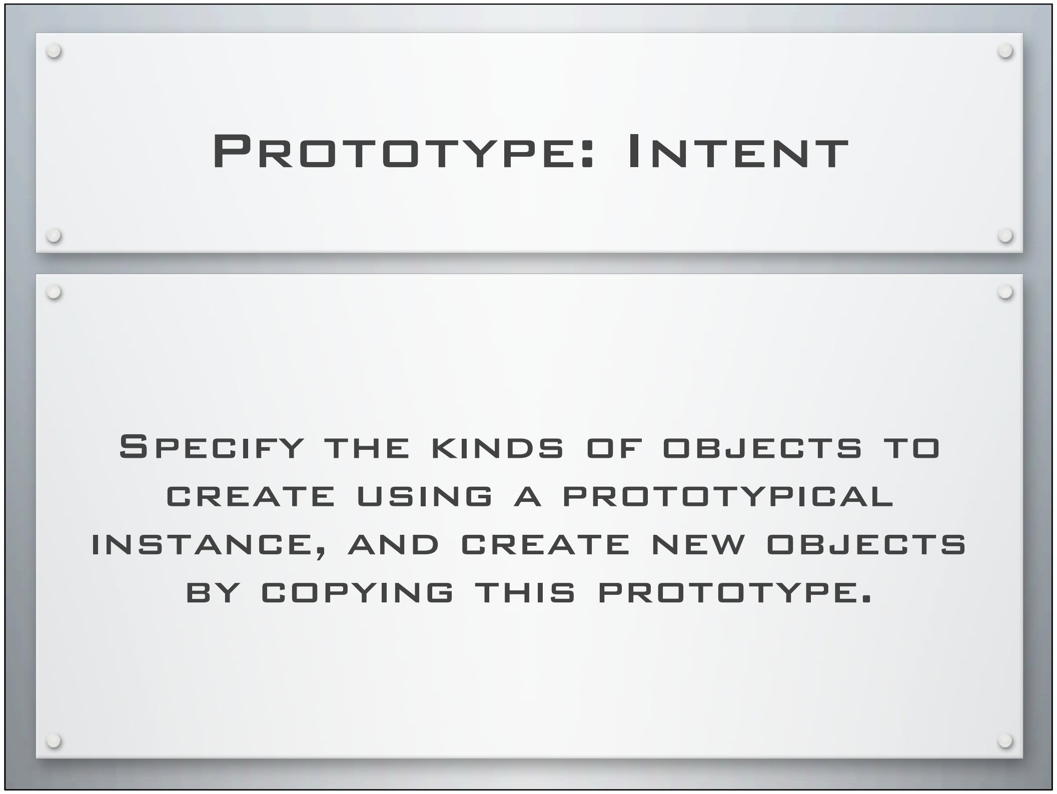
PROTOTYPE

DESIGN PATTERN

JUSTIN HENDRIX | CSPP 51023 | WINTER 2010

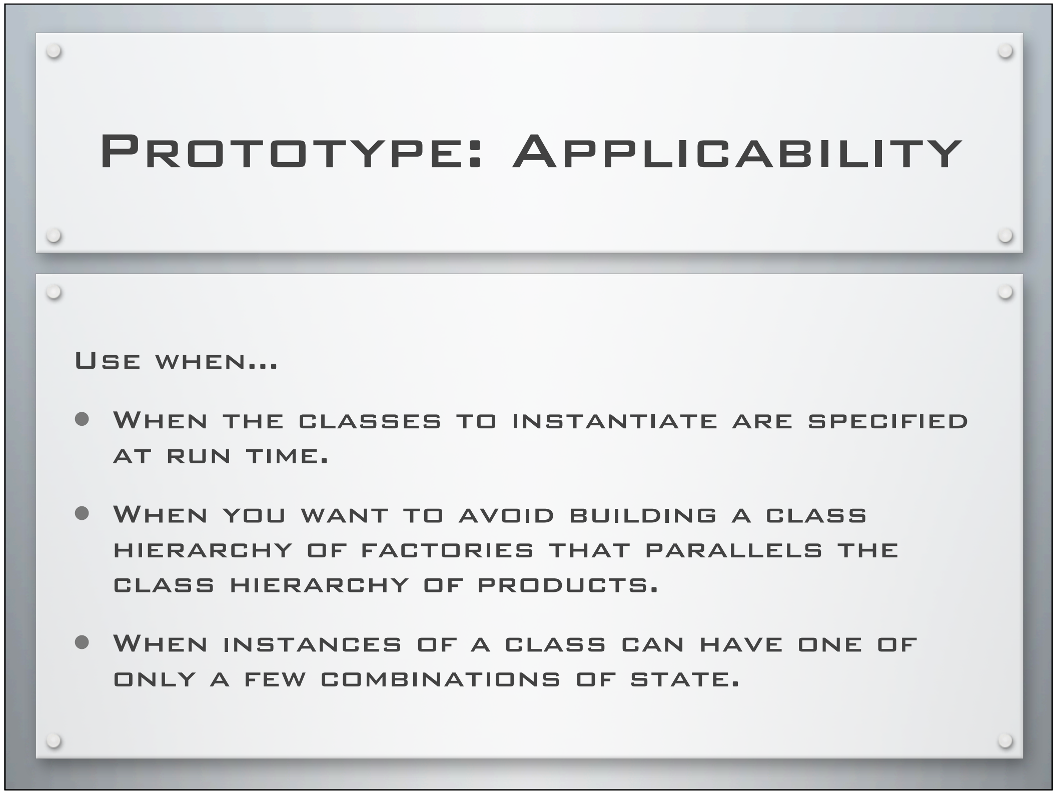
from: Design Patterns (Gamma, Helm, Johnson, and Vlissides)
p. 117- 126



Prototype: Intent

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Hide the concrete product classes from the client
-- reduce number of names clients know about
-- client work with application-specific classes without modification
when a system should be independent of how its products are created, composed and represented, and...

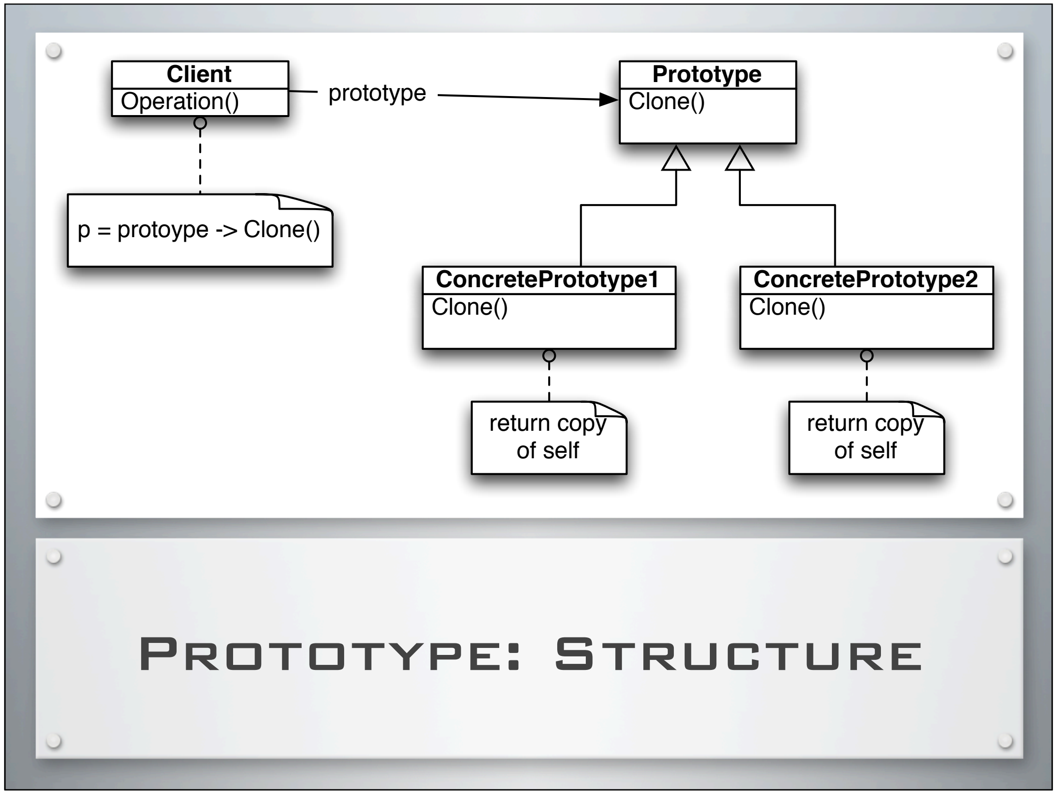


Prototype: Applicability

Use when...

- When the classes to instantiate are specified at run time.
- When you want to avoid building a class hierarchy of factories that parallels the class hierarchy of products.
- When instances of a class can have one of only a few combinations of state.

when a system should be independent of how its products are created, composed and represented, and...

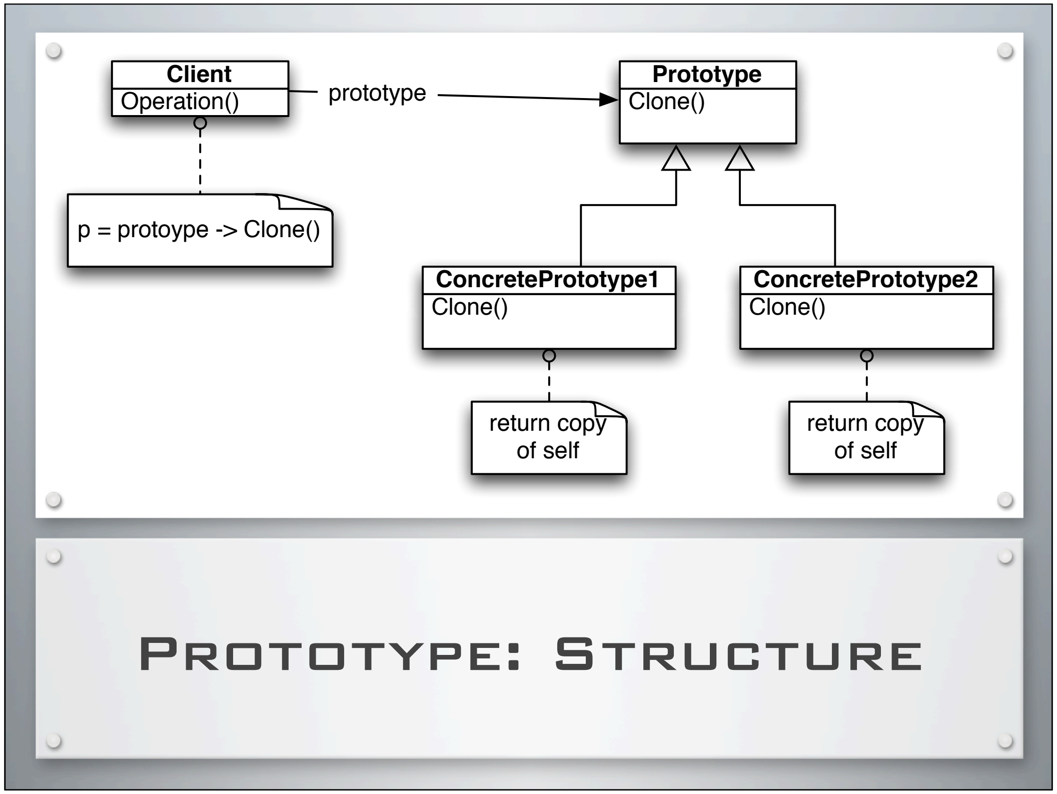


Prototype: Structure

A client asks a prototype to clone itself

Prototype: Participants

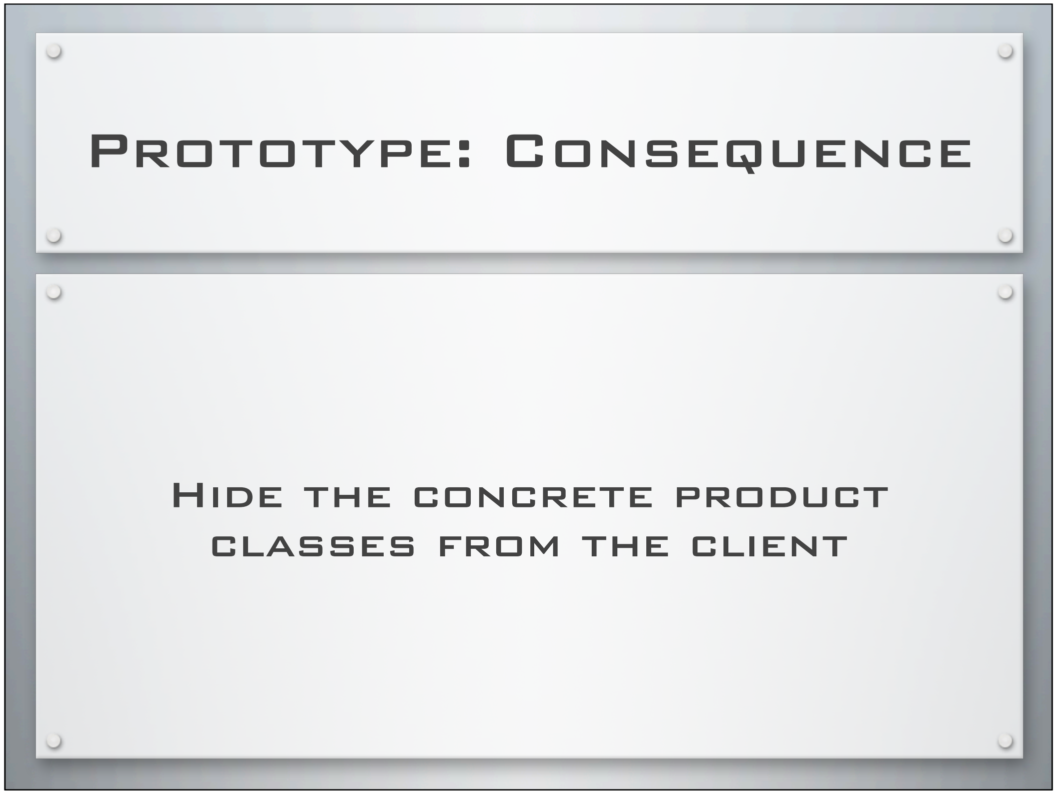
- Prototype
  - declares an interface for cloning itself
- ConcretePrototype
  - implements an operation for cloning itself
- Client
  - creates a new object by asking a prototype to clone itself



Prototype: Structure

A client asks a prototype to clone itself:

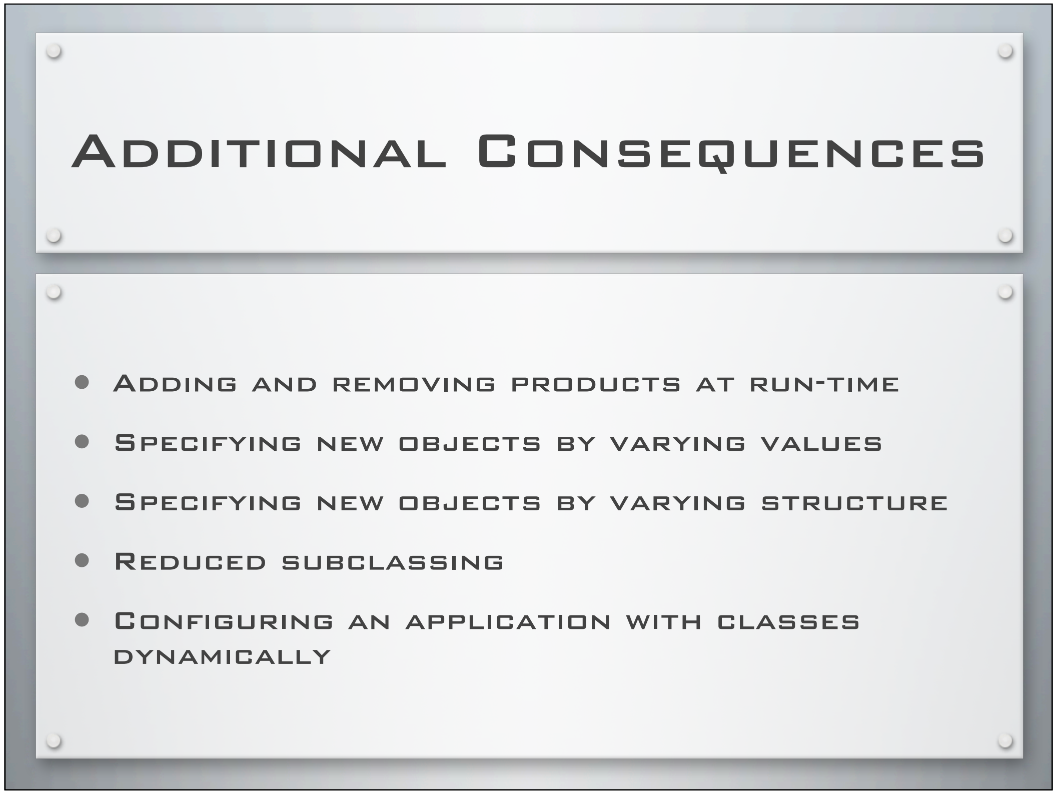
- Prototype
  - declares an interface for cloning itself
- ConcretePrototype
  - implements an operation for cloning itself
- Client
  - creates a new object by asking a prototype to clone itself



Prototype: Consequence

Hide the concrete product classes from the client

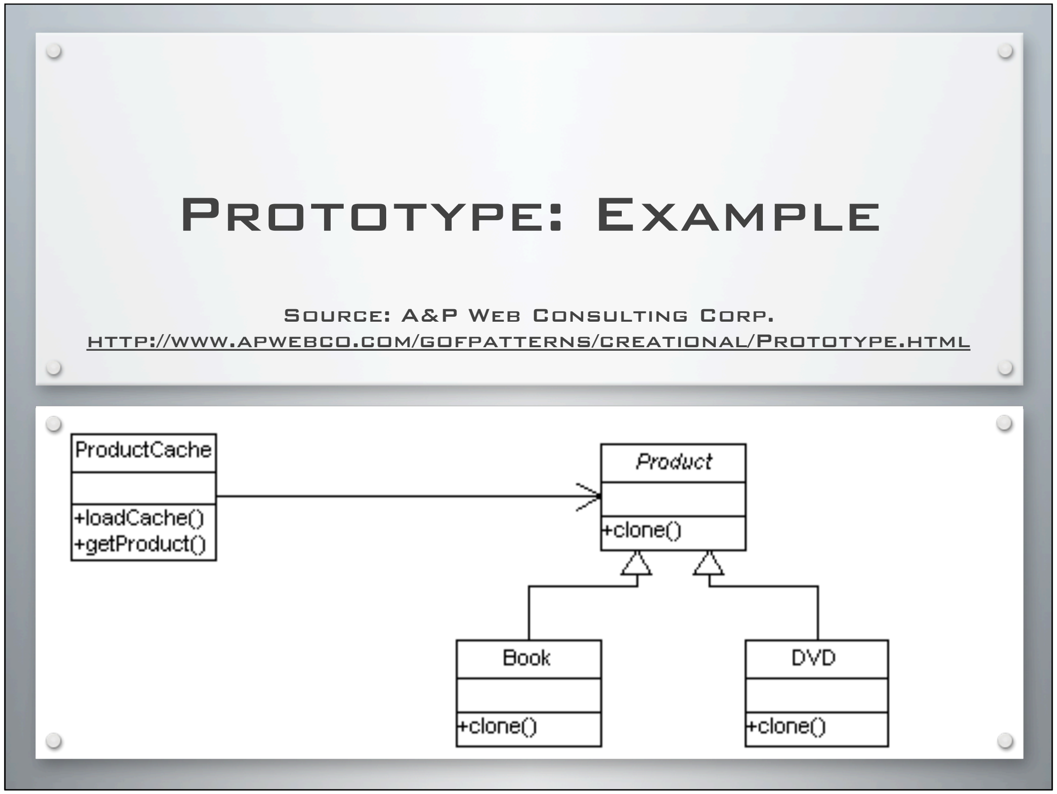
(Like the Abstract Factory and Builder)-- thus reducing the number of names clients know about
-- pattern lets a client work with application-specific classes without modification



Additional Consequences

- Adding and removing products at run-time
- Specifying new objects by varying values
- Specifying new objects by varying structure
- Reduced subclassing
- Configuring an application with classes dynamically

- incorporate new concrete product class in a system by registering a prototypical instance with the client, client can install an remove prototypes at run-time
- let users define new classes without “programming”, new objects defined by values not new classes
- i.e. subcircuits; using deep copy with Clone, circuits with different structures can be prototypes and reused
- clone a prototype rather than create new object; benefit mostly C++ where classses are not first-class objects; Smalltalk and Obj-C less benefit objects already act like prototypes
- run-time environment load classes dynamically through checking instance of each class into the prototype manager



Prototype: Example

Source: A&P Web Consulting Corp.
http://www.apwebco.com/gofpatterns/creational/Prototype.html

ProductCache
+loadCache()
+getProduct()

Product
+clone()

Book
+clone()

DVD
+clone()

Example assumptions:

- An e-commerce application gathers product information trough complex queries against a legacy database.
- The legacy database is updated at predefined intervals which are known.
- The number of products allows caching with a reasonable memory consumption.

When a user asks for information for a certain product the application could gather that information in two ways:

1. execute the complex query against legacy database, gather the information, and instantiate the object.
2. (prototype pattern) instantiate the objects at predefined intervals and keep them in a cache, when an object is requested, it is retrieved from cache and cloned. When the legacy database is updated, discard the content of the cache and re-load with new objects.

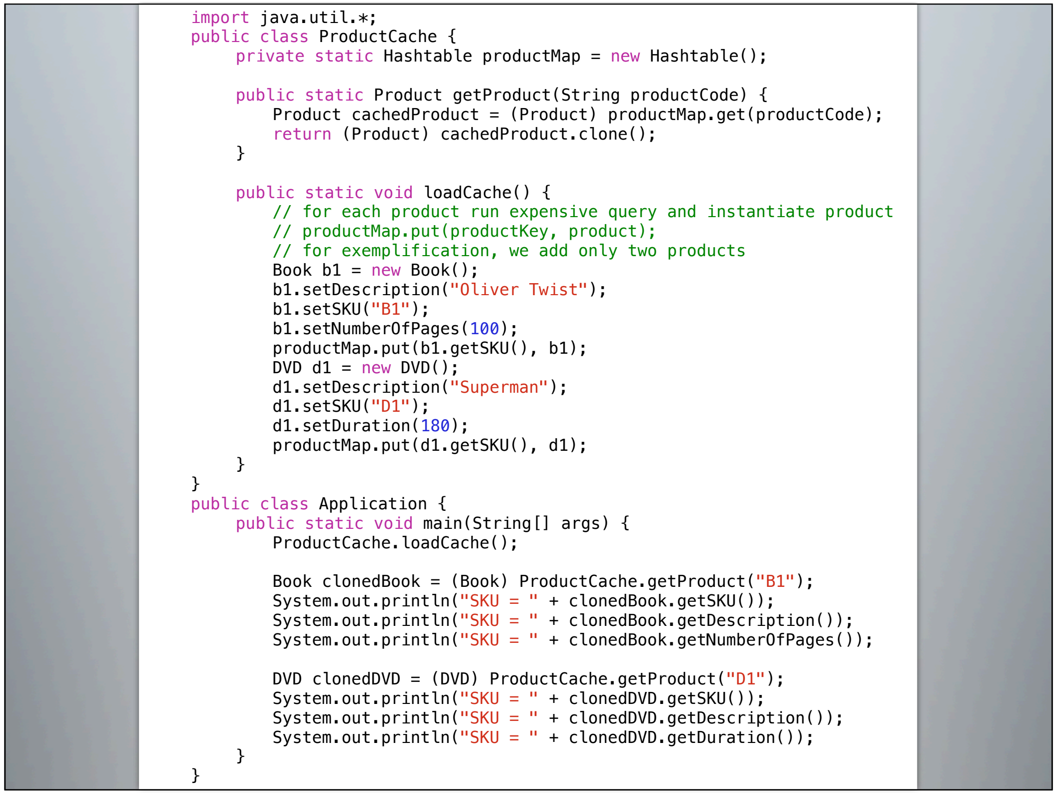
```java
public abstract class Product implements Cloneable {
        private String SKU;
        private String description;

        public Object clone() {
            Object clone = null;
            try {
                clone = super.clone();
            } catch (CloneNotSupportedException e) {
                e.printStackTrace();
            }
            return clone;
        }
        public String getDescription() {
            return description;
        }
        public String getSKU() {
            return SKU;
        }
        public void setDescription(String string) {
            description = string;
        }
        public void setSKU(String string) {
            SKU = string;
        }
   }
```

```java
public class Book extends Product {
    private int numberOfPages;

    public int getNumberOfPages() {
        return numberOfPages;
    }
    public void setNumberOfPages(int i) {
        numberOfPages = i;
    }
}
public class DVD extends Product {
    private int duration;

    public int getDuration() {
        return duration;
    }
    public void setDuration(int i) {
        duration = i;
    }
}
```



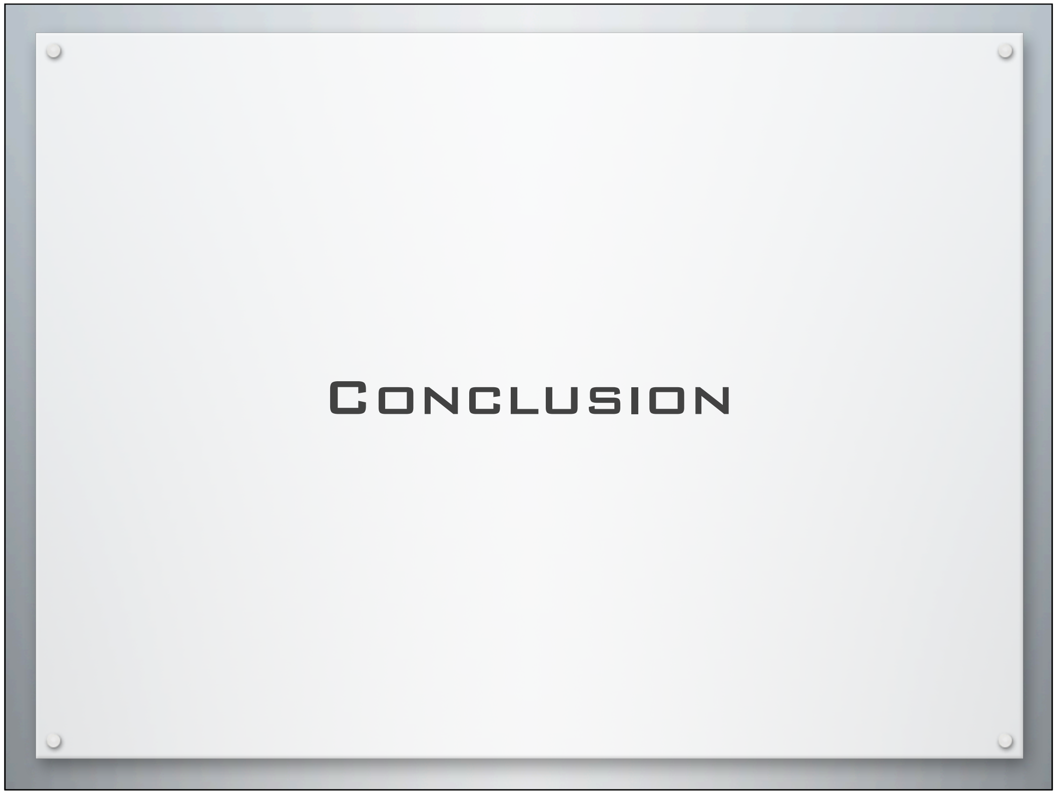
```java
import java.util.*;
public class ProductCache {
    private static Hashtable productMap = new Hashtable();

    public static Product getProduct(String productCode) {
        Product cachedProduct = (Product) productMap.get(productCode);
        return (Product) cachedProduct.clone();
    }

    public static void loadCache() {
        // for each product run expensive query and instantiate product
        // productMap.put(productKey, product);
        // for exemplification, we add only two products
        Book b1 = new Book();
        b1.setDescription("Oliver Twist");
        b1.setSKU("B1");
        b1.setNumberOfPages(100);
        productMap.put(b1.getSKU(), b1);
        DVD d1 = new DVD();
        d1.setDescription("Superman");
        d1.setSKU("D1");
        d1.setDuration(180);
        productMap.put(d1.getSKU(), d1);
    }
}
public class Application {
    public static void main(String[] args) {
        ProductCache.loadCache();

        Book clonedBook = (Book) ProductCache.getProduct("B1");
        System.out.println("SKU = " + clonedBook.getSKU());
        System.out.println("SKU = " + clonedBook.getDescription());
        System.out.println("SKU = " + clonedBook.getNumberOfPages());

        DVD clonedDVD = (DVD) ProductCache.getProduct("D1");
        System.out.println("SKU = " + clonedDVD.getSKU());
        System.out.println("SKU = " + clonedDVD.getDescription());
        System.out.println("SKU = " + clonedDVD.getDuration());
    }
}
```



Conclusion

-- Abstract Factory, Prototype, and Builder are more flexible than the Factory method, but also more complex
-- Typically designs start out using factory method and evolve toward the other creational patterns as the designer discovers where more flexibility is needed