Object Oriented Programming and Design in Java

Session 25

Instructor: Bert Huang

Announcements

- Homework 5 due
- Final sample problems posted
- Mon. May 10th, Final exam. 9 AM - noon
  - closed-book/notes, focus on post-midterm material, but material is inherently cumulative

Today's Plan

- Broad overview of topics on the exam
  - Key ideas from each topic area
- With remaining time, flip through tons of highlight slides from previous classes

Exam Material

- You are responsible for all material we covered in class
- Don't memorize minute details of Java; focus on the ideas
  - errors about Java specifics will receive little to no penalty
- These slides and this class session will not be comprehensive

Pre-Midterm

- Design tools (UML, CRC cards, etc)
- Designing classes, programming by contract
- Interfaces and polymorphism
- Programming patterns (Composite, Decorator, Strategy, Template Method)
- Inheritance and hierarchy
- Types in Java

Post-Midterm

- More Design Patterns
- Cloning and Serialization
- Reflection
- Generics
- Frameworks
- Multithreading
- Data Structures
- Networking

Design Patterns

- Understand the general ideas of the context and solution of each pattern
- Prototype, Adapter, Command, Factory Method, Proxy, Singleton, Visitor

Cloning and Serialization

- Cloneable and Serializable as tagging interfaces
- Shallow copy vs. deep copy
- Serialization of objects with references to other objects
  - transient fields
- Drawbacks of serialization

Reflection

- Reflection allows programs to get information about objects, classes, methods and fields at runtime
- Useful for extremely general code (i.e., automated testing, debugging, monitoring of programs at runtime)

Generics

- Generic classes, generic methods
- Type bounds, wild cards
- Type erasure
- Advantages of generics over using Objects (or other superclasses)

Frameworks

- Inversion of control
- Application frameworks (e.g., the graph editor framework)

Multithreading

- Java Thread states
- Locks and conditions
- Deadlock

Data Structures

- Abstract Data Types
- Data structures and their ideal applications
  - i.e., what operations are optimized in each data structure
- Don't study data structure implementations (unless it helps you remember what they're used for)

Networking

- Socket and ServerSocket classes
- Connecting via TCP/IP over a port and IP addresses

Slide Highlights

java.lang.Object

- All class variables extend the base Java class, java.lang.Object
- Object contains a few implemented methods:
  - String toString()
  - boolean equals(Object other)
  - Object clone()
  - int hashCode()

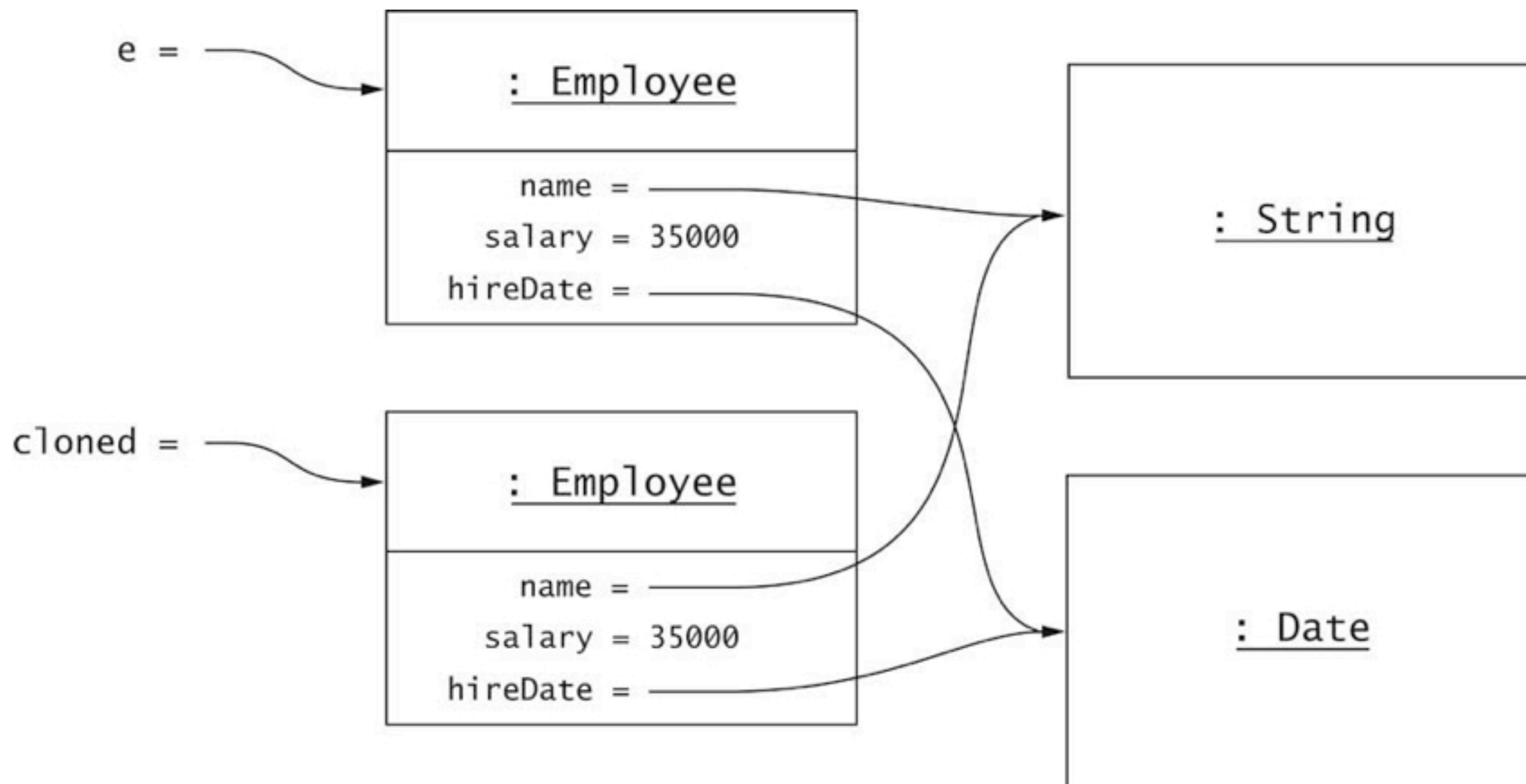
clone()

- Clone is meant to be used when you want an actual copy of an Object instead of another reference
- (x.clone() != x) && (x.clone().equals(x))
- Default clone() copies all fields
- clone() is a protected method by default and can only be used if your subclass implements the Cloneable interface

The Cloneable Interface

- Tagging interface; contains no methods
- But Object uses it to check that calls to clone() are only on Cloneable objects
  - otherwise throws `CloneNotSupportedException`
- Must be careful; copying fields may still share common aggregated objects

Shallow vs. Deep Copy



Shallow vs. Deep Copy

- Cloning all fields won’t clone any Class variables, like String or Date
- Then if the clone modifies the Date object, the original’s Date gets changed
- Instead, we can recursively clone all mutable class objects

Deep Copy Recursion

- Recursively cloning fields can cause very bad things to happen
- Consider MVC objects that store references to each other

Model → View
Controller

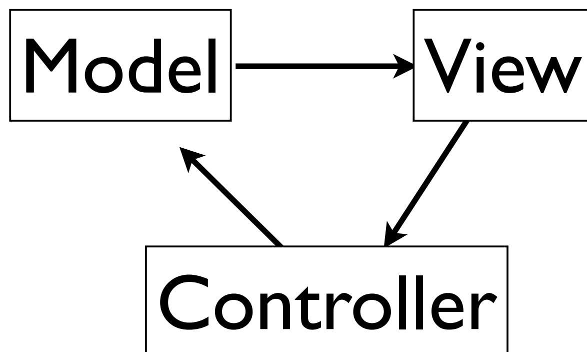
Model → View → Controller
Controller ← View ← Model
Model → View → Controller → M

Serializable Interface

- Another tagging interface
- Tells Java that a class is able to be written to file using `ObjectOutputStream`
- `new ObjectOutputStream(FileOutputStream f)`
- `ObjectOutputStream.writeObject(Serializable s)`
- Writes the object and *all its fields and referenced objects* to file
- Fields not to be written can be marked with keyword `transient`

Serializing Circular Structure

- Files assign serial numbers to Objects
- So circular structure can be saved without infinite recursion
- But we can only load one object
- Let's test this with an experiment



Reflection

- Reflection is the ability of a program to find out about the capabilities of objects at *runtime*
- Java provides these classes to describe features of types:
  - Class, Package, Field, Method, Constructor, Array

Class Objects

- `(obj instanceof Shape)` only tells you if variable `obj` is a subtype of `Shape`
- If you want to know the exact class, you need to use a class object `obj.getClass()`
- JVM keeps one object of each known class, so use == operator to check class equality
- Can also directly get class objects by `Shape.class == obj.getClass()`

Class Attributes

- `Shape.class.getSuperClass() //returns Class`
- `Shape.class.getInterfaces() //returns Interface[]`
- `Shape.class.getPackage() //returns Package`
- `Shape.class.getDeclaredMethods() //returns Method[]`
- `Shape.class.getDeclaredFields() //returns Field[]`
- `Shape.class.getDeclaredConstructors()//Constructor[]`

Method Objects

- `m.getName(), m.getParameterTypes()`
- Also can get Method objects using `Method m = getDeclaredMethod(name, params, ...)`
- Then call methods with `m.invoke(params)`
- Rarely useful, but can be used to build general testing programs

Field Objects

- `Class getType()`
- `int getModifiers() // binary flags`
  - `Modifier.isAbstract(), isPrivate(), isFinal(), etc`
- `Object get(Object obj) // reads field`
- `void set(Object obj, Object value)`
- `void setAccessible(boolean b) // changes whether private`
  `// fields are accessible. Wait, what???!`
  - Java programs allow this by default, applets and servlets do not.

Why Reflection?

- Pros:
  - Extremely powerful way to dynamically retrieve information about Classes by name
  - Retains Object Oriented ideas
  - Allows for meta-programs (like JUnit)
- Cons:
  - Can break encapsulation
  - Some anti-polymorphism ideas, e.g., checking an actual class type instead of trusting hierarchy

Old-Fashioned Generics

- ```
  public class ArrayList {
      void add(Object obj) { ... }
      Object get(int index) { ... }
  }
  ```
- Any Object subclass works
- Runtime exception when typecasting fails
- We could use reflection to check all casts
  ```

Generic Types

- Declared with a generic placeholder
- `public class Box<T> { ... }`
  - `Box<String> b = new Box<String>();`
  - `Box<Integer> b = new Box<Integer>();`
- `public class Pair<T,U> { ... }`
  - `Pair<String, Date> p = new Pair<String, Date>();`

Generic Methods

- We can use generic types in methods, which get resolved dynamically when the method is called

```
public static <E> void fill(ArrayList<E> a, E value, int count)
{
   for (int i = 0; i < count; i++)
      a.add(value);
}
```

- This checks that the ArrayList and value are of the appropriate type at compile time

Type bounds

- Occasionally, generic types are too restrictive

```
public static <E> void append(ArrayList<E> a,
      ArrayList<E> b, int count)
{
   for (int i = 0; i < count && i < b.size(); i++)
      a.add(b.get(i));
}
```

- We can use a *type bound* to relax restrictions

```
public static <E, F extends E> void append(ArrayList<E> a,
      ArrayList<F> b, int count)
```

Wildcards

- Type bounds still require that the client defines the generic types
- Sometimes this is undesirable, so we can use wildcards instead

```
public static <E> void append(ArrayList<E> a,
        ArrayList<? extends E> b, int count)
{
    for (int i = 0; i < count && i < b.size(); i++)
        a.add(b.get(i));
}
```

Type Erasure

- After javac checks correct type usage with generics, it strips all types from the code into *raw types*
- The resulting code is similar to old-fashioned “generic” code, using Object variables (or the most general superclass)
- This allows compatibility with older code
  - but unfortunately leads to some limitations

Frameworks

- Sets of cooperating classes that implement mechanisms essential for a particular problem domain
- Application frameworks implement services common to a certain type of application
- Programmers subclass some framework classes and implement additional functionality specific to the target application

Packages

- Typically, framework classes can be stored in packages
- javax.swing.*, java.awt.*, java.applet.*
- Allows clients to import easily

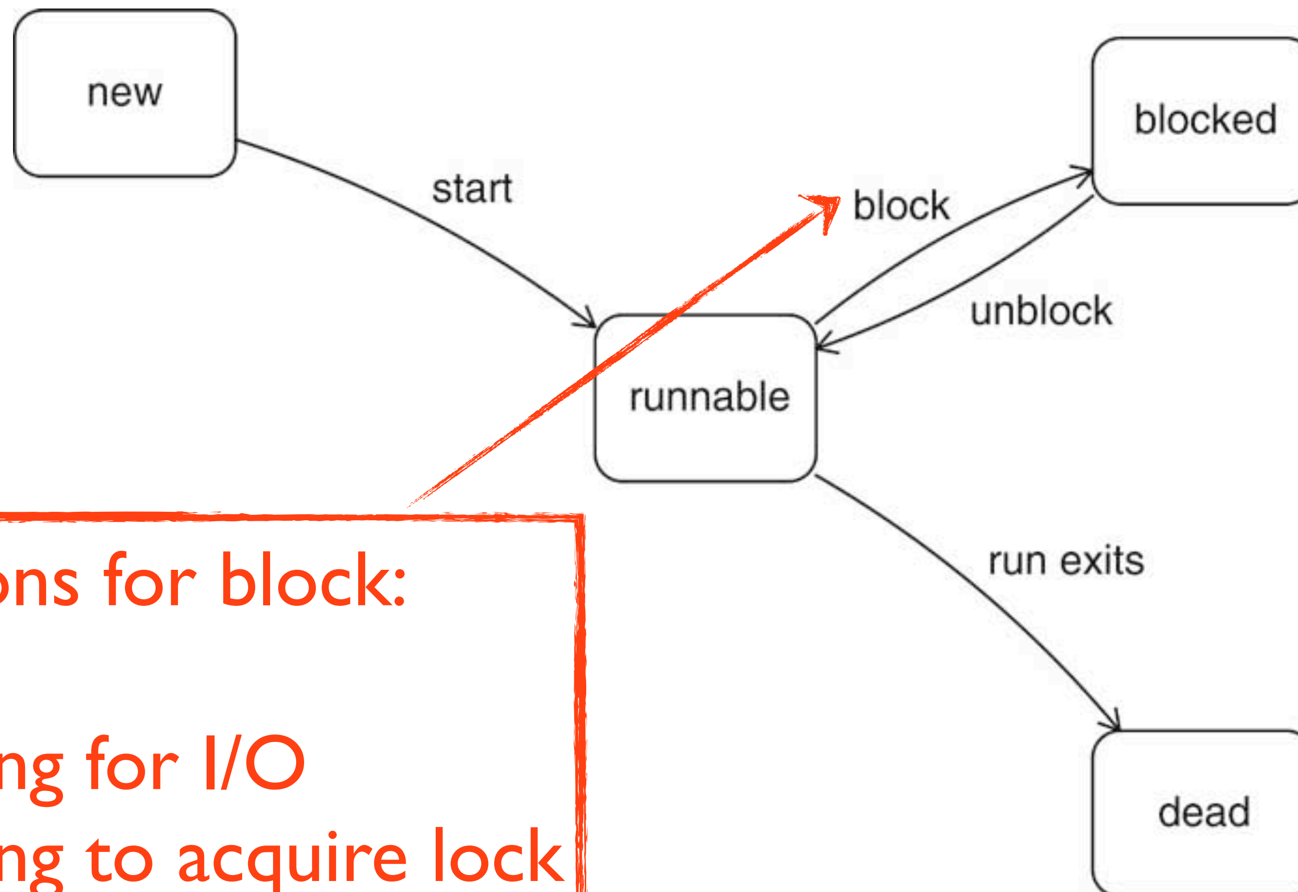
Inversion of Control

- Most of the work is done by the framework, as in the template method and strategy patterns
- The programmer doesn't need to be concerned with control flow, just the specifics of the applications

Multithreading

- Modern computer programs perform various calculations simultaneously
- Each parallel program unit is called a *thread*
- In most cases, threads are not actually run in parallel, but by taking turns
- But the OS is responsible for the turn-taking; we don’t know its policy

Thread States



Reasons for block:

Sleep

Waiting for I/O

Waiting to acquire lock

Waiting for condition

Thread (abridged)

- `void join()` - Waits for this thread to die
- `static void sleep(long millis)` - Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.
- `void start()` - Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.
- `static void yield()` - Causes the currently executing thread object to temporarily pause and allow other threads to execute.

Runnable

Method Summary

`void`	`run()` When an object implementing interface `Runnable` is used to create a thread, starting the thread causes the object's `run` method to be called in that separately executing thread.

Method Detail

run

`void run()`

When an object implementing interface `Runnable` is used to create a thread, starting the thread causes the object's `run` method to be called in that separately executing thread.

The general contract of the method `run` is that it may take any action whatsoever.

Object

protected Object	clone() Creates and returns a copy of this object.
boolean	equals(Object obj) Indicates whether some other object is "equal to" this one.
protected void	finalize() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<?>	getClass() Returns the runtime class of this Object.
int	hashCode() Returns a hash code value for the object.
void	notify() Wakes up a single thread that is waiting on this object's monitor.
void	notifyAll() Wakes up all threads that are waiting on this object's monitor.
String	toString() Returns a string representation of the object.
void	wait() Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
void	wait(long timeout) Causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
void	wait(long timeout, int nanos) Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

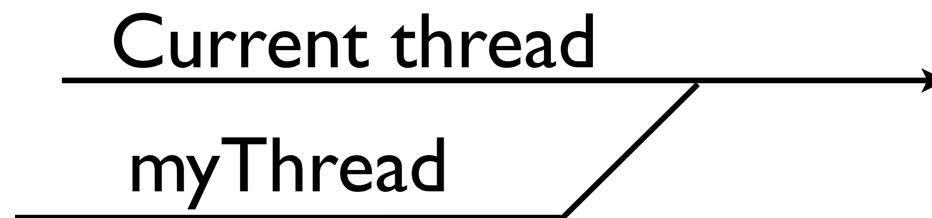
Interrupting Threads

- If you need to terminate a thread, call Thread.interrupt()
- Causes Thread.sleep() to throw InterruptedException
- Your run method should be structured to handle interrupts cleanly

```
public void run() {
   try {
      while(more_work_to_do) {
         // do work
         Thread.sleep(DELAY);
      }
   }
   catch(InterruptedException e)
   {
   }
   // clean up
}
```

Joining Threads

- myThread.join() joins Thread myThread with the current thread
  - i.e., waits for myThread to finish its run() method



Locks

- We can use *locks* to fix race conditions
- Threads temporarily acquire ownership of locks
- Only one thread can own a lock at a time
- If a thread tries to acquire a lock but it is owned by another, it waits
- When a lock owner releases the lock, all waiting threads are notified

Lock Interface

- `java.util.concurrent.locks` package includes the `Lock` interface
- Objects that implement `Lock` have
  - ```
    lock() // prevent other threads from
           // locking this object
    ```
  - `unlock() // allow other threads to lock this`
    ```

Condition Objects

- Each `Lock` can have any number of `Condition` objects
- `Condition setNonEmpty = setLock.newCondition()`
- ```
  setLock.lock()
  while(set.isEmpty())
      setNonEmpty.await() // releases the lock
  ```
- Whenever the condition could have changed, call `setNonEmpty.signalAll()`
  - Unblock all waiting threads, but a thread must reacquire the lock before returning from `await`
  ```

Dining Philosophers

- Example of deadlock when threads need two or more locks (e.g., moving objects from list to list)
- Each diner locks chopsticks then eats
  - leftChopstick.lock()
    rightChopstick.lock()
    eat()
    rightChopstick.unlock()
    leftChopstick.unlock()

First Problem: Starvation

- Since we don’t know how OS will schedule threads, two diners may never get to eat
- ReentrantLock has a **fairness** flag that makes sure locks are granted first-come-first-served
  - new ReentrantLock(true);

Second Problem: Deadlock

- If all diner threads start simultaneously, we can get stuck in a *deadlock*
- Each philosopher locks his left chopstick, waits for right chopstick
- Even if we use conditions and release the chopsticks, we could have *livelock*
  - Infinite loop of simultaneously locking and releasing the left chopsticks

Two Deadlock Solutions

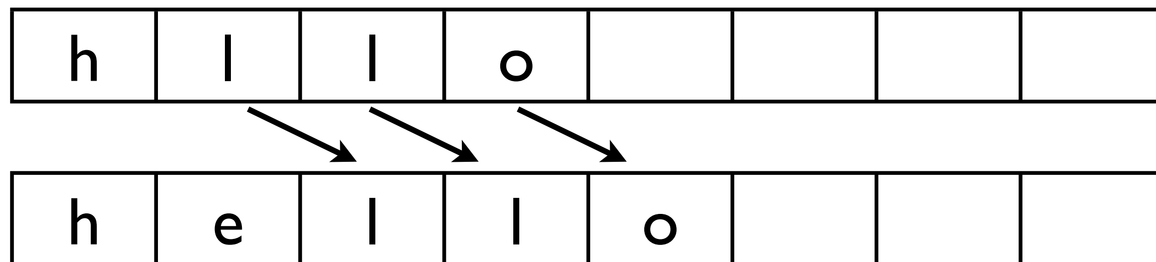
- Order the chopsticks; locks must be acquired in the same order
  - No circular deadlock, but now some threads have higher priority
- Require master lock to lock any chopsticks
  - master.lock()
    leftChopstick.lock(); rightChopstick.lock();
    master.unlock();
    eat()
    leftChopstick.unlock(); rightChopstick.unlock()

Lists

- An ordered series of objects
- Each object has a previous and next
  - Except **first** has no prev., **last** has no next
- We can insert an object (at location *k*)
- We can remove an object (at location *k*)
- We can read an object (from location *k*)

ArrayList

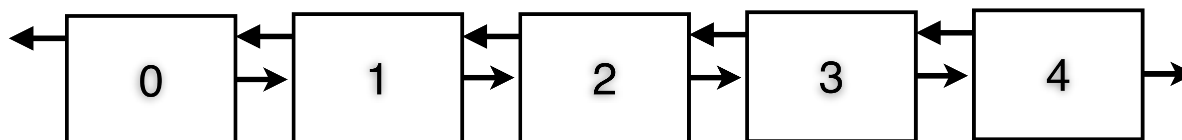
- Essentially a wrapper for an array
- Store elements in array, but handles list operations by shifting elements



- If array is full, copies into a new larger array
- O(1) get, O(N) insert/remove
    - O(1) insert/remove at the end of list

LinkedList

- Stores elements in Link objects
- Each link has reference to next (and prev)
  - prev links only in *doubly-linked* list
- Navigate by following next() references
- O(1) insert/remove with reference
  - But need O(N) to find (get) reference



Stacks and Queues

image from
http://bwog.net/2006/05/03/tray-spotting

- Stack - Last in first out
  - push() - add element to top of stack
  - pop() - remove element from top
- Queue - First in first out
  - enqueue (offer) - add element at back of line
  - dequeue (poll) - remove from front of line

Sets

- An unordered collection
- No duplicate entries
- We can insert an object
- We can check for an object – contains()
- We can remove an object

HashSet

- Uses hashCode() to index into an array
- Collisions occur when distinct elements *hash* into the same index
- Collisions resolved by trying empty spots in a systematic way

Maps

- Maps are collections of objects "indexed" by other objects
- key types map to value types
- No duplicate keys, duplicate values allowed
- aka "associative array"

HashMap

- `Map<String, Double> costs =`
  `            new HashMap<String, Double>();`
- `myMap.put("Big Mac", 2.99);`
- `myMap.get("Big Mac");`
- index by the key's hashCode()
  - but insert value instead of key

Sets, Maps, Collections

- Recall that Set is a subinterface of Collections that has no new methods
- HashMap doesn't implement Collection
- Has methods
  - `Set<K> keyset()`
  - `Collection<V> values()`

Sorted Map ADT

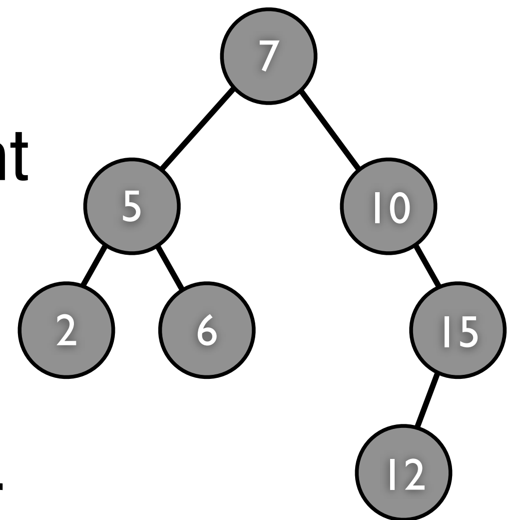
- Subtype of Map (can get value by key)
- `SortedMap<K implements Comparable,V>`
- `SortedMap<K,V> subMap(K fromKey, K toKey)`
  - firstKey, lastKey, headMap, tailMap

TreeMap

- Implements SortedMap
- put(), get(), contains() cost O(log N)
- Uses an advanced binary search tree called Red-Black Tree
  - a balanced BST
- Slower than HashMap, but keys have order

Binary Search Tree

- Tree nodes have left and right children
  - Left children are less than parent,
  - Right children are greater than parent
- At each node, O(1) comparison determines which child to move to
- Depth of tree is the worst-case time for each operation

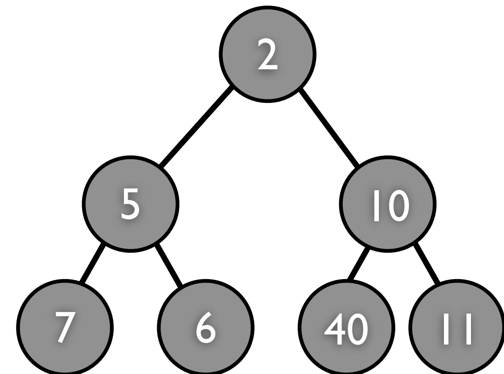


Priority Queue ADT

- Stores elements by priority (serves as the key)
  - Not really a queue, but used in similar applications
- add aka offer(E e)
- deleteMin aka poll()
- findMin aka peek()

Heaps

- Binary tree with heap order property: keys of children greater than parent's
- Running time:
  - O(log N) add,
  - O(log N) deleteMin,
  - O(1) findMin



Prototype Pattern

Context

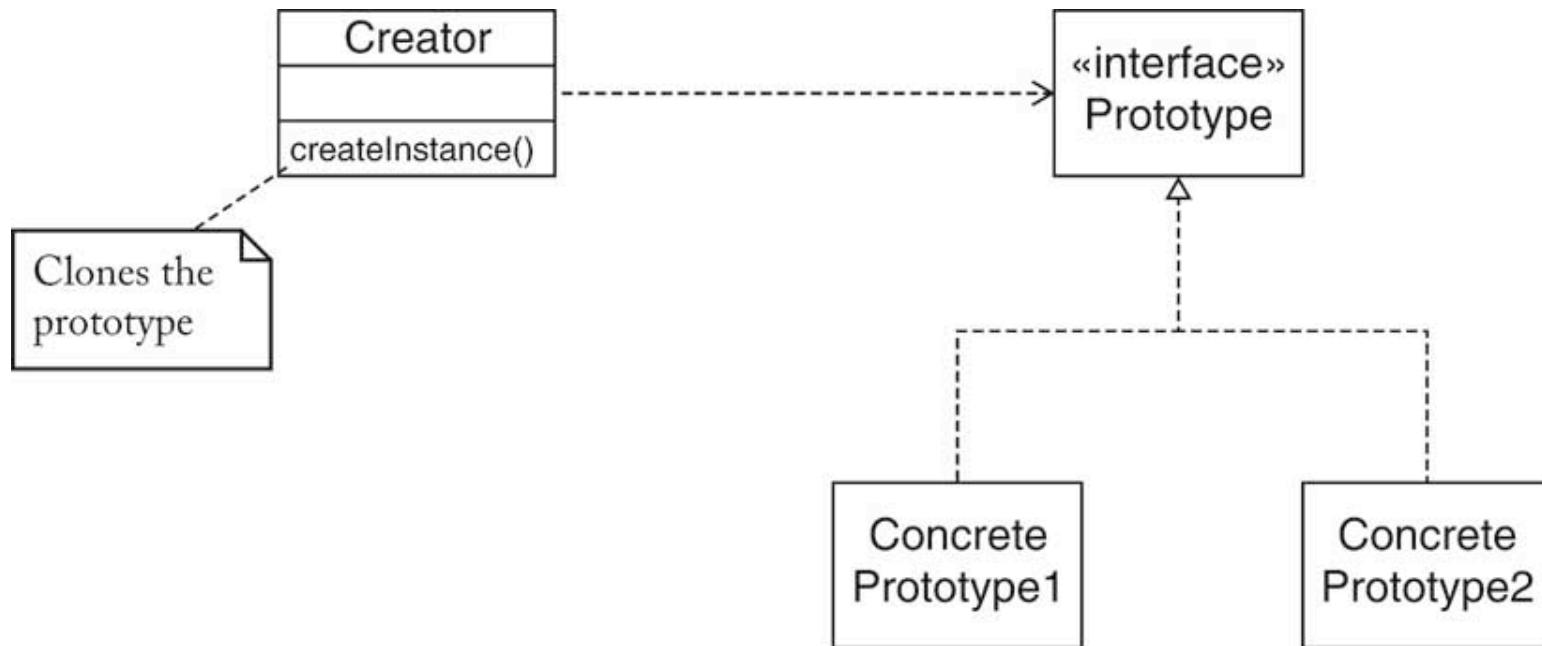
- A system needs to create several kinds of objects whose classes are not known when the system is built
- You don't want to require a separate class for each kind of object
- You want to avoid a separate hierarchy of classes whose responsibility it is to create the objects

---

Solution

- Define a prototype interface common to all created objects
- Supply a prototype object for each kind of object that the system creates
- Clone the prototype object whenever a new object of the given kind is required

Prototype Pattern



ADAPTER

Context

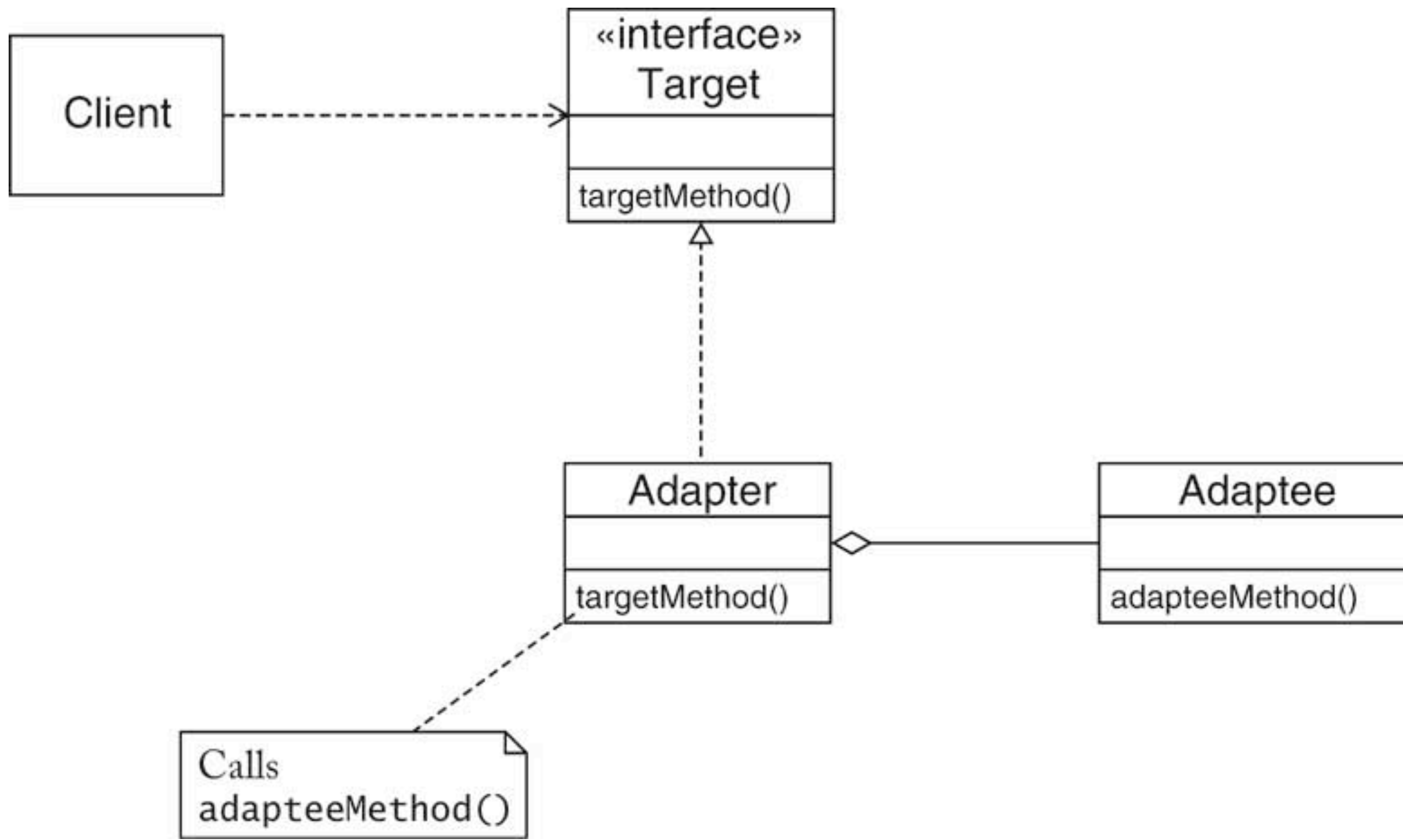
- You want to use an existing *adaptee* class without modifying it.
- The context in which you want to use the class requires conformance to a *target* interface
- The target interface and the adaptee interface are conceptually related

---

Solution

- Define an adapter class that implements the target interface
- The adapter class holds a reference to the adaptee. It translates target methods to adaptee methods
- The client wraps the adaptee into an adapter class object

Adapter Diagram



COMMAND

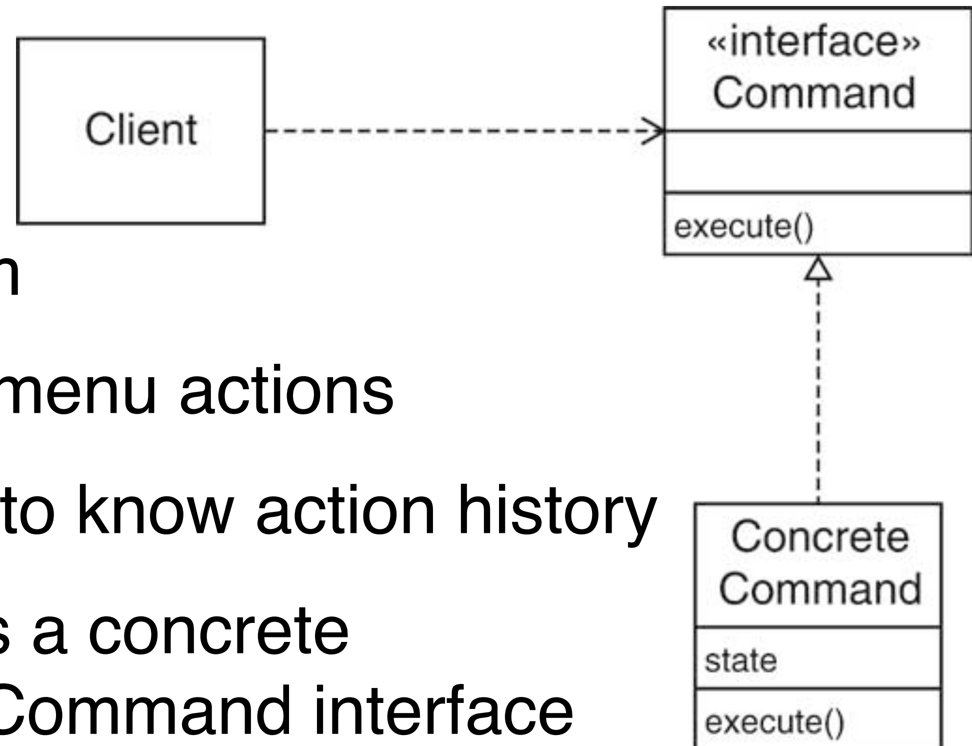
Context

- You want to implement commands that behave like objects, either because
  - you want to store additional information with commands,
  - or you want to collect commands

Solution

- Define a *command* interface type with a method to **execute** the command
- Supply methods in the command interface type to manipulate the state of command objects
- Each *concrete command* class implements the command interface type
- To invoke the command, call the **execute** method

Command Example



- Client: painting program
- User performs various menu actions
- Multi-level undo needs to know action history
  - Each type of action is a concrete implementation of a Command interface
  - Each action also implements an undo() method
- Client program stores stack of commands; pop().undo() to undo most recent command

FACTORY-METHOD

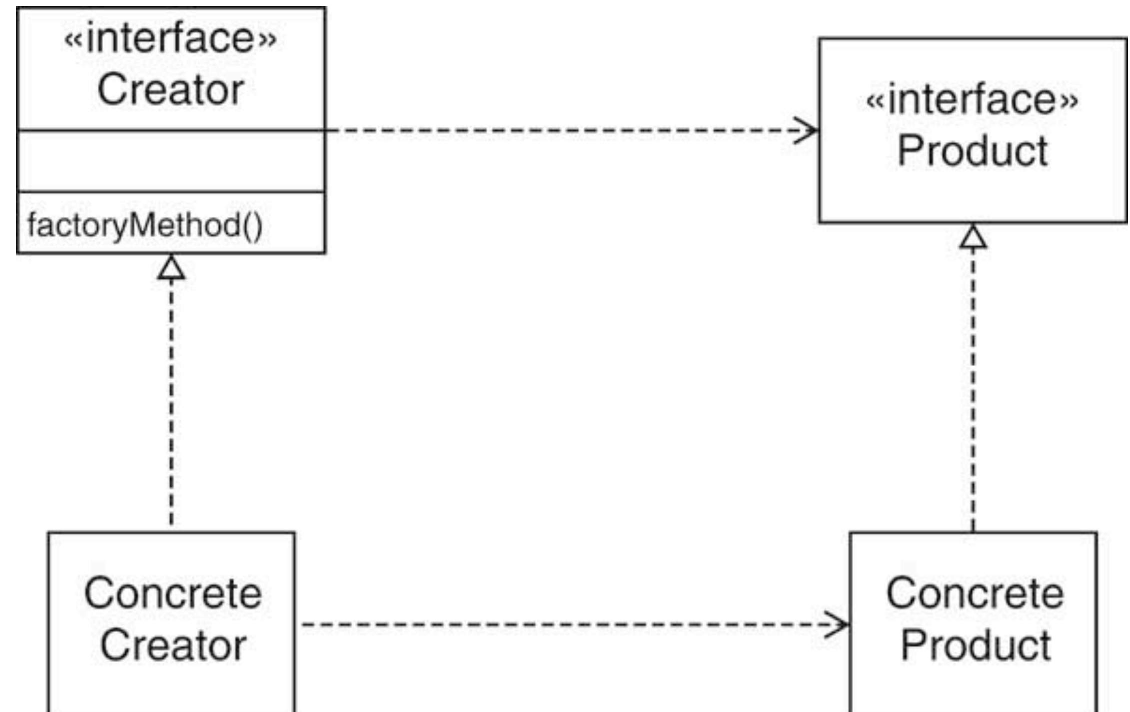
Context

- A creator type creates objects of another *product* type
- Subclasses of the creator type need to create different kinds of product objects
- Clients do not need to know the exact type of product objects

Solution

- Define a creator type that expresses the commonality of all creators
- Define a product type that expresses the commonality of all products
- Define a *factory method* in the creator type. The factory method yields a product object
- Each concrete creator class implements the factory method so that it returns an object of a concrete product class

Example Factory-Method



- Creator: Collection
- Concrete Creator: LinkedList
- factoryMethod(): iterator()
- Product: Iterator
- ConcreteProduct: LinkedListIterator

PROXY

Context

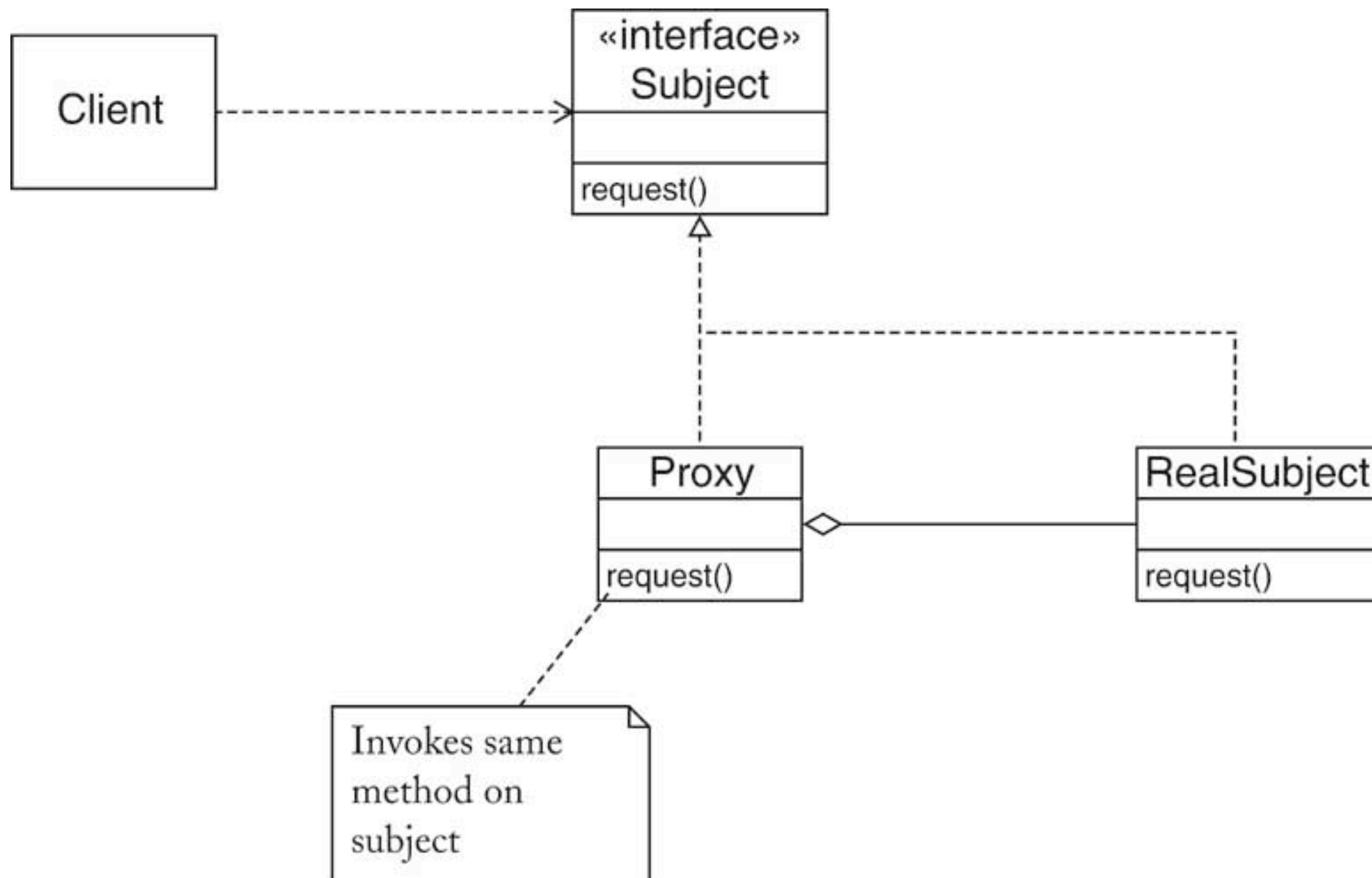
- A *real subject class* provides a service that is specified by an *subject interface* type
- There is a need to modify the service in order to make it more versatile
- Neither the client nor the real subject should be affected by the modification

---

Solution

- Define a *proxy* class that implements the subject interface type. The proxy holds a reference to the real subject
- The client uses a proxy object
- Each proxy method invokes the same method on the real subject and provides the necessary modifications

Proxy Diagram



SINGLETON

Context

- All clients need to access a single shared instance of a class
- You want to ensure that no additional instances can be created accidentally

---

Solution

- Define a class with a private constructor
- The class constructs a single instance of itself
- Supply a static method that returns a reference to the single instance

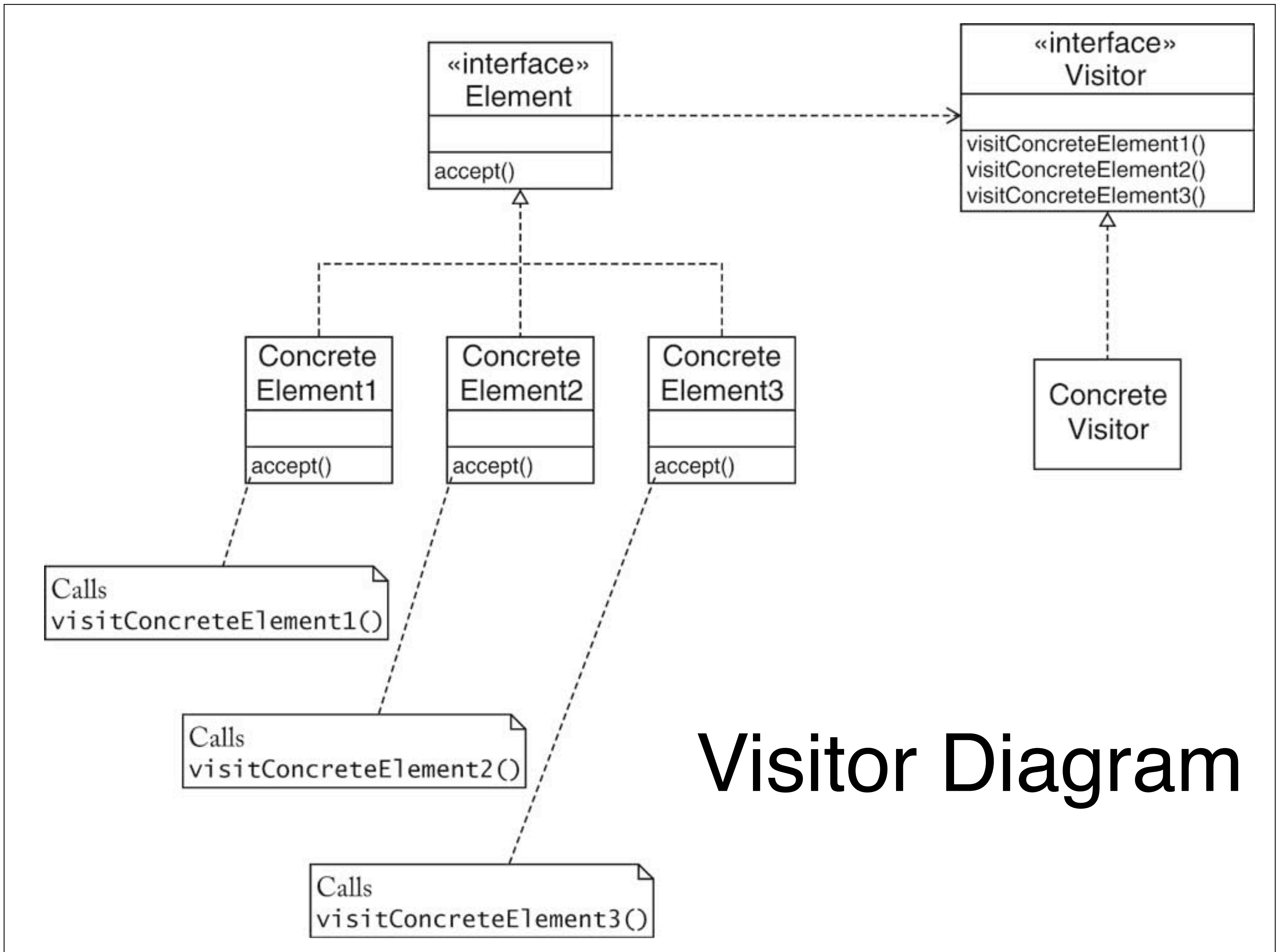
VISITOR

Context

- An object structure contains element classes of multiple types, and you want to carry out operations that depend on the object types
- The set of operations should be extensible over time
- The set of element classes is fixed

Solution

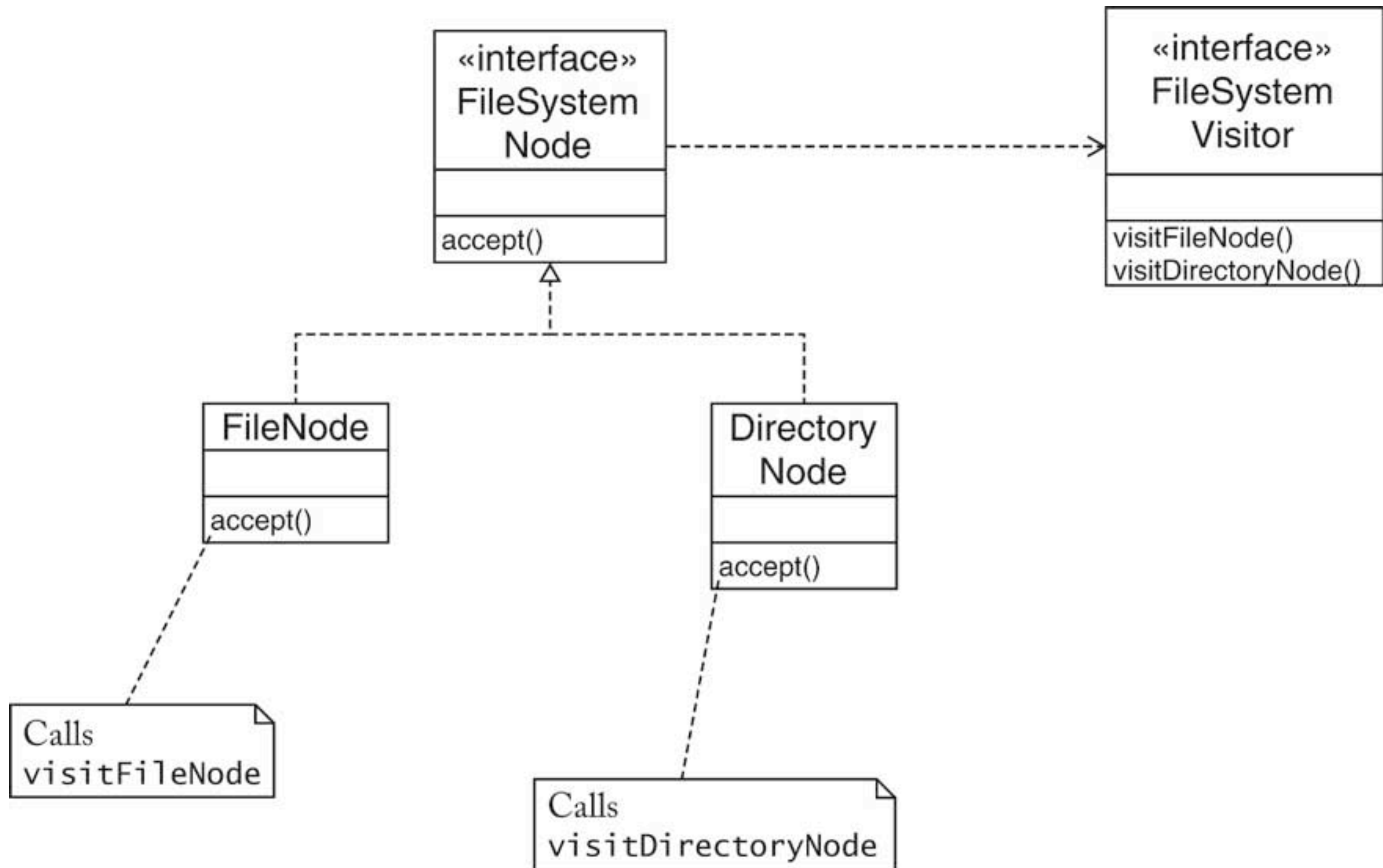
- Define a *visitor* interface that has methods for visiting elements of each of the given types
- Each element class defines an **accept** method that invokes the matching element visitation method on the visitor parameter
- To implement an operation, define a class that implements the visitor interface type and supplies the operation's action for each element type


«interface»
Element
accept()
«interface»
Visitor
visitConcreteElement1()
visitConcreteElement2()
visitConcreteElement3()
Concrete Element1
accept()
Concrete Element2
accept()
Concrete Element3
accept()
Concrete Visitor
Calls visitConcreteElement1()
Calls visitConcreteElement2()
Calls visitConcreteElement3()
Visitor Diagram

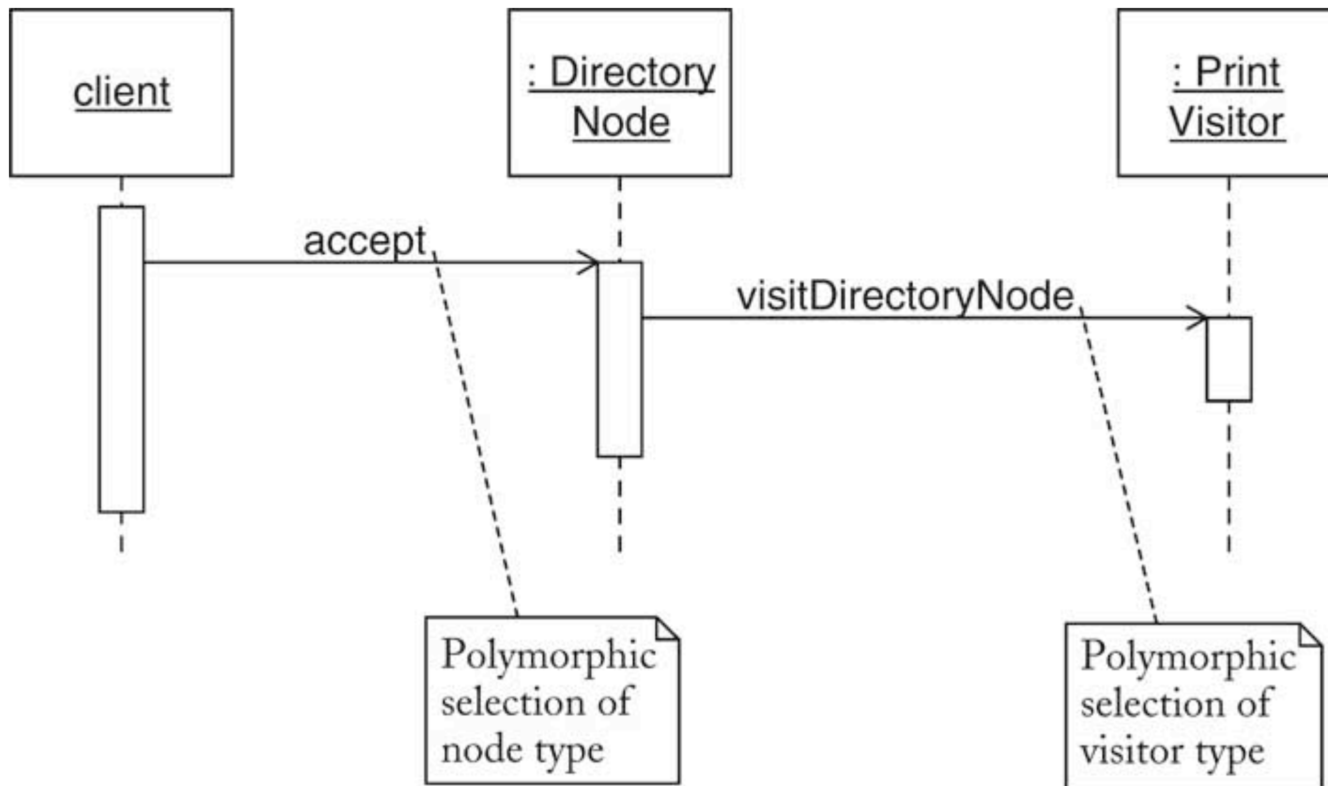
Double Dispatch

- This pattern uses polymorphism twice to make code very general
  - 1st, element.accept() calls Visitor method based on type of element
  - 2nd, the Visitor method performs operation based on type of Visitor
- Both actions called through interfaces
- Concrete classes need not be known at runtime

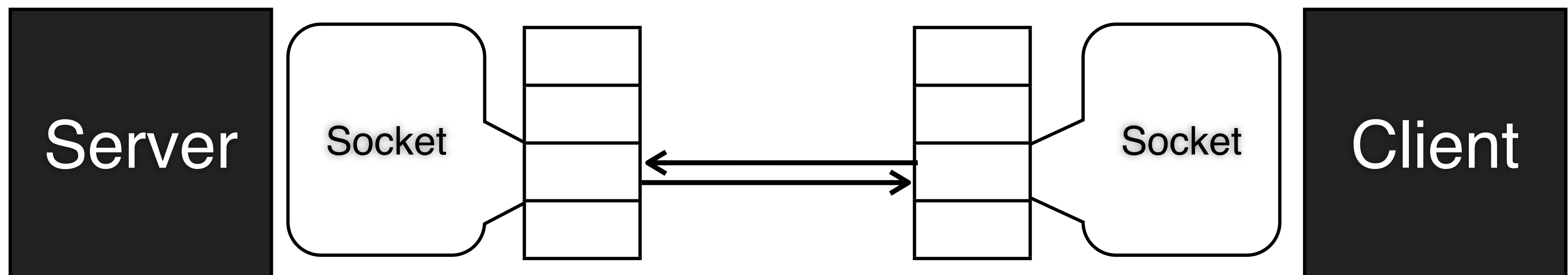
Example Visitor



Double Dispatch in FileSystemNode



Sockets



- On each side of a TCP/IP connection, there is a *socket*
- Java lets us abstract away the details and work with the sockets
- One end is a *server* and the other is a *client*
- Each socket has an InputStream and an OutputStream

Socket API

- Common constructor:
  `new Socket(String host, int port)`
- `InputStream getInputStream()`
- `OutputStream getOutputStream()`
- `void close()`
- IOExceptions everywhere; lots of try/catch blocks in Socket code
- But how do we set up the host? ServerSocket

ServerSocket API

- Constructor:
  `new ServerSocket(int port)`
- `Socket accept() // Listens for a connection to be made and accepts it`
- `close();`
- Get Sockets using accept() and perform logic with them using their InputStream and OutputStream
- Usually, wrap with `new BufferedReader(socket.getInputStream())`
  - and `new PrintWriter(socket.getOutputStream())`

Why Study OOP and Design?

- Writing a code is easy
- Understanding code is hard
- Good organization and design of programs makes understanding easier