

Object Oriented Programming in Java

2023 H2 semester

Summary

- These slides contain material about objects, classes, and object-oriented programming (OOP) in Java.
- Contains:
 - Key syntax of Java
 - Key principles of OOP in Java like interfaces, their implementation and inheritance
 - OOP patterns with examples in Java
 - Part of the lecture are demonstrated source codes

Why Java?

- TIOBE index
<https://www.tiobe.com/tiobe-index/>

Apr 2023	Apr 2022	Programming Language	Ratings	Change
1	1	Python	14.51%	+0.59%
2	2	C	14.41%	+1.71%
3	3	Java	13.23%	+2.41%
4	4	C++	12.96%	+4.68%
5	5	C#	8.21%	+1.39%
6	6	Visual Basic	4.40%	-1.00%
7	7	JavaScript	2.10%	-0.31%
8	9	SQL	1.68%	-0.61%
9	10	PHP	1.36%	-0.28%
10	13	Go	1.28%	+0.20%
11	12	Delphi/Object Pascal	1.23%	+0.05%
12	8	Assembly language	1.03%	-1.31%
13	16	Classic Visual Basic	0.92%	+0.09%
14	20	MATLAB	0.86%	+0.12%
15	24	Scratch	0.79%	+0.13%
16	11	R	0.76%	-0.79%
17	14	Swift	0.72%	-0.28%
18	15	Ruby	0.66%	-0.22%
19	28	Rust	0.63%	+0.18%
20	31	Fortran	0.59%	+0.24%

Intro

- **Jeff Goodell:** Would you explain, in simple terms, exactly what object-oriented software is?
- **Steve Jobs:** Objects are like people. They're living, breathing things that have knowledge inside them about how to do things and have memory inside them so they can remember things. And rather than interacting with them at a very low level, you interact with them at a very high level of abstraction, like we're doing right here.
- Here's an example: If I'm your laundry object, you can give me your dirty clothes and send me a message that says, "Can you get my clothes laundered, please." I happen to know where the best laundry place in San Francisco is. And I speak English, and I have dollars in my pockets. So I go out and hail a taxicab and tell the driver to take me to this place in San Francisco. I go get your clothes laundered, I jump back in the cab, I get back here. I give you your clean clothes and say, "Here are your clean clothes."
- You have no idea how I did that. You have no knowledge of the laundry place. Maybe you speak French, and you can't even hail a taxi. You can't pay for one, you don't have dollars in your pocket. Yet I knew how to do all of that. And you didn't have to know any of it. All that complexity was hidden inside of me, and we were able to interact at a very high level of abstraction. That's what objects are. They encapsulate complexity, and the interfaces to that complexity are high level.

Primitives vs. objects; value and reference semantics

A swap method?

- Does the following swap method work? Why or why not?

```
public static void main(String[] args) {  
    int a = 7;  
    int b = 35;  
  
    // swap a with b?  
    swap(a, b);  
  
    System.out.println(a + " " + b);  
}  
  
public static void swap(int a, int b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

Value semantics

- **value semantics:** Behavior where values are copied when assigned, passed as parameters, or returned.
- All primitive types in Java use value semantics.
- When one variable is assigned to another, its value is copied.
- Modifying the value of one variable does not affect others.

```
int x = 5;  
int y = x;      // x = 5, y = 5  
y = 17;          // x = 5, y = 17  
x = 8;           // x = 8, y = 17
```

Reference semantics (objects)

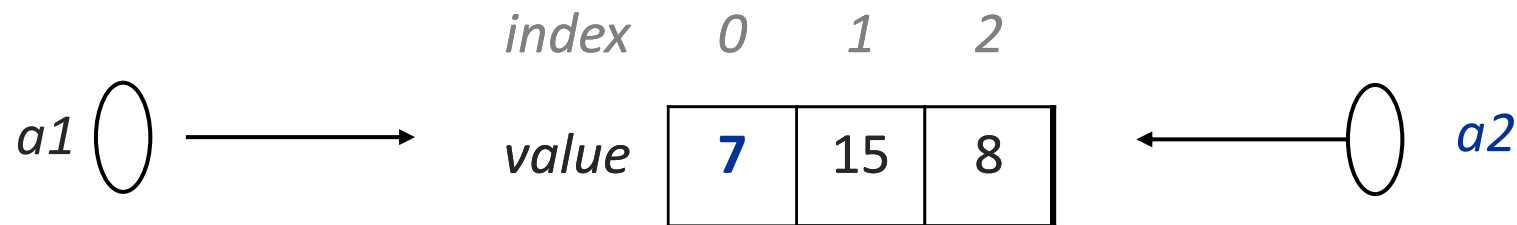
- **reference semantics:** Behavior where variables actually store the address of an object in memory.
- When one variable is assigned to another, the object is *not* copied; both variables refer to the *same object*.
- Modifying the value of one variable *will* affect others.

```
int[] a1 = {4, 15, 8};
```

```
int[] a2 = a1;           // refer to same array as a1
```

```
a2[0] = 7;
```

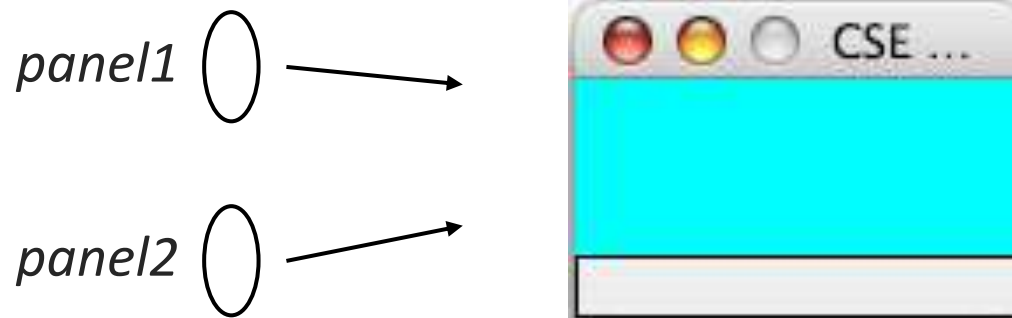
```
System.out.println(Arrays.toString(a1)); // [7, 15, 8]
```



References and objects

- Arrays and objects use reference semantics. Why?
 - *efficiency*. Copying large objects slows down a program.
 - *sharing*. It's useful to share an object's data among methods.

```
DrawingPanel panel1 = new DrawingPanel(80, 50);  
DrawingPanel panel2 = panel1;    // same window  
panel2.setBackground(Color.CYAN);
```




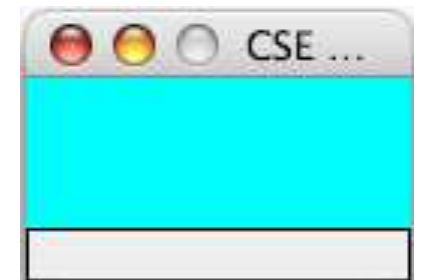
Objects as parameters

- When an object is passed as a parameter, the object is *not* copied. The parameter refers to the same object.
- If the parameter is modified, it *will* affect the original object.

```
public static void main(String[] args) {  
    DrawingPanel window = new DrawingPanel(80, 50);  
    window.setBackground(Color.YELLOW);  
    example(window);  
}
```

```
public static void example(DrawingPanel panel) window   
    panel.setBackground(Color.CYAN);  
    ...  
}
```

panel  →



Arrays as parameters

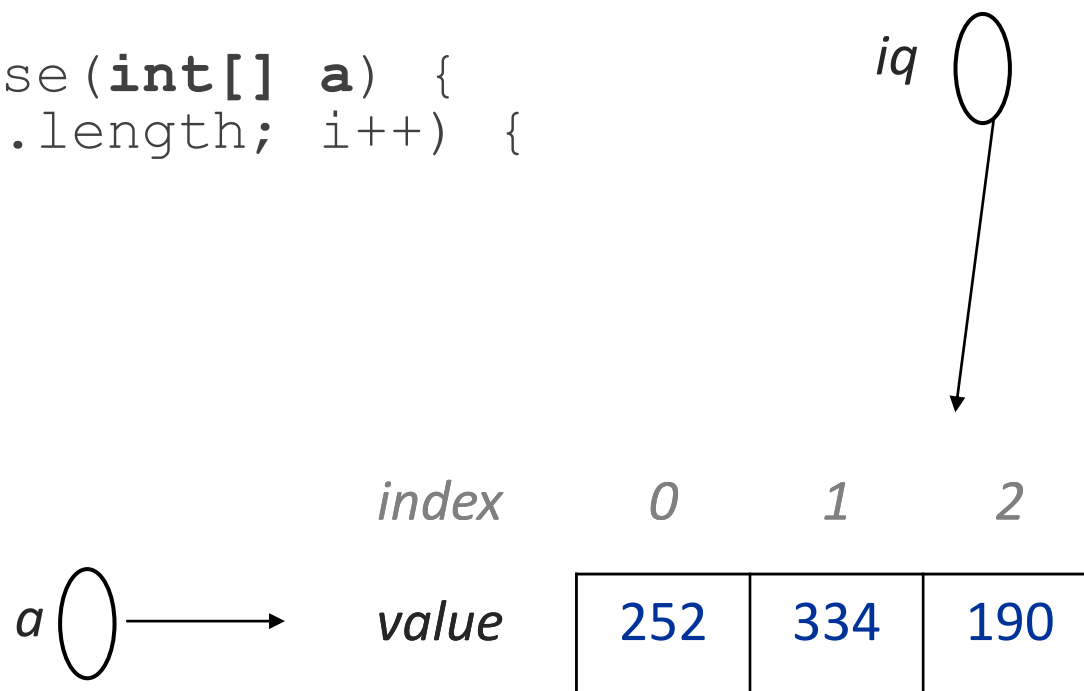
- Arrays are also passed as parameters by reference.
- Changes made in the method are also seen by the caller.

```
public static void main(String[] args) {  
    int[] iq = {126, 167, 95};  
    increase(iq);  
    System.out.println(Arrays.toString(iq));  
}
```

```
public static void increase(int[] a) {  
    for (int i = 0; i < a.length; i++) {  
        a[i] = a[i] * 2;  
    }  
}
```

- Output:

[252, 334, 190]



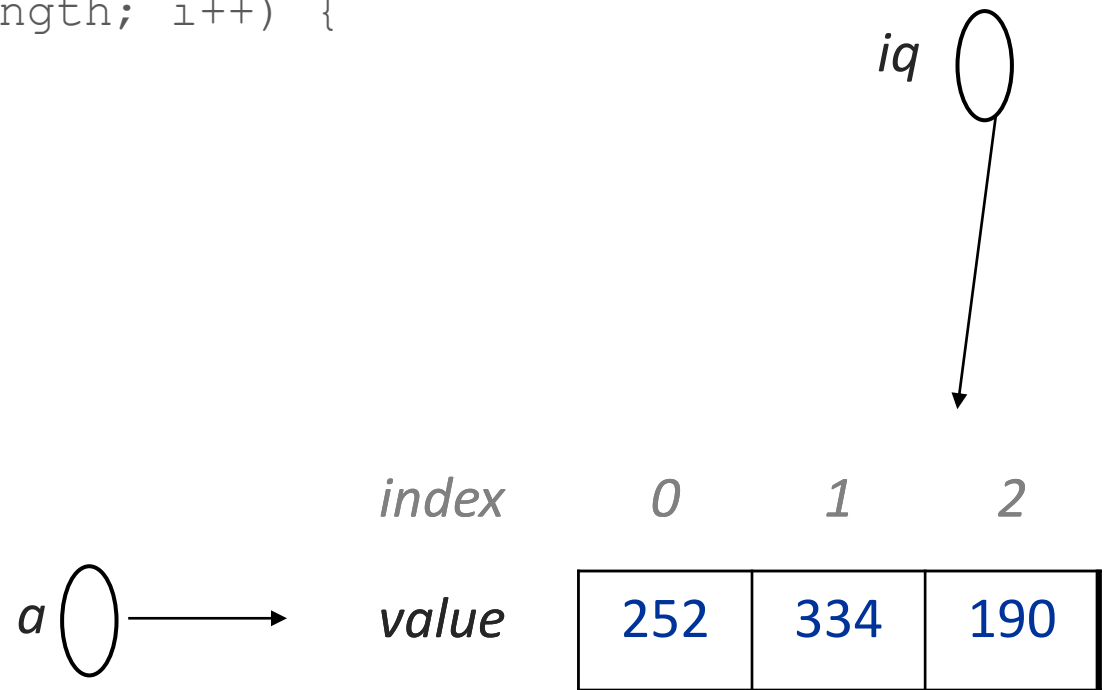
Arrays pass by reference

- Arrays are also passed as parameters by reference.
 - Changes made in the method are also seen by the caller.

```
public static void main(String[] args) {  
    int[] iq = {126, 167, 95};  
    increase(iq);  
    System.out.println(Arrays.toString(iq));  
}  
  
public static void increase(int[] a) {  
    for (int i = 0; i < a.length; i++) {  
        a[i] = a[i] * 2;  
    }  
}
```

- Output:

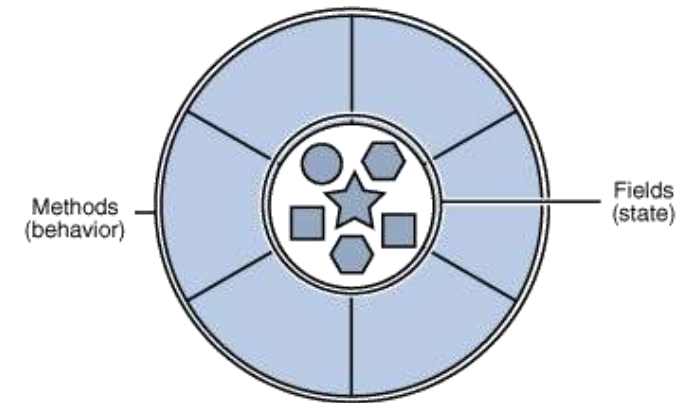
[252, 334, 190]



Classes and Objects

Objects

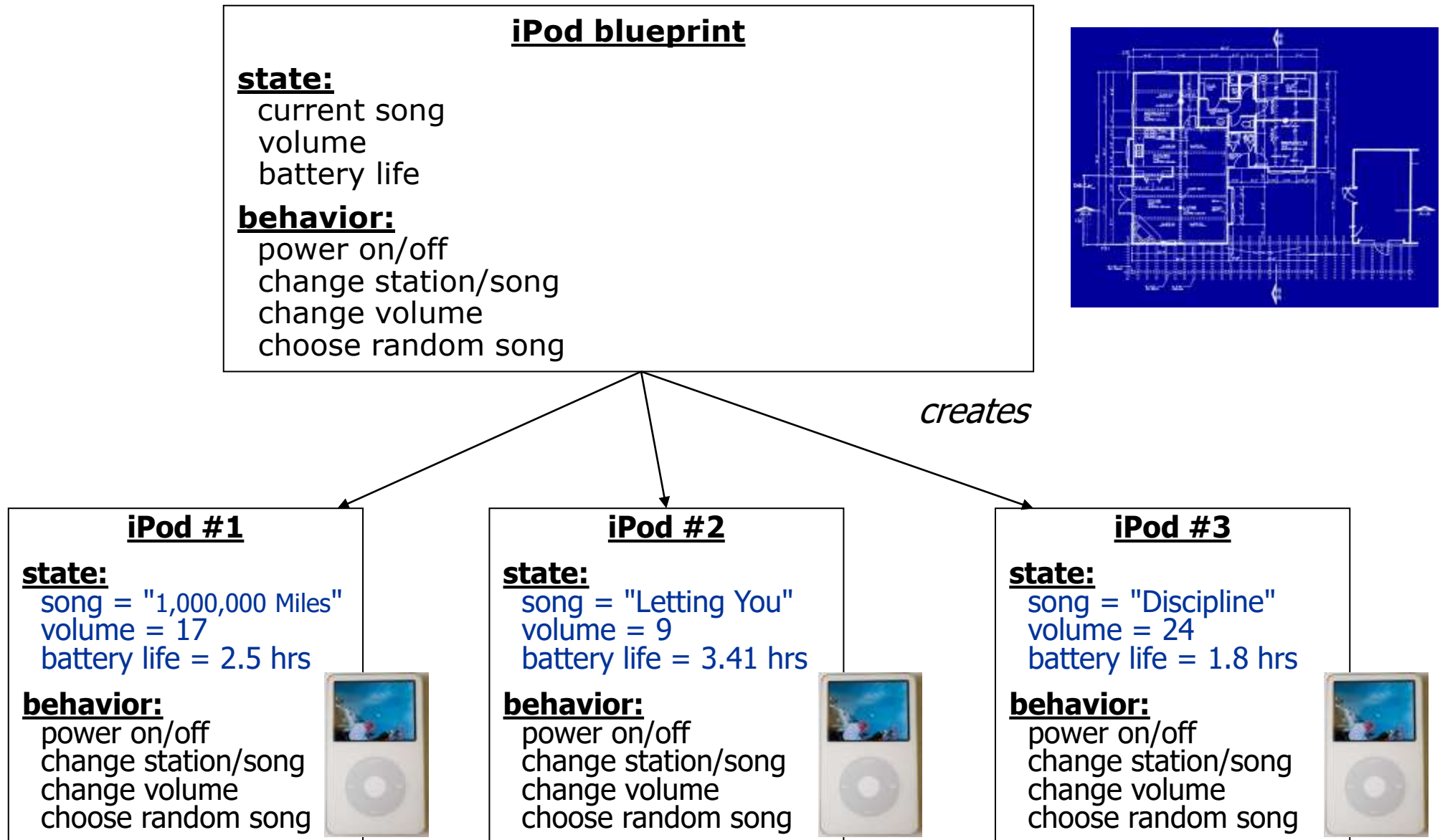
- **object:** An entity that encapsulates data and behavior.
 - *data:* variables inside the object
 - *behavior:* methods inside the object
 - You interact with the methods; the data is hidden in the object.
- Constructing (creating) an object:
Type **objectName** = new **Type** (**parameters**) ;
- Calling an object's method:
objectName . **methodName** (**parameters**) ;



Classes

- **class**: A program entity that represents either:
 1. A program / module, or
 2. A template for a new type of objects.
- **object-oriented programming (OOP)**: Programs that perform their behavior as interactions between objects.
- **abstraction**: Separation between concepts and details.
Objects and classes provide abstraction in programming.

Blueprint analogy



Point objects

```
import java.awt.*;  
...  
Point p1 = new Point(5, -2);  
Point p2 = new Point();           // origin (0, 0)
```

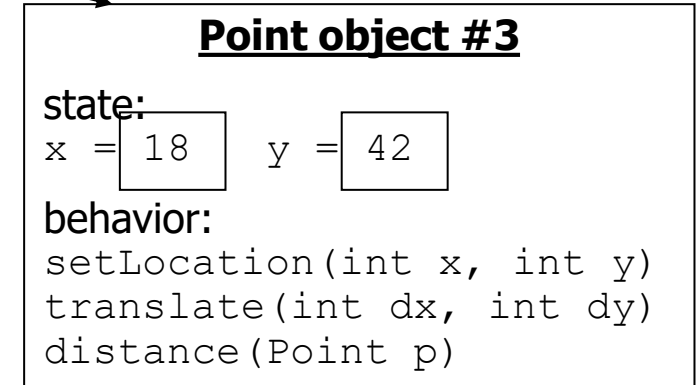
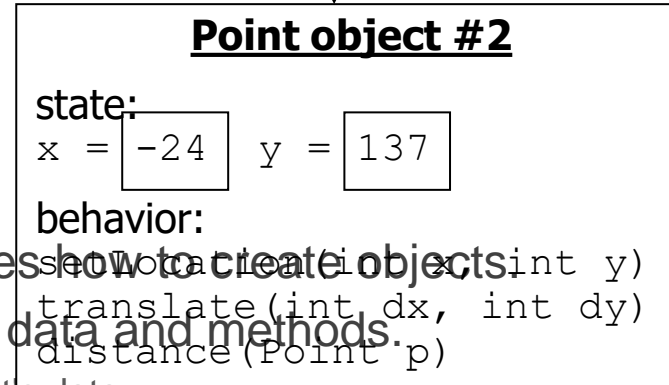
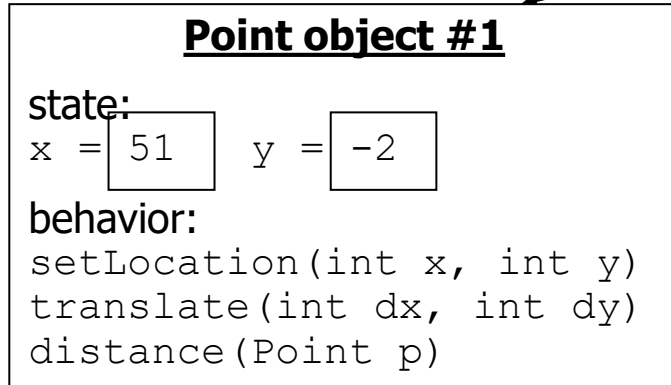
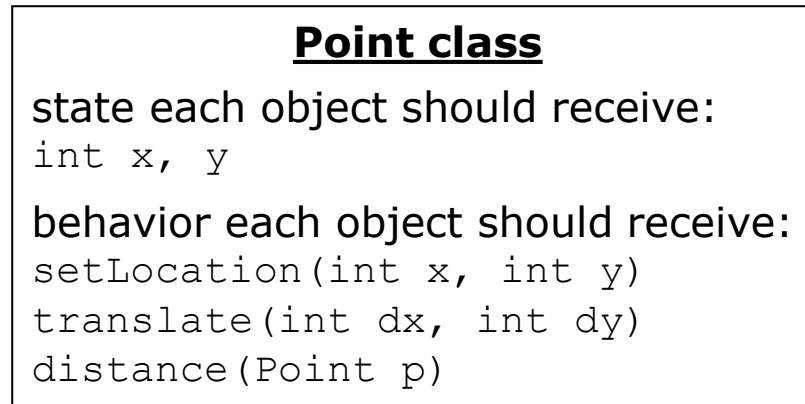
- Data:

Name	Description
x	the point's x-coordinate
y	the point's y-coordinate

- Methods:

Name	Description
setLocation(x , y)	sets the point's x and y to the given values
translate(dx , dy)	adjusts the point's x and y by the given amounts
distance(p)	how far away the point is from point <i>p</i>

Point class as blueprint

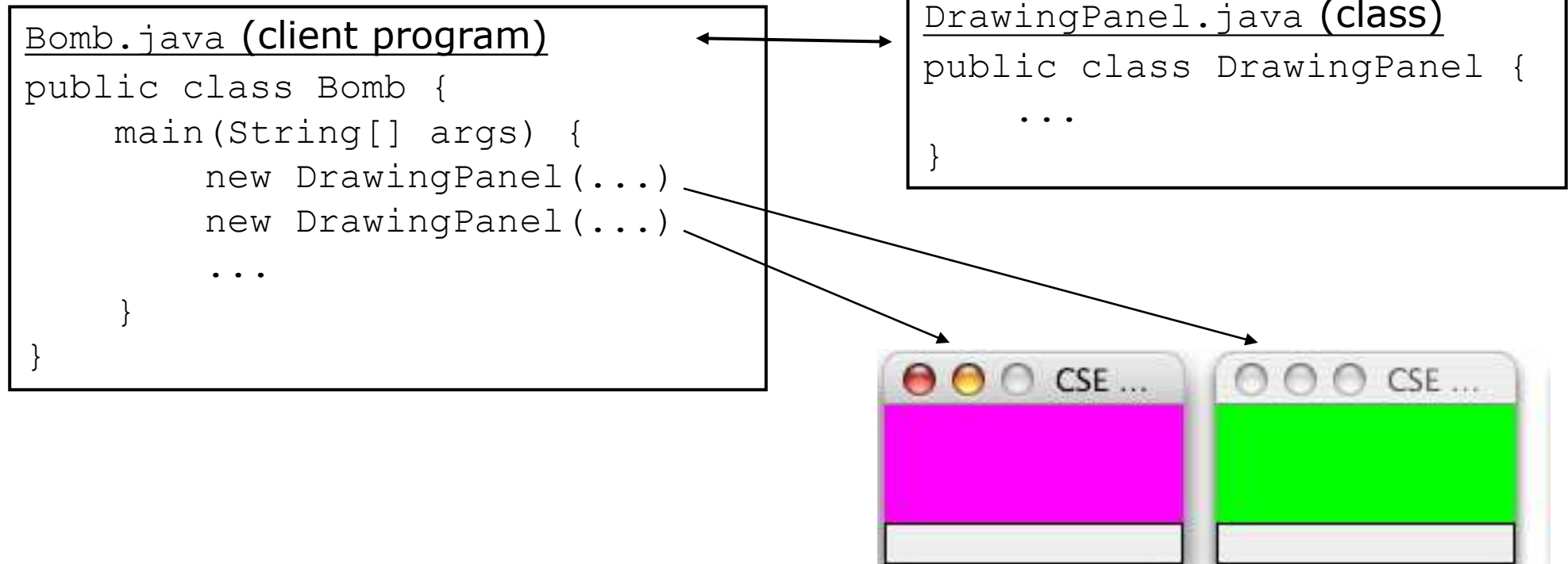


as how to create objects
data and methods.

- The methods operate on that object's data.

Clients of objects

- **client program:** A program that uses objects.
 - Example: Bomb is a client of `DrawingPanel` and `Graphics`.



Fields

- **field**: A variable inside an object that is part of its state.
 - Each object has *its own copy* of each field.
- Declaration syntax:

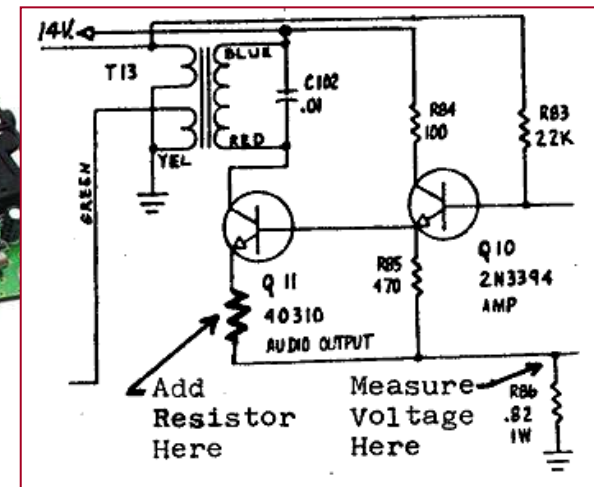
```
private type name;
```

- Example:

```
public class Point {  
    private int x;  
    private int y;  
  
    . . .  
}
```

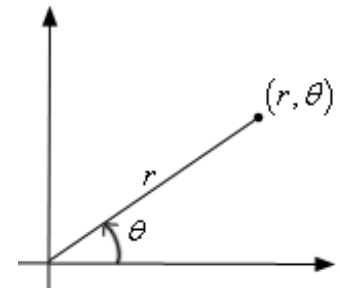
Encapsulation

- **encapsulation:** Hiding implementation details from clients.
- Encapsulation enforces *abstraction*.
 - separates external view (behavior) from internal view (state)
 - protects the integrity of an object's data



Benefits of encapsulation

- Abstraction between object and clients
- Protects object from unwanted access
 - Example: Can't fraudulently increase an `Account`'s balance.
- Can change the class implementation later
 - Example: `Point` could be rewritten in polar coordinates (r, θ) with the same methods.
- Can constrain objects' state (**invariants**)
 - Example: Only allow `Accounts` with non-negative balance.
 - Example: Only allow `Dates` with a month from 1-12.



Instance methods

- **instance method** (or **object method**): Exists inside each object of a class and gives behavior to each object.

```
public type name (parameters) {  
    statements;  
}
```

- same syntax as static methods, but without `static` keyword

Example:

```
public void tranlate(int dx, int dy) {  
    x += dx;  
    y += dy;  
}
```

The implicit parameter

- **implicit parameter:**

The object on which an instance method is being called.

- If we have a `Point` object `p1` and call `p1.translate(5, 3)`; the object referred to by `p1` is the implicit parameter.
- If we have a `Point` object `p2` and call `p2.translate(4, 1)`; the object referred to by `p2` is the implicit parameter.
- The instance method can refer to that object's fields.
 - We say that it executes in the *context* of a particular object.
 - `translate` can refer to the `x` and `y` of the object it was called on.

Categories of methods

- **accessor:** A method that lets clients examine object state.
 - Examples: `distance`, `distanceFromOrigin`
 - often has a non-`void` return type
- **mutator:** A method that modifies an object's state.
 - Examples: `setLocation`, `translate`
- **helper:** Assists some other method in performing its task.
 - often declared as `private` so outside clients cannot call it

The toString method

tells Java how to convert an object into a String for printing

```
public String toString() {  
    code that returns a String representing this object;  
}
```

- Method name, return, and parameters must match *exactly*.
- Example:

```
// Returns a String representing this Point.  
public String toString() {  
    return "(" + x + ", " + y + ")";  
}
```

Constructors

- **constructor**: Initializes the state of new objects.

```
public type (parameters) {  
    statements;  
}
```

- runs when the client uses the `new` keyword
- no return type is specified; implicitly "returns" the new object

```
public class Point {  
    private int x;  
    private int y;  
  
    public Point(int initialX, int initialY) {  
        x = initialX;  
        y = initialY;  
    }  
}
```

Multiple constructors

- A class can have multiple constructors.
 - Each one must accept a unique set of parameters.
- *Example:* A `Point` constructor with no parameters that initializes the point to (0, 0).

// Constructs a new point at (0, 0) .

```
public Point() {  
    x = 0;  
    y = 0;  
}
```

The keyword `this`

- `this` : Refers to the implicit parameter inside your class.
(a variable that stores the object on which a method is called)
- Refer to a field: `this.field`
- Call a method: `this.method(parameters) ;`
- One constructor `this(parameters) ;`
can call another:

Calling another constructor

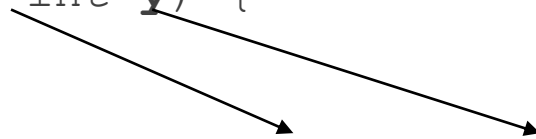
```
public class Point {  
    private int x;  
    private int y;
```

```
    public Point() {  
        this(0, 0);  
    }
```

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

```
    ...
```

```
}
```



- Avoids redundancy between constructors
- Only a constructor (not a method) can call another constructor

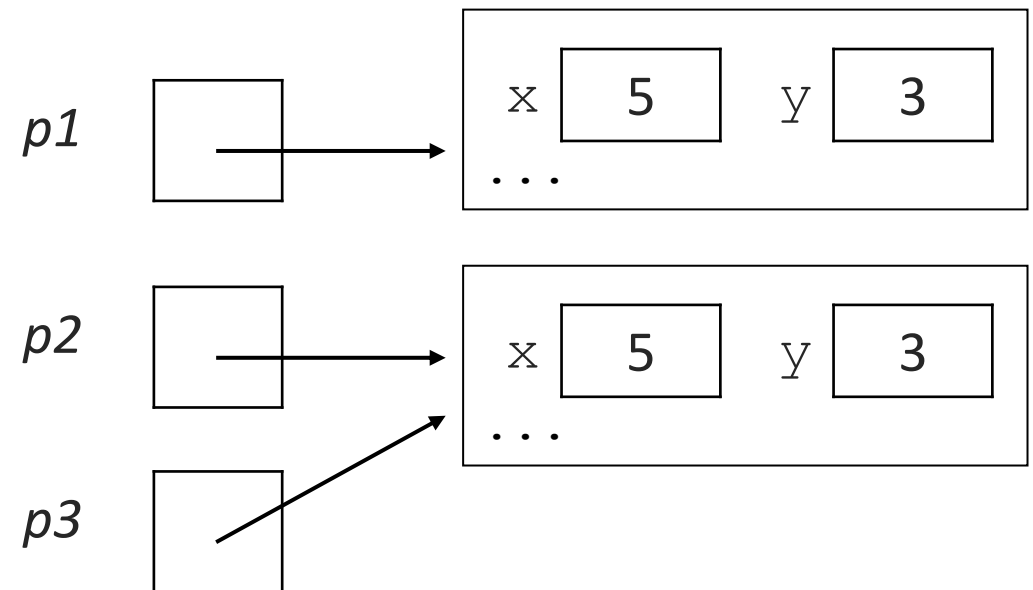
Comparing objects for equality and ordering

Comparing objects

- The `==` operator does not work well with objects.
 - `==` compares references to objects, not their state.
 - It only produces `true` when you compare an object to itself.

```
Point p1 = new Point(5, 3);  
Point p2 = new Point(5, 3);  
Point p3 = p2;
```

```
// p1 == p2 is false;  
// p1 == p3 is false;  
// p2 == p3 is true
```



The equals method

- The `equals` method compares the state of objects.

```
if (str1.equals(str2)) {  
    System.out.println("the strings are equal");  
}
```

- But if you write a class, its `equals` method behaves like `==`

```
if (p1.equals(p2)) {    // false :- (  
    System.out.println("equal");  
}
```

- This is the default behavior we receive from class `Object`.
- Java doesn't understand how to compare new classes by default.

The `compareTo` method (10.2)

- The standard way for a Java class to define a comparison function for its objects is to define a `compareTo` method.
 - Example: in the `String` class, there is a method:

```
public int compareTo(String other)
```
- A call of `A.compareTo(B)` will return:
 - a value < 0 if **A** comes "before" **B** in the ordering,
 - a value > 0 if **A** comes "after" **B** in the ordering,
 - or 0 if **A** and **B** are considered "equal" in the ordering.

Using compareTo

- `compareTo` can be used as a test in an `if` statement.

```
String a = "alice";  
String b = "bob";  
if (a.compareTo(b) < 0) { // true  
    ...  
}
```

Primitives	Objects
<code>if (a < b) { ...</code>	<code>if (a.compareTo(b) < 0) { ...</code>
<code>if (a <= b) { ...</code>	<code>if (a.compareTo(b) <= 0) { ...</code>
<code>if (a == b) { ...</code>	<code>if (a.compareTo(b) == 0) { ...</code>
<code>if (a != b) { ...</code>	<code>if (a.compareTo(b) != 0) { ...</code>
<code>if (a >= b) { ...</code>	<code>if (a.compareTo(b) >= 0) { ...</code>
<code>if (a > b) { ...</code>	<code>if (a.compareTo(b) > 0) { ...</code>

compareTo and collections

- You can use an array or list of strings with Java's included binary search method because it calls `compareTo` internally.

```
String[] a = {"al", "bob", "cari", "dan", "mike"};
int index = Arrays.binarySearch(a, "dan"); // 3
```

- Java's `TreeSet/Map` use `compareTo` internally for ordering.

```
Set<String> set = new TreeSet<String>();
for (String s : a) {
    set.add(s);
}
System.out.println(s);
// [al, bob, cari, dan, mike]
```

Comparable (10.2)

```
public interface Comparable<E> {  
    public int compareTo(E other);  
}
```

- A class can implement the `Comparable` interface to define a natural ordering function for its objects.
- A call to your `compareTo` method should return:
 - a value < 0 if this object comes "before" the other object,
 - a value > 0 if this object comes "after" the other object,
 - or 0 if this object is considered "equal" to the other.
- If you want multiple orderings, use a `Comparator` instead (see Ch. 13.1)

Comparable template

```
public class name implements Comparable<name> {  
  
    . . .  
  
    public int compareTo(name other) {  
        . . .  
    }  
}
```

Comparable example

```
public class Point implements Comparable<Point> {  
    private int x;  
    private int y;  
    ...  
  
    // sort by x and break ties by y  
    public int compareTo(Point other) {  
        if (x < other.x) {  
            return -1;  
        } else if (x > other.x) {  
            return 1;  
        } else if (y < other.y) {  
            return -1;    // same x, smaller y  
        } else if (y > other.y) {  
            return 1;    // same x, larger y  
        } else {  
            return 0;    // same x and same y  
        }  
    }  
}
```

compareTo tricks

- *subtraction trick* - Subtracting related numeric values produces the right result for what you want compareTo to return:

```
// sort by x and break ties by y
public int compareTo(Point other) {
    if (x != other.x) {
        return x - other.x;    // different x
    } else {
        return y - other.y;    // same x; compare y
    }
}
```

- The idea:

- | | |
|------------------------------|--------------------------------|
| • if $x > \text{other.x}$, | then $x - \text{other.x} > 0$ |
| • if $x < \text{other.x}$, | then $x - \text{other.x} < 0$ |
| • if $x == \text{other.x}$, | then $x - \text{other.x} == 0$ |

- NOTE: This trick doesn't work for doubles (but see `Math.signum`)

compareTo tricks 2

- *delegation trick* - If your object's fields are comparable (such as strings), use their `compareTo` results to help you:

```
// sort by employee name, e.g. "Jim" < "Susan"
public int compareTo(Employee other) {
    return name.compareTo(other.getName());
}
```

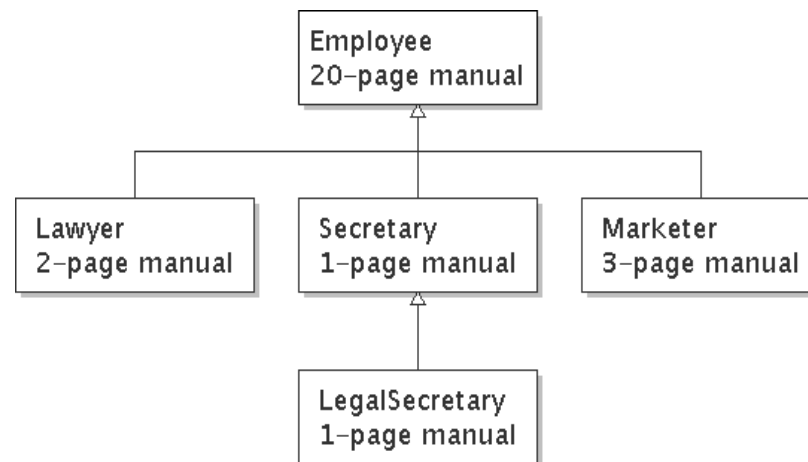
- *toString trick* - If your object's `toString` representation is related to the ordering, use that to help you:

```
// sort by date, e.g. "09/19" > "04/01"
public int compareTo(Date other) {
    return toString().compareTo(other.toString());
}
```

Inheritance

Inheritance

- **inheritance**: Forming new classes based on existing ones.
 - a way to share/**reuse code** between two or more classes
- **superclass**: Parent class being extended.
- **subclass**: Child class that inherits behavior from superclass.
 - gets a copy of every field and method from superclass
- **is-a relationship**: Each object of the subclass also "is a(n)" object of the superclass and can be treated as one.



Inheritance syntax

```
public class name extends superclass {
```

- Example:

```
public class Lawyer extends Employee {  
    ...  
}
```

- By extending `Employee`, each `Lawyer` object now:
 - receives a copy of each method from `Employee` automatically
 - can be treated as an `Employee` by client code
- Lawyer can also replace ("override") behavior from `Employee`.

Overriding Methods

- **override:** To write a new version of a method in a subclass that replaces the superclass's version.
- No special syntax required to override a superclass method. Just write a new version of it in the subclass.

```
public class Lawyer extends Employee {  
    // overrides getVacationForm in Employee class  
    public String getVacationForm() {  
        return "pink";  
    }  
    . . .  
}
```

The super keyword

- A subclass can call its parent's method/constructor:

```
super.method (parameters)    // method  
super (parameters) ;       // constructor
```

```
public class Lawyer extends Employee {  
    public Lawyer(String name) {  
        super (name) ;  
    }  
  
    // give Lawyers a $5K raise (better)  
    public double getSalary() {  
        double baseSalary = super.getSalary() ;  
        return baseSalary + 5000.00 ;  
    }  
}
```

Subclasses and fields

```
public class Employee {  
    private double salary;  
    ...  
}
```

```
public class Lawyer extends Employee {  
    ...  
    public void giveRaise(double amount) {  
        salary += amount;    // error; salary is private  
    }  
}
```

- Inherited private fields/methods cannot be directly accessed by subclasses. (*The subclass has the field, but it can't touch it.*)
 - How can we allow a subclass to access/modify these fields?

Protected fields/methods

```
protected type name;    // field
```

```
protected type name (type name, ..., type name) {  
    statement(s);        // method  
}
```

- a **protected field** or **method** can be seen/called only by:
 - the class itself, and its subclasses
 - also by other classes in the same "package" (discussed later)
 - useful for allowing selective access to inner class implementation

```
public class Employee {  
    protected double salary;  
    ...  
}
```


Inheritance and constructors

- If we add a constructor to the `Employee` class, our subclasses do not compile. The error:

```
Lawyer.java:2: cannot find symbol
symbol   : constructor Employee()
location: class Employee
public class Lawyer extends Employee {
      ^
```

- The short explanation: Once we write a constructor (that requires parameters) in the superclass, we must now write constructors for our employee subclasses as well.

Inheritance and constructors

- Constructors are not inherited.
 - Subclasses don't inherit the `Employee(int)` constructor.
- Subclasses receive a default constructor that contains:

```
public Lawyer() {  
    super();          // calls Employee() constructor  
}
```

- But our `Employee(int)` replaces the default `Employee()`.
 - The subclasses' default constructors are now trying to call a non-existent default `Employee` constructor.

Calling superclass constructor

`super (parameters) ;`

- Example:

```
public class Lawyer extends Employee {  
    public Lawyer(int years) {  
        super(years); // calls Employee c'tor  
    }  
    ...  
}
```

- The `super` call must be the first statement in the constructor.

Polymorphism

Polymorphism

- **polymorphism:** Ability for the same code to be used with different types of objects and behave differently with each.
- `System.out.println` can print any type of object.
 - Each one displays in its own way on the console.
- `CritterMain` can interact with any type of critter.
 - Each one moves, fights, etc. in its own way.

Coding with polymorphism

- A variable of type T can hold an object of any subclass of T .

```
Employee ed = new Lawyer();
```

- You can call any methods from the `Employee` class on `ed`.
- When a method is called on `ed`, it behaves as a `Lawyer`.

```
System.out.println(ed.getSalary()); // 50000.0  
System.out.println(ed.getVacationForm()); // pink
```

Polymorphic parameters

- You can pass any subtype of a parameter's type.

```
public static void main(String[] args) {
    Lawyer lisa = new Lawyer();
    Secretary steve = new Secretary();
    printInfo(lisa);
    printInfo(steve);
}

public static void printInfo(Employee e) {
    System.out.println("pay   : " + e.getSalary());
    System.out.println("vdays: " + e.getVacationDays());
    System.out.println("vform: " + e.getVacationForm());
    System.out.println();
}
```

OUTPUT:

pay : 50000.0	pay : 50000.0
vdays: 15	vdays: 10
vform: pink	vform: yellow

Polymorphism and arrays

- Arrays of superclass types can store any subtype as elements.

```
public static void main(String[] args) {  
    Employee[] e = {new Lawyer(),    new Secretary(),  
                    new Marketer(), new LegalSecretary()};  
  
    for (int i = 0; i < e.length; i++) {  
        System.out.println("pay   : " + e[i].getSalary(););  
        System.out.println("vdays: " + i].getVacationDays(););  
        System.out.println();  
    }  
}
```

Output:

pay : 50000.0	pay : 60000.0
vdays: 15	vdays: 10
pay : 50000.0	pay : 55000.0
vdays: 10	vdays: 10

Casting references

- A variable can only call that type's methods, not a subtype's.

```
Employee ed = new Lawyer();  
int hours = ed.getHours();    // ok; in Employee  
ed.sue();                     // compiler error
```

- The compiler's reasoning is, variable `ed` could store any kind of employee, and not all kinds know how to `sue`.
- To use `Lawyer` methods on `ed`, we can type-cast it.

```
Lawyer theRealEd = (Lawyer) ed;  
theRealEd.sue();           // ok  
  
( (Lawyer) ed ).sue();     // shorter version
```

More about casting

- The code crashes if you cast an object too far down the tree.

```
Employee eric = new Secretary();  
(Secretary) eric).takeDictation("hi");           // ok  
((LegalSecretary) eric).fileLegalBriefs(); // error  
// (Secretary doesn't know how to file briefs)
```

- You can cast only up and down the tree, not sideways.

```
Lawyer linda = new Lawyer();  
((Secretary) linda).takeDictation("hi");           // error
```

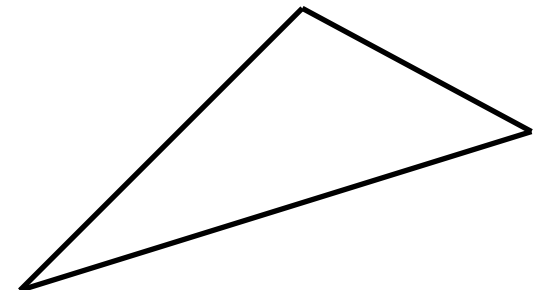
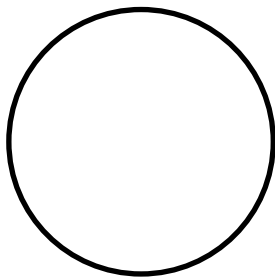
- Casting doesn't actually change the object's behavior.
It just gets the code to compile/run.

```
((Employee) linda).getVacationForm()           // pink
```

Interfaces

Shapes example

- Consider the task of writing classes to represent 2D shapes such as `Circle`, `Rectangle`, and `Triangle`.
- Certain operations are common to all shapes:
 - perimeter: distance around the outside of the shape
 - area: amount of 2D space occupied by the shape
- Every shape has these, but each computes them differently.



Shape area and perimeter

- Circle (as defined by radius r):

$$\text{area} = \pi r^2$$

$$\text{perimeter} = 2 \pi r$$

- Rectangle (as defined by width w and height h):

$$\text{area} = w h$$

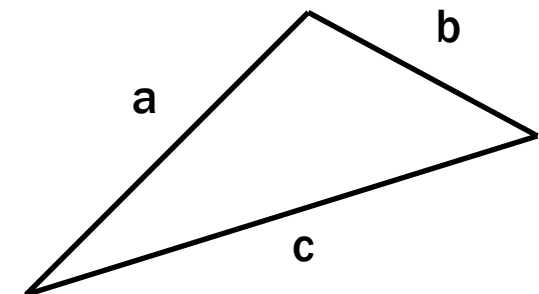
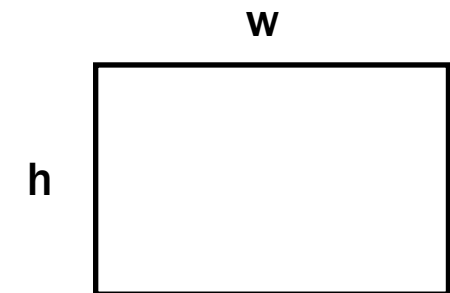
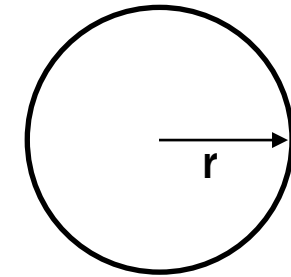
$$\text{perimeter} = 2w + 2h$$

- Triangle (as defined by side lengths a , b , and c)

$$\text{area} = \sqrt{s(s-a)(s-b)(s-c)}$$

$$\text{where } s = \frac{1}{2}(a+b+c)$$

$$\text{perimeter} = a + b + c$$



Common behavior

- Suppose we have 3 classes `Circle`, `Rectangle`, `Triangle`.
 - Each has the methods `perimeter` and `area`.
- We'd like our client code to be able to treat different kinds of shapes in the same way:
 - Write a method that prints any shape's area and perimeter.
 - Create an array to hold a mixture of the various shape objects.
 - Write a method that could return a rectangle, a circle, a triangle, or any other kind of shape.
 - Make a `DrawingPanel` display many shapes on screen.

Interfaces

- **interface:** A list of methods that a class can promise to implement.
 - Inheritance gives you an is-a relationship *and* code sharing.
 - A `Lawyer` can be treated as an `Employee` and inherits its code.
 - Interfaces give you an is-a relationship *without* code sharing.
 - A `Rectangle` object can be treated as a `Shape` but inherits no code.
 - Analogous to non-programming idea of roles or certifications:
 - "I'm certified as a CPA accountant.
This assures you I know how to do taxes, audits, and consulting."
 - "I'm 'certified' as a Shape, because I implement the Shape interface.
This assures you I know how to compute my area and perimeter."

Interface syntax

```
public interface name {  
    public type name(type name, ..., type name);  
    public type name(type name, ..., type name);  
    ...  
    public type name(type name, ..., type name);  
}
```

Example:

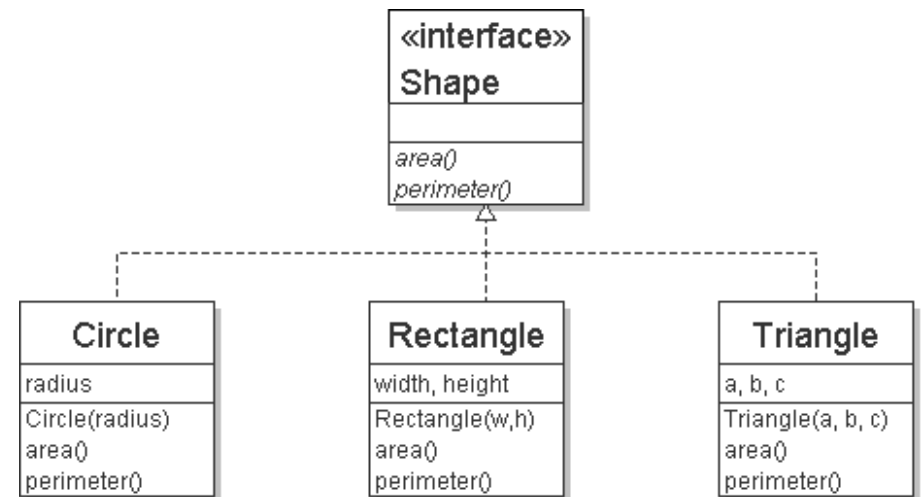
```
public interface Vehicle {  
    public int getSpeed();  
    public void setDirection(int direction);  
}
```


Shape interface

// Describes features common to all shapes.

```
public interface Shape {  
    public double area();  
    public double perimeter();  
}
```

- Saved as Shape.java



- **abstract method:** A header without an implementation.
 - The actual bodies are not specified, because we want to allow each class to implement the behavior in its own way.

Implementing an interface

```
public class name implements interface {  
    ...  
}
```

- A class can declare that it "implements" an interface.
- The class promises to contain each method in that interface.
(Otherwise it will fail to compile.)

- Example:

```
public class Bicycle implements Vehicle {  
    ...  
}
```

Interface requirements

```
public class Banana implements Shape {  
    // haha, no methods! pwned  
}
```

- If we write a class that claims to be a Shape but doesn't implement area and perimeter methods, it will not compile.

```
Banana.java:1: Banana is not abstract and does not override abstract  
method area() in Shape  
public class Banana implements Shape {  
    ^
```

Interfaces + polymorphism

- Interfaces benefit the *client code* author the most.
- they allow **polymorphism**
(the same code can work with different types of objects)

```
public static void printInfo(Shape s) {  
    System.out.println("The shape: " + s);  
    System.out.println("area : " + s.area());  
    System.out.println("perim: " + s.perimeter());  
    System.out.println();  
}  
  
...  
Circle circ = new Circle(12.0);  
Triangle tri = new Triangle(5, 12, 13);  
printInfo(circ);  
printInfo(tri);
```

Abstract Classes

List classes example

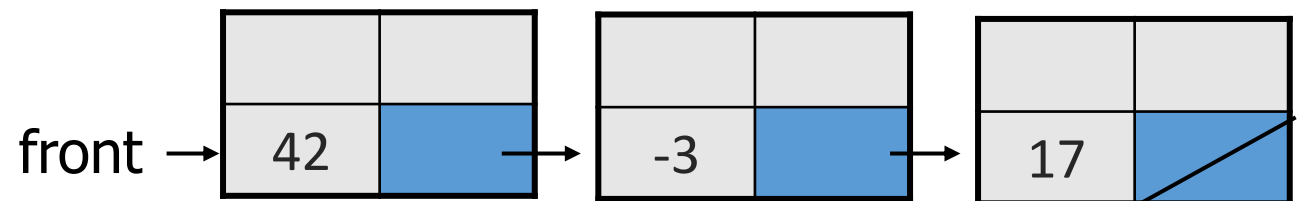
- Suppose we have implemented the following two list classes:

- `ArrayList`

	42	-3	17

- `LinkedList`

- We have a `List` interface to indicate that both implement a List ADT.
- Problem:
 - Some of their methods are implemented the same way (redundancy).



Common code

- Notice that some of the methods are implemented the same way in both the array and linked list classes.
 - `add(value)`
 - `contains`
 - `isEmpty`
- Should we change our interface to a class? Why / why not?
 - How can we capture this common behavior?

Abstract classes (9.6)

- **abstract class:** A hybrid between an interface and a class.
 - defines a superclass type that can contain method declarations (like an interface) and/or method bodies (like a class)
 - like interfaces, abstract classes that cannot be instantiated (cannot use `new` to create any objects of their type)
- What goes in an abstract class?
 - implementation of common state and behavior that will be inherited by subclasses (parent class role)
 - declare generic behaviors that subclasses implement (interface role)

Abstract class syntax

```
// declaring an abstract class
public abstract class name {
    ...

    // declaring an abstract method
    // (any subclass must implement it)
    public abstract type name(parameters) ;
}
```

- A class can be `abstract` even if it has no abstract methods
- You can create variables (but not objects) of the abstract type

Abstract and interfaces

- Normal classes that claim to implement an interface must implement all methods of that interface:

```
public class Empty implements List {} // error
```

- Abstract classes can claim to implement an interface without writing its methods; subclasses must implement the methods.

```
public abstract class Empty implements List {} // ok
```

```
public class Child extends Empty {} // error
```

An abstract list class

// Superclass with common code for a list of integers.

```
public abstract class AbstractList implements List {  
    public void add(int value) {  
        add(size(), value);  
    }  
  
    public boolean contains(int value) {  
        return indexOf(value) >= 0;  
    }  
  
    public boolean isEmpty() {  
        return size() == 0;  
    }  
}
```

```
public class ArrayList extends AbstractList { ...
```

```
public class LinkedList extends AbstractList { ...
```

Abstract class vs. interface

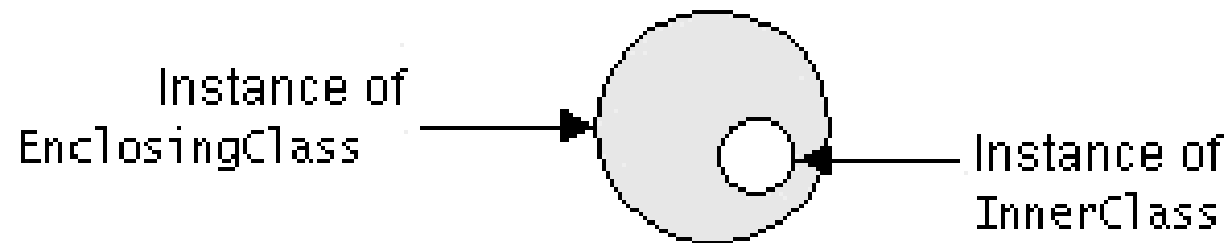
- Why do both interfaces and abstract classes exist in Java?
 - An abstract class can do everything an interface can do and more.
 - So why would someone ever use an interface?
- Answer: Java has single inheritance.
 - can extend only one superclass
 - can implement many interfaces
- Having interfaces allows a class to be part of a hierarchy (polymorphism) without using up its inheritance relationship.

```
public class Pokus extends Shape implements interface 1, interface 2, ... {  
  
{
```

Inner Classes

Inner classes

- **inner class:** A class defined inside of another class.
 - can be created as `static` or non-static
 - we will focus on standard non-static ("nested") inner classes
- usefulness:
 - inner classes are hidden from other classes (encapsulated)
 - inner objects can access/modify the fields of the outer object



Inner class syntax

```
// outer (enclosing) class
public class name {
    ...

    // inner (nested) class
    private class name {
        ...
    }
}
```

- Only this file can see the inner class or make objects of it.
- Each inner object is associated with the outer object that created it, so it can access/modify that outer object's methods/fields.
 - If necessary, can refer to outer object as **OuterClassName**.`this`

Example: Array list iterator

```
public class ArrayList extends AbstractList {  
    ...  
    // not perfect; doesn't forbid multiple removes in a row  
    private class ArrayIterator implements Iterator<Integer> {  
        private int index;    // current position in list  
  
        public ArrayIterator() {  
            index = 0;  
        }  
  
        public boolean hasNext() {  
            return index < size();  
        }  
  
        public Integer next() {  
            index++;  
            return get(index - 1);  
        }  
  
        public void remove() {  
            ArrayList.this.remove(index - 1);  
            index--;  
        }  
    }  
}
```

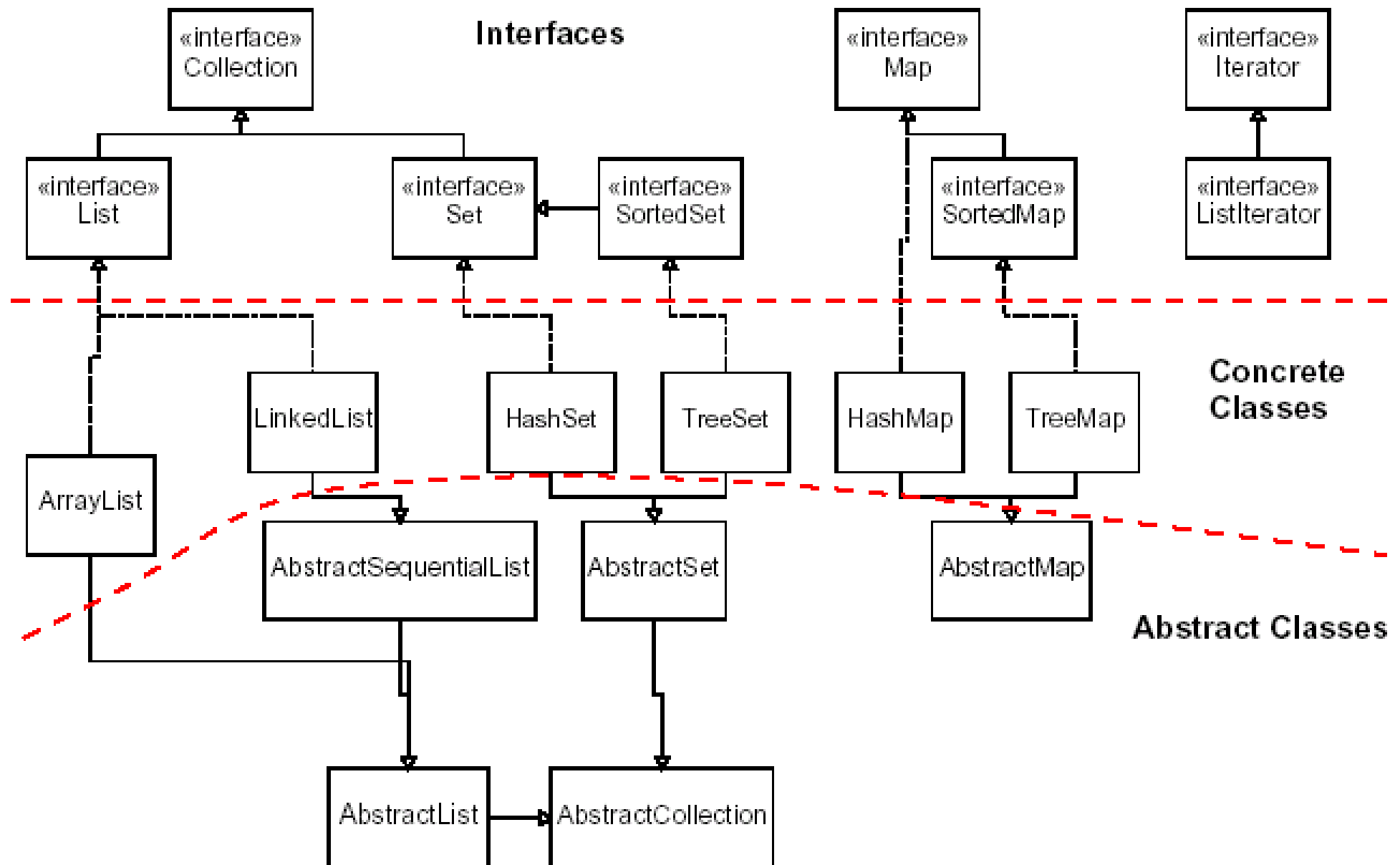

Collections

Collections

- **collection**: an object that stores data; a.k.a. "data structure"
 - the objects stored are called **elements**
 - some collections maintain an ordering; some allow duplicates
 - typical operations: *add*, *remove*, *clear*, *contains* (search), *size*
- examples found in the Java class libraries:
 - `ArrayList`, `LinkedList`, `HashMap`, `TreeSet`, `PriorityQueue`
- all collections are in the `java.util` package

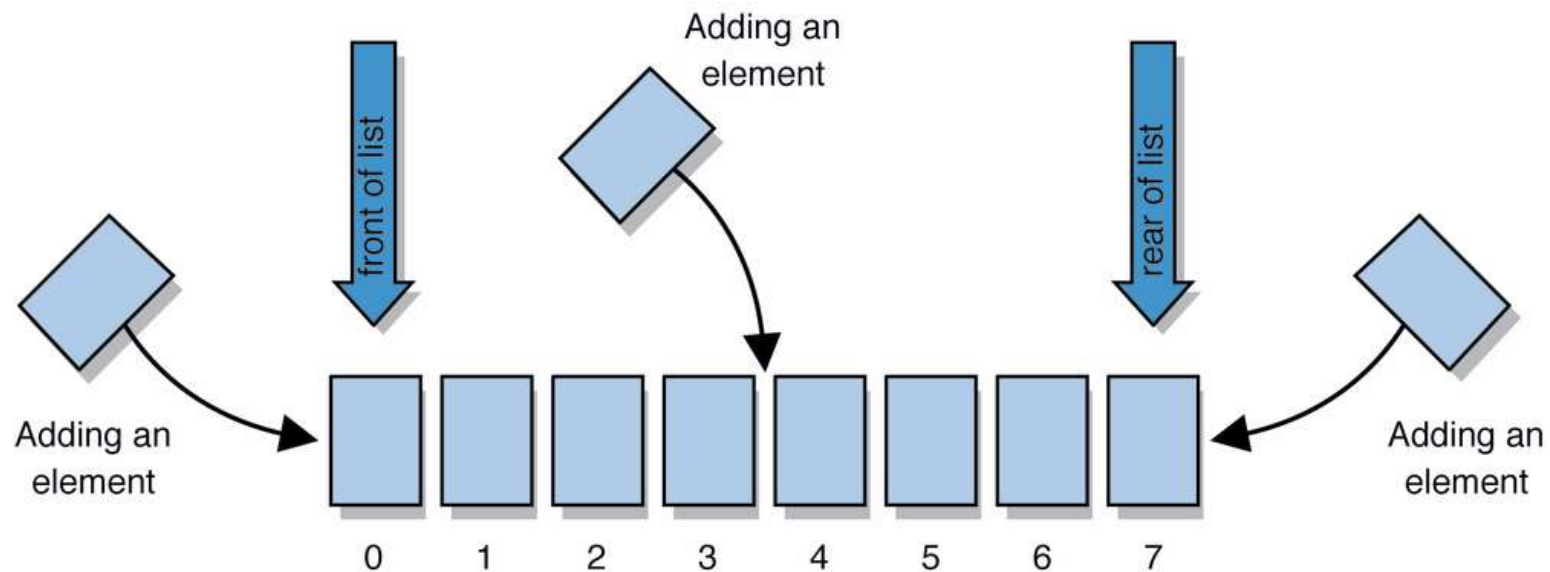
```
import java.util.*;
```

Java collection framework



Lists

- **list**: a collection storing an ordered sequence of elements
 - each element is accessible by a 0-based **index**
 - a list has a **size** (number of elements that have been added)
 - elements can be added to the front, back, or elsewhere
 - in Java, a list can be represented as an **ArrayList** object



Idea of a list

- Rather than creating an array of boxes, create an object that represents a "list" of items. (initially an empty list.)

```
[]
```

- You can add items to the list.
 - The default behavior is to add to the end of the list.

```
[hello, ABC, goodbye, okay]
```

- The list object keeps track of the element values that have been added to it, their order, indexes, and its total size.
 - Think of an "array list" as an automatically resizing array object.
 - Internally, the list is implemented using an array and a size field.

ArrayList methods (10.1)

<code>add (value)</code>	appends value at end of list
<code>add (index, value)</code>	inserts given value just before the given index, shifting subsequent values to the right
<code>clear ()</code>	removes all elements of the list
<code>indexOf (value)</code>	returns first index where given value is found in list (-1 if not found)
<code>get (index)</code>	returns the value at given index
<code>remove (index)</code>	removes/returns value at given index, shifting subsequent values to the left
<code>set (index, value)</code>	replaces value at given index with given value
<code>size ()</code>	returns the number of elements in list
<code>toString ()</code>	returns a string representation of the list such as "[3, 42, -7, 15]"

ArrayList methods 2

<code>addAll (list)</code> <code>addAll (index, list)</code>	adds all elements from the given list to this list (at the end of the list, or inserts them at the given index)
<code>contains (value)</code>	returns true if given value is found somewhere in this list
<code>containsAll (list)</code>	returns true if this list contains every element from given list
<code>equals (list)</code>	returns true if given other list contains the same elements
<code>iterator()</code> <code>listIterator()</code>	returns an object used to examine the contents of the list
<code>lastIndexOf (value)</code>	returns last index value is found in list (-1 if not found)
<code>remove (value)</code>	finds and removes the given value from this list
<code>removeAll (list)</code>	removes any elements found in the given list from this list
<code>retainAll (list)</code>	removes any elements <i>not</i> found in given list from this list
<code>subList (from, to)</code>	returns the sub-portion of the list between indexes from (inclusive) and to (exclusive)
<code>toArray()</code>	returns the elements in this list as an array

Type Parameters (Generics)

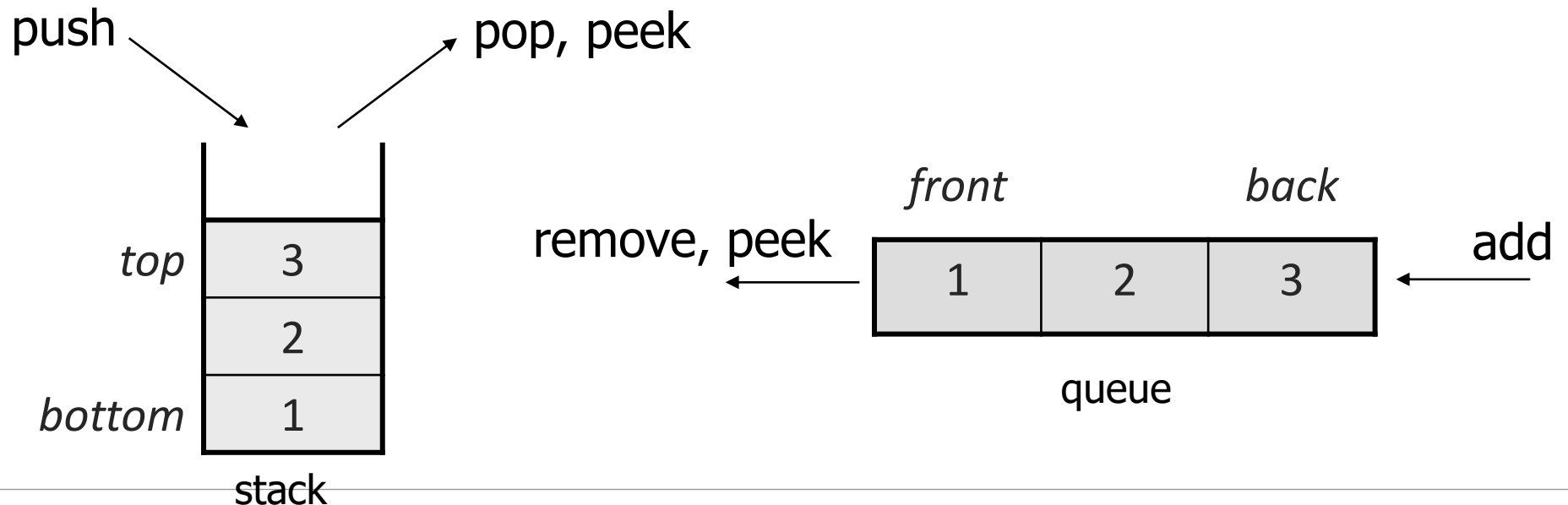
```
List<Type> name = new ArrayList<Type>();
```

- When constructing an `ArrayList`, you must specify the type of elements it will contain between `<` and `>`.
 - This is called a *type parameter* or a *generic* class.
 - Allows the same `ArrayList` class to store lists of different types.

```
List<String> names = new ArrayList<String>();  
names.add("Marty Stepp");  
names.add("Stuart Reges");
```

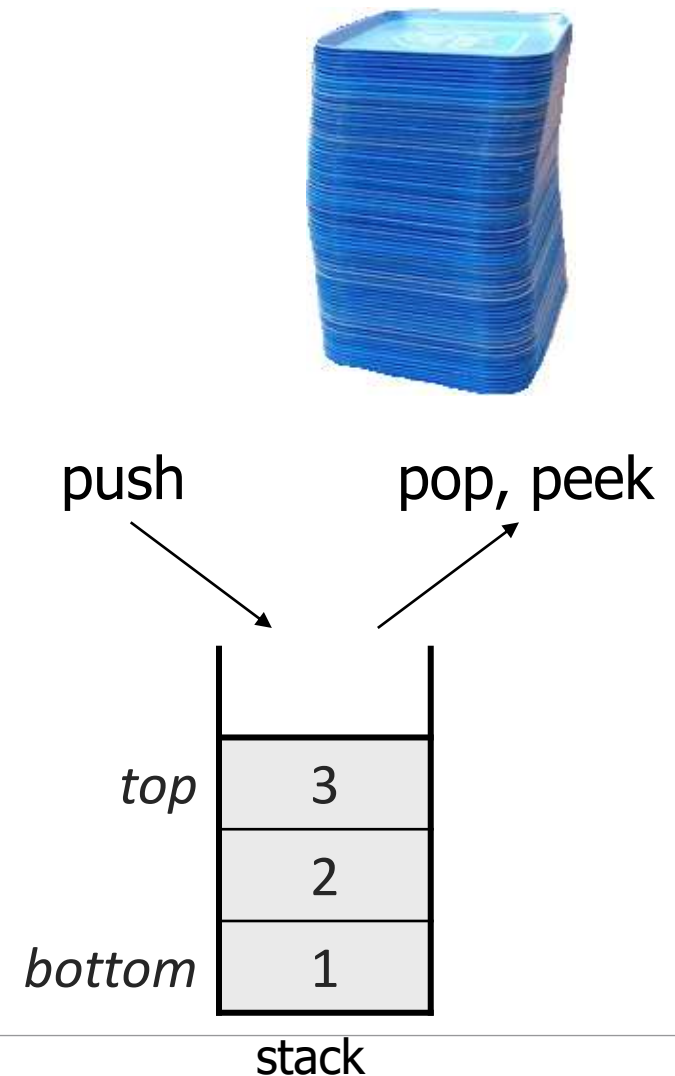

Stacks and queues

- Sometimes it is good to have a collection that is less powerful, but is optimized to perform certain operations very quickly.
- Two specialty collections:
 - **stack**: Retrieves elements in the reverse of the order they were added.
 - **queue**: Retrieves elements in the same order they were added.



Stacks

- **stack**: A collection based on the principle of adding elements and retrieving them in the opposite order.
 - Last-In, First-Out ("LIFO")
 - The elements are stored in order of insertion, but we do not think of them as having indexes.
 - The client can only add/remove/examine the last element added (the "top").
- basic stack operations:
 - **push**: Add an element to the top.
 - **pop**: Remove the top element.
 - **peek**: Examine the top element.



Class Stack

<code>Stack<E>()</code>	constructs a new stack with elements of type E
<code>push(value)</code>	places given value on top of stack
<code>pop()</code>	removes top value from stack and returns it; throws <code>EmptyStackException</code> if stack is empty
<code>peek()</code>	returns top value from stack without removing it; throws <code>EmptyStackException</code> if stack is empty
<code>size()</code>	returns number of elements in stack
<code>isEmpty()</code>	returns <code>true</code> if stack has no elements

```
Stack<Integer> s = new Stack<Integer>();  
s.push(42);  
s.push(-3);  
s.push(17);                                // bottom [42, -3, 17] top  
  
System.out.println(s.pop()); // 17
```

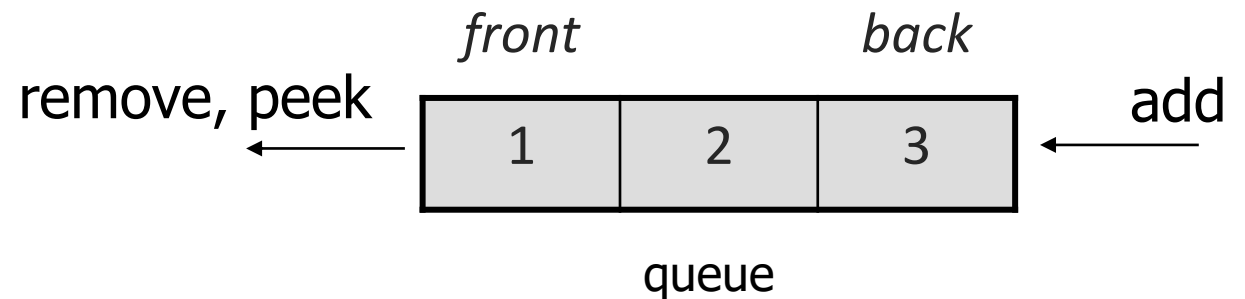
- `Stack` has other methods, but you should not use them.

Queues

- **queue**: Retrieves elements in the order they were added.
 - First-In, First-Out ("FIFO")
 - Elements are stored in order of insertion but don't have indexes.
 - Client can only add to the end of the queue, and can only examine/remove the front of the queue.



- basic queue operations:
 - **add** (enqueue): Add an element to the back.
 - **remove** (dequeue): Remove the front element.
 - **peek**: Examine the front element.



Programming with Queues

<code>add (value)</code>	places given value at back of queue
<code>remove ()</code>	removes value from front of queue and returns it; throws a <code>NoSuchElementException</code> if queue is empty
<code>peek ()</code>	returns front value from queue without removing it; returns <code>null</code> if queue is empty
<code>size ()</code>	returns number of elements in queue
<code>isEmpty ()</code>	returns <code>true</code> if queue has no elements

```
Queue<Integer> q = new LinkedList<Integer> ();  
q.add(42);  
q.add(-3);  
q.add(17);           // front [42, -3, 17] back  
  
System.out.println(q.remove()); // 42
```

- **IMPORTANT:** When constructing a queue you must use a new `LinkedList` object instead of a new `Queue` object.

Queue idioms

- As with stacks, must pull contents out of queue to view them.

```
// process (and destroy) an entire queue
```

```
while (!q.isEmpty()) {  
    do something with q.remove();  
}
```

- another idiom: Examining each element exactly once.

```
int size = q.size();  
for (int i = 0; i < size; i++) {  
    do something with q.remove();  
    (including possibly re-adding it to the queue)  
}
```

Abstract data types (ADTs)

- **abstract data type (ADT):** A specification of a collection of data and the operations that can be performed on it.
 - Describes *what* a collection does, not *how* it does it
- We don't know exactly how a stack or queue is implemented, and we don't need to.
 - We just need to understand the idea of the collection and what operations it can perform.

(Stacks are usually implemented with arrays; queues are often implemented using another structure called a linked list.)

ADTs as interfaces (11.1)

- **abstract data type (ADT):** A specification of a collection of data and the operations that can be performed on it.
 - Describes *what* a collection does, not *how* it does it.
- Java's collection framework uses interfaces to describe ADTs:
 - Collection, Deque, List, Map, Queue, Set
- An ADT can be implemented in multiple ways by classes:
 - ArrayList and LinkedList implement List
 - HashSet and TreeSet implement Set
 - LinkedList, ArrayDeque, etc. implement Queue
- They messed up on Stack; there's no Stack interface, just a class.

Using ADT interfaces

When using Java's built-in collection classes:

- It is considered good practice to always declare collection variables using the corresponding ADT interface type:

```
List<String> list = new ArrayList<String>();
```

- Methods that accept a collection as a parameter should also declare the parameter using the ADT interface type:

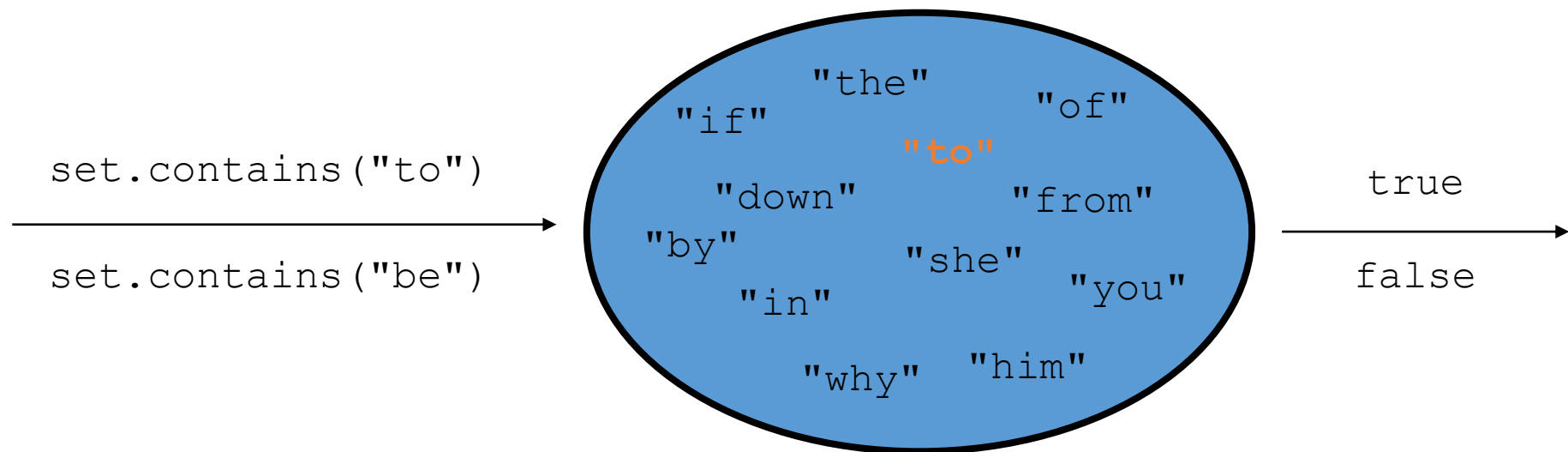
```
public void stutter(List<String> list) {  
    ...  
}
```

Why use ADTs?

- Why would we want more than one kind of list, queue, etc.?
- Answer: Each implementation is more efficient at certain tasks.
 - `ArrayList` is faster for adding/removing at the end;
 - `LinkedList` is faster for adding/removing at the front/middle.
 - Etc.
- You choose the optimal implementation for your task, and if the rest of your code is written to use the ADT interfaces, it will work.

Sets

- **set**: A collection of unique values (no duplicates allowed) that can perform the following operations efficiently:
 - add, remove, search (contains)
- We don't think of a set as having indexes; we just add things to the set in general and don't worry about order



set

Set implementation

- in Java, sets are represented by `Set` interface in `java.util`
- `Set` is implemented by `HashSet` and `TreeSet` classes
- `HashSet`: implemented using a "hash table" array;
very fast: **$O(1)$** for all operations
elements are stored in unpredictable order
- `TreeSet`: implemented using a "binary search tree";
pretty fast: **$O(\log N)$** for all operations
elements are stored in sorted order
- `LinkedHashSet`: **$O(1)$** but stores in order of insertion

Set methods

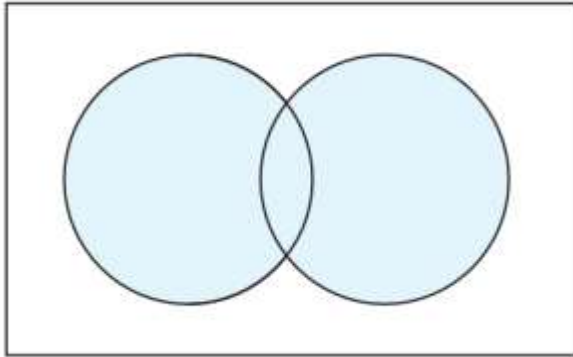
```
List<String> list = new ArrayList<String>();  
...  
Set<Integer> set = new TreeSet<Integer>();           // empty  
Set<String> set2 = new HashSet<String>(list);
```

- can construct an empty set, or one based on a given collection

<code>add(value)</code>	adds the given value to the set
<code>contains(value)</code>	returns <code>true</code> if the given value is found in this set
<code>remove(value)</code>	removes the given value from the set
<code>clear()</code>	removes all elements of the set
<code>size()</code>	returns the number of elements in list
<code>isEmpty()</code>	returns <code>true</code> if the set's size is 0
<code>toString()</code>	returns a string such as "[3, 42, -7, 15]"

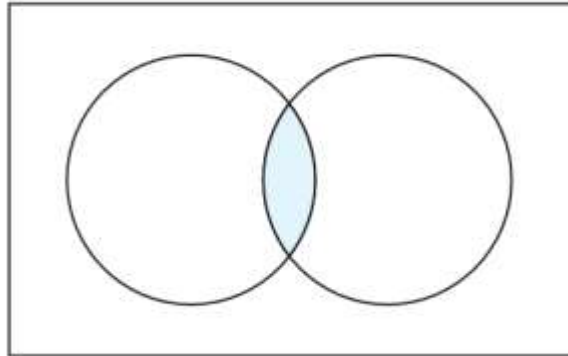
Set operations

$A \cup B$ Union



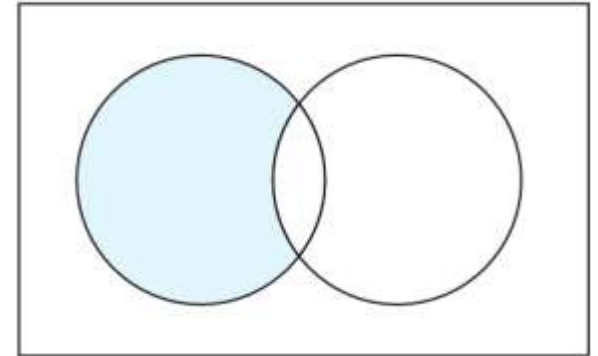
`addAll`

$A \cap B$ Intersection



`retainAll`

$A - B$ Difference



`removeAll`

<code>addAll (collection)</code>	adds all elements from the given collection to this set
<code>containsAll (coll)</code>	returns <code>true</code> if this set contains every element from given set
<code>equals (set)</code>	returns <code>true</code> if given other set contains the same elements
<code>iterator ()</code>	returns an object used to examine set's contents (<i>seen later</i>)
<code>removeAll (coll)</code>	removes all elements in the given collection from this set
<code>retainAll (coll)</code>	removes elements <i>not</i> found in given collection from this set
<code>toArray ()</code>	returns an array of the elements in this set

Sets and ordering

- HashSet : elements are stored in an unpredictable order

```
Set<String> names = new HashSet<String>();  
names.add("Jake");  
names.add("Robert");  
names.add("Marisa");  
names.add("Kasey");  
System.out.println(names);  
// [Kasey, Robert, Jake, Marisa]
```

- TreeSet : elements are stored in their "natural" sorted order

```
Set<String> names = new TreeSet<String>();  
...  
// [Jake, Kasey, Marisa, Robert]
```

- LinkedHashSet : elements stored in order of insertion

```
Set<String> names = new LinkedHashSet<String>();  
...  
// [Jake, Robert, Marisa, Kasey]
```

The "for each" loop (7.1)

```
for (type name : collection) {  
    statements;  
}
```

- Provides a clean syntax for looping over the elements of a `Set`, `List`, array, or other collection

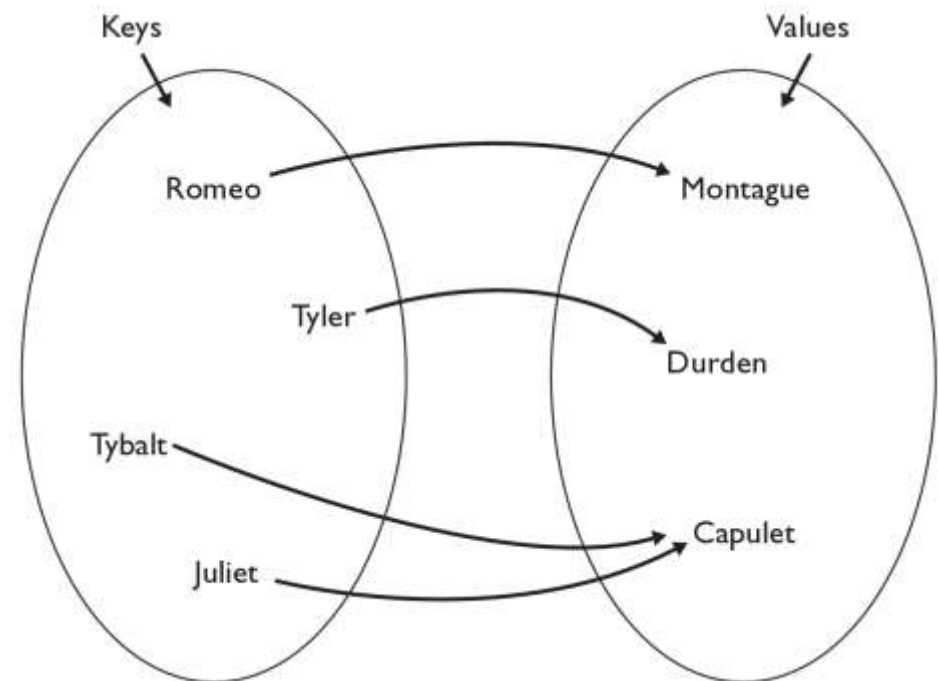
```
Set<Double> grades = new HashSet<Double>();  
...
```

```
for (double grade : grades) {  
    System.out.println("Student's grade: " + grade);  
}
```

- needed because sets have no indexes; can't get element i

The Map ADT

- **map**: Holds a set of unique *keys* and a collection of *values*, where each key is associated with one value.
 - a.k.a. "dictionary", "associative array", "hash"
- basic map operations:
 - **put**(*key*, *value*): Adds a mapping from a key to a value.
 - **get**(*key*): Retrieves the value mapped to the key.
 - **remove**(*key*): Removes the given key and its mapped value.



`myMap.get("Juliet")` returns "Capulet"

Map concepts

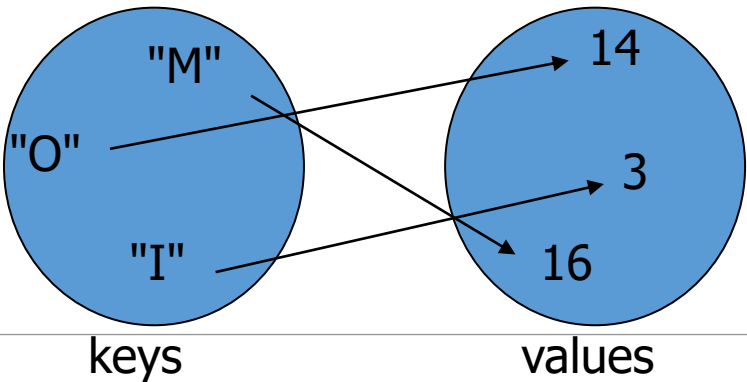
- a map can be thought of as generalization of a tallying array
 - the "index" (key) doesn't have to be an `int`
- recall previous tallying examples from CSE 142
 - count digits: 22092310907

index	0	1	2	3	4	5	6	7	8	9
value	3	1	3	0	0	0	0	1	0	2

// (M)cCain, (O)bama, (I)ndependent

- count votes: "MOOOOOOMMMMMOOOOOOMOMMIMOMMIMOMMMIO"

key	"M"	"O"	"I"
value	16	14	3



Map implementation

- in Java, maps are represented by `Map` interface in `java.util`
- `Map` is implemented by the `HashMap` and `TreeMap` classes
 - `HashMap`: implemented using an array called a "hash table"; extremely fast: **$O(1)$** ; keys are stored in unpredictable order
 - `TreeMap`: implemented as a linked "binary tree" structure; very fast: **$O(\log N)$** ; keys are stored in sorted order
- A map requires 2 type parameters: one for keys, one for values.

// maps from String keys to Integer values

```
Map<String, Integer> votes = new HashMap<String, Integer>();  
votes.put("Obama", Integer(0));  
votes.put("McCain", Integer(0));  
a = votes.get("Obama");  
votes.put("Obama", a+1);
```

Map methods

<code>put (key, value)</code>	adds a mapping from the given key to the given value; if the key already exists, replaces its value with the given one
<code>get (key)</code>	returns the value mapped to the given key (<code>null</code> if not found)
<code>containsKey (key)</code>	returns <code>true</code> if the map contains a mapping for the given key
<code>remove (key)</code>	removes any existing mapping for the given key
<code>clear ()</code>	removes all key/value pairs from the map
<code>size ()</code>	returns the number of key/value pairs in the map
<code>isEmpty ()</code>	returns <code>true</code> if the map's size is 0
<code>toString ()</code>	returns a string such as " <code>{ a=90, d=60, c=70 }</code> "

<code>keySet ()</code>	returns a set of all keys in the map
<code>values ()</code>	returns a collection of all values in the map
<code>putAll (map)</code>	adds all key/value pairs from the given map to this map
<code>equals (map)</code>	returns <code>true</code> if given map has the same mappings as this one

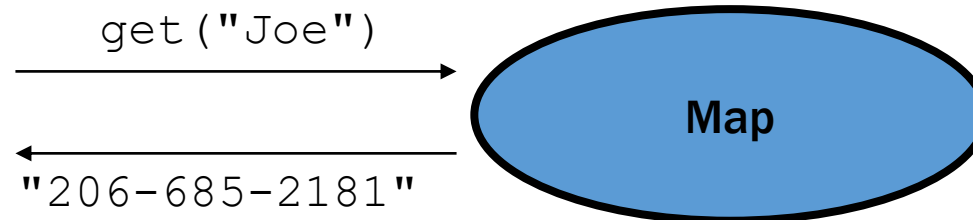
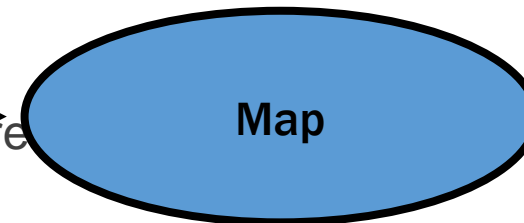
Using maps

- A map allows you to get from one half of a pair to the other.
 - Remembers one piece of information about every index (key).

```
//   key      value  
put ("Joe", "206-685-2181")
```

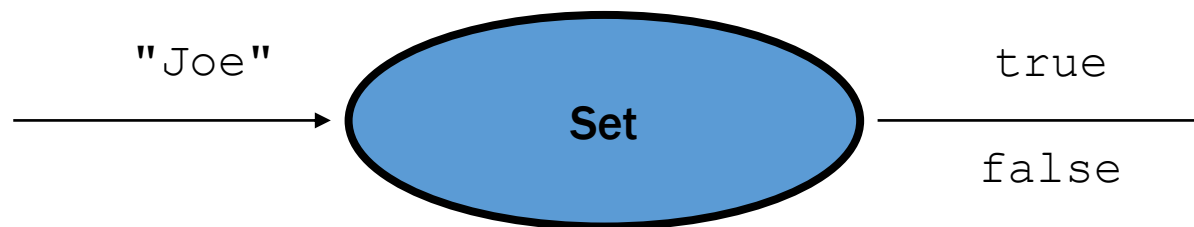
- Later, we can supply only the key and get back the value

Allows us to ask: *What is Joe's phone number?*

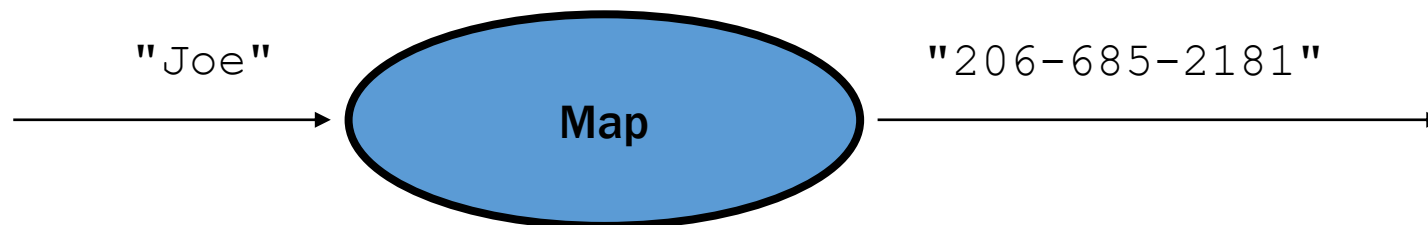


Maps vs. sets

- A set is like a map from elements to `boolean` values.
 - *Set: Is Joe found in the set? (true/false)*



- *Map: What is Joe's phone number?*



keySet and values

- `keySet` method returns a `Set` of all keys in the map
 - can loop over the keys in a `foreach` loop
 - can get each key's associated value by calling `get` on the map

```
Map<String, Integer> ages = new TreeMap<String, Integer>();
ages.put("Joe", 19);
ages.put("Geneva", 2); // ages.keySet() returns Set<String>
ages.put("Vicki", 57);
for (String name : ages.keySet()) { // Geneva -> 2
    int age = ages.get(name); // Joe -> 19
    System.out.println(name + " -> " + age); // Vicki -> 57
}
```

- `values` method returns a collection of all values in the map
 - can loop over the values in a `foreach` loop
 - no easy way to get from a value to its associated key(s)

Priority queue ADT

- **priority queue**: a collection of ordered elements that provides fast access to the minimum (or maximum) element
- usually implemented using a tree structure called a *heap*
- priority queue operations:
 - add adds in order; $O(\log N)$ worst
 - peek returns **minimum** value; $O(1)$ always
 - remove removes/returns **minimum** value; $O(\log N)$ worst
 - isEmpty,
clear,
size,
iterator $O(1)$ always

Java's PriorityQueue class

```
public class PriorityQueue<E> implements Queue<E>
```

Method/Constructor	Description	Runtime
<code>PriorityQueue<E>()</code>	constructs new empty queue	$O(1)$
<code>add(E value)</code>	adds value in sorted order	$O(\log N)$
<code>clear()</code>	removes all elements	$O(1)$
<code>iterator()</code>	returns iterator over elements	$O(1)$
<code>peek()</code>	returns minimum element	$O(1)$
<code>remove()</code>	removes/returns min element	$O(\log N)$

```
Queue<String> pq = new PriorityQueue<String>();  
pq.add("Stuart");  
pq.add("Marty");  
...
```

Priority queue ordering

- For a priority queue to work, elements must have an ordering
 - in Java, this means implementing the `Comparable` interface

- Reminder:

```
public class Foo implements Comparable<Foo> {  
    ...  
    public int compareTo(Foo other) {  
        // Return positive, zero, or negative number  
    }  
}
```

Design patterns

- **Creational Patterns**
- **Structural Patterns**
- **Behavioral Patterns**

Patterns

Creational patterns

- **Factory method**
- **Abstract factory**
- **Builder**
- **Prototype**
- **Singleton (8)**

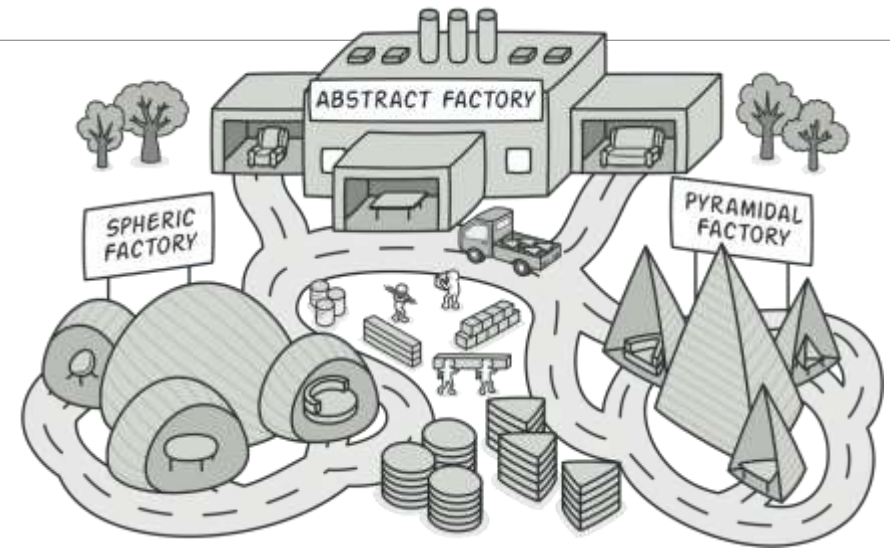
Structural patterns

- **Adapter**
- **Bridge (12)**
- **Composite**
- **Decorator**
- **Facade**
- **Flyweight (12)**
- **Proxy**

Creational patterns:










- **Chain of responsibility (12)**
- **Command**
- **Iterator**
- **Mediator**
- **Memento**
- **Observer**
- **State**
- **Strategy**
- **Template method**
- **Visitor**

Creational patterns: Abstract factory



Problem

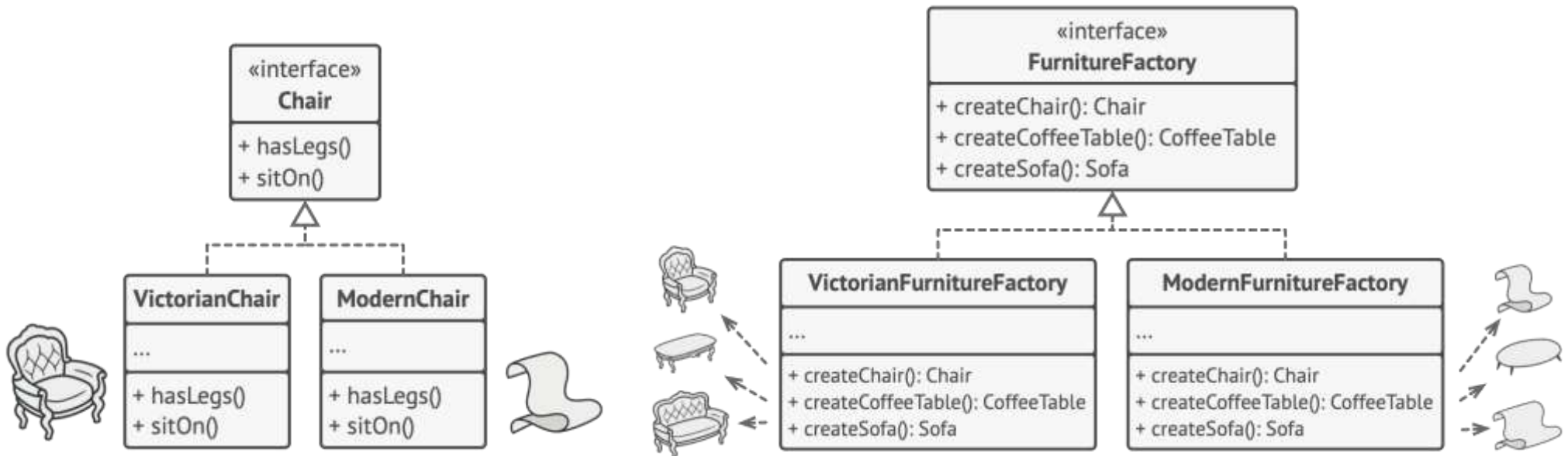
- Several variants of object of the same type
- Need a way to create individual furniture objects so that they match other objects of the same family.
- Do not change code when new type is added

	Chair	Sofa	Coffee Table
Art Deco			
Victorian			
Modern			

Creational patterns: Abstract factory (continued)

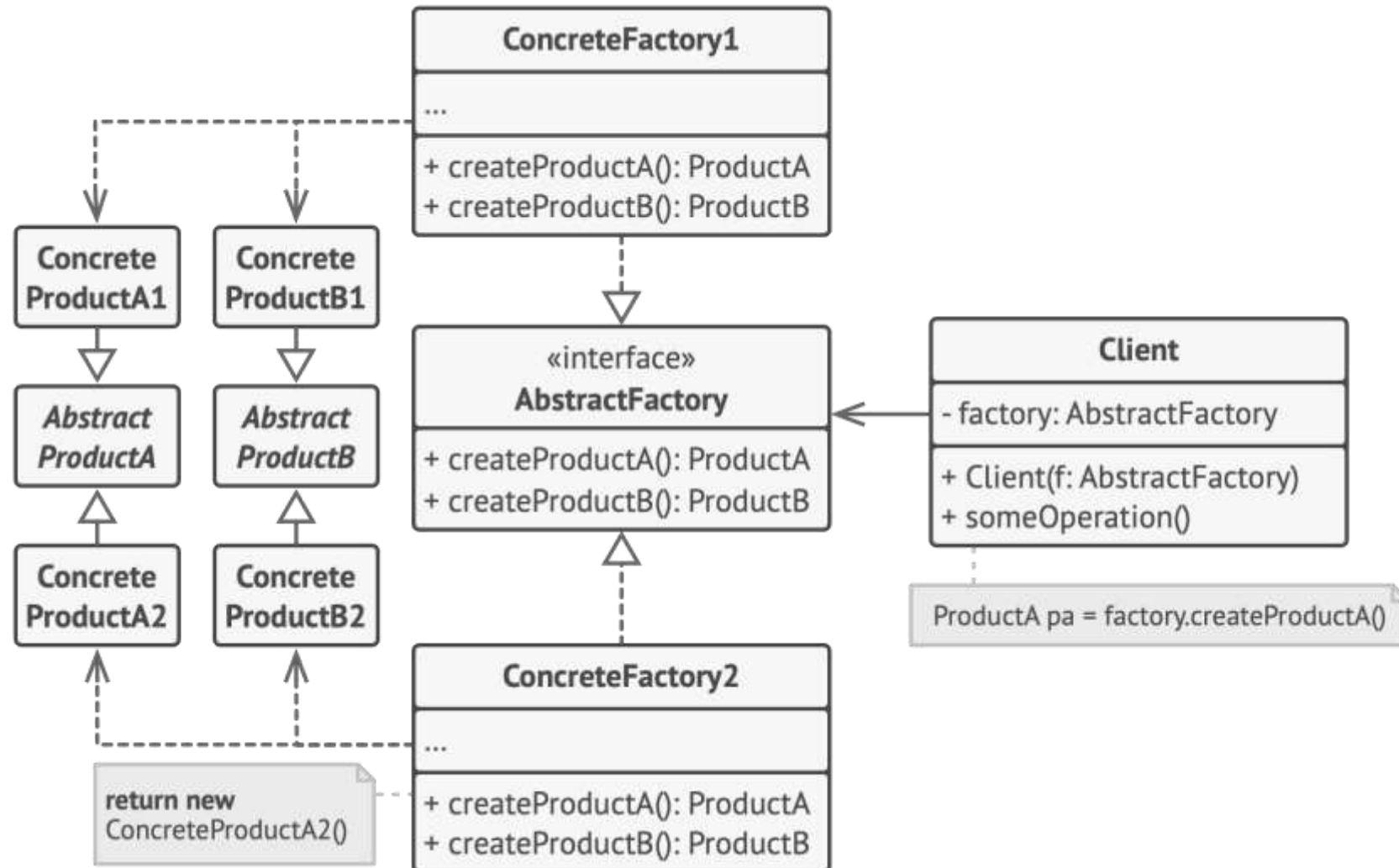
Solution

- Abstract Factory pattern suggests is to explicitly declare interfaces for each distinct product of the product family (e.g., chair, sofa or coffee table)
- Declare the *Abstract Factory*—an interface with a list of creation methods for all products that are part of the product family (for example, createChair, createSofa and createCoffeeTable). These methods must return **abstract** product types represented by the interfaces we extracted previously: Chair, Sofa, CoffeeTable and so on

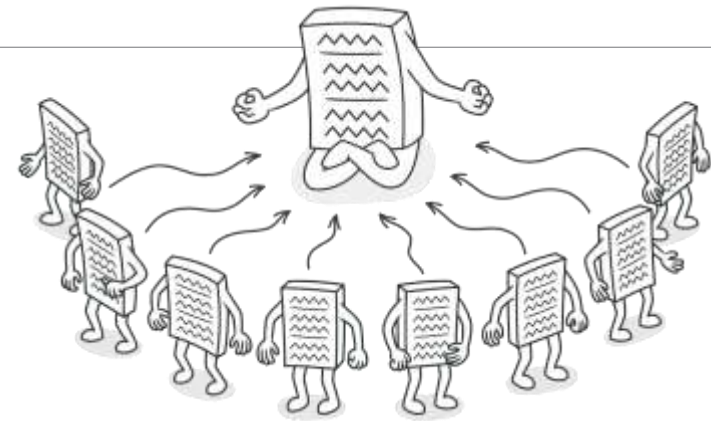


Creational patterns: Abstract factory (continued)

Usually, the application creates a concrete factory object at the initialization stage. Just before that, the app must select the factory type depending on the configuration or the environment settings.



Creational patterns: Singleton



Problem

- **Ensure that a class has just a single instance.** Why would anyone want to control how many instances a class has? The most common reason for this is to control access to some shared resource—for example, a database or a file.
- **Provide a global access point to that instance.** Remember those global variables that you (all right, me) used to store some essential objects? While they're very handy, they're also very unsafe since any code can potentially overwrite the contents of those variables and crash the app.

Note: This behaviour is impossible to implement with a regular constructor since a constructor call **must** always return a new object by design.

Creational patterns: Singleton (continued)

Solution

All implementations of the Singleton have these two steps in common:

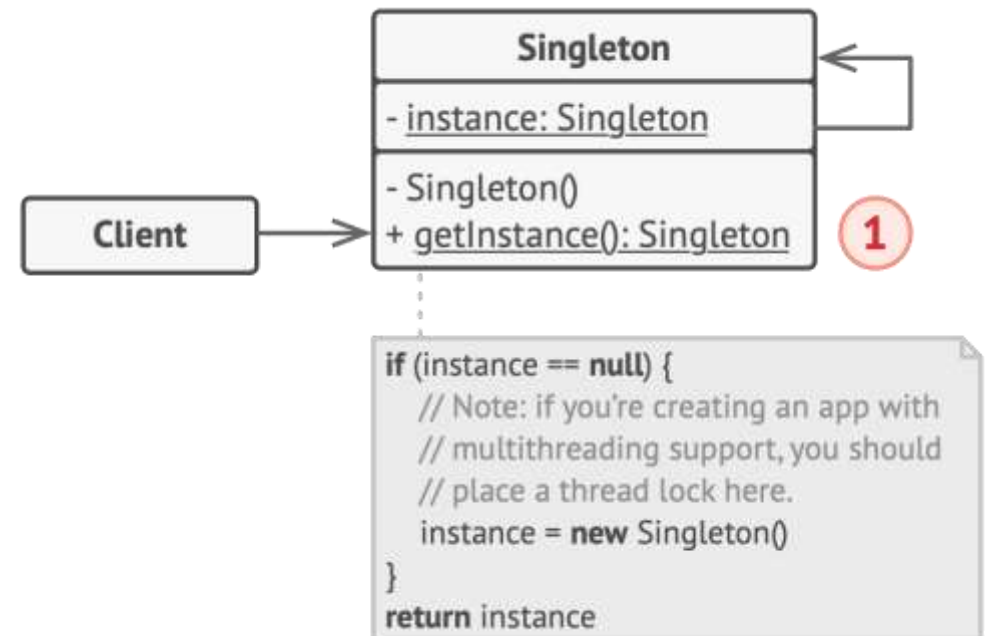
- Make the default constructor private, to prevent other objects from using the new operator with the Singleton class.
- Create a static creation method that acts as a constructor. Under the hood, this method calls the private constructor to create an object and saves it in a static field. All following calls to this method return the cached object.

If your code has access to the Singleton class, then it's able to call the Singleton's static method. So whenever that method is called, the same object is always returned.

Structure

The **Singleton** class declares the static method `getInstance` that returns the same instance of its own class.

The Singleton's constructor should be hidden from the client code. Calling the `getInstance` method should be the only way of getting the Singleton object.

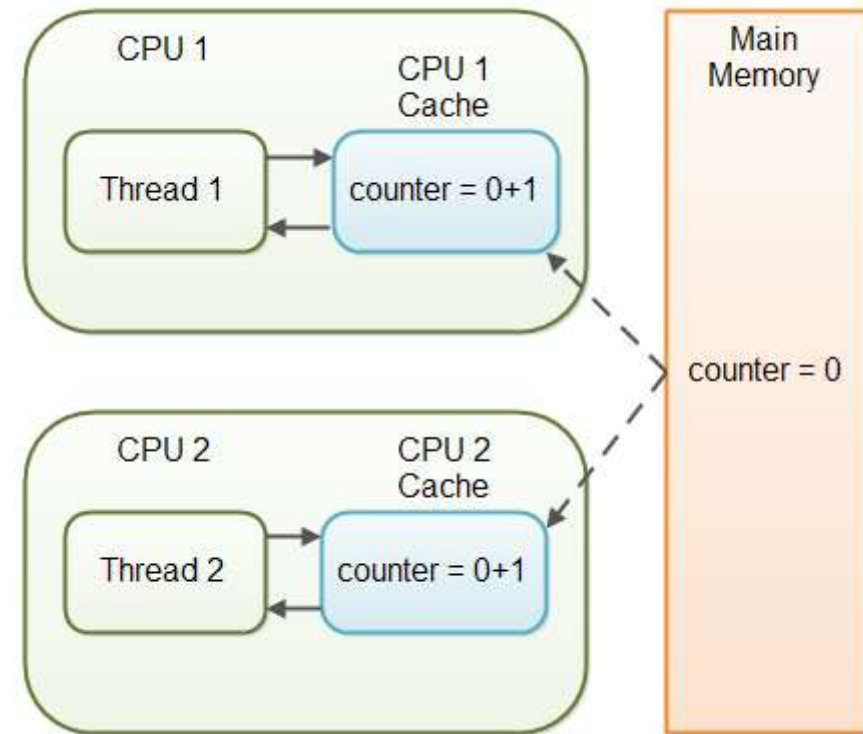


Creational patterns: Singleton (continued)

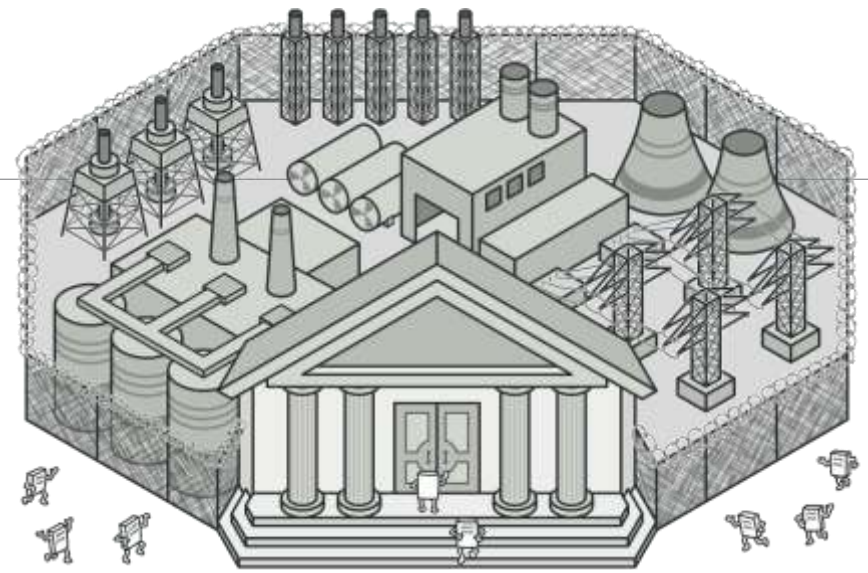
Java language notes:

Volatile keyword is **used to modify the value of a variable by different threads**. It is also used to make classes thread safe. It means that multiple threads can use a method and instance of the classes at the same time without any problem. The volatile keyword can be used either with primitive type or objects.

A *Java synchronized block* marks a method or a block of code as *synchronized*. A synchronized block in Java can only be executed a single thread at a time (depending on how you use it). Java synchronized blocks can thus be used to avoid [race conditions](#).

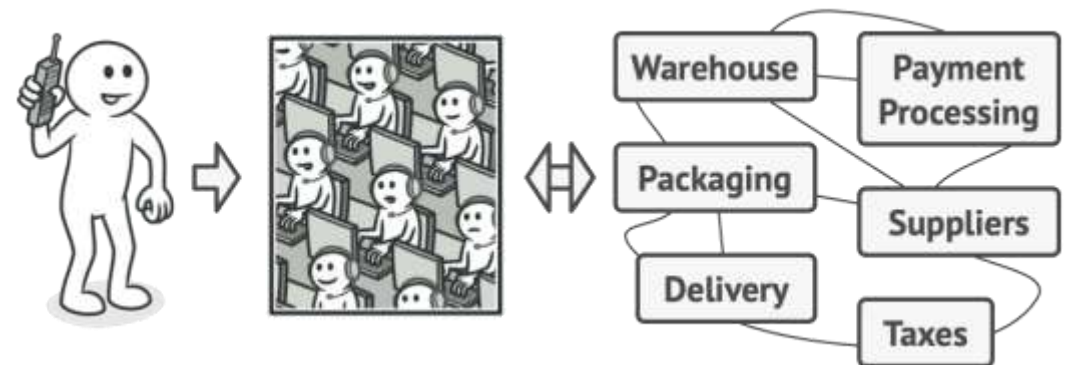


Structural patterns: Facade



Problem

- You must make your code work with a broad set of objects that belong to a sophisticated library or framework. Ordinarily, you'd need to initialize all of those objects, keep track of dependencies, execute methods in the correct order, and so on.
- As a result, the business logic of your classes would become tightly coupled to the implementation details of 3rd-party classes, making it hard to comprehend and maintain.



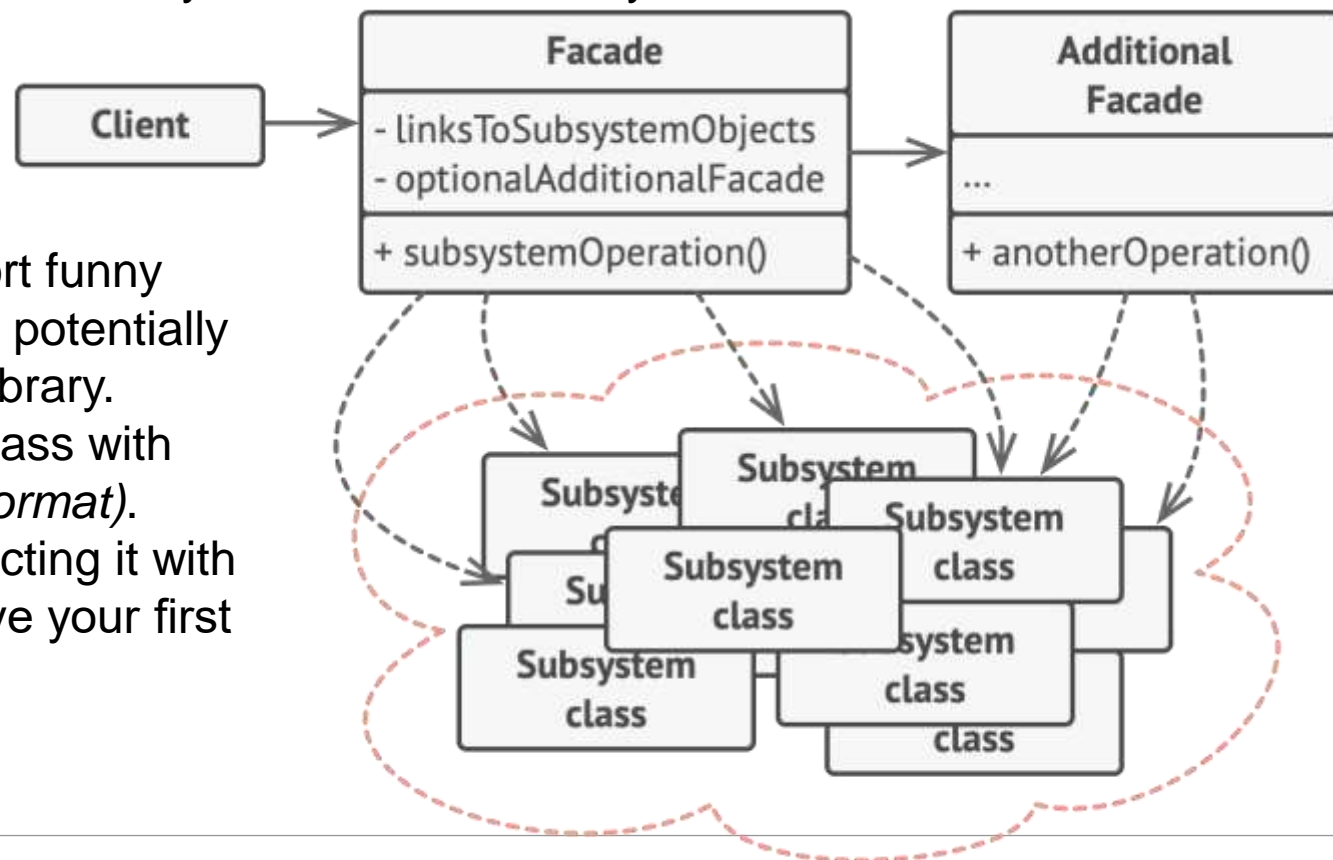
Structural patterns: Facade (continued)

Solution

A facade is a class that provides a simple interface to a complex subsystem which contains lots of moving parts. A facade might provide limited functionality in comparison to working with the subsystem directly. However, it includes only those features that clients really care about.

Having a facade is handy when you need to integrate your app with a sophisticated library that has dozens of features, but you just need a tiny bit of its functionality.

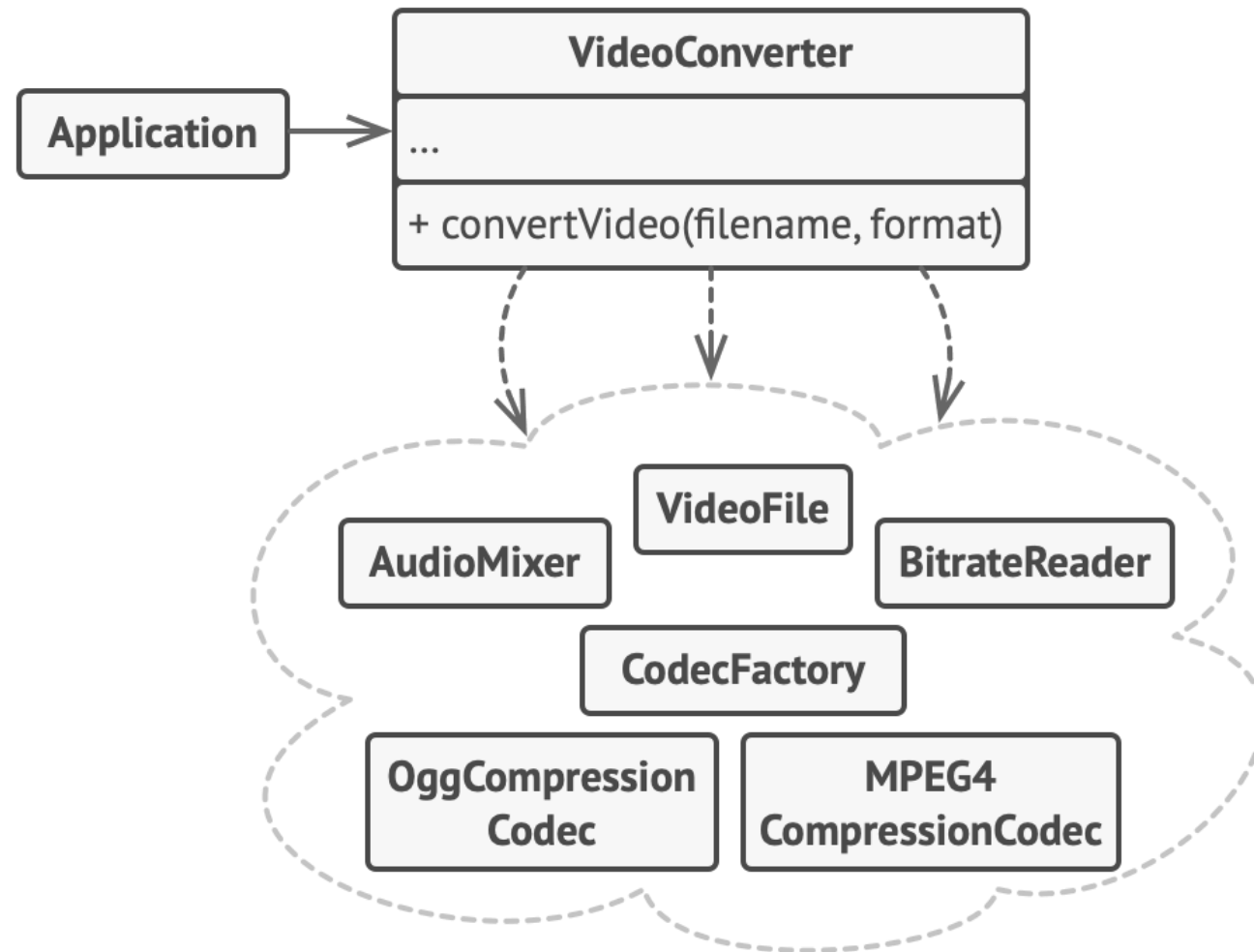
For instance, an app that uploads short funny videos with cats to social media could potentially use a professional video conversion library. However, all that it really needs is a class with the single method *encode(filename, format)*. After creating such a class and connecting it with the video conversion library, you'll have your first facade.



Structural patterns: Facade (continued)

Example

Complex video processing

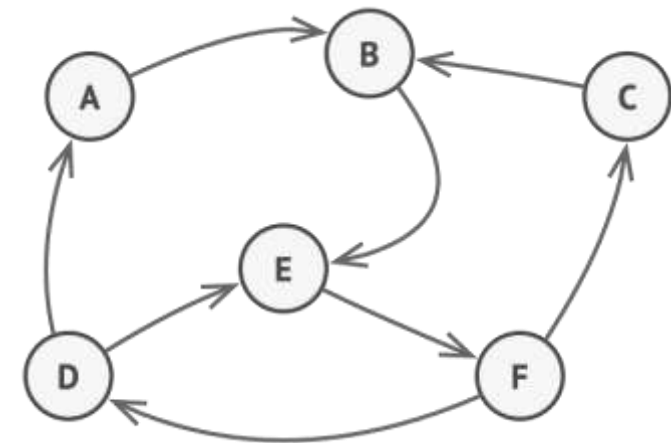
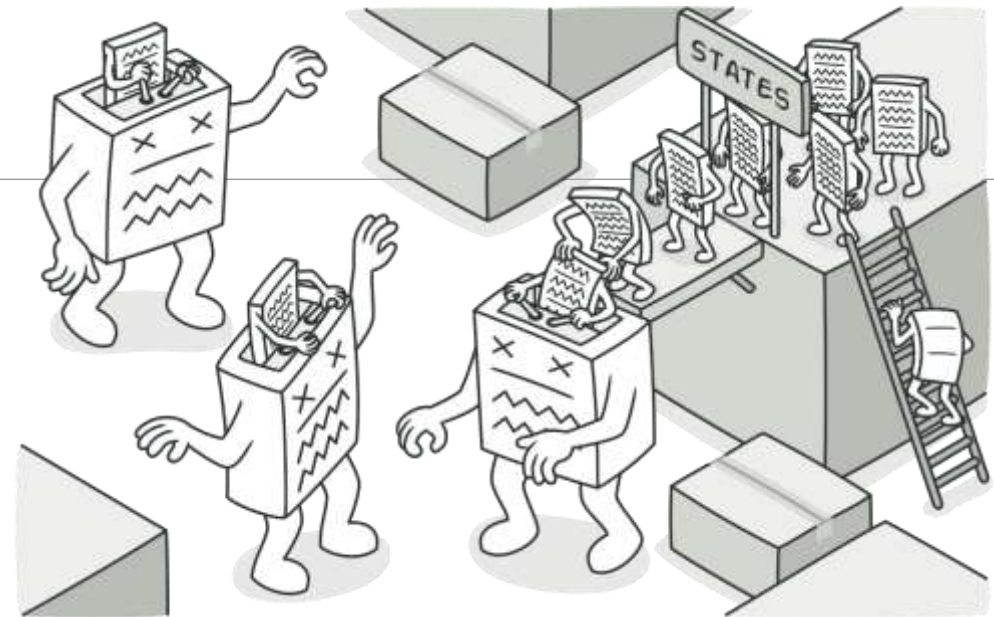


Behavioral patterns: State

Problem

At any given moment, there's a *finite* number of *states* which a program can be in. Within any unique state, the program behaves differently, and the program can be switched from one state to another instantaneously. However, depending on a current state, the program may or may not switch to certain other states. These switching rules, called *transitions*, are also finite and predetermined. You can also apply this approach to objects. Imagine that we have a Document class. A document can be in one of three states: Draft, Moderation and Published. The publish method of the document works a little bit differently in each state:

- In Draft, it moves the document to moderation.
- In Moderation, it makes the document public, but only if the current user is an administrator.
- In Published, it doesn't do anything at all.



Behavioral patterns: State (continued)

The biggest weakness of a state machine based on conditionals reveals itself once we start adding more and more states and state-dependent behaviors to the *Document* class. Most methods will contain monstrous conditionals that pick the proper behavior of a method according to the current state. Code maintenance becomes complex because any change to the transition logic may require changing state conditionals in every method.

The problem tends to get bigger as a project evolves. It's quite difficult to predict all possible states and transitions at the design stage. Hence, a lean state machine built with a limited set of conditionals can grow into a bloated mess over time.

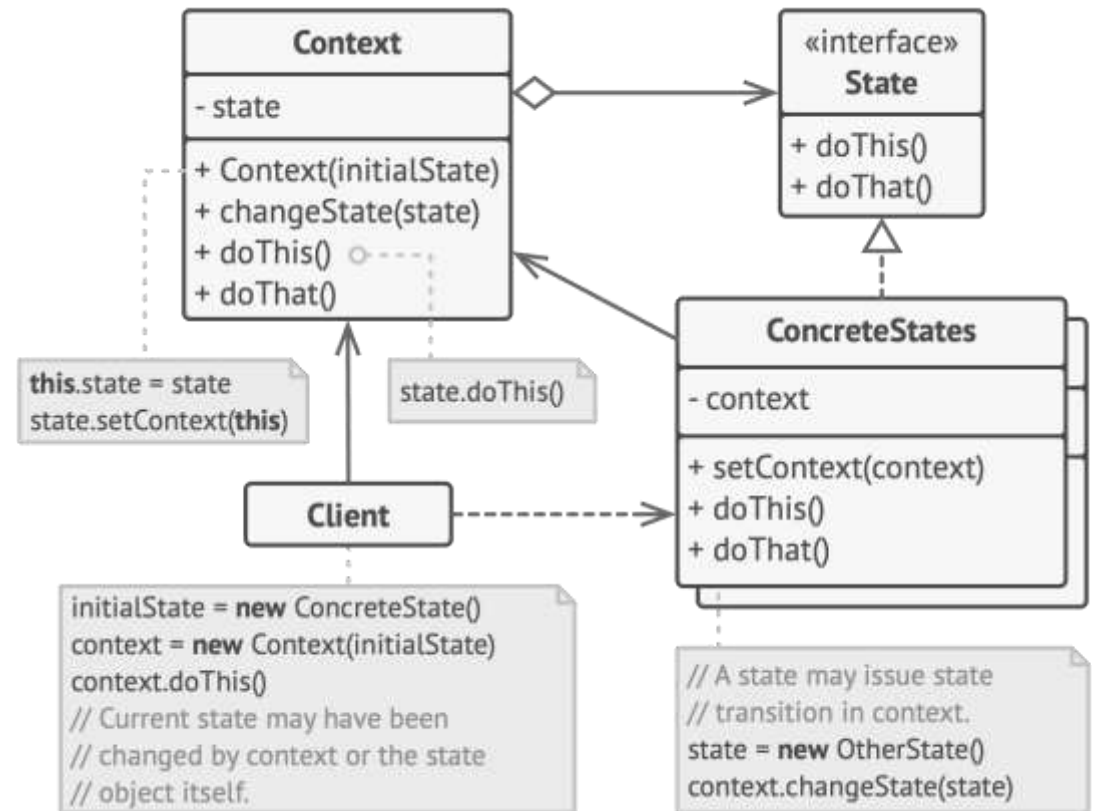
Solution

- The State pattern suggests that you create new classes for all possible states of an object and extract all state-specific behaviors into these classes.
- Instead of implementing all behaviors on its own, the original object, called *context*, stores a reference to one of the state objects that represents its current state, and delegates all the state-related work to that object.



Behavioral patterns: State (continued)

- The buttons and switches in your smartphone behave differently depending on the current state of the device:
- When the phone is unlocked, pressing buttons leads to executing various functions.
- When the phone is locked, pressing any button leads to the unlock screen.
- When the phone's charge is low, pressing any button shows the charging screen.

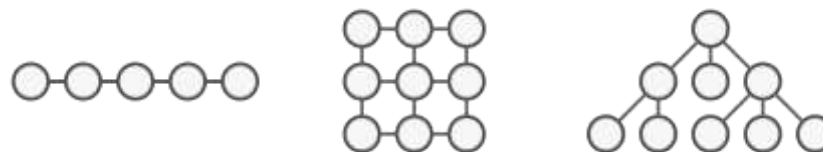


Behavioral patterns: Iterator



Problem

- Collections are one of the most used data types in programming. Nonetheless, a collection is just a container for a group of objects.
- Most collections store their elements in simple lists. However, some of them are based on stacks, trees, graphs and other complex data structures.
- But no matter how a collection is structured, it must provide some way of accessing its elements so that other code can use these elements. There should be a way to go through each element of the collection without accessing the same elements over and over.

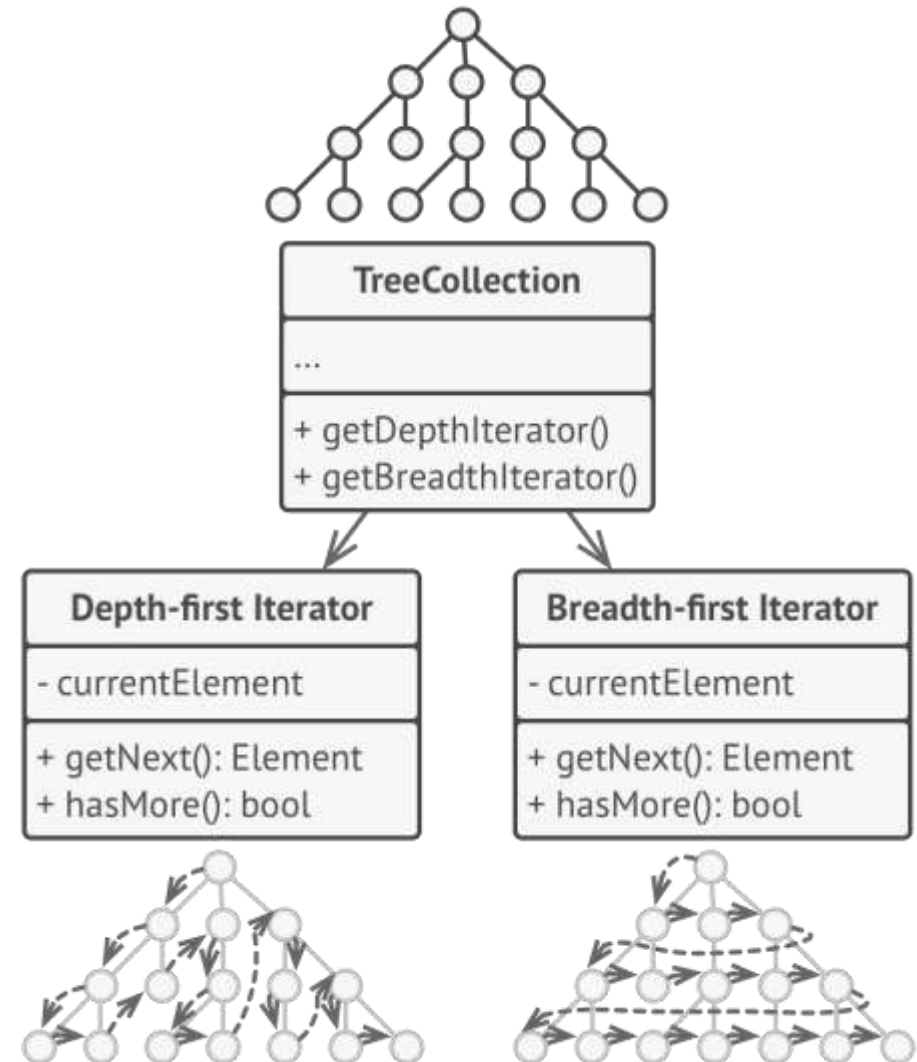


Behavioral patterns: Iterator (continued)

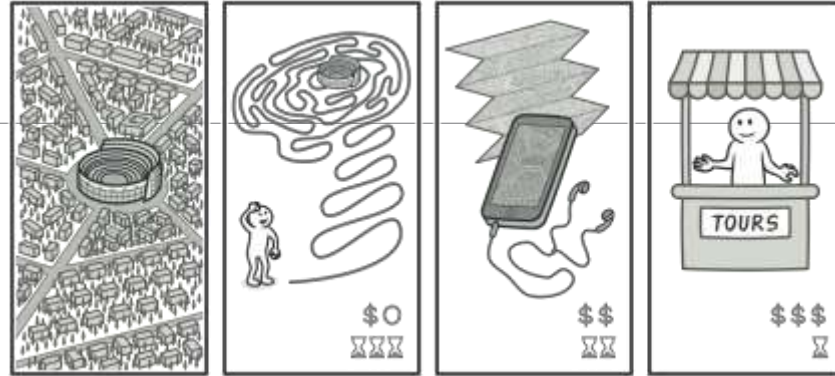
Solution

The main idea of the Iterator pattern is to extract the traversal behavior of a collection into a separate object called an *iterator*.

- In addition to implementing the algorithm itself, an **iterator object encapsulates all of the traversal details**, such as the current position and how many elements are left till the end. Because of this, several iterators can go through the same collection at the same time, independently of each other.
- Usually, iterators provide one primary method for fetching elements of the collection. The client can keep running this method until it doesn't return anything, which means that the iterator has traversed all of the elements.
- All iterators must implement the same interface. This makes the client code compatible with any collection type or any traversal algorithm as long as there's a proper iterator. If you need a special way to traverse a collection, you just create a new iterator class, without having to change the collection or the client.

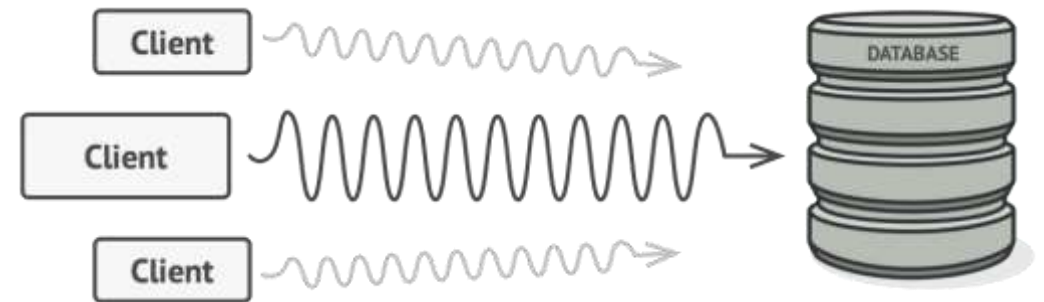


Behavioral patterns: Iterator (continued)



1. The **Iterator** interface declares the operations required for traversing a collection: fetching the next element, retrieving the current position, restarting iteration, etc.
2. **Concrete Iterators** implement specific algorithms for traversing a collection. The iterator object should track the traversal progress on its own - traverse the same collection independently of each other.
3. The **Collection** interface declares one or multiple methods for getting iterators compatible with the collection.
4. **Concrete Collections** return new instances of a particular concrete iterator class each time the client requests one. It's just that these details aren't crucial to the actual pattern.
5. The **Client** works with both collections and iterators via their interfaces. This way the client isn't coupled to concrete classes, allowing you to use various collections and iterators with the same client code.
6. Typically, clients don't create iterators on their own, but instead get them from collections. Yet, in certain cases, the client can create one directly; for example, when the client defines its own special iterator.

Structural patterns: Proxy



Proxy is a structural design pattern that lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

Problem

- You have a massive object that consumes a vast amount of system resources. You need it from time to time, but not always.
- You could implement lazy initialization: create this object only when it's actually needed. All of the object's clients would need to execute some deferred initialization code. Unfortunately, this would probably cause a lot of code duplication.
- You could **put this code directly into our object's class, but that isn't possible**. For instance, the class may be part of a closed 3rd-party library.

Structural patterns: Proxy (continued)

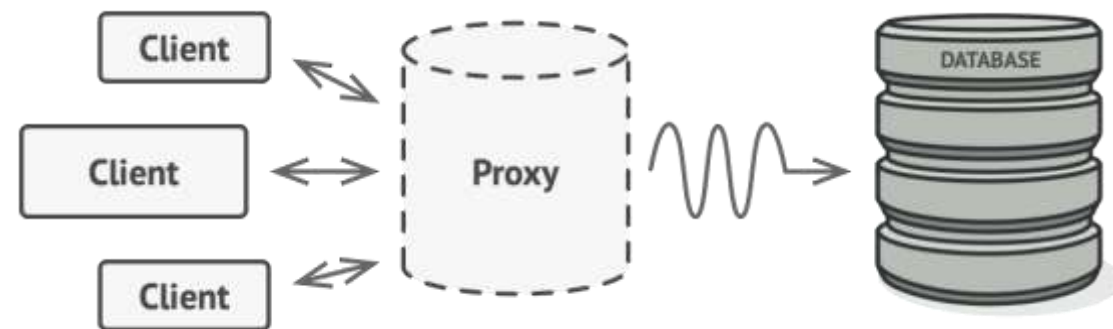
Solution

The Proxy pattern suggests that you create a new proxy class with the same interface as an original service object. Then you update your app so that it passes the proxy object to all of the original object's clients. Upon receiving a request from a client, the proxy creates a real service object and delegates all the work to it.

Benefit: If you need to execute something either before or after the primary logic of the class, the proxy lets you do this without changing that class. Since the proxy implements the same interface as the original class, it can be passed to any client that expects a real service object.

Example

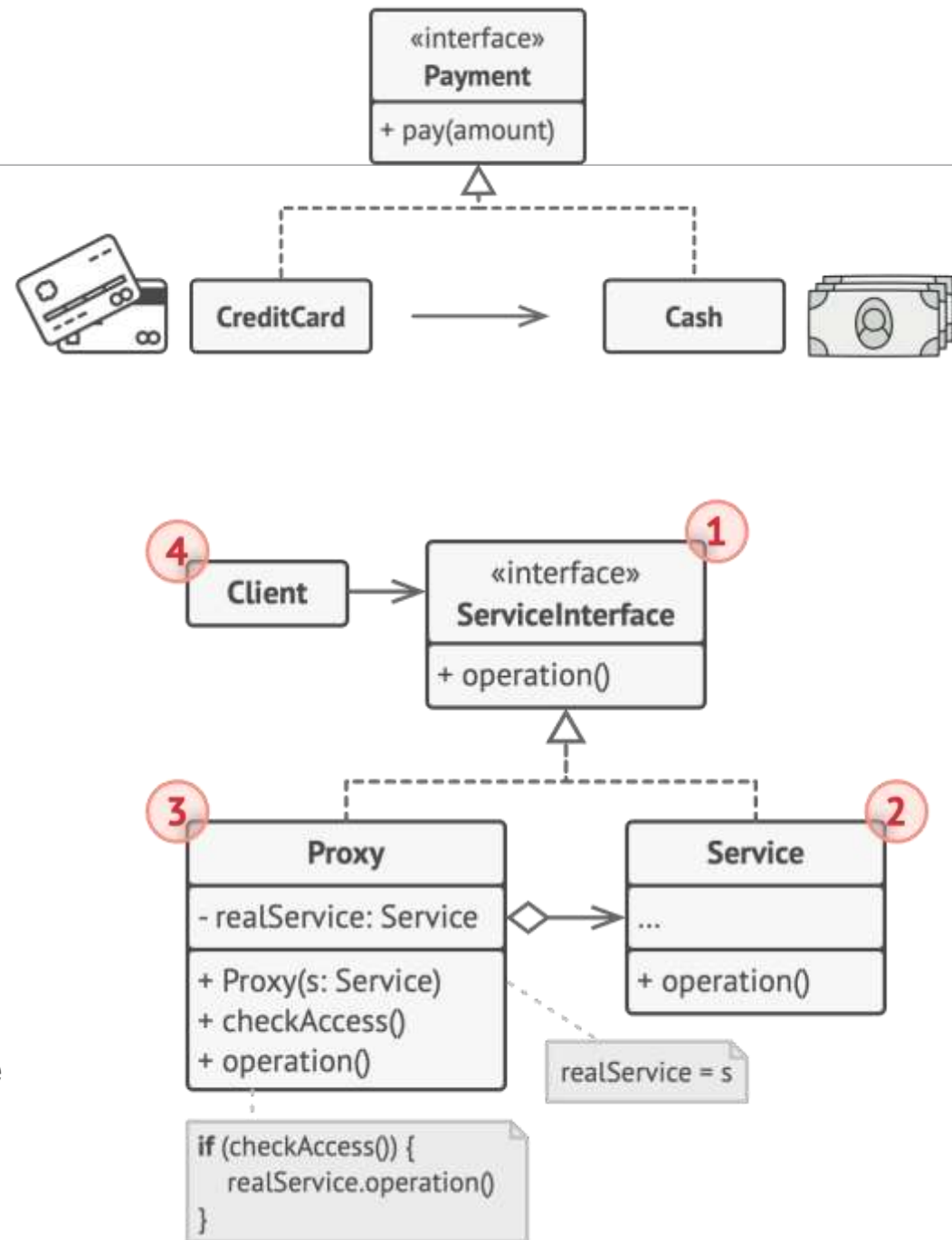
A credit card is a proxy for a bank account, which is a proxy for a bundle of cash. Both implement the same interface: they can be used for making a payment. A consumer feels great because there's no need to carry loads of cash around. A shop owner is also happy since the income from a transaction gets added electronically to the shop's bank account without the risk of losing the deposit or getting robbed on the way to the bank.



Structural patterns: Proxy (continued)

Structure

1. The **Service Interface** declares the interface of the Service. The proxy must follow this interface to be able to disguise itself as a service object.
2. The **Service** is a class that provides some useful business logic.
3. The **Proxy** class has a reference field that points to a service object. After the proxy finishes its processing (e.g., lazy initialization, logging, access control, caching, etc.), it passes the request to the service object.
4. Usually, proxies manage the full lifecycle of their service objects.
5. The **Client** should work with both services and proxies via the same interface. This way you can pass a proxy into any code that expects a service object.

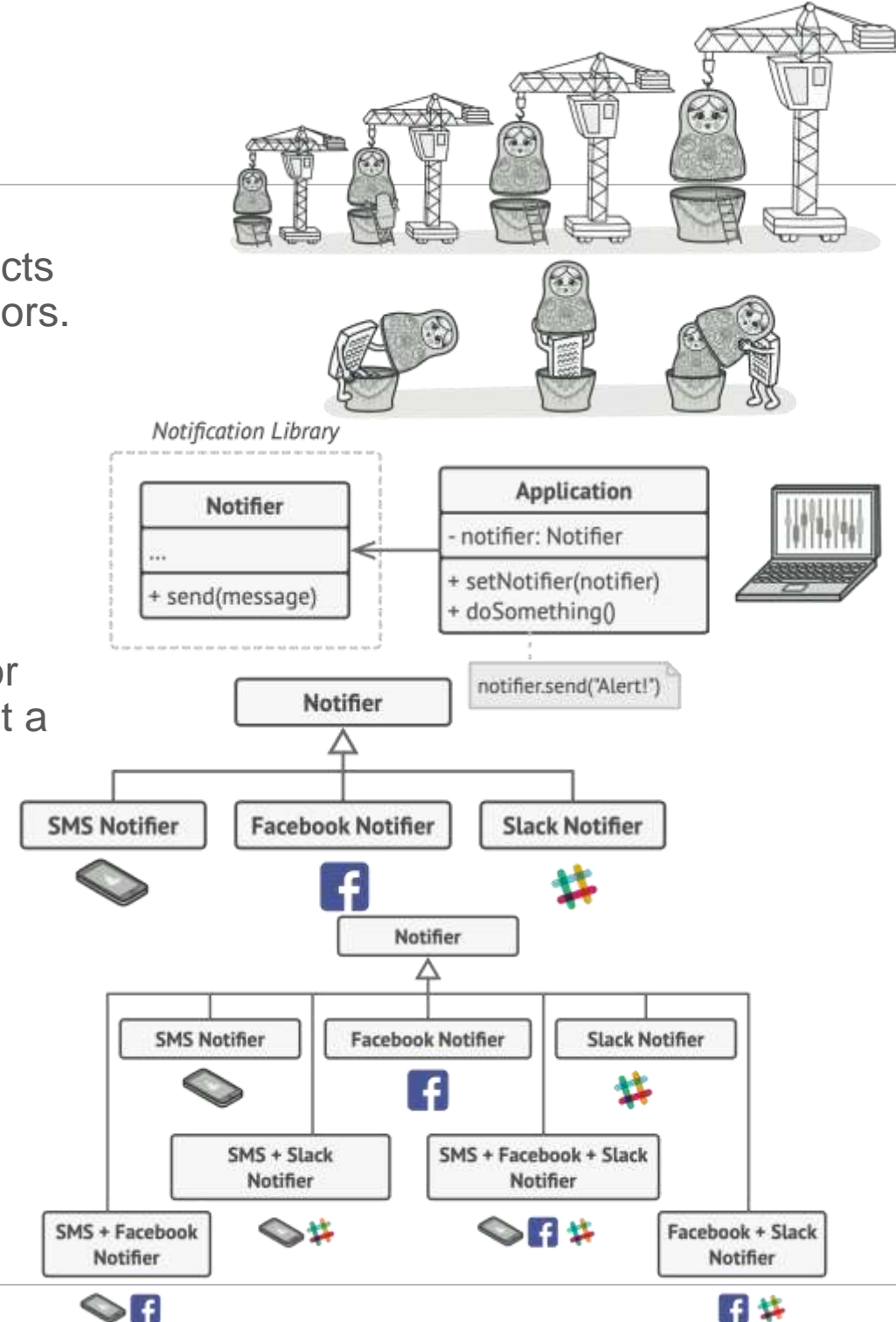


Structural patterns: Decorator

Decorator is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

Problem

- Imagine that you're working on a notification library which lets other programs notify their users about important events.
- The initial version of the library was based on the *Notifier* class that had only a few fields, a constructor and a single send method. The method could accept a message argument from a client and send the message to a list of emails that were passed to the notifier via its constructor.
- Then need more types of notifiers
- Need a combination – not only statically via inheritance, but also dynamically during run time



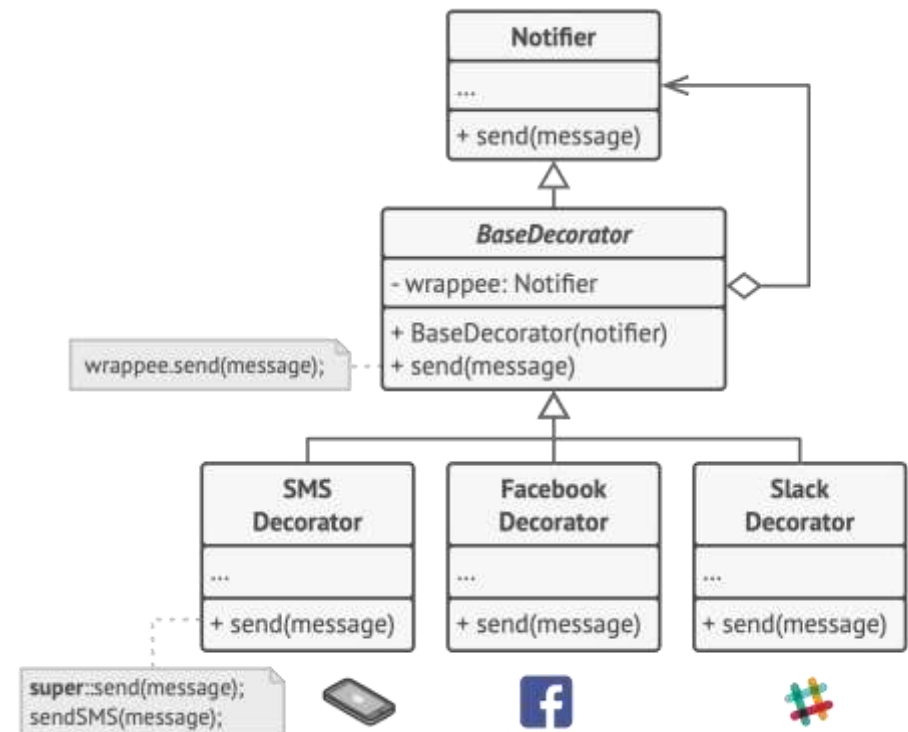
Structural patterns: Decorator (continued)

Solution

To overcome these caveats is by using Aggregation or Composition instead of Inheritance. One object has a reference to another and delegates it some work, whereas with inheritance, the object itself is able to do that work, inheriting the behavior from its superclass.

We substitute the linked “helper” object with another, changing the behavior of the container at runtime. An object can use the behavior of various classes, having references to multiple objects and delegating them all kinds of work. Aggregation/composition is the key principle behind many design patterns, including Decorator.

A *wrapper* is an object that can be linked with some *target* object. The wrapper contains the same set of methods as the target and delegates to it all requests it receives. However, the wrapper may alter the result by doing something either before or after it passes the request to the target.



Structural patterns: Decorator (continued)

Structure

- The last decorator in the stack would be the object that the client works with. Since all decorators implement the same interface as the base notifier, the rest of the client code won't care whether it works with the "pure" notifier object or the decorated one.
- We could apply the same approach to other behaviours such as formatting messages or composing the recipient list. The client can decorate the object with any custom decorators, as long as they follow the same interface as the others.

Real-life

Wearing clothes is an example of using decorators. When you're cold, you wrap yourself in a sweater. If you're still cold with a sweater, you can wear a jacket on top. If it's raining, you can put on a raincoat. All of these garments "extend" your basic behavior but aren't part of you, and you can easily take off any piece of clothing whenever you don't need it.

```
stack = new Notifier()
if (facebookEnabled)
    stack = new FacebookDecorator(stack)
if (slackEnabled)
    stack = new SlackDecorator(stack)

app.setNotifier(stack)
```

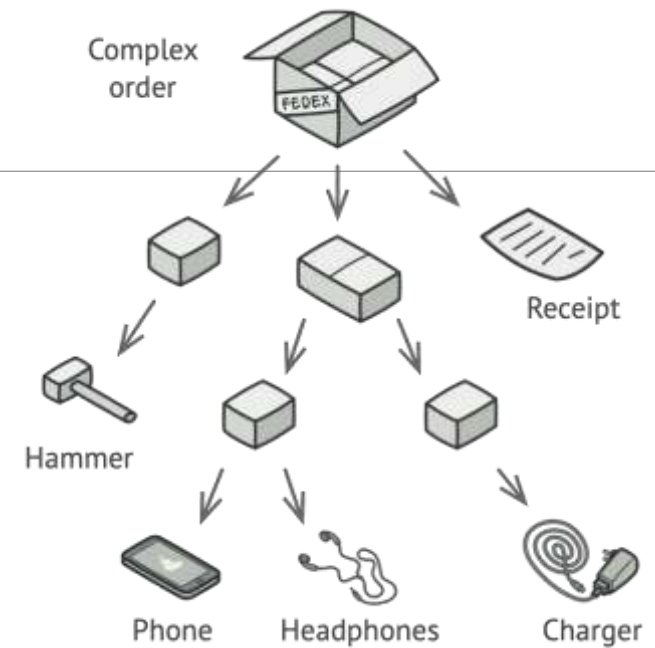


```
notifier.send("Alert!")
// Email → Facebook → Slack
```



Structural patterns: Composite

Composite is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.



Problem

- Using the Composite pattern makes sense only when the core model of your app can be represented as a tree.
- For example, imagine that you have two types of objects: Products and Boxes. A Box can contain several Products as well as a number of smaller Boxes. These little Boxes can also hold some Products or even smaller Boxes, and so on.
- Say you decide to create an ordering system that uses these classes. Orders could contain simple products without any wrapping, as well as boxes stuffed with products...and other boxes. How would you determine the total price of such an order?

Complication

- Direct approach: unwrap all the boxes, go over all the products and then calculate the total. That would be doable in the real world; but in a program, it's not as simple as running a loop. You have to know the classes of Products and Boxes you're going through, the nesting level of the boxes and other nasty details beforehand.

Structural patterns: Composite (continued)

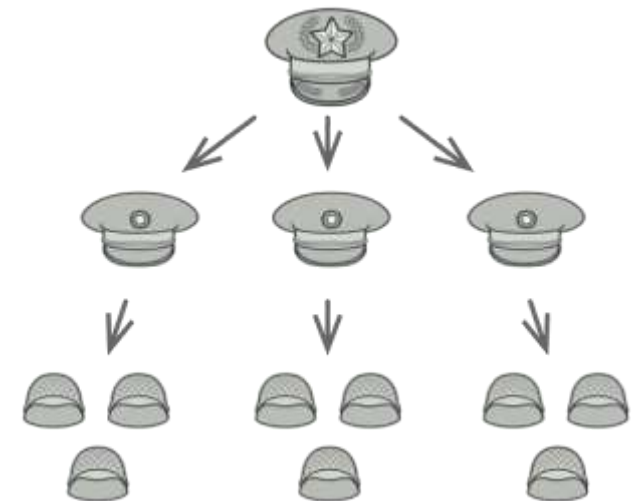
Solution

The Composite pattern suggests that you work with Products and Boxes through a common interface which declares a method for calculating the total price.

Approach:

- For a product, it'd simply return the product's price.
- For a box, it'd go over each item the box contains, ask its price and then return a total for this box. If one of these items were a smaller box, that box would also start going over its contents and so on, until the prices of all inner components were calculated.
- A box could even add some extra cost to the final price, such as packaging cost.

The greatest benefit of this approach is that you don't need to care about the concrete classes of objects that compose the tree. You don't need to know whether an object is a simple product or a sophisticated box. You can treat them all the same via the common interface. When you call a method, the objects themselves pass the request down the tree.



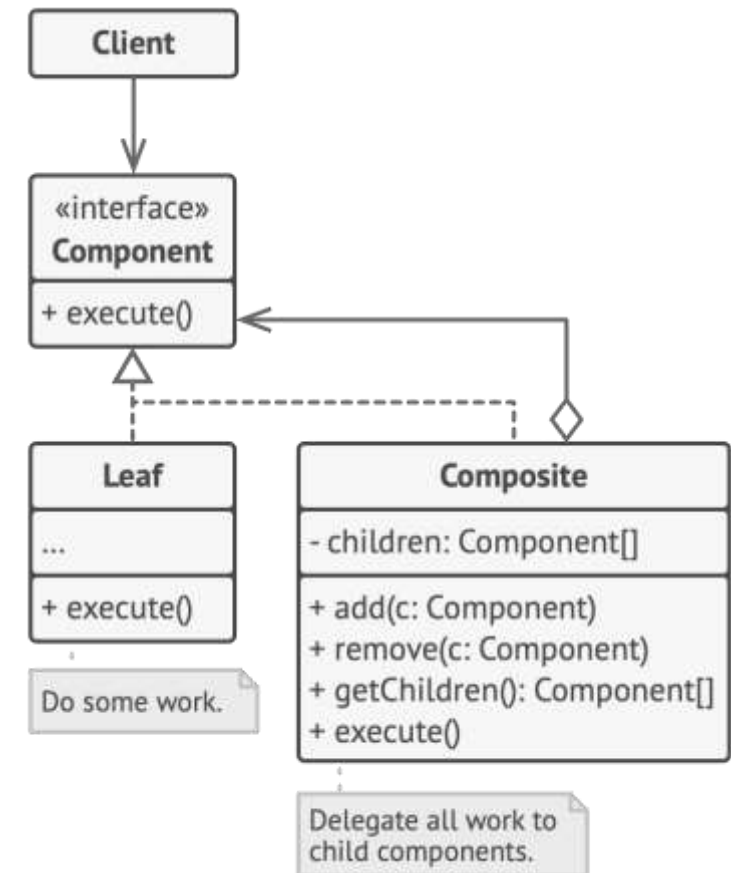
Real world alternative – army units:

Structural patterns: Composite (continued)

Structure

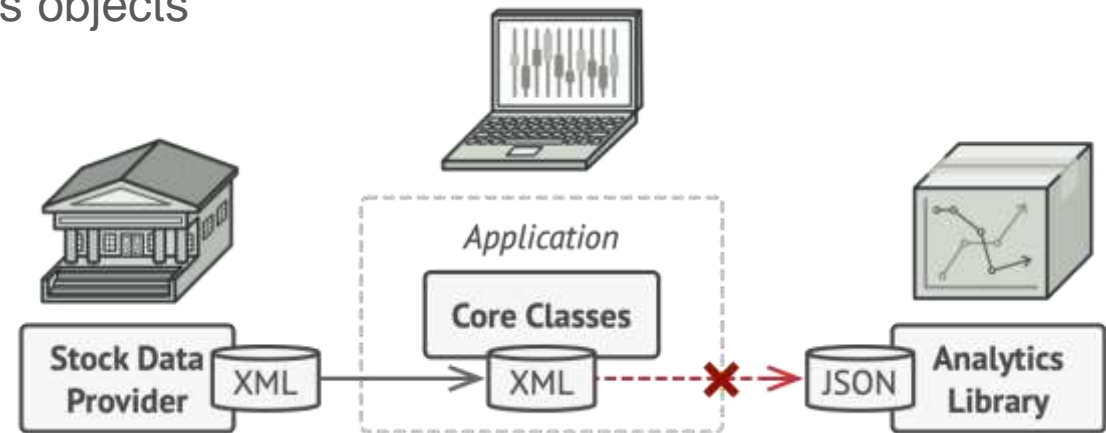
- The **Component** interface describes operations that are common to both simple and complex elements of the tree.
- The **Leaf** is a basic element of a tree that doesn't have sub-elements. Usually, leaf components end up doing most of the real work, since they don't have anyone to delegate the work to.
- The **Client** works with all elements through the component interface. As a result, the client can work in the same way with both simple or complex elements of the tree.
- The **Container** (aka *composite*) is an element that has sub-elements: leaves or other containers. A container doesn't know the concrete classes of its children. It works with all sub-elements only via the component interface.

Upon receiving a request, a container delegates the work to its sub-elements, processes intermediate results and then returns the final result to the client.



Structural patterns: Adapter

Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate.



Problem

- Creating a stock market monitoring app. The app downloads the stock data from multiple sources in XML format and then displays nice-looking charts and diagrams for the user.
- At some point, you decide to improve the app by integrating a smart 3rd-party analytics library. But there's a catch: the analytics library only works with data in JSON format.

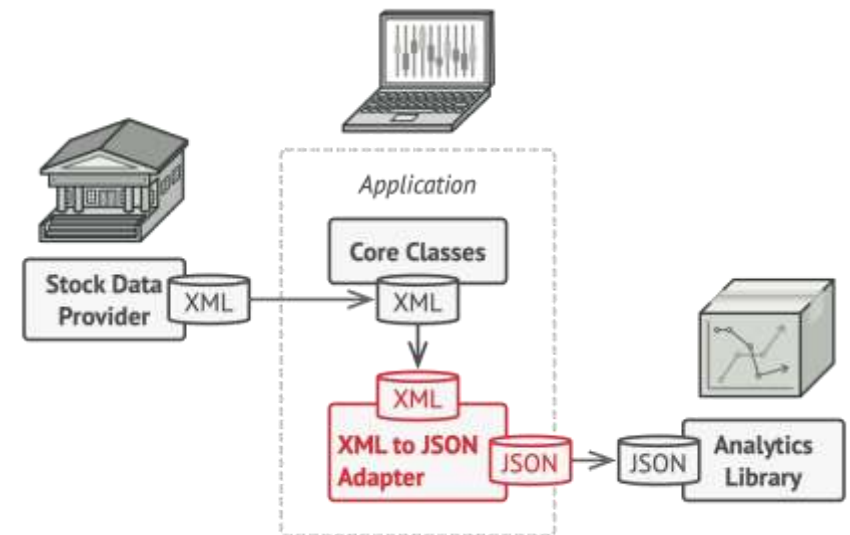
Typical complications:

- You could change the library to work with XML. However, this might break some existing code that relies on the library. And worse, you might not have access to the library's source code in the first place, making this approach impossible.

Structural patterns: Adapter (continued)

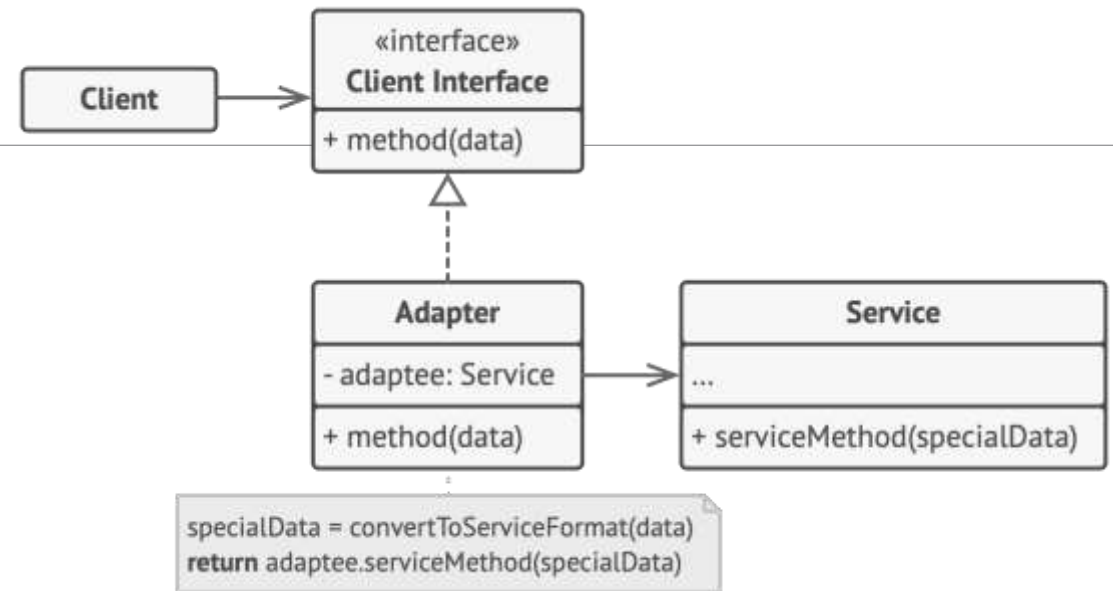
Solution

- You can create an *adapter*. This is a special object that converts the interface of one object so that another object can understand it.
- An adapter wraps one of the objects to hide the complexity of conversion happening behind the scenes. The wrapped object isn't even aware of the adapter. For example, you can wrap an object that operates in meters and kilometers with an adapter that converts all of the data to imperial units such as feet and miles.
- Adapters can not only convert data into various formats but can also help objects with different interfaces collaborate. Here's how it works:
 1. The adapter gets an interface, compatible with one of the existing objects.
 2. Using this interface, the existing object can safely call the adapter's methods.
 3. Upon receiving a call, the adapter passes the request to the second object, but in a format and order that the second object expects.
- Sometimes it's even possible to create a two-way adapter that can convert the calls in both directions.



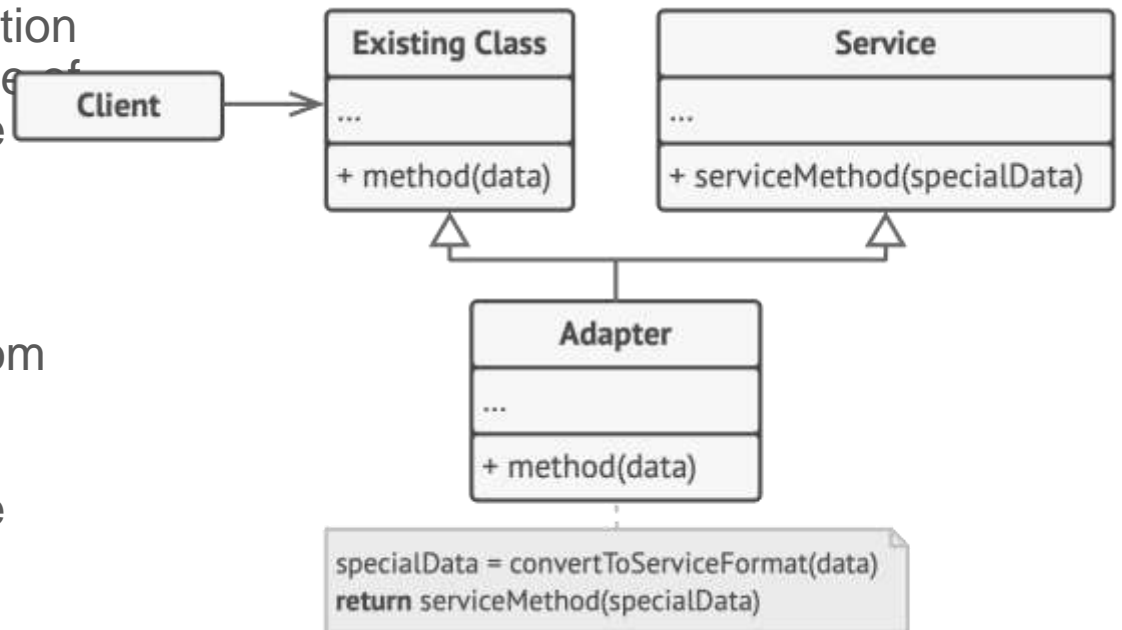
Structural patterns: Adapter (continued)

A *wrapper* is an object that can be linked with some *target* object. The wrapper contains the same set of methods as the target and delegates to it all requests it receives. However, the wrapper may alter the result by doing something either before or after it passes the request to the target.



Structure

- This implementation uses the object composition principle: the adapter implements the interface of one object and wraps the other one. It can be implemented in all popular programming languages.
- Note: Another implementation can use inheritance: the adapter inherits interfaces from both objects at the same time. Note that this approach can only be implemented in programming languages that support multiple inheritance, such as C++.

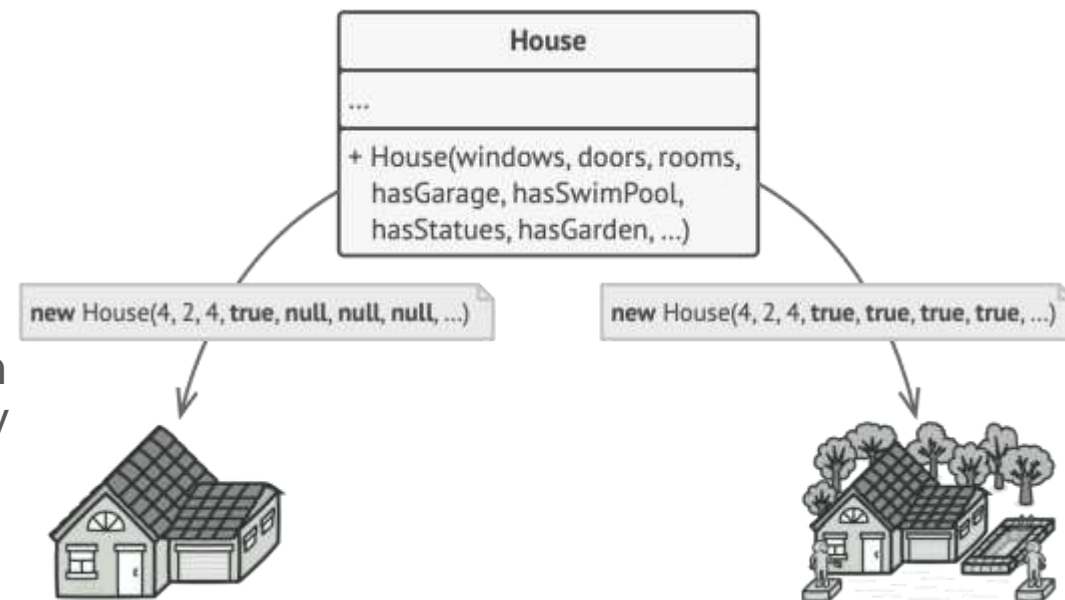
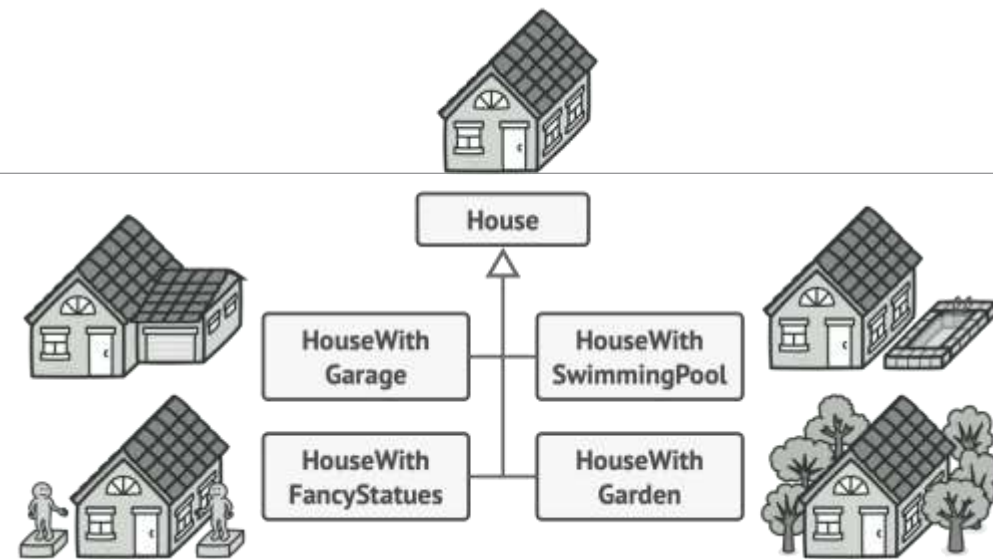


Creational patterns: Builder

Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows to produce different types and representations of an object using the same construction code.

Problem

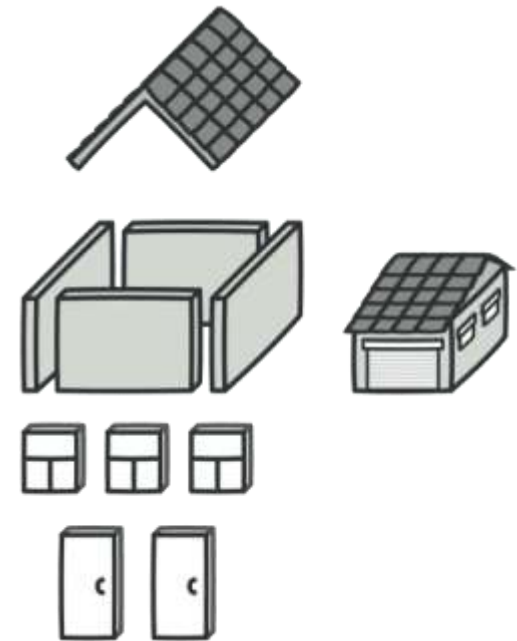
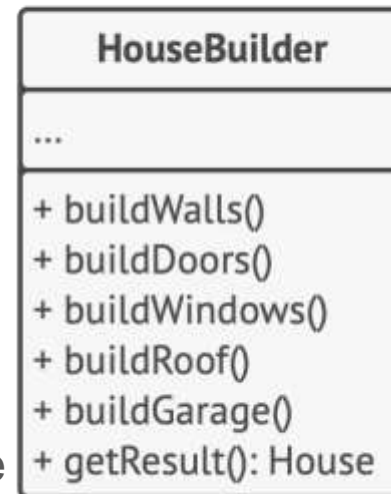
- A complex object that requires laborious, step-by-step initialization of many fields and nested objects.
- Such initialization code is usually buried inside a monstrous constructor with lots of parameters.
- The simplest solution is to extend the base House class and create a set of subclasses to cover all combinations of the parameters. But eventually you'll end up with a considerable number of subclasses. Any new parameter, such as a roof style, will require growing this hierarchy even more.



Creational patterns: Builder (continued)

Solution

- The Builder pattern suggests that you extract the object construction code out of its own class and move it to separate objects called *builders*.
- You don't need to call all of the steps. You can call only those steps that are necessary for producing a particular configuration of an object.
- Some of the construction steps might require different implementation when you need to build various representations of the product. For example, walls of a cabin may be built of wood, but the castle walls must be built with stone.
- In this case, you can create several different builder classes that implement the same set of building steps, but in a different manner. Then you can use these builders in the construction process (i.e., an ordered set of calls to the building steps) to produce different kinds of objects.
- You can go further and extract a series of calls to the builder steps you use to construct a product into a separate class called **director**. The director class defines the order in which to execute the building steps, while the builder provides the implementation for those steps.



Creational patterns: Builder (continued)

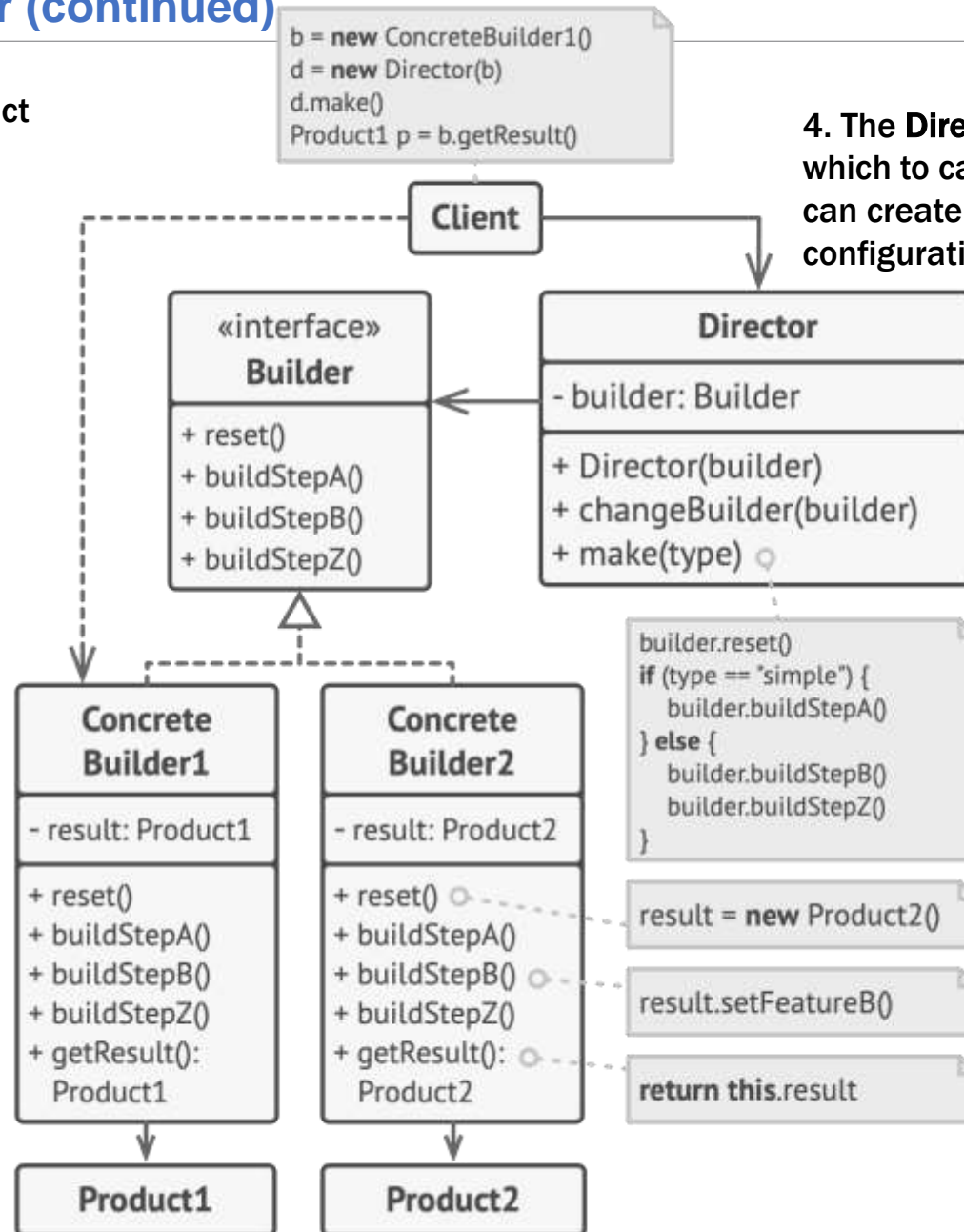
1. The **Builder** interface declares product construction steps that are common to all types of builders.

2. **Concrete Builders** provide different implementations of the construction steps. Concrete builders may produce products that don't follow the common interface.

3. **Products** are resulting objects. Products constructed by different builders don't have to belong to the same class hierarchy or interface.

4. The **Director** class defines the order in which to call construction steps, so you can create and reuse specific configurations of products.

5. The **Client** must associate one of the builder objects with the director. Usually, it's done just once, via parameters of the director's constructor. Then the director uses that builder object for all further construction. However, there's an alternative approach for when the client passes the builder object to the production method of the director. In this case, you can use a different builder each time you produce something with the director.



Creational patterns: Prototype

Prototype is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.

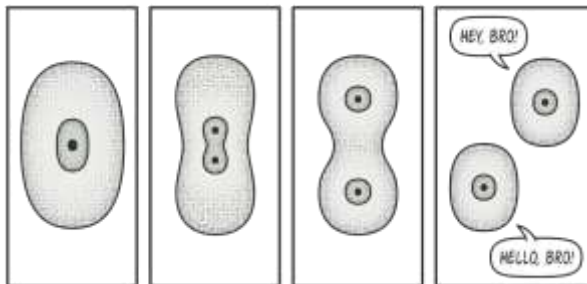
Problem

- You have an object, and you want to create an exact copy of it. How would you do it? First, you have to create a new object of the same class. Then you have to go through all the fields of the original object and copy their values over to the new object.
- BUT - not all objects can be copied that way because some of the object's fields may be private and not visible from outside of the object itself.
- There's one more problem with the direct approach. Since you have to know the object's class to create a duplicate, your code becomes dependent on that class. If the extra dependency doesn't scare you, there's another catch. Sometimes you only know the interface that the object follows, but not its concrete class, when, for example, a parameter in a method accepts any objects that follow some interface.



Creational patterns: Prototype (continued)

- The Prototype pattern delegates the cloning process to the actual objects that are being cloned. The pattern declares a common interface for all objects that support cloning. This interface lets you clone an object without coupling your code to the class of that object. Usually, such an interface contains just a single clone method.
- The implementation of the clone method is very similar in all classes. The method creates an object of the current class and carries over all of the field values of the old object into the new one. You can even copy private fields because most programming languages let objects access private fields of other objects that belong to the same class.
- An object that supports cloning is called a prototype. When your objects have dozens of fields and hundreds of possible configurations, cloning them might serve as an alternative to subclassing.

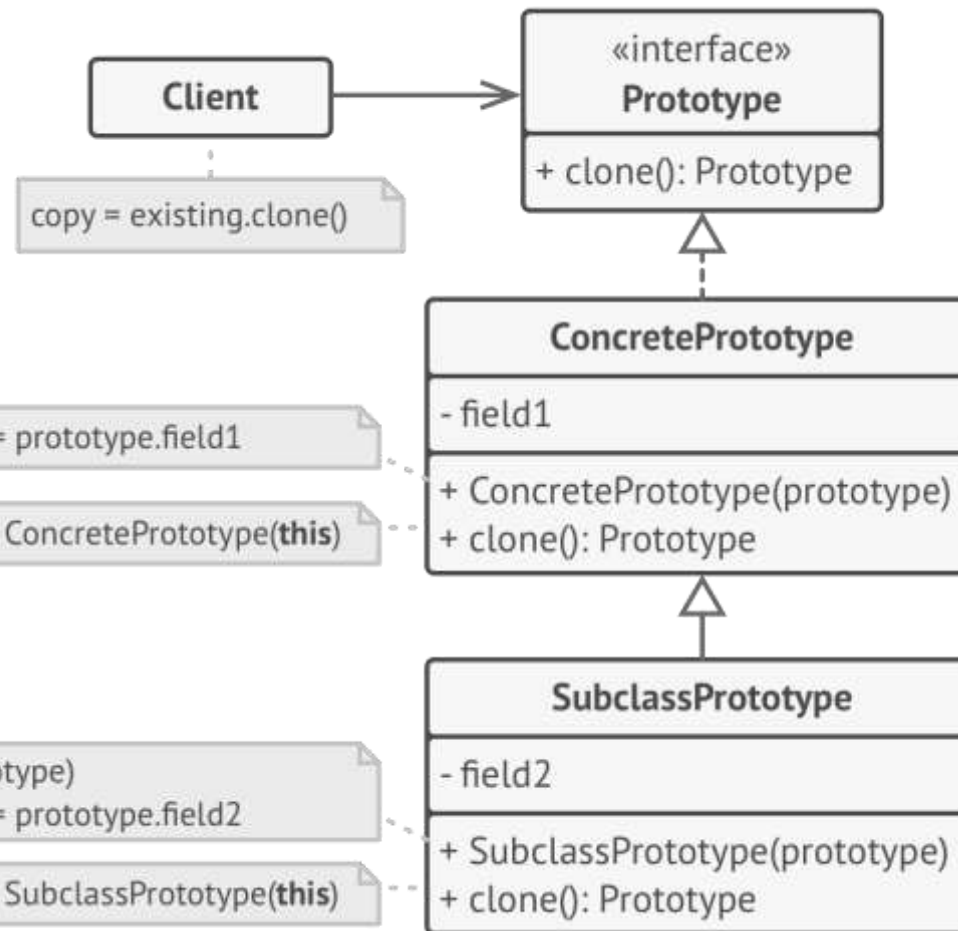


Real world analogy to the pattern is the process of mitotic cell division (biology, remember?). After mitotic division, a pair of identical cells is formed. The original cell acts as a prototype and takes an active role in creating the copy.

Creational patterns: Prototype (continued)

Use the Prototype pattern when your code shouldn't depend on the concrete classes of objects that you need to copy.

1. The Prototype interface declares the cloning methods. In most cases, it's a single clone method.

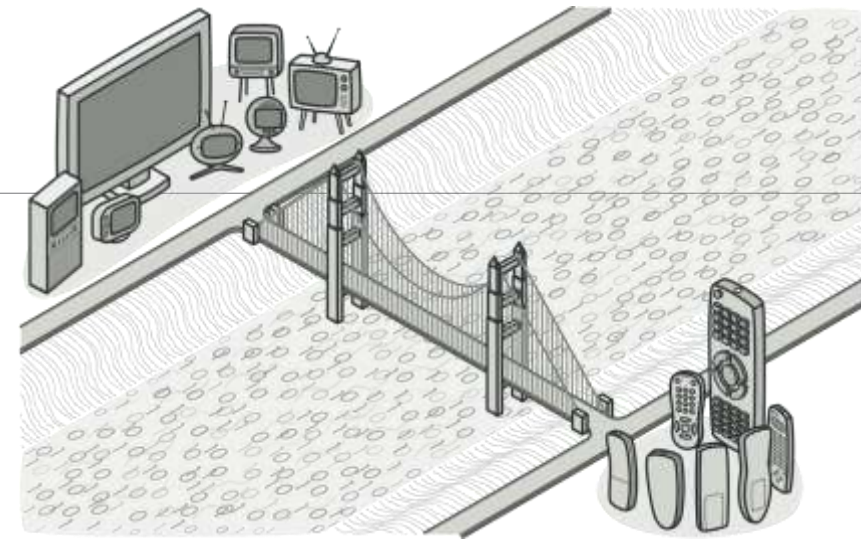


2. The **Client** can produce a copy of any object that follows the prototype interface.

2. The **Concrete Prototype** class implements the cloning method. In addition to copying the original object's data to the clone, this method may also handle some edge cases of the cloning process related to cloning linked objects, untangling recursive dependencies, etc.

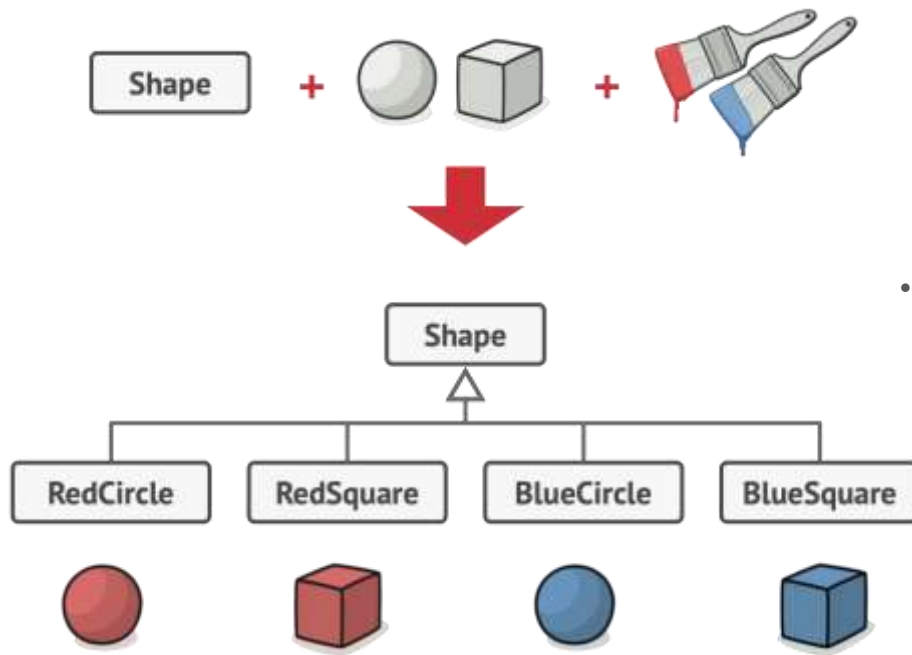
Structural patterns: Bridge

Bridge is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.



Problem

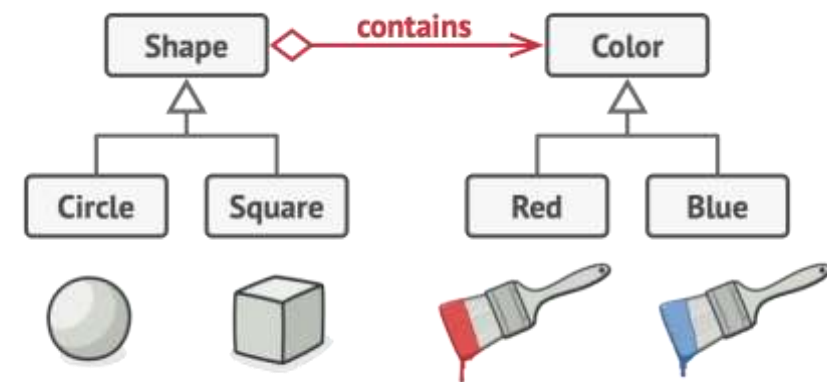
- Say you have a geometric Shape class with a pair of subclasses: Circle and Square. You want to extend this class hierarchy to incorporate colors, so you plan to create Red and Blue shape subclasses. However, since you already have two subclasses, you'll need to create four class combinations such as BlueCircle and RedSquare.
- **Adding new shape types and colors to the hierarchy will grow it exponentially.** For example, to add a triangle shape you'd need to introduce two subclasses, one for each color. And after that, adding a new color would require creating three subclasses, one for each shape type. The further we go, the worse it becomes.



Structural patterns: Bridge (continued)

Solution

- This problem occurs because we're trying to extend the shape classes in two independent dimensions: by form and by color. That's a very common issue with class inheritance.
- The Bridge pattern attempts to solve this problem by switching from inheritance to the object composition. What this means is that you extract one of the dimensions into a separate class hierarchy, so that the original classes will reference an object of the new hierarchy, instead of having all of its state and behaviors within one class.
- Following this approach, we can extract the color-related code into its own class with two subclasses: Red and Blue. The Shape class then gets a reference field pointing to one of the color objects. Now the shape can delegate any color-related work to the linked color object. That reference will act as a bridge between the Shape and Color classes. From now on, adding new colors won't require changing the shape hierarchy, and vice versa.



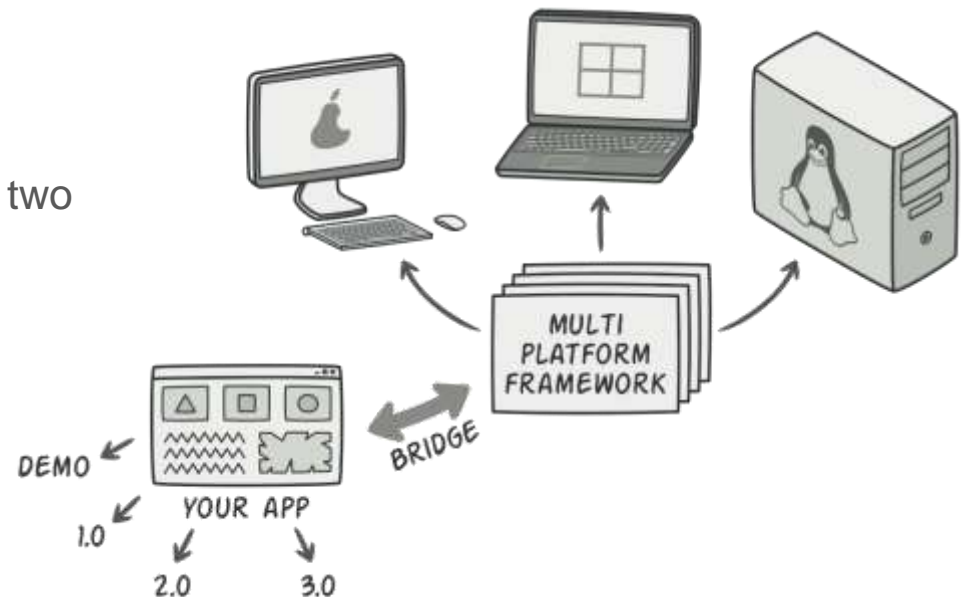
Structural patterns: Bridge (continued)

Abstraction and *Implementation* are alternative names of the Bridge definition.

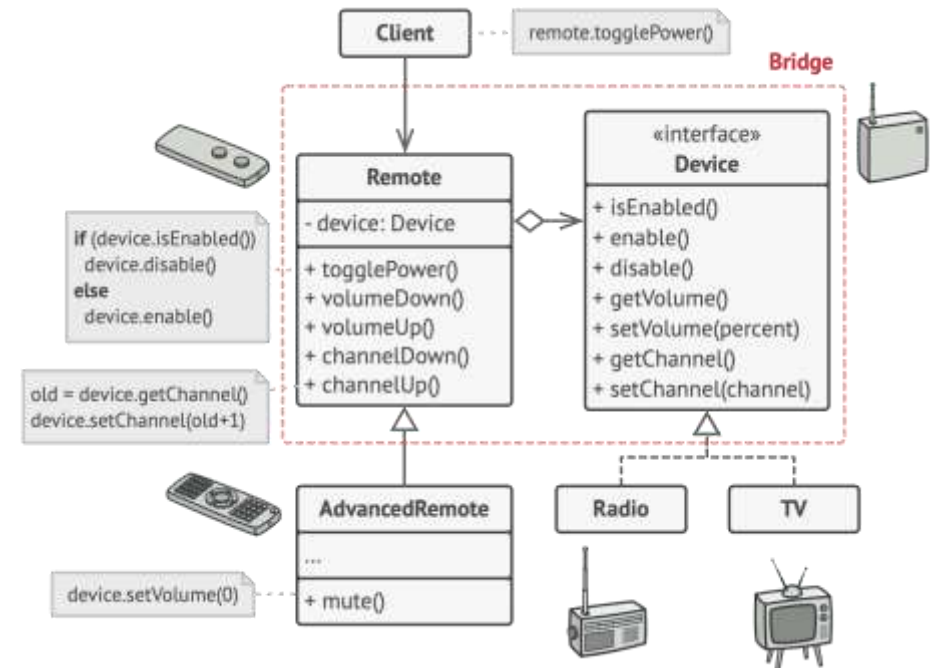
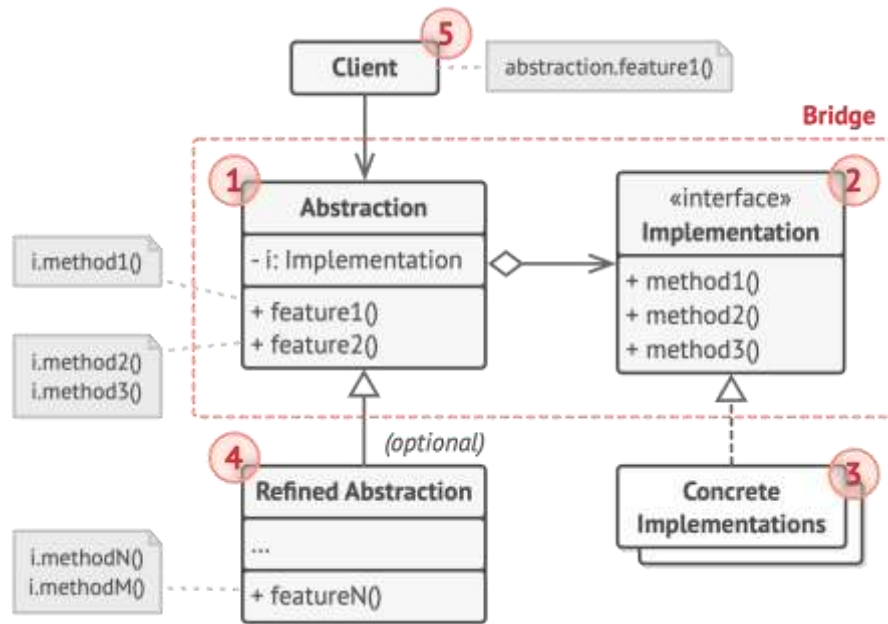
In real applications, the abstraction can be represented by a graphical user interface (GUI), and the implementation could be the underlying operating system code (API) which the GUI layer calls in response to user interactions.

- Generally speaking, you can extend such an app in two independent directions:
- Have several different GUIs (for instance, tailored for regular customers or admins).
- Support several different APIs (for example, to be able to launch the app under Windows, Linux, and macOS).
- In a worst-case scenario, this app might look like a giant spaghetti bowl, where hundreds of conditionals connect different types of GUI with various APIs all over the code.

- Bridge pattern suggests that we divide the classes into two hierarchies:
- Abstraction: the GUI layer of the app.
- Implementation: the operating systems' APIs.



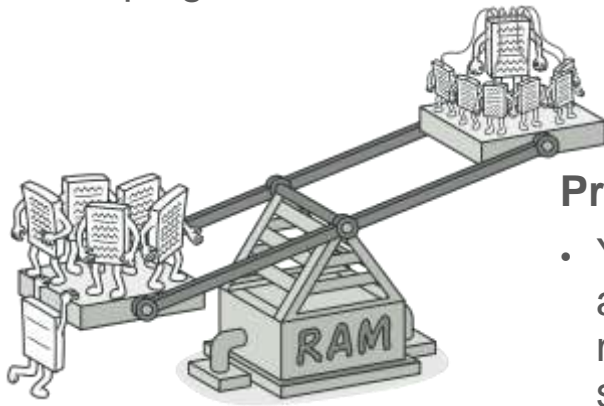
Structural patterns: Bridge (continued)



1. The **Abstraction** provides high-level control logic. It relies on the implementation object to do the actual low-level work.
2. The **Implementation** declares the interface that's common for all concrete implementations. An abstraction can only communicate with an implementation object via methods that are declared here.
3. The abstraction may list the same methods as the implementation, but usually the abstraction declares some complex behaviors that rely on a wide variety of primitive operations declared by the implementation.
4. **Concrete Implementations** contain platform-specific code.
5. **Refined Abstractions** provide variants of control logic. Like their parent, they work with different implementations via the general implementation interface.
6. Usually, the **Client** is only interested in working with the abstraction. However, it's the client's job to link the abstraction object with one of the implementation objects.

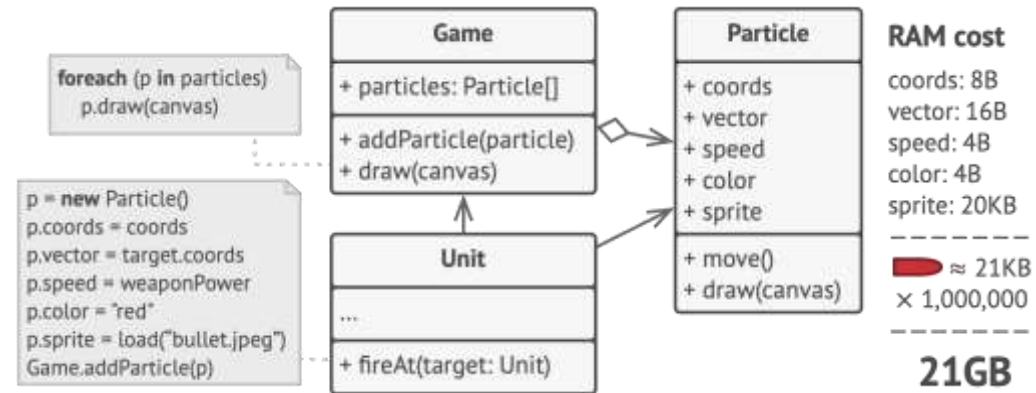
Structural patterns: Flyweight

Flyweight is a structural design pattern that lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.



Problem

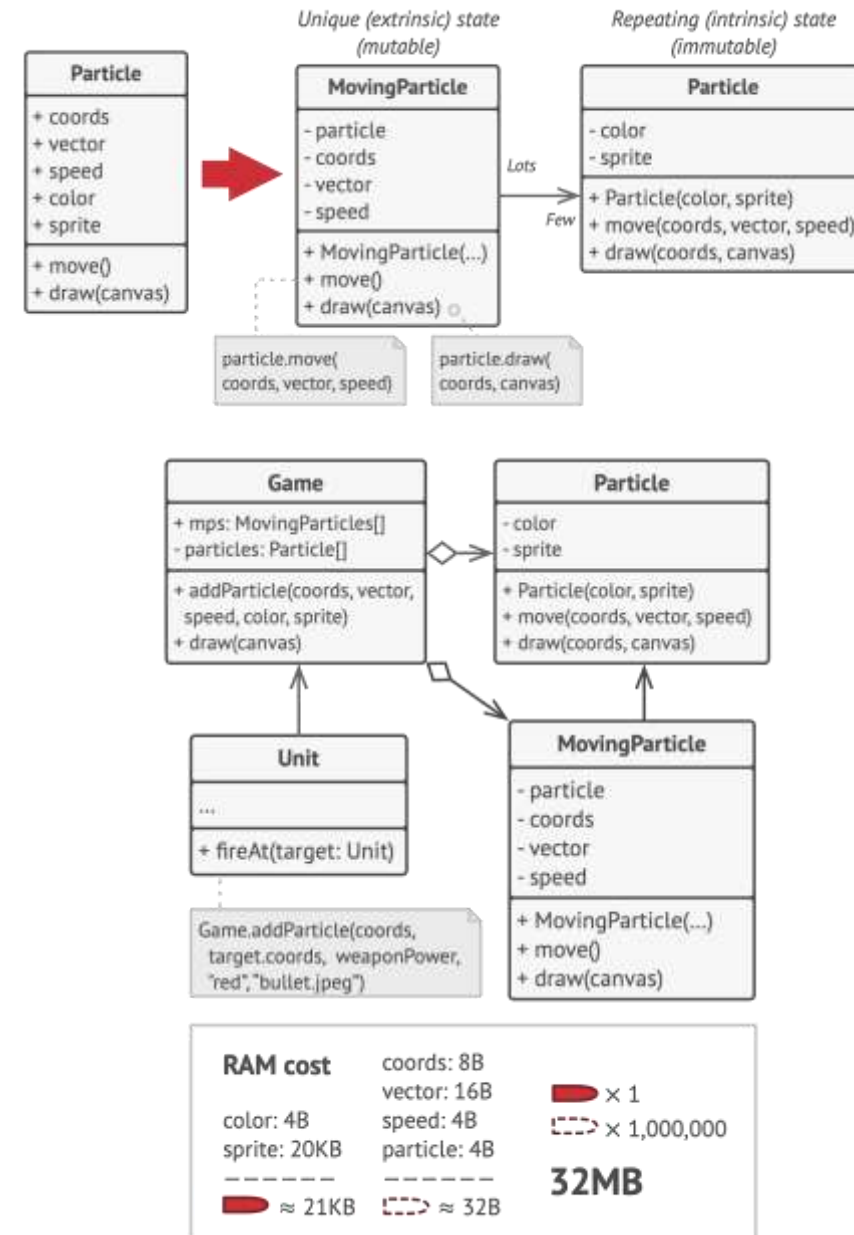
- You decided to create a simple video game: players would be moving around a map and shooting each other. You chose to implement a realistic particle system and make it a distinctive feature of the game. Vast quantities of bullets, missiles, and shrapnel from explosions should fly all over the map and deliver a thrilling experience to the player.
- Upon its completion, you built the game and sent it to your friend for a test drive. Although the game was running flawlessly on your machine, your friend wasn't able to play for long. On his computer, the game kept crashing after a few minutes of gameplay. After spending several hours digging through debug logs, you discovered that the game crashed because of an insufficient amount of RAM. It turned out that your friend's rig was much less powerful than your own computer, and that's why the problem emerged so quickly on his machine.
- The actual problem was related to your particle system. Each particle, such as a bullet, a missile or a piece of shrapnel was represented by a separate object containing plenty of data. At some point, when the carnage on a player's screen reached its climax, newly created particles no longer fit into the remaining RAM, so the program crashed.



Structural patterns: Flyweight (continued)

Solution

- On closer inspection of the Particle class, you may notice that the color and sprite fields consume a lot more memory than other fields. What's worse is that these two fields store almost identical data across all particles. For example, all bullets have the same color and sprite.
- Other parts of a particle's state, such as coordinates, movement vector and speed, are unique to each particle. After all, the values of these fields change over time. This data represents the always changing context in which the particle exists, while the color and sprite remain constant for each particle.
- This constant data of an object is usually called the *intrinsic state*. It lives within the object; other objects can only read it, not change it. The rest of the object's state, often altered "from the outside" by other objects, is called the *extrinsic state*.
- The Flyweight pattern suggests that you stop storing the extrinsic state inside the object. Instead, you should pass this state to specific methods which rely on it. Only the intrinsic state stays within the object, letting you reuse it in different contexts. As a result, you'd need fewer of these objects since they only differ in the intrinsic state, which has much fewer variations than the extrinsic.

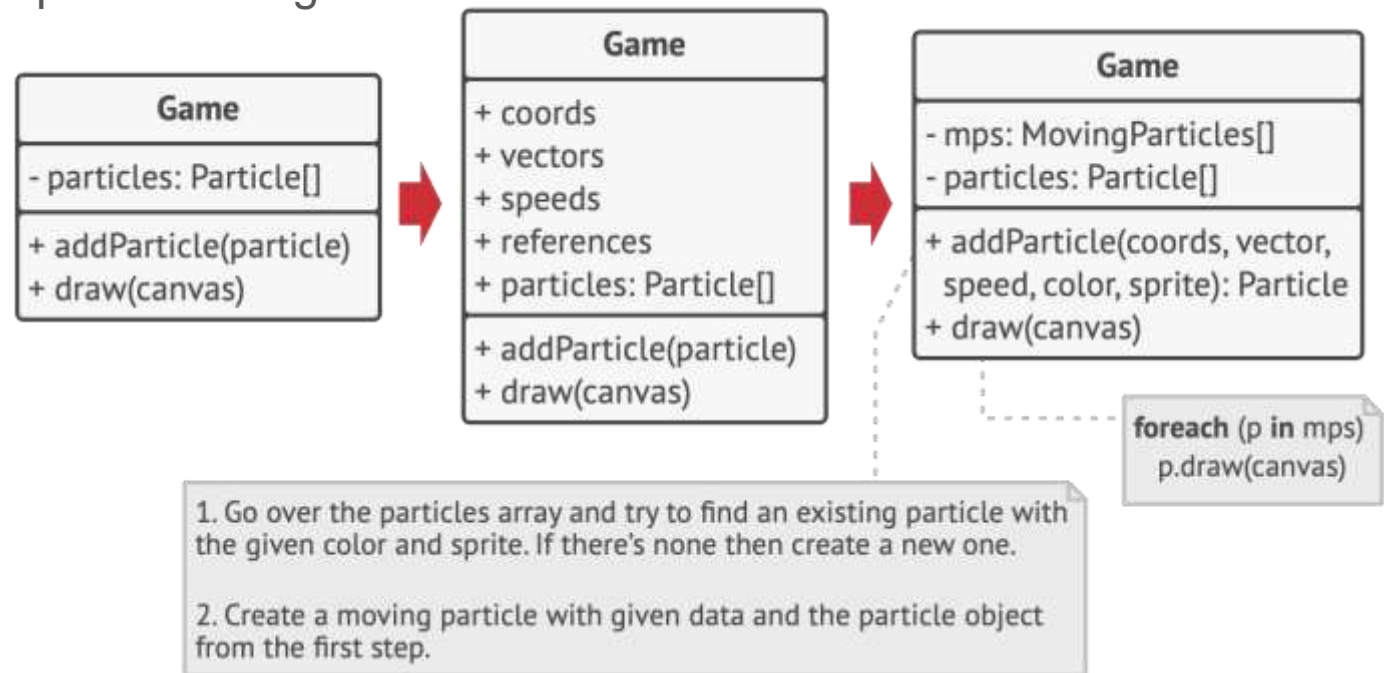


Structural patterns: Flyweight (continued)

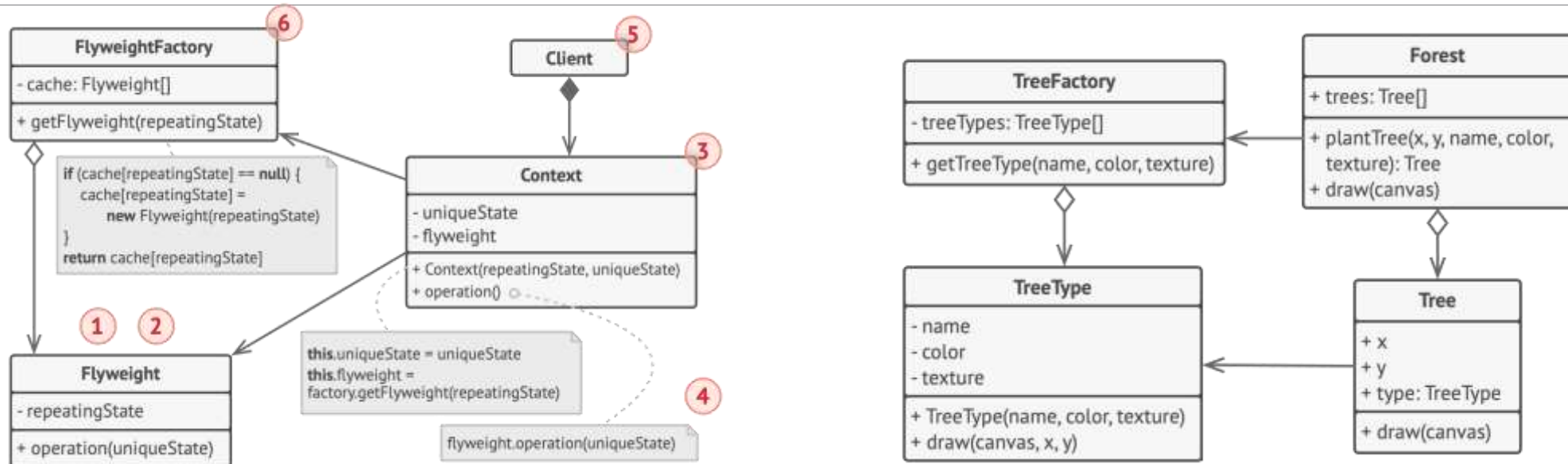
Where does the extrinsic state move to? Some class should still store it, right? In most cases, it gets moved to the container object, which aggregates objects before we apply the pattern.

In our case, that's the main Game object that stores all particles in the particles field. To move the extrinsic state into this class, you need to create several array fields for storing coordinates, vectors, and speed of each individual particle. But that's not all. You need another array for storing references to a specific flyweight that represents a particle. These arrays must be in sync so that you can access all data of a particle using the same index.

- A more elegant solution is to create a separate context class that would store the extrinsic state along with reference to the flyweight object. This approach would require having just a single array in the container class.



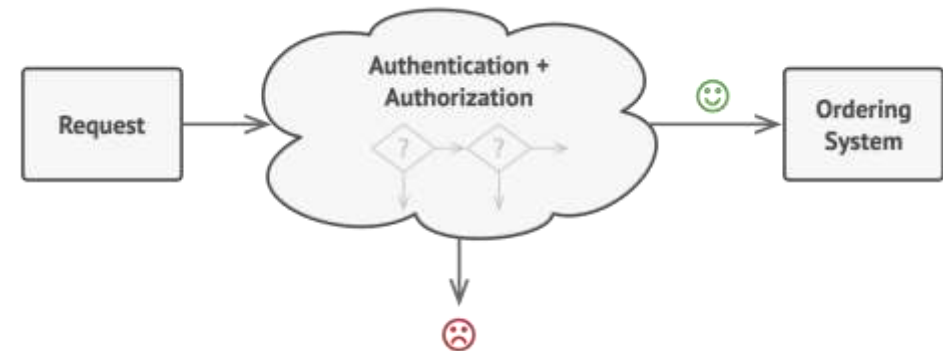
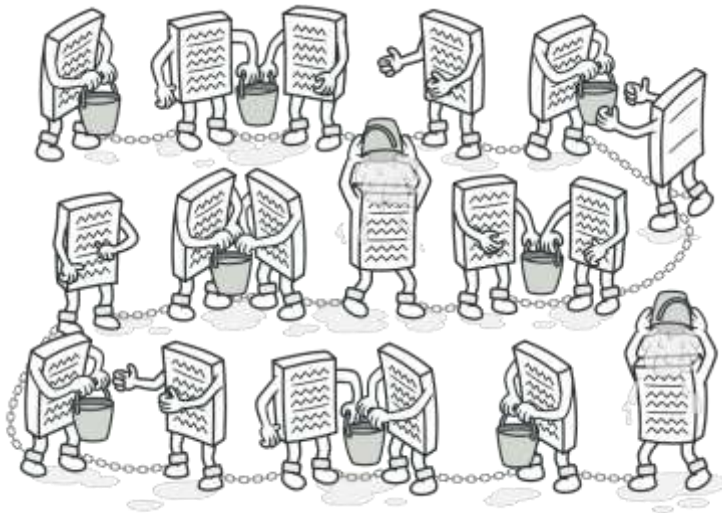
Structural patterns: Flyweight (continued)



1. The Flyweight pattern is merely an optimization.
2. The **Flyweight** class contains the portion of the original object's state that can be shared between multiple objects. The same flyweight object can be used in many different contexts. The state stored inside a flyweight is called *intrinsic*. The state passed to the flyweight's methods is called *extrinsic*.
3. The **Context** class contains the extrinsic state, unique across all original objects. When a context is paired with one of the flyweight objects, it represents the full state of the original object.
4. Usually, the behavior of the original object remains in the flyweight class. In this case, whoever calls a flyweight's method must also pass appropriate bits of the extrinsic state into the method's parameters. On the other hand, the behavior can be moved to the context class, which would use the linked flyweight merely as a data object.
5. The **Client** calculates or stores the extrinsic state of flyweights. From the client's perspective, a flyweight is a template object which can be configured at runtime by passing some contextual data into parameters of its methods.
6. The **Flyweight Factory** manages a pool of existing flyweights. With the factory, clients don't create flyweights directly. Instead, they call the factory, passing it bits of the intrinsic state of the desired flyweight. The factory looks over previously created flyweights and either returns an existing one that matches search criteria or creates a new one if nothing is found.

Behavioral patterns: Chain on command / responsibility

Chain of Responsibility is a behavioral design pattern that lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.



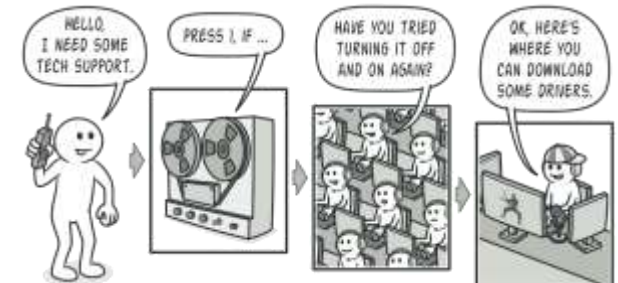
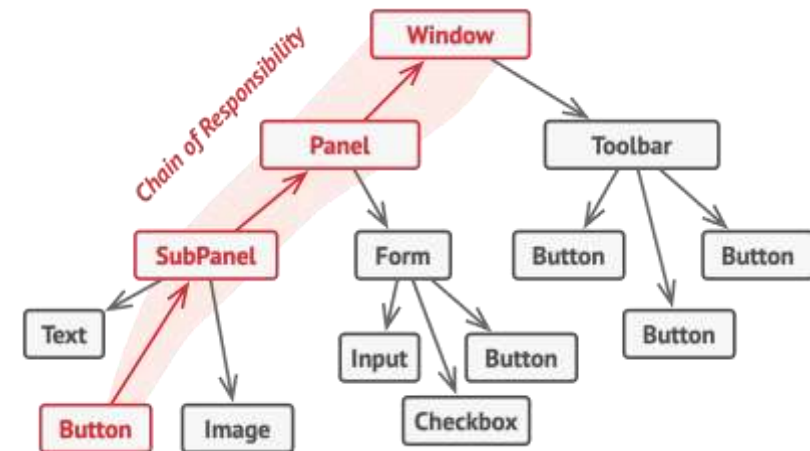
Problem

- Example of an online ordering system. You want to restrict access to the system so only authenticated users can create orders. Also, users who have administrative permissions must have full access to all orders.
- These checks must be performed sequentially. The application can attempt to authenticate a user to the system whenever it receives a request that contains the user's credentials. However, if those credentials aren't correct and authentication fails, there's no reason to proceed with any other checks.
- The code of the checks, which had already looked like a mess, became more and more bloated as you added each new feature. Changing one check sometimes affected the others. When you tried to reuse the checks to protect other components of the system, you had to duplicate some of the code since those components required some of the checks, but not all of them.
- The system became very hard to comprehend and expensive to maintain. You struggled with the code for a while, until one day you decided to refactor the whole thing.

Behavioral patterns: Chain on command (continued)

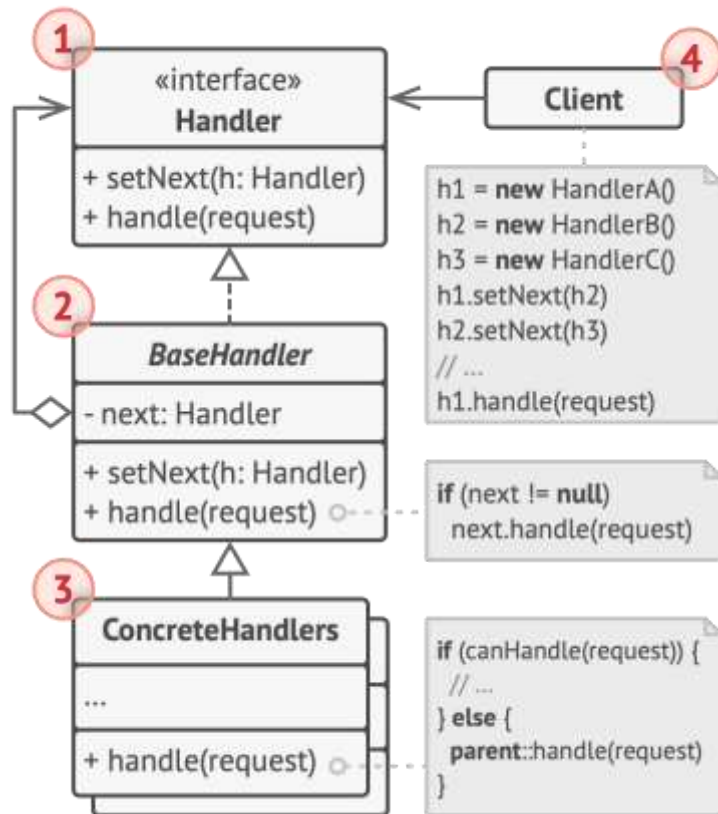
Solution

- Like many other behavioral design patterns, the **Chain of Responsibility** relies on transforming particular behaviors into stand-alone objects called *handlers*. In our case, each check should be extracted to its own class with a single method that performs the check. The request, along with its data, is passed to this method as an argument.
- The pattern suggests that you link these handlers into a chain. Each linked handler has a field for storing a reference to the next handler in the chain. In addition to processing a request, handlers pass the request further along the chain. The request travels along the chain until all handlers have had a chance to process it.
- Here's the best part: a handler can decide not to pass the request further down the chain and effectively stop any further processing.
- In our example with ordering systems, a handler performs the processing and then decides whether to pass the request further down the chain. Assuming the request contains the right data, all the handlers can execute their primary behavior, whether it's authentication checks or caching.



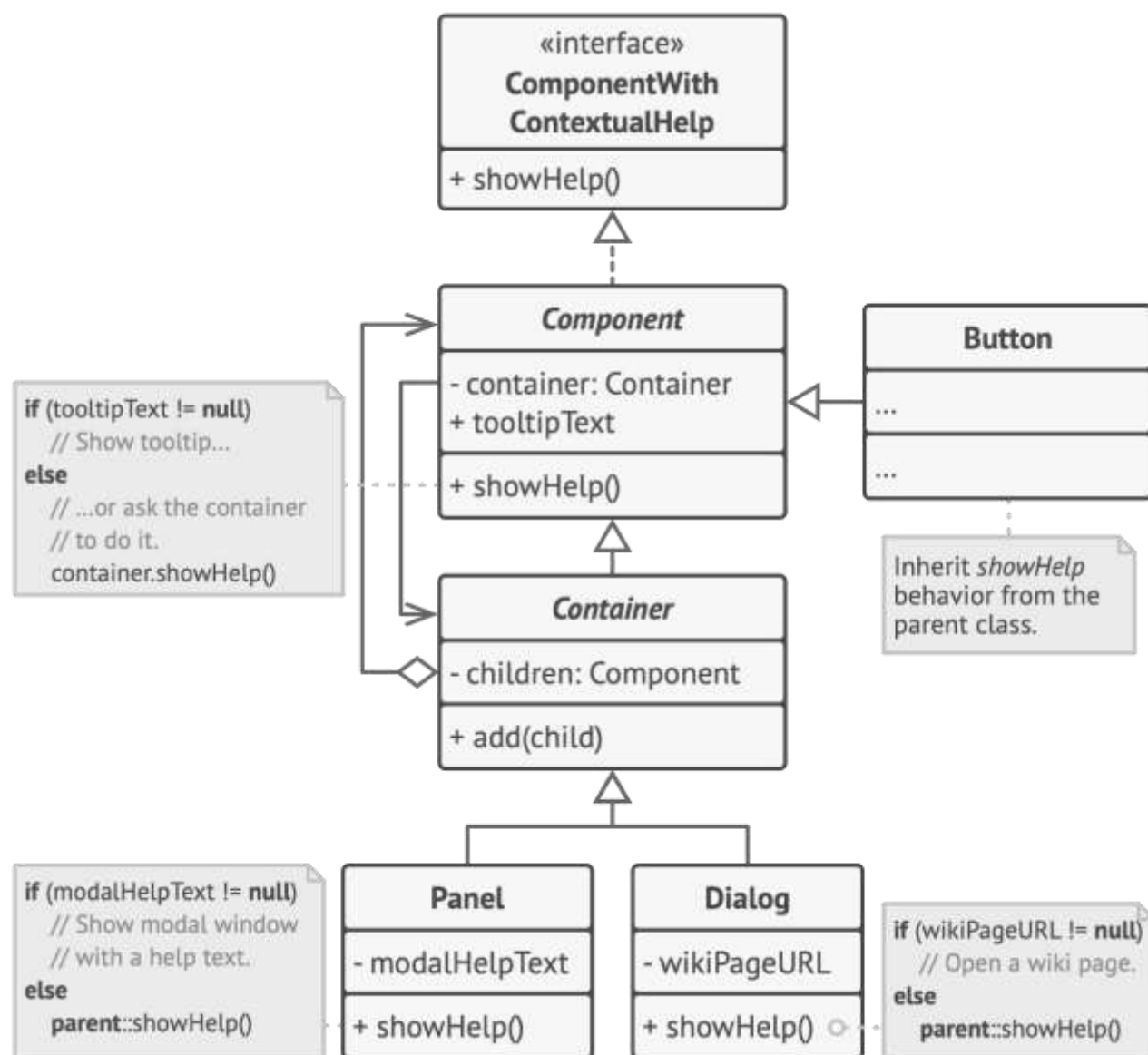
Real world analogy

Behavioral patterns: Chain on command (continued)



1. The **Handler** declares the interface, common for all concrete handlers. It usually contains just a single method for handling requests, but sometimes it may also have another method for setting the next handler on the chain.
2. The **Base Handler** is an optional class where you can put the boilerplate code that's common to all handler classes.
3. Usually, this class defines a field for storing a reference to the next handler. The clients can build a chain by passing a handler to the constructor or setter of the previous handler. The class may also implement the default handling behavior: it can pass execution to the next handler after checking for its existence.
4. **Concrete Handlers** contain the actual code for processing requests. Upon receiving a request, each handler must decide whether to process it and, additionally, whether to pass it along the chain.
5. Handlers are usually self-contained and immutable, accepting all necessary data just once via the constructor.
6. The **Client** may compose chains just once or compose them dynamically, depending on the application's logic. Note that a request can be sent to any handler in the chain—it doesn't have to be the first one.

Behavioral patterns: Chain on command (continued)



In this example, the Chain of Responsibility pattern is responsible for displaying contextual help information for active GUI elements.

Sources & other reading

- **Design Patterns: Elements of Reusable Object-Oriented Software** by [Erich Gamma](#) (Author), [Richard Helm](#) (Author), [Ralph Johnson](#) (Author), [John Vlissides](#) (Author)
- Design Patterns [Design Patterns in Java \(refactoring.guru\)](#)
- [CSE 331 23wi \(washington.edu\)](#)

Contact:

RNDr. Július Šiška, PhD.

E: julius.siska at gmail.com