

Abstract Decorators

For abstract decorators, just implement a decorator which have a no op. So implement the interface, and for the method exposed by the interface, just do: `method() { return aElement.method(); }`

Then you can extend the abstract decorator. Why do it, well, it's a boilerplate investment. More boilerplate for the abstract decorator, but the specific decorator will just have to do `super(pElement)` for the constructor, and `super.method()` for the other. Kinda useless, but hey...

Only use is that some method can have a default implementation. So you skip on the boilerplate of undecorated method.

==== It allows you to make the `.aElement` that you are decorating private, and it doesn't need to be protected

Comparator and Itererator

Interface	Functional Interface?	Function Object?	Purpose	Key Method(s)
<code>Comparator<T></code>	✓ Yes	✓ Yes	Compares two objects externally	<code>int compare(T o1, T o2)</code>
<code>Comparable<T></code>	✓ Yes	✓ Yes	Compares self to another object	<code>int compareTo(T other)</code>
<code>Iterator<T></code>	✗ No	✗ No	Traverses elements, maintains state	<code>boolean hasNext()</code> , <code>T next()</code>
<code>Iterable<T></code>	✗ No	✗ No	Represents a collection you can iterate	<code>Iterator<T> iterator()</code>

Code:

```
class MySpecialIterator<T> implements Iterator<T> {

    private T currentElement;
    private int currentElementIndex;

    private MySpecialCollection<T> data;
    public MySpecialIterator(MySpecialCollection<T> collection) {
        // Copy all.
        data = collection.deepcopy();

        currentElementIndex = 0;
        currentElement = data.getFirst();
        // Assuming a getFirst exist.
    }

    public bool hasNext() {
        // if can return current element
        // If null, can't return.
    }

    public T next() {
        T ret = currentElement;

        currentElement = currentElement.getNext(currentElementIndex);
        currentElementIndex++;

        return ret;
    }
}
```

```
class MySpecialCollection<T> implements Iterable<T>, Comparable<T> {

    ?Collection data;

    MySpecialIterator<T> iterator() {
        return MySpecialIterator(this);
    }

    Optional<T> getNext(int currentIndex) {
        // if there is a next element, returns it.
        // otherwise, returns null.
    }

    static Optional<T> getFirst() {
        // if there is a first, returns it, otherwise returns null;

        return Optional.of("#<First element>")
        // or
        return Optional.ofNullable("#<First element, but it might be null>")
        //or
        return Optional.empty() // first element is null;
    }

    // Other methods related to the collection.
    // add element, get element, remove

    // compare to doesn't go here. It goes for the special element.

    // Sort with comparable
    public static <T extends Comparable<T>> void sort(SpecialCollection col) {
        // ^Return type is void, <T extends Comparable<T>> is just some information.
        // for the compiler about T's information.
    }
}
```

```
// current, afterCurrent;

// if current.compareTo(after) > 0 , swap current and afterCurrent
// in the collection
}

public static <T> void sort(SpecialCollection col, Comparator<T> comp ) {
    // current, afterCurrent;

    // if comp.compare(current, afterCurrent) > 0 , swap current and afterCurrent
    // in the collection
}

// sort with coolness comparator

}

class SpecialElement implements Comparable <SpecialElement> {
    int compareTo(T other) {

        // returns (this - other) {for ints}

        // returns <0 if this is before other (other is after this) (this < other)
        // returns >0 if this is after other (other is before this). (this > other)
        // returns 0 if this == other.

        // It won't allow you to use the <, <=, >, >= operators.
        // No operator overloading
        // So you'd need wrappers for isLessThen, isBiggerThen, isBiggerEqual, isLessEqual
```

```
}

}

class CoolnessComparator<T> implements Comparator<T> {

    int compare(T o1, T o2) {
        // returns <0 if o1 is before o2 (o2 is after o1)
        // returns >0 if o1 is after o2 (o2 is before o1).
        // returns (o1 - o2) {for ints}

        // equal to o1.compareTo(o2)
        // if same comparaison metric

    }
}
```

====

Java optionals:

```
Optional<String> some = Optional.of("hello");           // ✓ non-null
Optional<String> maybeNull = Optional.ofNullable(null); // ✓ maybe-null
Optional<String> empty = Optional.empty();              // ✓ explicitly empty
```

```
Optional<String> name = Optional.of("François");
name.ifPresent(n -> System.out.println("Hello " + n));
```

```
String result = name.orElse("Default");  
String lazy = name.orElseGet(() -> expensiveDefault());  
String sure = name.orElseThrow();
```

Java functional interface:

Interface Name	Return Type	Method Signature	Package
Function<T, R>	R	apply(T)	java.util.function
BiFunction<T, U, R>	R	apply(T, U)	java.util.function
Consumer<T>	void	accept(T)	java.util.function
Supplier<T>	T	get()	java.util.function
Predicate<T>	boolean	test(T)	java.util.function
UnaryOperator<T>	T	apply(T)	java.util.function
BinaryOperator<T>	T	apply(T, T)	java.util.function
Runnable	void	run()	java.lang



Java Functional Methods Cheat Sheet



Core Functional Methods (Beyond `map()` and `flatMap()`)

Method	Input Type	Output Type	Description	Available In
<code>map(f: T → R)</code>	Regular function	<code>Optional<R></code> , <code>Stream<R></code>	Transforms value	Optional, Stream
<code>flatMap(f: T → Optional<R>)</code>	Optional-returning function	<code>Optional<R></code> , <code>Stream<R></code>	Flattens nested optionals or streams	Optional, Stream
<code>filter(f: T → boolean)</code>	Predicate	<code>Optional<T></code> , <code>Stream<T></code>	Keeps value if condition passes	Optional, Stream
<code>peek(f: T → void)</code>	Side-effect consumer	<code>Stream<T></code>	Debug/log each value, doesn't alter stream	Stream
<code>orElse(T)</code>	Value	<code>T</code>	Returns value if present, else fallback	Optional
<code>orElseGet(Supplier<T>)</code>	Supplier	<code>T</code>	Lazily compute fallback	Optional
<code>orElseThrow()</code>	(none)	<code>T</code> or throws	Throws <code>NoSuchElementException</code> if empty	Optional
<code>orElseThrow(Supplier)</code>	Exception supplier	<code>T</code> or throws	Throws custom exception	Optional
<code>or(Supplier<Optional<T>>)</code>	Supplier of fallback optional	<code>Optional<T></code>	Fallback optional	Optional
<code>ifPresent(Consumer<T>)</code>	Consumer	<code>void</code>	Executes block if value present	Optional
<code>ifPresentOrElse(...)</code>	Consumer + Runnable	<code>void</code>	Handles both present and empty	Optional (Java 9+)
<code>distinct()</code>	(none)	<code>Stream<T></code>	Removes duplicate elements	Stream

Method	Input Type	Output Type	Description	Available In
<code>sorted()</code> / <code>sorted(Comparator)</code>	(none) / <code>comparator</code>	<code>Stream<T></code>	Sorts elements	Stream
<code>limit(n)</code>	<code>int</code>	<code>Stream<T></code>	Truncates after n elements	Stream
<code>skip(n)</code>	<code>int</code>	<code>Stream<T></code>	Skips the first n elements	Stream
<code>collect(...)</code>	Collector	Varies	Terminal operation to gather into collection	Stream
<code>reduce(...)</code>	Accumulator	Optional / T	Combines stream elements into one	Stream
<code>anyMatch(Predicate)</code>	Predicate	boolean	True if any element matches	Stream
<code>allMatch(Predicate)</code>	Predicate	boolean	True if all elements match	Stream
<code>findFirst()</code> / <code>findAny()</code>	(none)	<code>Optional<T></code>	Retrieves first/any value	Stream



Optional vs Stream Example

```
Optional<String> name = Optional.of("Alice");
Optional<Integer> length = name.map(String::length); // Optional[5]

Stream<String> names = Stream.of("Alice", "Bob");
Stream<Integer> lengths = names.map(String::length); // Stream of 5, 3

// collect a list of optional
```



```
List<Optional<T>> optionals = ...;

List<T> results = optionals.stream()
    .filter(Optional::isPresent)    // keep only non-empty optionals
    .map(Optional::get)             // unwrap the values
    .collect(Collectors.toList());  // collect into a new list
```

Java Lambdas:

```
// different way of creating a java function
Predicate<Card> blackCardFilter =
    (Card card) -> card.getSuit().getColor() == Suit.Color.BLACK;
```

```
Predicate<Card> blackCards = (Card card) ->
{ return card.getSuit().getColor() == Suit.Color.BLACK; }
```

```
Predicate<Card> blackCardFilter =
    (card) -> card.getSuit().getColor() == Suit.Color.BLACK;
```

```
Predicate<Card> blackCardFilter =
    card -> card.getSuit().getColor() == Suit.Color.BLACK;
```

```
Supplier<Integer> get5 = () -> 5;
```

```
// Java lambdas (Anonymous functions)
```


```
// ===== Class (Static) methods =====
cards.removeIf(card -> CardUtils.hasBlackSuit(card));
```

```
cards.removeIf(CardUtils::hasBlackSuit);

// ===== Instance methods =====
cards.removeIf(card -> deck.topSameColorAs(card));
cards.removeIf(deck::topSameColorAs)
// Works for 0 or many arguments

// Using a predicate
cards.removeIf(blackCardFilter)
```

Java FlatMap vs Map:

Feature	map()	flatMap()
Input	Function: $T \rightarrow R$	Function: $T \rightarrow \text{Optional}<R>$
Output	<code>Optional<R></code>	<code>Optional<R></code>
Use When	You have a regular value result	You already return an <code>optional</code> inside the lambda
Avoids	Nested <code>Optional<Optional<R>></code>	 Yes

==== Map: if $f: T \rightarrow R$

$T.\text{map}(f: T \rightarrow R) \rightarrow R$ $\text{Optional}.\text{map}(f: T \rightarrow R) \rightarrow \text{Optional}(R)$

==== FlatMap

if $f: T \rightarrow \text{Optional}$ Then,

`Optional.flatMap(f: T -> Optional) --> Optional T.flatMap(f: T -> Optional) --> Optional`

So this is for functions of

==== if use flatmap when `f: T->R`, compile time error

if use map whne `f: T->Optional`

optional nesting: `Optional<Optional>`



JUnit & Unit Testing Cheat Sheet



1. JUnit Basics (5.2, 5.5, 5.6)



Key Annotations

Annotation	Purpose
@Test	Marks a test method
@BeforeEach	Runs before every test method
@AfterEach	Runs after every test method
@BeforeAll	Runs once before all tests (must be static)
@AfterAll	Runs once after all tests (must be static)
@Disabled	Skips the test
@DisplayName	Custom name for the test

Assertions (`org.junit.jupiter.api.Assertions`)

Assertion	Purpose
<code>assertEquals(expected, actual)</code>	Checks equality
<code>assertTrue(condition)</code>	Checks that condition is true
<code>assertFalse(condition)</code>	Checks that condition is false
<code>assertNull(value)</code>	Asserts the value is null
<code>assertNotNull(value)</code>	Asserts the value is not null
<code>assertThrows(Exception.class, () -> ...)</code>	Verifies an exception is thrown
<code>assertAll(...)</code>	Groups multiple assertions
<code>assertTimeout(Duration, Executable)</code>	Fails if exceeds time limit

Java Reflection API Cheat Sheet

Class-Level Methods (`java.lang.Class`)

Method	Description
<code>getName()</code>	Fully qualified class name
<code>getSimpleName()</code>	Class name without package
<code>getPackage()</code>	Gets the package object

Method	Description
<code>getSuperclass()</code>	Gets the superclass class
<code>getInterfaces()</code>	Interfaces implemented by the class
<code>getModifiers()</code>	Returns class modifiers (use with <code>Modifier</code>)
<code>isInterface()</code>	Checks if it's an interface
<code>isEnum()</code>	Checks if it's an enum
<code>newInstance()</code> ⚠	Creates a new instance (deprecated in Java 9+)

Field Methods (`java.lang.reflect.Field`)

Method	Description
<code>getDeclaredFields()</code>	All fields declared in class (incl. private)
<code>getFields()</code>	Only public fields (incl. inherited)
<code>getName()</code>	Gets the field name
<code>getType()</code>	Gets the field's type
<code>get(Object obj)</code>	Gets field value from object
<code>set(Object obj, val)</code>	Sets field value on object
<code>setAccessible(true)</code>	Bypasses access control
<code>isSynthetic()</code>	Checks if field was generated by compiler

Method Methods (`java.lang.reflect.Method`)

Method	Description
<code>getDeclaredMethods()</code>	All methods declared in class (incl. private)
<code>getMethods()</code>	Only public methods (incl. inherited)
<code>getName()</code>	Gets the method name
<code>getParameterTypes()</code>	Gets parameter types
<code>getReturnType()</code>	Gets return type
<code>getModifiers()</code>	Gets method modifiers
<code>invoke(obj, args...)</code>	Invokes method on object with arguments
<code>setAccessible(true)</code>	Allows access to private methods

Constructor Methods (`java.lang.reflect.Constructor`)

Method	Description
<code>getDeclaredConstructors()</code>	All constructors declared (incl. private)
<code>getConstructors()</code>	Only public constructors
<code>getParameterTypes()</code>	Gets constructor parameter types
<code>newInstance(args...)</code>	Creates new instance using constructor
<code>setAccessible(true)</code>	Allows access to private constructors



Annotation Methods

Method	Description
<code>getAnnotations()</code>	All annotations on element
<code>getAnnotation(Class<A>)</code>	Gets a specific annotation
<code>isAnnotationPresent(Class<A>)</code>	Checks if annotation is present



Common: Access Control

Method	Description
<code>setAccessible(true)</code>	Used with fields/methods/constructors to allow access to private members

Reflections Example

```
Class<Card> cardClass1 = Card.class
Class<?> cardClass2 = card.getClass()
// ? is a wildcard, meaning any class. Because if we have Card class, then why use an instance

// if you have only the name:

try {
    String fullyQualifiedName = "cards.Card";
    // cards is the package name.
    Class<Card> cardClass = (Class<Card>) Class.forName(fullyQualifiedName);
} catch (ClassNotFoundException e) {
```

```
        e.printStackTrace();
    }

// Creating a new instance
try {
    Card card1 = Card.get(Rank.ACE, Suit.CLUBS);
    Constructor<Card> cardConstructor = Card.class.getDeclaredConstructor(Rank.class, Suit.class);
    cardConstructor.setAccessible(true);
    Card card2 = cardConstructor.newInstance(Rank.ACE, Suit.CLUBS);
    System.out.println(card1 == card2);
} catch (ReflectiveOperationException e) {
    e.printStackTrace();
}

try {
    Card card = Card.get(Rank.TWO, Suit.CLUBS);
    Field rankField = Card.class.getDeclaredField("aRank");
    rankField.setAccessible(true);
    rankField.set(card, Rank.ACE);
    System.out.println(card);
} catch (ReflectiveOperationException e) {
    e.printStackTrace();
}

// getDeclaredMethod("Method name", ClassType(Method arg1), ClassType(method arg2), ... )
// getDeclaredMethod("Method name", ClassType(Method arg1), ClassType(method arg2), ClassType(Method arg3), ... )
// getDeclaredMethod("Method name", Integer.class, float.class, Banana.class, ... )
// Same for get Declared constructor

// Types:
// Field, Constructor<T>, Method.
// Type.class.getDeclared#X("name", if args: Type(arg1).class, Type(arg2).class, ... __VARGS__ = *args)
```


Testing `private` methods

```
public class TestFoundationPile {
    private FoundationPile aPile = new FoundationPile();

    // Private method getPreviousCard:
    private Optional<Card> getPreviousCard(Card pCard) {
        try {
            Method method = FoundationPile.class.getDeclaredMethod("getPreviousCard", Card.class);
            method.setAccessible(true);
            return (Optional<Card>) method.invoke(aPile, pCard);
        }
        catch (ReflectiveOperationException exception) {
            fail();
            return Optional.empty();
        }
    }

    @Test
    public void testGetPreviousCard_empty() {
        assertFalse( getPreviousCard( Card.get(Rank.ACE, Suit.CLUBS) ).isPresent() );
    }
}
```

Exmample of tests:

```
@Test
public void testPeek_Empty() {
    assertThrows(EmptyStackException.class, () -> aPile.peek());
}
```

```
// Don't write test for stuff that has  
  
@Test  
void testWithdrawReducesBalance() {  
    // Arrange  
    Account acc = new Account(100);  
  
    // Act  
    acc.withdraw(30);  
  
    // Assert  
    assertEquals(70, acc.getBalance());  
}
```

2. Unit Testing Principles (5.1, 5.5, 5.6, 5.7)

Good tests are:

- Fast / Speed
- Independent
- Deterministic / Repeatable
- Focus
- Readable

Liskov Substitution Principle (LSP)

- Per LSP, methods of a subclass: • cannot have stricter preconditions; • cannot have less strict postconditions; • cannot take more specific types as parameters; • cannot make the method less accessible (e.g., public -> protected); • cannot throw more checked exceptions; and • cannot have a less specific return type. • The last four are automatically checked by the compil

Law of Demeter

- Code in a method should only access: • the instance variables of its implicit parameter (this); • the arguments passed to the method; • any new object(s) created within the method; • (if need be) globally available objects. • Max delegation chain length: | Inside the class: 2 | Outside : 1

Access modifier:



Summary Table

Modifier	Class	Package	Subclass	World (everyone)
private	✓	✗	✗	✗
default (no modifier)	✓	✓	✗	✗
protected	✓	✓	✓	✗
public	✓	✓	✓	✓

Final keyword

- Final has a different meaning for fields, methods, classes:
- A final field cannot be assigned a value more than once.

- A final method cannot be overridden.
- A final class cannot be inherited

Todo:

Prototype, Clone, Gui, event, Reflection

Anti pattern

Anti P Name	Explanation
appropriate Intimacy	Class uses another's internal state Delegate behavior, enforce encapsulation
Temporary Field	Field not always needed Move to local var or extract class
Message Chain	Long access chains (a.b().c().d()) Add intermediate methods, Law of Demeter
Long Method	Method does too much, hard to follow Extract smaller methods, clarify logic

More generally:



Java Design Anti-Patterns Cheat Sheet

Anti-Pattern	Reference	Description
Primitive Obsession	2.2	Overusing primitive types instead of small objects for concepts (e.g. using <code>String</code> for phone numbers).

Anti-Pattern	Reference	Description
Inappropriate Intimacy	2.5	One class accessing the private details of another too frequently; tight coupling.
Switch Statement	3.4	Using many <code>switch / if</code> statements to control logic instead of polymorphism.
Speculative Generality	4.4	Writing general-purpose code "just in case", before it's needed.
Temporary Field	4.4	Fields that are only used in certain circumstances, often left <code>null</code> .
Long Method	4.6	A method that is excessively long and hard to read or test.
God Class	6.1	A class that does too much — centralizes all functionality and logic.
Message Chain	6.9	A sequence of method calls like <code>a.getB().getC().doSomething()</code> ; violates Law of Demeter.
Pairwise Dependencies	8.1, 8.2	Classes/modules that require knowledge of each other's internal workings or lifecycle.

Prototype

when

Object creation is expensive (e.g. loading data, network requests)

You need many similar objects with small variations

You want to avoid subclassing factories

You need to preserve state or structure in co

how

You define a base interface or abstract class with a clone() method.

Concrete classes implement clone() to duplicate themselves.

Clients call clone() on a prototype to get a new instance.

```
Circle original = new Circle();
original.color = "red";
original.radius = 10;
```

```
Circle copy = original.clone();
copy.radius = 20;
```

Clone

Make all class in the class heirachy implement clonable interface. Recursively. Use constructor or cloning

```
public MemorizingDeck clone() {
    MemorizingDeck clone = (MemorizingDeck) super.clone();
    clone.aDrawnCards = new CardStack(aDrawnCards);
    return clone;
}
```

}

```
# super(args) : constructor
# super.method(args) : methods
# can't super for static methods
```

Object Diagram:

	objectName: ObjectType				

	<fields>				
	aggragetedFieldNName = -----		----->	:Type	# annonymous

				Fields = ...	
	primitiveField:String = "Value"			# because name already said.	

Sequence diagrams:

Like object diagram

```
| Object |
|
|
```

```
(--- horizontal)

Rectangle _____> Calls (full line)

<----- return value. (doted line)
```

```
# Top: Start of the code.
# Bottom, end of the method code.
```

```
wizardPlayer.getActiveItemMO()
```

client	Wizard Player	Item
-	-	-
R	R	R
R	R_getActiveItemMO_>	R
R	R <-----	R
R <-----	R	R
R	R	R
R	R	R
R	R	R

Observer:

```
=====Model=====
```

```
Field:
```

```
    Data and state
```

```
    List<Observers>
```


Methods:

- add/remove observer

- get/Set Data.

- #optional: notifyObservers (not called in set if batch processing)

=====Observer=====

Field:

- Representation of accurate model data

- Or model reference

method:

- newNumber / numberChanged.

- newNumber(int 5): (push strategy)

- newNumber(pModel) pModel.getIntegerData() (pull strategy)

- ^^ Create a new interface which the Model implement.

- The interface only has getter method. So unmodifiable.

```
// Flexibility, doesn't need to just be number changed. Doesn't always need to be notified.
```

```
public interface Observer {  
    default void increased(int pNumber) { }  
    default void decreased(int pNumber) { }  
    default void changedToMax(int pNumber) { }  
    default void changedToMin(int pNumber) { }  
}
```

```
// Only implement what you need, and can split into multiple interface if needed
```

Java FX, GUI, Event Handling:

Gui components

Gui components (Object diagram)

```
public class LuckyNumber extends Application {
    @Override
    public void start(Stage pStage) {
        Model model = new Model(); // observer
        GridPane root = new GridPane();
        // Panel classes defined earlier.
        root.add(new SliderPanel(model), 0, 0, 1, 1);
        root.add(new IntegerPanel(model), 0, 1, 1, 1);
        root.add(new TextPanel(model), 0, 2, 1, 1);
        pStage.setScene(new Scene(root));
        pStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}

public class IntegerPanel extends Parent implements Observer {
    private TextField aText = new TextField();
    private Model aModel;
    public IntegerPanel(Model pModel) {
        aModel = pModel;
        // register as an observer of the model
        aModel.addObserver(this);
        aText.setText(new Integer(aModel.getNumber()).toString());
        // add the text field to the component graph
        getChildren().add(aText);
        ...
    }
}
```

```
        // will be called when notified by the model that number has changed
        public void numberChanged(int pNumber) {
            aText.setText(new Integer(pNumber).toString());
        }
    }
```

```
// with event loop:
// our defined observer.
```

```
public class IntegerPanel extends Parent implements Observer {
    private TextField aText = new TextField();
    private Model aModel;

    public IntegerPanel(Model pModel) {

        aModel = pModel;
        aModel.addObserver(this);
        aText.setText(new Integer(aModel.getNumber()).toString());
        getChildren().add(aText);

        aText.setOnAction(new EventHandler<ActionEvent>() {
            public void handle(ActionEvent pEvent) {
                int number;
                try {
                    number = Integer.parseInt(aText.getText());
                }
                catch(NumberFormatException pException ) {
                    number = 1;
                }
                aModel.setNumber(number);
            }
        });
    }
}
```

// Observer code:

```
public class Model {
    private int aNumber = 5;
    private List<Observer> aObservers = new ArrayList<>();
    public void addObserver(Observer pObserver) {
        aObservers.add(pObserver);
    }
    public void removeObserver(Observer pObserver) {
        aObservers.remove(pObserver);
    }

    private void notifyObservers() {
        for (Observer observer : aObservers) {
            observer.numberChanged(aNumber);
        }
    }

    public void setNumber(int pNumber) {
        aNumber = pNumber;
        notifyObservers();
    }
}

public interface Observer {
    void numberChanged(int pNumber);
}
```

End

Write nothing here