Design Patterns in Java: Creational

Introduction & Prerequisites

Bryan Hansen

twitter: bh5k | http://www.linkedin.com/in/hansenbryan

Why

Communication

Common Vocabulary

Abstract Topic

Revisit

More than just a Singleton!

Pattern Groups

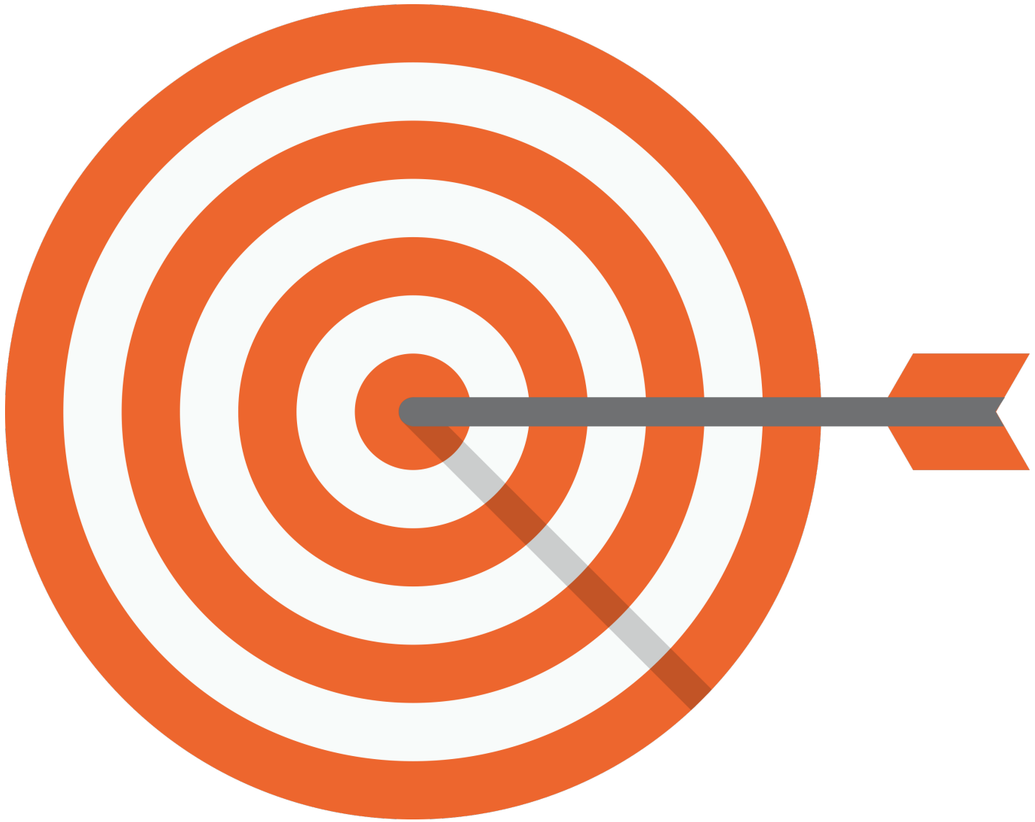
Creational

Structural

Behavioral

What

- Singleton
- Builder
- Prototype
- Factory
- AbstractFactory



How

- Overview
- Concepts
- Design
- Live example
- Demo, code your own
- Pitfalls
- Contrast
- Summary

Prerequisites

Java 7+

Eclipse or Spring STS

Singleton Pattern

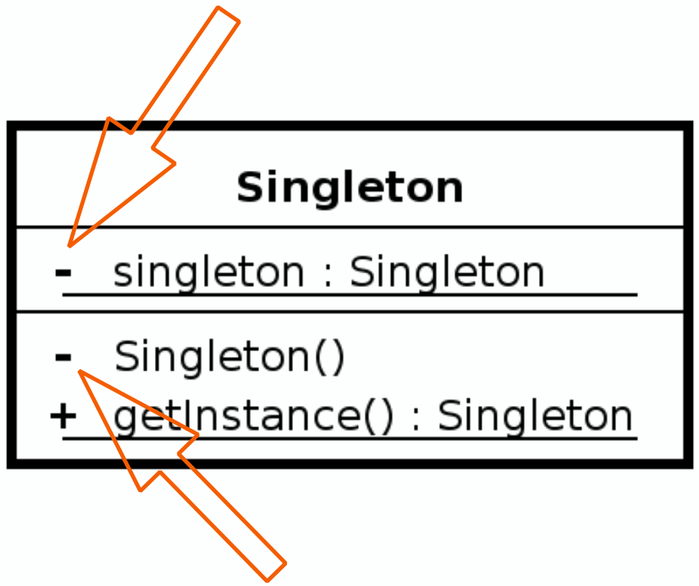
Bryan Hansen

twitter: bh5k | http://www.linkedin.com/in/hansenbryan

Concepts

- Only one instance created
- Guarantees control of a resource
- Lazily loaded
- Examples:
  - Runtime
  - Logger
  - Spring Beans
  - Graphic Managers

Design



Class is responsible for lifecycle

Static in nature

Needs to be thread safe

Private instance

Private constructor

No parameters required for construction

Everyday Example - Runtime Env

```
Runtime singletonRuntime = Runtime.getRuntime();

singletonRuntime.gc();

System.out.println(singletonRuntime);

Runtime anotherInstance = Runtime.getRuntime();

System.out.println(anotherInstance);

if(singletonRuntime == anotherInstance) {
   System.out.println(“They are the same instance”);
}
```

Exercise Singleton

Create Singleton

Demonstrate only one instance created

Lazy Loaded

Thread safe operation

Pitfalls

- Often overused
- Difficult to unit test
- If not careful, not thread-safe
- Sometimes confused for Factory
- java.util.Calendar is NOT a Singleton
  - Prototype

Contrast

Singleton

- Returns same instance
  - One constructor method - no args
- No Interface

Factory

- Returns various instances
  - Multiple constructors
- Interface driven
- Adaptable to environment more easily

Singleton Summary

- Guarantee one instance
- Easy to implement
- Solves a well defined problem
- Don’t abuse it

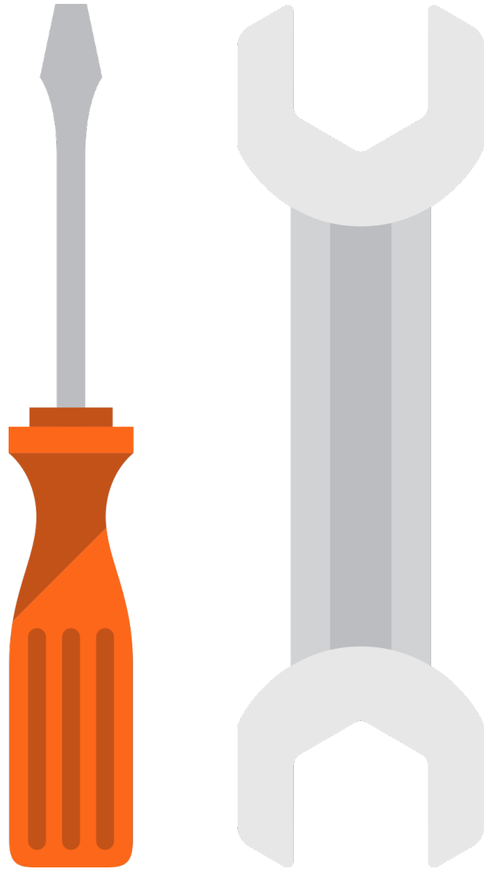
Builder Pattern

Bryan Hansen

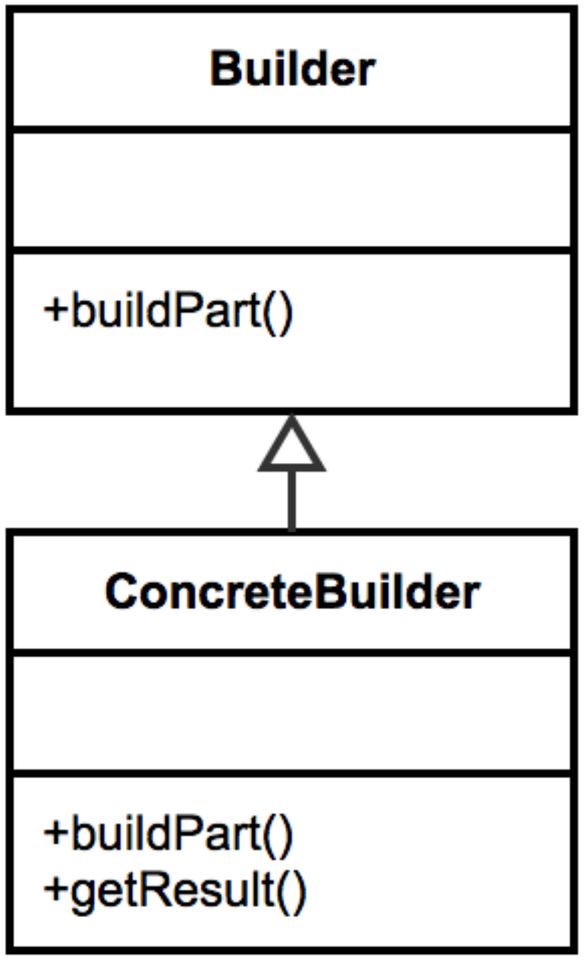
twitter: bh5k | http://www.linkedin.com/in/hansenbryan

Concepts

- Handles complex constructors
- Large number of parameters
- Immutability
- Examples:
  - StringBuilder
  - DocumentBuilder
  - Locale.Builder



Design



Flexibility over telescoping constructors

Static inner class

Calls appropriate constructor

Negates the need for exposed setters

Java 1.5+ can take advantage of Generics

Everyday Example - StringBuilder

```
StringBuilder builder = new StringBuilder();

builder.append("This is an example ");

builder.append("of the builder pattern. ");

builder.append("It has methods to append ");

builder.append("almost anything we want to a String. ");

builder.append(42);
```

Exercise Builder

Demonstrate Exposed Setters

Demonstrate Telescoping Constructors

Create Builder

Build Out Example

Pitfalls

- Immutable
- Inner static class
- Designed first
- Complexity
- Method returns object

Contrast

Builder

- Handles complex constructors
- No interface required
- Can be a separate class
- Works with legacy code

Prototype

- Implemented around a clone
- Avoids calling complex constructors
- Difficult to implement in legacy code

Builder Summary

- Creative way to deal with complexity
- Easy to implement
- Few drawbacks
- Can refactor in with separate class

Prototype Pattern

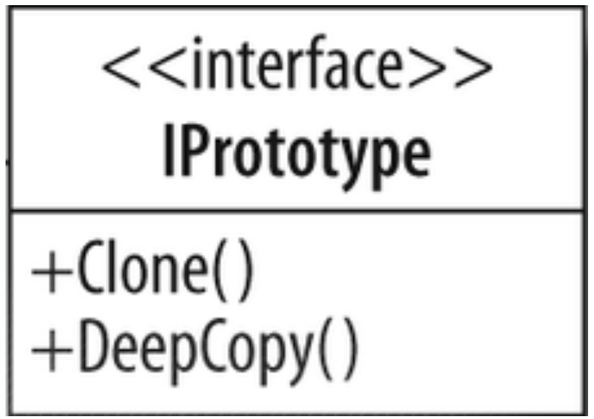
Bryan Hansen

twitter: bh5k | http://www.linkedin.com/in/hansenbryan

Concepts

- Avoids costly creation
- Avoids subclassing
- Typically doesn't use "new"
- Often utilizes an Interface
- Usually implemented with a Registry
- Example:
  - java.lang.Object#clone()

Design



Clone / Cloneable

Avoids keyword “new”

Although a copy, each instance unique

Costly construction not handled by client

Can still utilize parameters for construction

Shallow VS Deep Copy

Everyday Example - Object Clone

```
public class Statement implements Cloneable {

   public Statement(String sql, List<String> parameters, Record record) {
      this.sql = sql;
      this.parameters = parameters;
      this.record = record;
   }

   public Statement clone() {
      try {
         return (Statement) super.clone();
      } catch (CloneNotSupportedException e) {}
      return null;
   }
}
```

Exercise Prototype

Create Prototype

Demonstrate shallow copy

Create with a Registry

Pitfalls

- Sometimes not clear when to use
- Used with other patterns
  - Registry
- Shallow VS Deep Copy

Contrast

Prototype

- Lighter weight construction
  - Copy Constructor or Clone
- Shallow or Deep
- Copy of itself

Factory

- Flexible Objects
  - Multiple constructors
- Concrete Instance
- Fresh Instance

Prototype Summary

- Guarantee unique instance
- Often refactored in
- Can help with performance issues
- Don’t always jump to a Factory

Factory Method Pattern

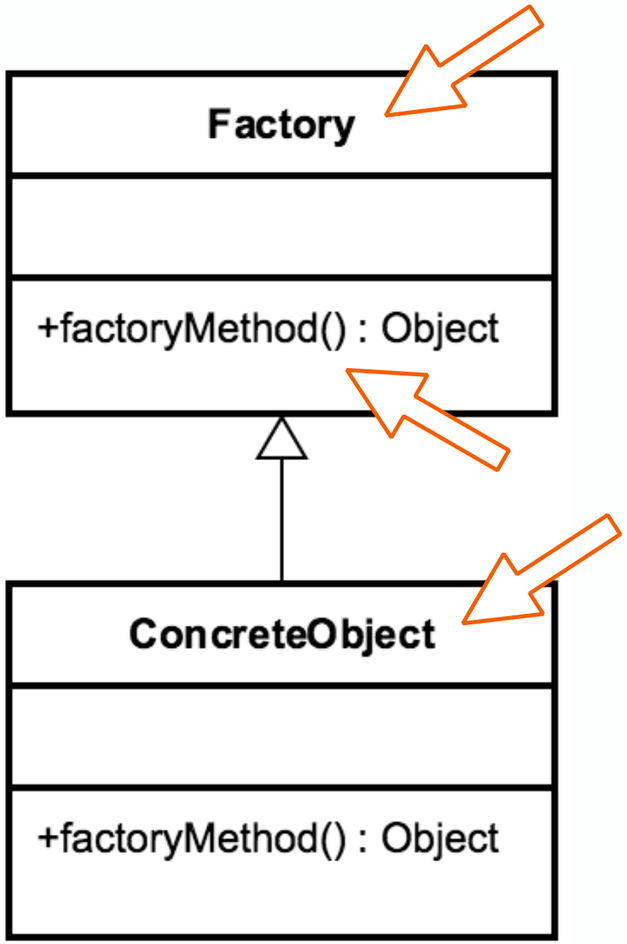
Bryan Hansen

twitter: bh5k | http://www.linkedin.com/in/hansenbryan

Concepts

- Doesn’t expose instantiation logic
- Defer to subclasses
- Common interface
- Specified by architecture, implemented by user
- Examples:
  - Calendar
  - ResourceBundle
  - NumberFormat

Design



Factory is responsible for lifecycle

Common Interface

Concrete Classes

Parameterized create method

Everyday Example - Calendar

```
Calendar cal = Calendar.getInstance();

System.out.println(cal);

System.out.println(cal.get(Calendar.DAY_OF_MONTH));
```

Exercise Factory

Create Pages

Create Website

Create Concrete Classes

Create Factory

Enum

Pitfalls

- Complexity
- Creation in subclass
- Refactoring

Contrast

Singleton

- Returns same instance
  - One constructor method - no args
- No Interface
- No Subclasses

Factory

- Returns various instances
  - Multiple constructors
- Interface driven
- Subclasses
- Adaptable to environment more easily

Factory Summary

- Parameter Driven
- Solves complex creation
- A little complex
- Opposite of a Singleton

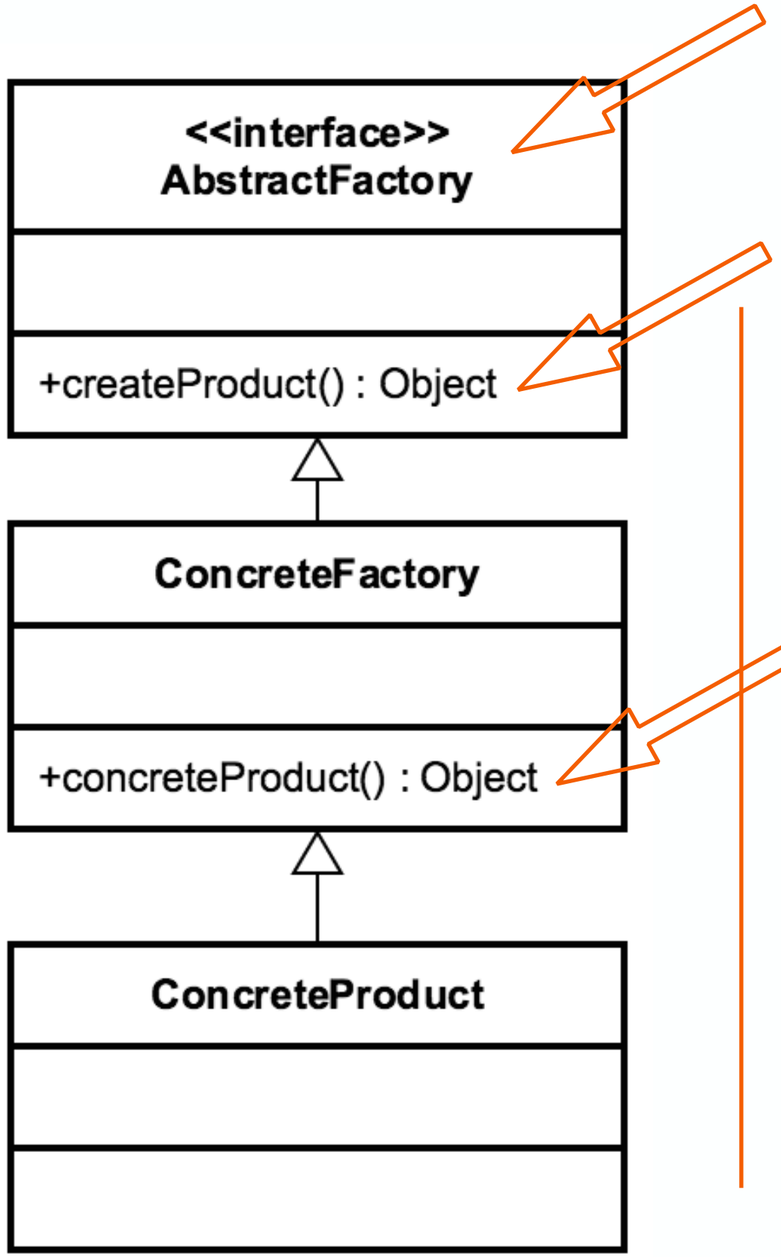
AbstractFactory Pattern

Bryan Hansen

twitter: bh5k | http://www.linkedin.com/in/hansenbryan

Concepts

- Factory of Factories
- Factory of related objects
- Common Interface
- Defer to Subclasses
- Examples:
  - DocumentBuilder
  - Frameworks



Design

Groups Factories together

Factory is responsible for lifecycle

Common Interface

Concrete Classes

Parameterized create method

Composition

Everyday Example - DocumentBuilderFactory

```
DocumentBuilderFactory abstractFactory =
DocumentBuilderFactory.newInstance();

DocumentBuilder factory = abstractFactory.newDocumentBuilder();

Document doc = factory.parse(bais);
```

Exercise AbstractFactory

Code Walkthrough

AbstractFactory

Factory

Product

Pitfalls

- Complexity
- Runtime switch
- Pattern within a pattern
- Problem specific
- Starts as a Factory

Contrast

Factory

- Returns various instances
  - Multiple constructors
- Interface driven
- Adaptable to environment more easily

AbstractFactory

- Implemented with a Factory
- Hides the Factory
- Abstracts Environment
- Built through Composition

AbstractFactory Summary

- Group of similar Factories
- Complex
- Heavy abstraction
- Framework pattern