e. the whole object graph) are copied.
  Problem: Alias-references (as e.g. cycles)



[1]According to the API documentation these requirements are not absolute (SHOULD-requirements and not MUST-requirements).

Remarks:

- Method `Object.clone` only performs a shallow copy!

  If the object contains references to other objects, then only these *references* are copied (but not the referenced objects, i.e. original and copy share the same referenced objects).
  If a different behavior is required, then `clone()` has to be overridden and the referenced objects have to be copied "manually".

- Signature of method `clone()`:

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

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

  Since Java 5 the result type of overriding methods can be strengthened. This allows to omit type casts on the client side.

- The `clone`-implementations of the collection classes defined in the Java Collection Framework only return a shallow copy. A deep copy cannot be implemented due to the following reasons:

  1. Method `clone` is not declared public in class `Object`.
     Therefore, the clone method cannot be called on an instance with the static type `Object`.

  2. The `Cloneable` interface is empty.
     It does not help to extend the generic type parameter of a collection class from the `Cloneable` interface (`T extends Cloneable`) as `Cloneable` is a marker interface only, i.e. an invocation of method `clone` over this interface is not possible.

- Problem: final fields
  Fields declared final cannot be initialized with fresh values on the new instance in method clone: Java cloning simply copies this values from the prototype (mode: shallow copy).

  Example:

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

     private final int id = nextId++;    // each car instance contains
                                         // its own serial number which
                                         // cannot be changed after
                                         // cloning
     private int color;
     ...
}
```

  If a car is cloned, then the serial number is copied as well as part of the default shallow copy process, but once set, it cannot be changed! The clone and the original share the same final fields (and probably this is also the correct and desired behavior).

  Solution: If final fields should be changed after having cloned a prototype, then use the copy-constructor approach, i.e. pass the prototype to the constructor which is used to create a new instance (see next page).

Cloning based on copy constructors

Many programming languages do not have a "built-in" cloning support. Cloning in such languages is typically implemented using so-called copy constructors. Copy constructors are constructors that receive an instance of their own type as parameter. They then initialize the new instance with the same values as the prototype passed.

```
class A extends Base {
    private int value;
    private B reference;

    A() { … } // Default constructor
    A(int value, B reference) { … } // Another constructor

    A(A original) {  // Copy constructor
        super(original);  // initialize attributes of the superclass
        this.value = original.value;  // REMEMBER: original.value is accessible
                                      // (although it has been declared private)
        this.reference = original.reference.clone(); // call B's clone method
    }

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

Question:
Let us assume that copy constructors are provided for all classes implemented in a particular system:

1. Is it then necessary at all to implement method `clone` in these classes?
2. If yes, how would such a `clone` method look like?
3. Is it then necessary to extend all classes from the `Cloneable` interface?

Cloning based on serialization

If all the classes which should be cloneable are serializable as well (i.e. implement the `Serializable` interface) then cloning can also be implemented with the help of Java serialization. Objects which do not implement the marker interface `Serializable` cannot be cloned with this approach, and fields declared as transient will not be cloned. The `clone` method which follows this approach looks as follows:

```
Object clone {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    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;
}
```