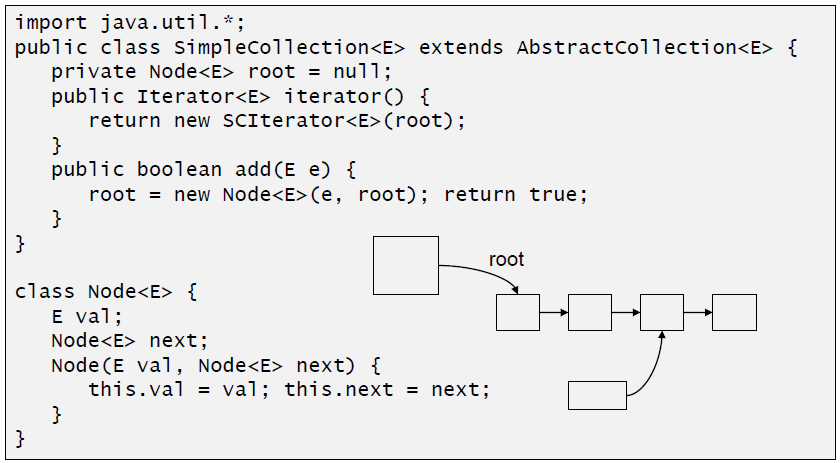
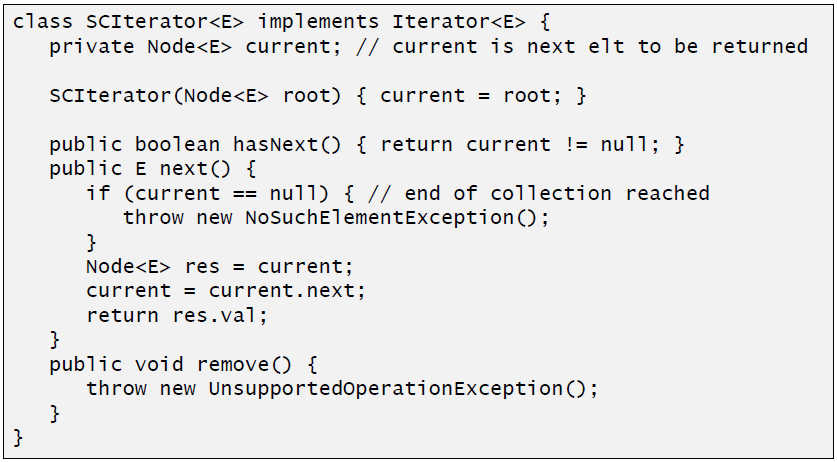
# Core Components of OOP

* Encapsulation: Information hiding & abstraction
* Inheritance: By extending and modifying existing classes (Subclasses)
* Polymorphism: Dynamic invocation where an object in a type hierarchy can have different types and static types
* Interfaces: Define a type and set of methods
* Abstract class (or default methods in Java): Predefined general functions, in general default behaviour (Semi finished product)
* Concrete class
* General design principle: Program to an interface and not an implementation Set s = new HashSet();

# Collections

* Small set of interfaces: No special interfaces for immutable collections (add/remove disabled), extensible only (remove disabled) or collections which accept null objects 🡪 Throw exceptions
* Set and List are based on Collection, Map is freestanding

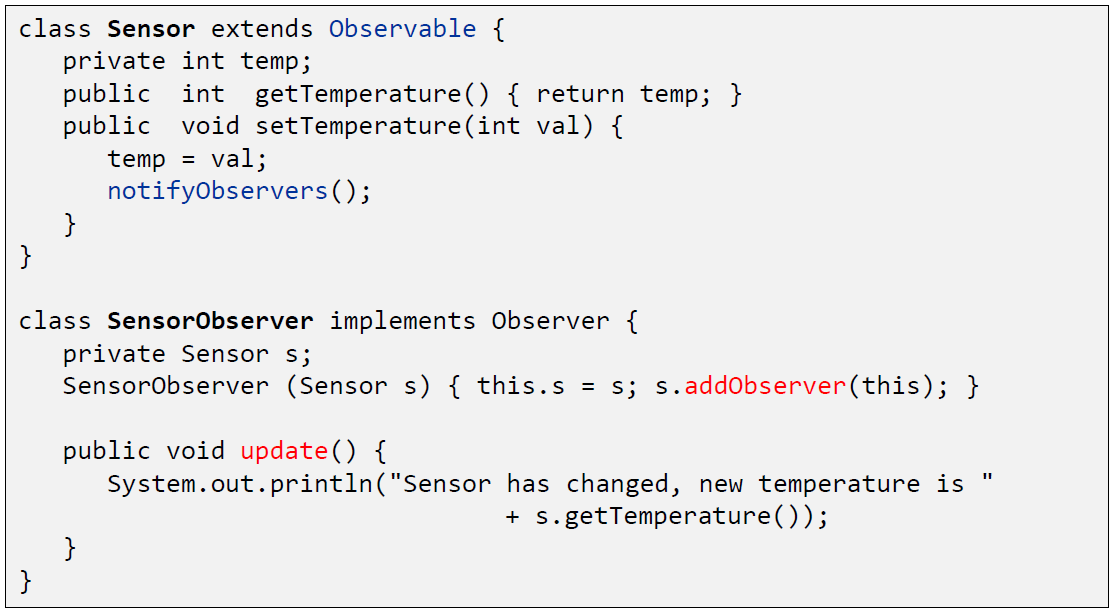
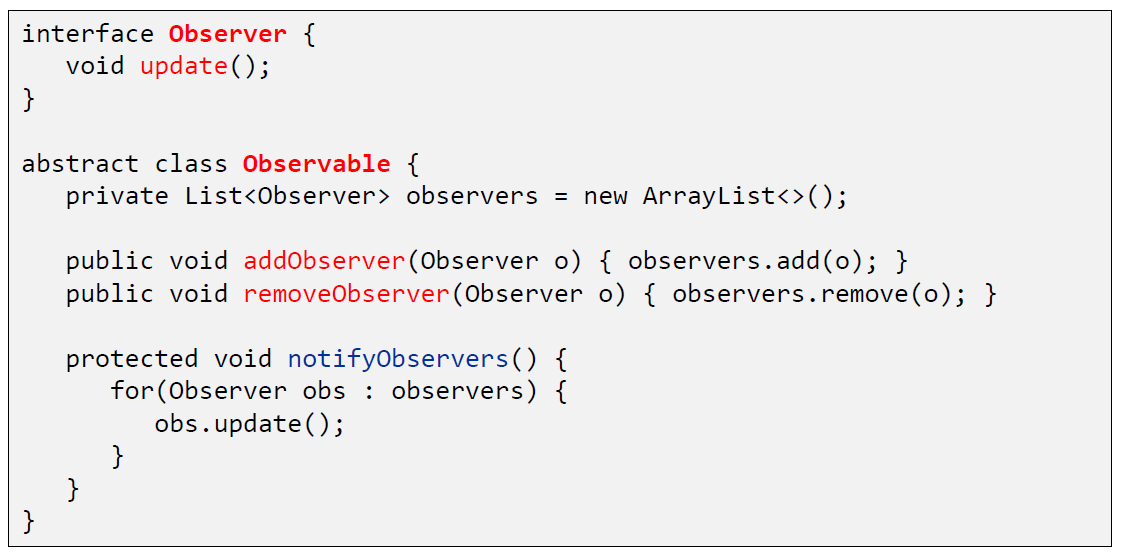
 

# Design Pattern

* Description: Reusable solutions to recurring problems that we encounter during software development
* Description: Description of a frequently occurring structure of cooperating components, which solves a general design problem in a particular context
* Reasons: Patterns enable programmers to recognize a problem and immediately determine the solution without having to stop and analyse the problem first
* Patterns do provide common vocabulary for communicating about design
* Patterns do not provide an exact solution, only provide a solution schema
* Types of patterns: Architectural patterns (System design), design patterns (Micro architectures) and coding patterns, Idioms, recipes

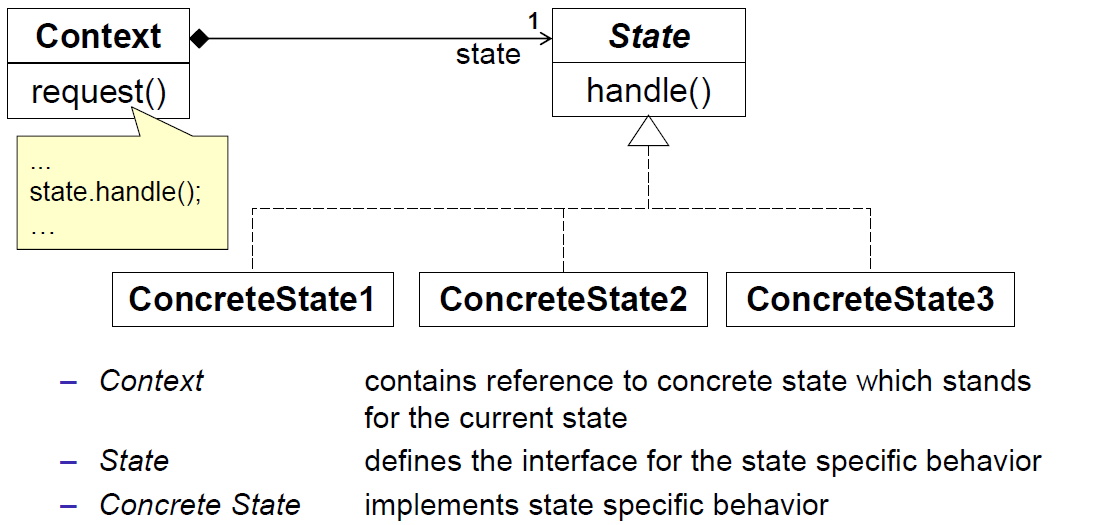
# Design Pattern: Observer Pattern

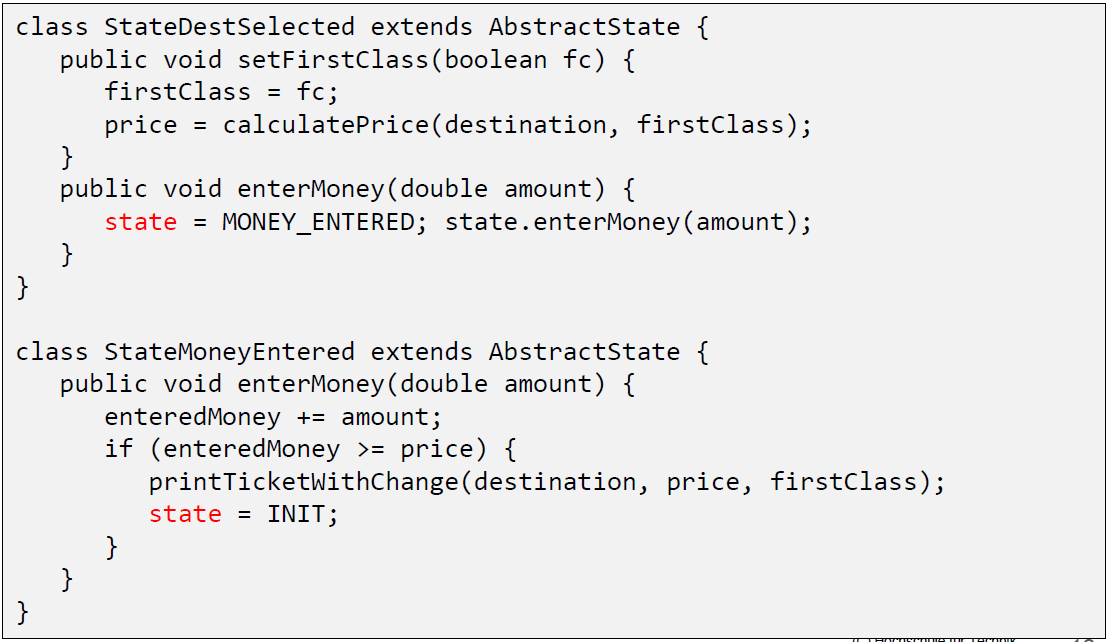
* Intent
  + One to many relation between objects which allows to inform the dependent objects about state changes
  + Consistency assurance between cooperating objects without connecting them too much
  + Notification of a dependent object without knowing it
* Consequences
  + Decoupling: Subject only knows observer interface, no concrete observes
  + Support for broadcast communication: Subject does not care about number of observers and observers can ignore the update
  + Unexpected updates: A simple operation on a subject may case a cascade of updates
* Rules
  + Use setChanged() to indicate a state change
  + In JavaFX, implement a getValue() method that returns a new instance. The framework will compare the old and new values and trigger the notify if a change occurred
  + Unsubscribe from the observer, or garbage collection might not happen (JavaFX uses weak references)
  + Don’t add or remove an observer during a notification or use a copy of the list (CopyOnWriteArrayList)
  + Only trigger a notification in case of a real change
  + Only trigger another notification after the first one was handled by all observers or delay the notification
  + Changes and the resulting notification should be atomic and grouped together

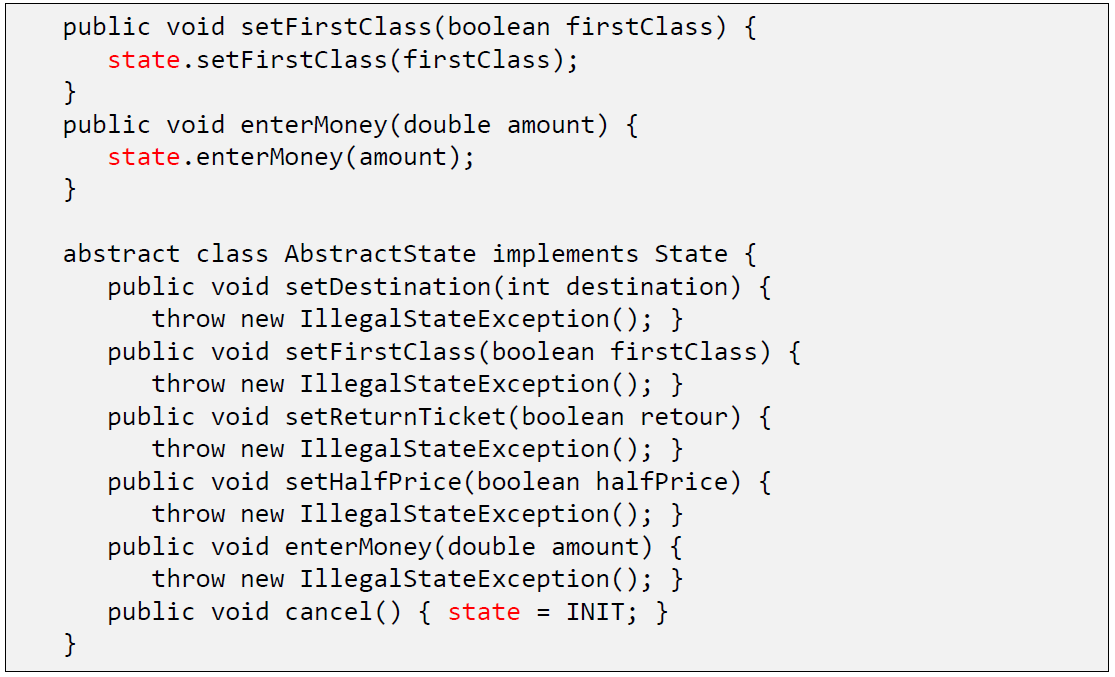
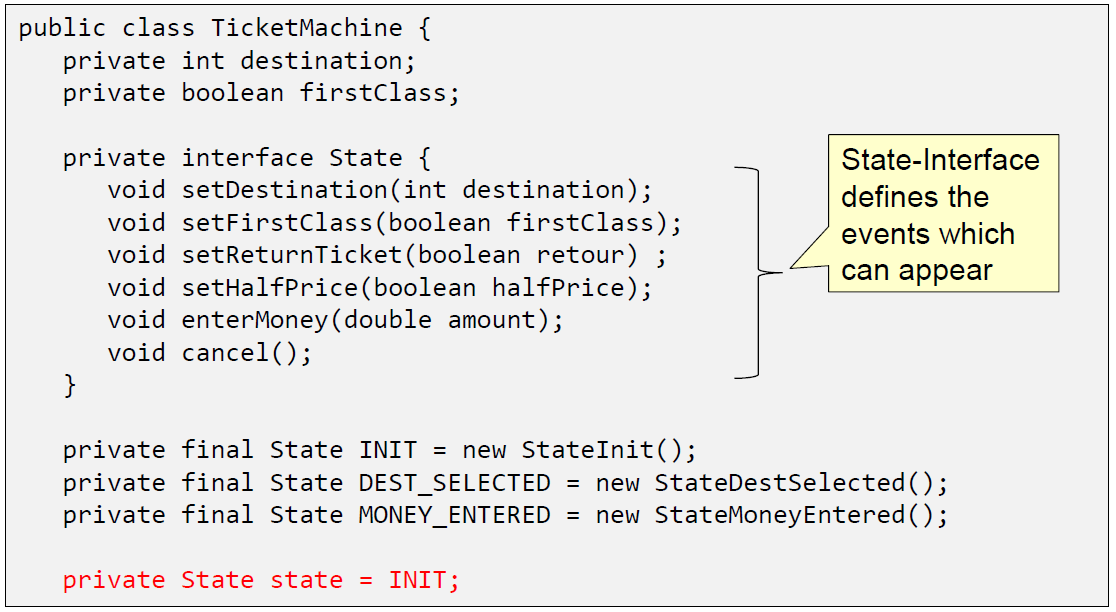


# Design Pattern: State Pattern

* Intent: Allow an object to alter its [internal] behaviour when its internal state changes 🡪 Outsourcing of state dependent behaviour
* Examples are the draw tool with handles (Ctrl/Shift pressed) and the point constrainer
* State Transition
  + Decentralized: transition initiated by state objects
    - States must know its successors
    - States need access to a state transition method in the context
      * context.setState(s); (or inner classes)
    - State methods may also return the new state (which is then set by the context)
  + Parameterized: transition signaled by state, executed by context
    - State returns a key (e.g. a String) which describes the new state
    - Association between keys and states is hold in the context
      * State transitions are configurable
      * Separation state-behavior vs transitions
      * State machine may be changed without changing state implementations
  + Centralized: transition initiated by the context
    - State should be informed that it is activated or passivated
* Creation of State Objects
  + Created when needed, so a state has to know the concrete state classes and depend on them. Useful if state changes happen infrequently: context.setState(new StateB());
  + Created ahead of time and stored in the context. Useful if state changes occur rapidly: context.setState(context.STATE\_B);
* Applicability
  + State dependency (Design / Architecture)
    - An object’s behavior depends on its state, and it must change its behavior at run-time depending on that state
  + Separation (Refactoring)
    - A class contains many behaviors which appear in multiple conditional statements (switch / if)
    - State is usually represented by one or more enumerated constants
    - Often several operations will contain this same conditional structure 🡪 move related conditional branches into their own strategy class

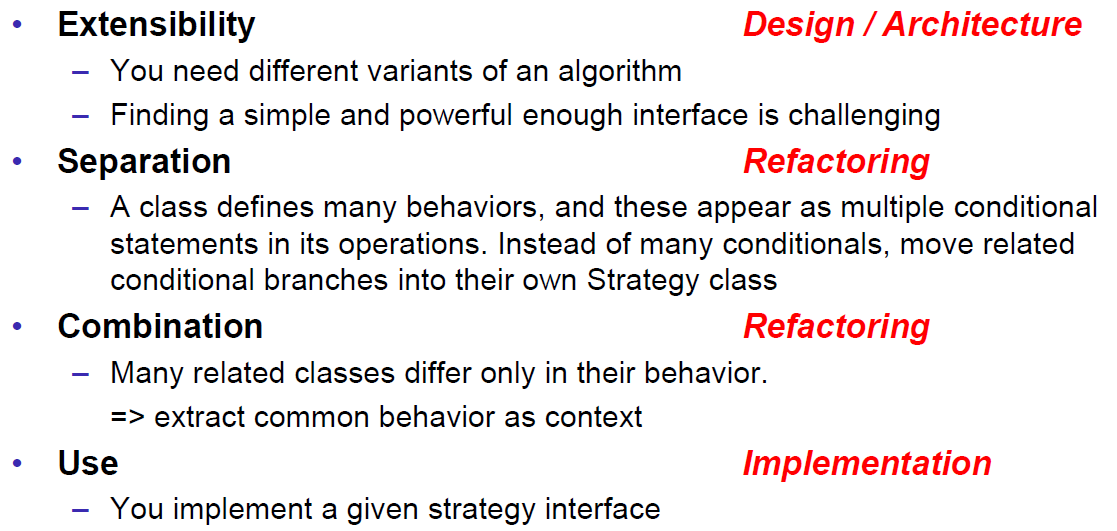


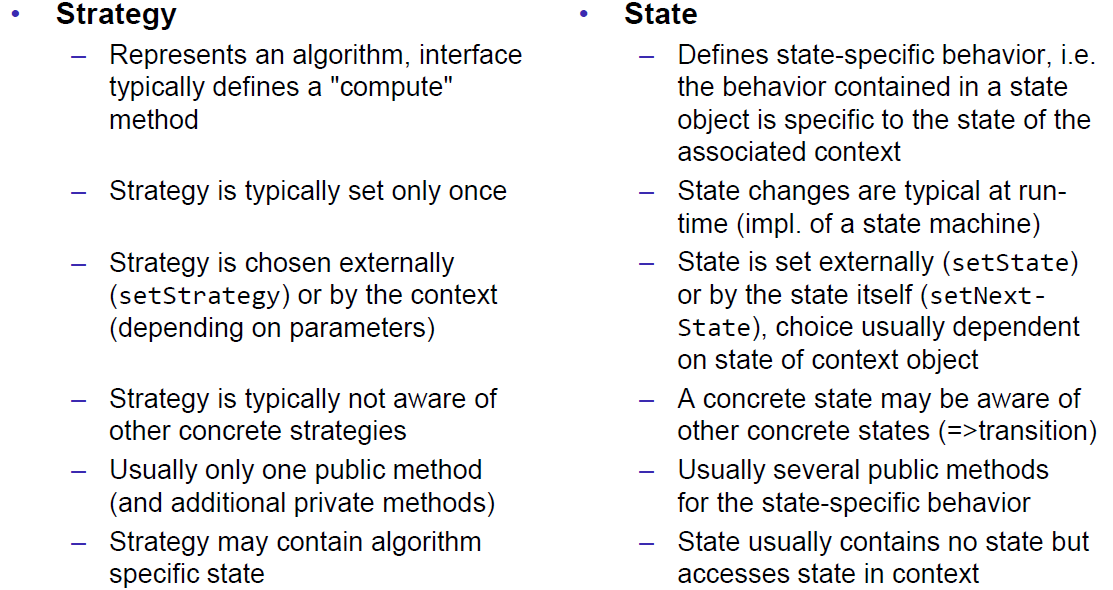




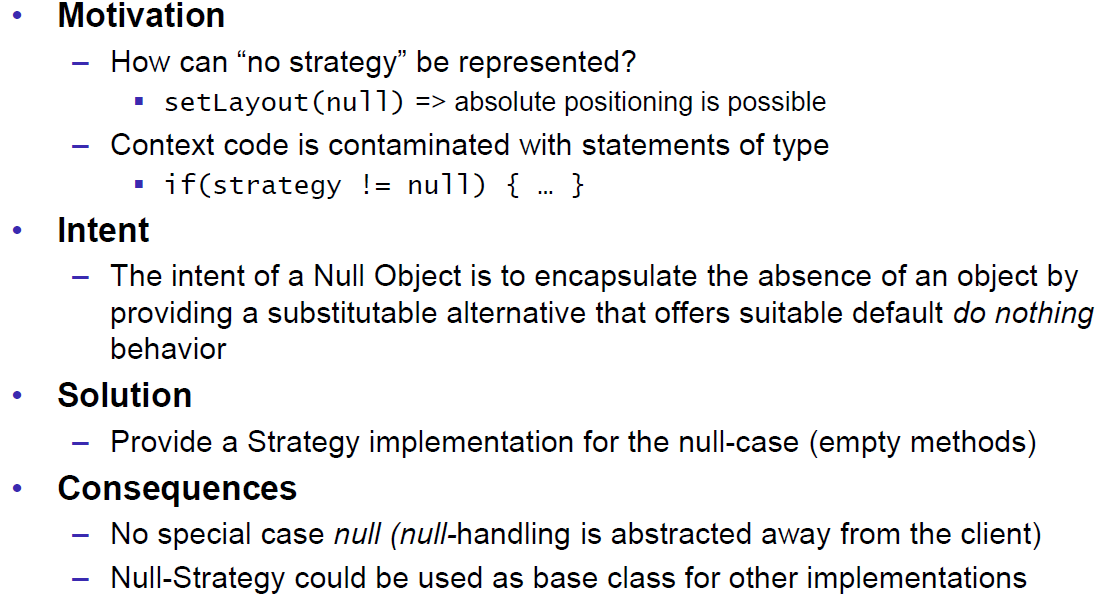
# Design Pattern: Strategy Pattern

* Intent: Define a family of algorithms where each one is encapsulated, interchangeable and where the system is extensible for new algorithms
* Example: Look & Feel in Swing, layout managers, point constrainer
* When can we speak of a Strategy Pattern?
  + A context class must want to use different variants of an algorithm
  + The context can also deal with new implementations of the algorithm
  + There must be an interface type that is an abstraction for the algorithm
  + Concrete strategy classes must implement the strategy interface type
  + The context class uses the strategy object to invoke the algorithm
  + A client supplies an object of a concrete strategy class to the context
* When do we not speak of a Strategy Pattern?
  + Only one algorithm which depends on parameters
  + E.g. filename for internationalization
  + Size of a Grid: 5/10/20 pixels



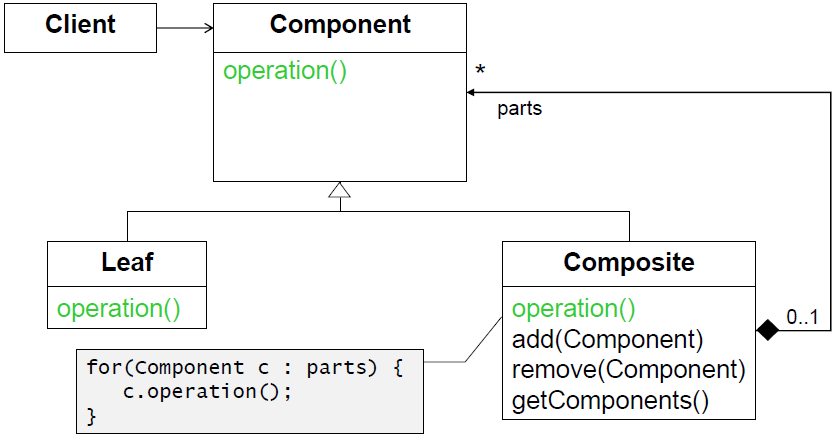
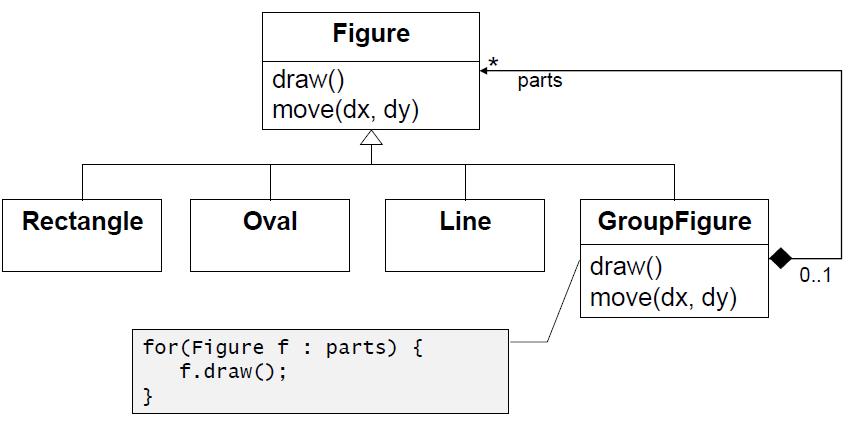


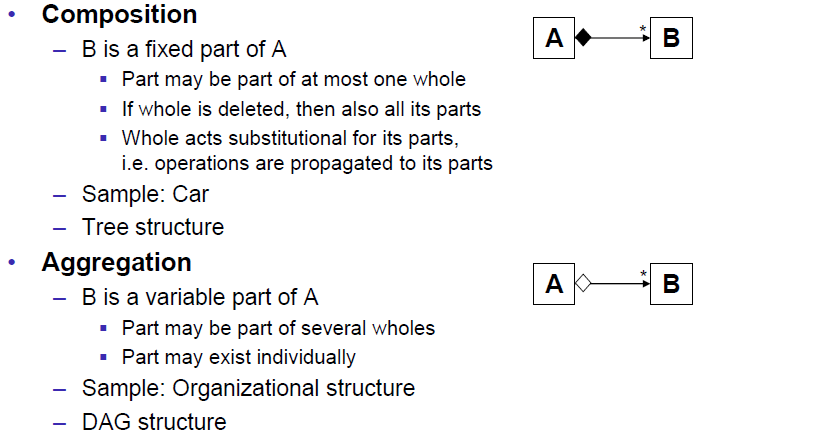
# Design Pattern: Null Object Pattern



# Design Pattern: Composite Pattern

* Intent: Representation of recursive part-whole hierarchies. Individual objects and compositions of objects should be treated uniformly
* Examples: File structure, menus or graphical user interface with common functionality like move, draw, resize or copy
* Participants
  + Component
    - Declares the interface for the objects in the composition
    - Implements the default behavior
    - Optionally: declares methods for accessing and managing child objects
  + Leaf
    - Defines behavior for primitive objects
  + Composite
    - Stores children references
    - Defines behavior for components having children
    - Implements methods for accessing and managing child objects
  + Client
    - Manipulates objects through the Component interface
* Problems
  + Don’t share children of a composition or ensure that the sharing doesn’t corrupt the children
  + Don’t create cyclic trees (Child is adding its parent as child)
* Where to define the composite specific methods
  + In component itself: Transparent (Swing): Swing JComponent extends AWT Container extends AWT Component 🡪 Not splitted up
  + In own composite: Safe (AWT/JavaFX): AWT Container extends AWT component (Analog to JavaFX Parent extends JavaFX Node)





# Design Pattern: Prototype Pattern

* Intent: Specify the kind of objects to create using a prototypical instance and create a new instance by copying this prototype
* Motivation: Copy & paste function in the editor or tool palette with prototype objects which can be copied
* Method Object.clone()
  + General intent of this method (not absolute requirements, SHOULD)
    - x.clone() != x 🡪 new instance
    - x.clone().getClass() == x.getClass() 🡪 same dynamic type
    - x.clone().equals(x) 🡪 equal
  + Visibility: protected
    - Can only be invoked if clone is overridden in a subclass with a method visible by the client
    - Cannot be invoked on objects of static type Object
* Implementation
  + Implementation checks whether the class implements the Cloneable interface
    - Cloneable is a marker interface
    - Cloneable is used to declare that a class supports cloning
    - If Object.clone is called on a class without Cloneable 🡪 CloneNotSupportedException
  + New instance is created via super.clone() (no constructor invocation!!!)
  + All attributes are copied (memory-copy)
* Problem: Shallow versus deep copy
  + Shallow copy: Copy all references 🡪 Both objects have the same references as attributes 🡪 Default behaviour
  + Deep copy: Do a normal shallow copy, but also clone all attributes 🡪 All attributes also must do a deep copy or strange errors might occur
* Remarks
  + References to immutable objects need not to be cloned as they cannot be changed (and typically cannot be cloned)
  + Final fields cannot be changed in the clone methods
  + Java allows covariant (House instead of Object), e.g. type of result may be strengthened
  + Alias references and cycles have to be handled manually when implementing a deep copy
* Clone with copy constructor
  + Every cloneable class contains a copy constructor (For a deep copy the attributes need to be copied in the constructor)
  + Method clone invokes this copy constructor and returns the result
  + Subclass invokes copy constructor of the base class via super(c) before doing their regular copy

# Information Hiding and Immutability

* Definition: The principle of information hiding is the hiding of design decisions (implementation details) in a computer program that are most likely to change, thus protecting other parts of the program from change if the design decision is changed. The protection involves providing a stable interface which shields the remainder of the program from the implementation (the details that are most likely to change).
* Mutable Parameters
  + Mutable parameters can be modified by the caller
  + Modifications can cause applications to behave incorrectly
  + Modifications to sensitive security state may result in elevated privileges for the attacker 🡪 E.g. altering the signers of a class can give the class access to unauthorized resources
* Robust programming
  + Make a copy of mutable non-primitive and non-immutable input/output parameters (Constructor, setter and getter) via clone() or the copy constructor (Perform deep cloning on arrays if necessary)
* Immutability
  + Definition: Immutable objects are objects whose state cannot be modified after creation (they can only be in one state)
  + Benefits
    - Consistency: Can freely be shared and cannot become inconsistent
    - Thread safety: Are inherently thread safe and require no synchronization
    - Safe in the presence of ill-behaved code: Methods that take objects as parameters should not change their state unless documented
    - Caching: Can be referenced in another class (Scheduler) without fearing that the cached object might change (Mutable Date)
    - Good Keys: Can be used as key of a hash map because their hashCode is not changing
    - Protection: Is going to protect the object from state changes with itself (Fraction example)

