Objects Analysis Threads Design

15-214

toad

Fall 2014

Principles of Software Construction: Objects, Design, and Concurrency

Design Goals and Process

Jonathan Aldrich Charlie Garrod

School of Computer Science

institute for SOFTWARE RESEARCH

Review: In the Previous Lecture…

- We designed a forestry simulation.
  What made it challenging?

- What is it that generally drives design decisions?

- What is the feature of object-oriented programming that facilitates extensibility?

Review: In the Previous Lecture…

- We designed a forestry simulation.
  What made it challenging?
  - Extensibility and evolvability requirements
- What is it that generally drives design decisions?
  - Quality Attributes (extensibility, evolvability, and others)
- What is the feature of object-oriented programming that facilitates extensibility?
  - Dispatch

Review/Elaboration: Quality Attributes

- Design driven by extensibility, modifiability requirements
  - Framework should work **unmodified** with other simulations
  - Can **extend** with new agents
    - without **modifying** existing agents or the framework
  - Can **modify** the simulation setup or **modify** the set of agents
    - Without **modifying** the agents or the framework

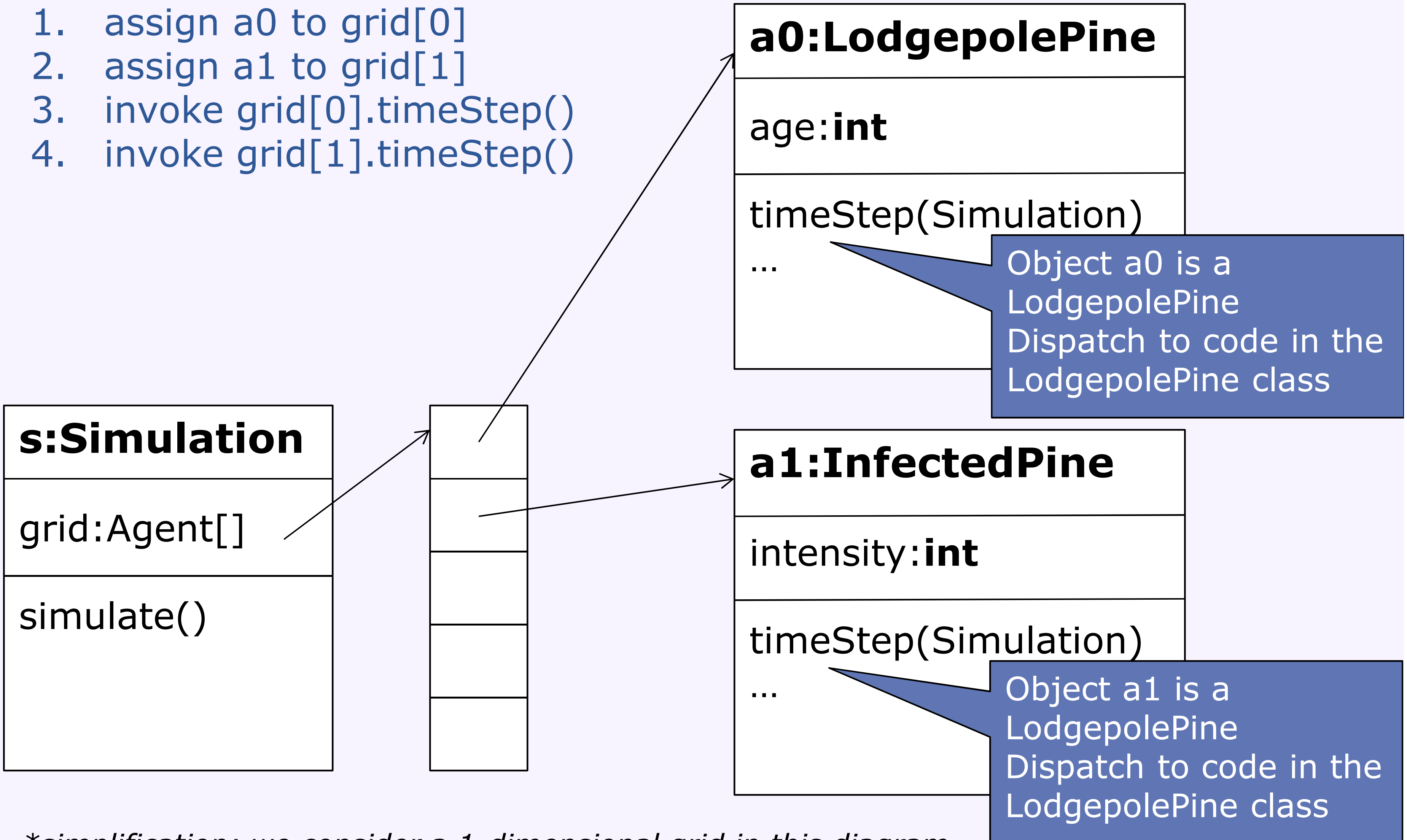
- Extensibility and modifiability are **quality attributes**
  - Properties of software that describe its fitness for further development and use
  - Not **what** the system does but **how well** it does it

- Other quality attributes
  - Performance
  - Availability
  - Security
  - Testability
  - Usability

The major focus of design is achieving quality attributes

Review: Dispatch in the Previous Lecture

1. assign a0 to grid[0]
2. assign a1 to grid[1]
3. invoke grid[0].timeStep()
4. invoke grid[1].timeStep()



**simplification: we consider a 1-dimensional grid in this diagram*

Puzzle: What Does This Code Print?

```
class Table {
    BallState state;
    void set(BallState s) {
        state = s;
    }
    void play(int n) {
        for (int i = 0; i < n; ++i)
            state.bounce(this);
        state.print();
    }
}
void main(…) {
    Table t = new Table();
    t.set(new Ping());
    t.play(3);
}
```

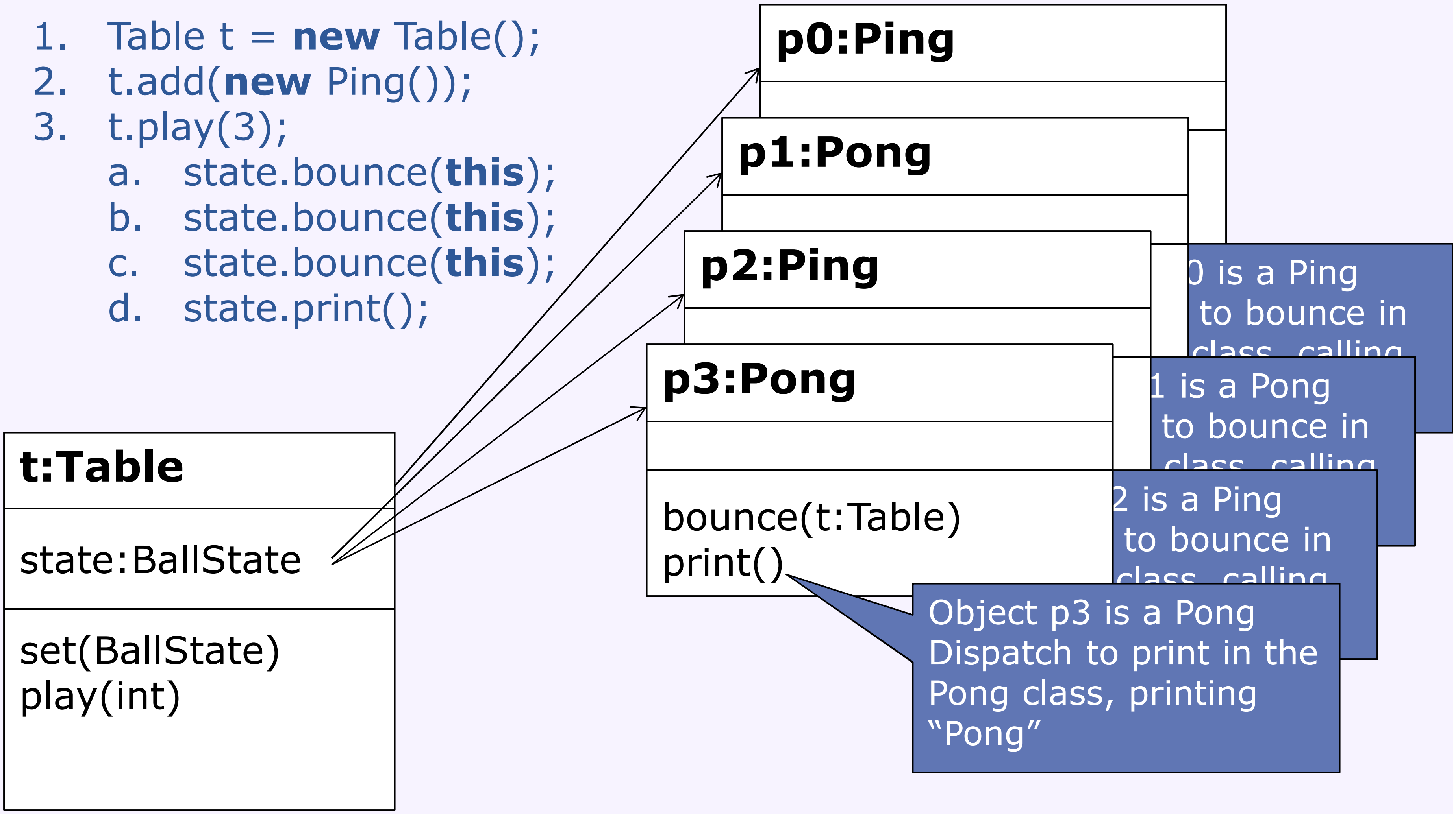
```
interface BallState {
    void bounce(Table t);
    void print();
}
                allState {

    void print() {
        System.out.println("Ping");
}   }
class Pong implements BallState {
    void bounce(Table t) {
        t.set(new Ping());
    }
    void print() {
        System.out.println("Pong");
}   }
```

Refers to the state field of the current object **this**. Could have written: **this**.state = s;

System.out is an object representing standard output.

Dispatch in Ping-Pong

1. Table t = **new** Table();
2. t.add(**new** Ping());
3. t.play(3);
   a. state.bounce(**this**);
   b. state.bounce(**this**);
   c. state.bounce(**this**);
   d. state.print();



**simplification: we consider a 1-dimensional grid in this diagram*

Learning Goals

- Review quality attributes, extensibility, and dispatch ✓
- Know the steps of the design process
- Understand quality attributes in more depth
- Learn how several design guidelines promote quality attributes
- Illustrate the design process through an example
- Learn Java’s encapsulation constructs

Why a Design Process?

- Without a process, how do you know what to do?
  - A process tells you what is the next thing you should be doing
- A process structures learning
  - We can discuss individual steps in isolation
  - You can practice individual steps, too
- If you follow a process, we can help you better
  - You can show us what steps you have done
  - We can target our advice to where you are stuck

Steps in the Design Process *(example: forest simulation)*

- Precondition: understand functional requirements
  - Step-by-step simulation of the forest, according to a spec
- Precondition: understand quality attribute requirements
  - Extension with new agents; easily change simulation setup
- Design a logical architecture
  - Driven by quality attributes: what code must change independently
- Design a behavioral model
  - The interactions between components, and their order
- Responsibility assignment
  - Which components store data and implement behavior
- Interface design
  - The operations of each component, and their signatures
- Algorithm and data structure design – pseudo-code
- Postcondition: ready to code

Steps in the Design Process *(example: forest simulation)*

- Precondition: understand functional requirements
  - Step-by-step simulation of the forest, according to a spec
- Precondition: understand quality attribute requirements
  - Extension with new agents; easily change simulation setup
- Design a logical architecture
  - Driven by quality attributes: what code must change independently
- Design a behavioral model
  - The interactions between components, and their order
- Responsibility assignment
  - Which components store data and implement behavior
- Interface design
  - The operations of each c
- Algorithm and data stru
- Postcondition: ready to code

Caveats:

- **You may skip steps**
- **You may backtrack**
- **Some steps break down further**

Data Structure Design: Mathematical Sets of Integers

- Design a library representing mathematical sets of integers
- The library should support:
  - Creating empty and singleton sets
  - Computing the union, intersection, and difference of two sets
  - Testing membership of an integer in a set
  - *(likely more goes here…)*
- Plan to extend the library with efficient representations:
  - A representation that represents singleton sets with little space
  - A representation for which union and intersection are fast
  - A representation for which testing membership is fast
  - A representation that is well-balanced across all operations
- Should be able to modify each representation independently
- Different representations should interoperate

Which of these requirements are quality attributes?

Steps in the Design Process *(example: mathematical sets)*

- Precondition: understand functional requirements
  - Operations the sets must support
- Precondition: understand quality attribute requirements
  - Extensible representations; Interoperability; Performance
- Design a logical architecture
  - Trivial: an interface and multiple representations
- Design a behavioral model
  - No interesting constraints on order of operations
- Responsibility assignment
  - Trivial: the set object does it all
- Interface design
  - Mostly specified by the functional requirements
- Algorithm and data structure design – pseudo-code
  - This will be our main focus

IntSet Interface Design *(in-class version)*

```
interface IntSet {

}
```

- The library should support:
  - Computing the union of two sets
  - Testing membership of an integer in a set
  - *… (and other operations)*

IntSet Interface Design *(prepared version)*

```
interface IntSet {
   /** @return the union of this and s */
   IntSet union(IntSet s);

   /** @return true if this contains i */
   boolean contains(int i);

   // and other operations
}
```

- The library should support:
  - Computing the union of two sets
  - Testing membership of an integer in a set
  - *… (and other operations)*

Algorithm and Data Structure Design

- What choices will support:
  - A representation that represents singleton sets with little space?
  - A representation for which union and intersection are fast?
  - A representation for which testing membership is fast?
  - A representation that is well-balanced across all operations?

Algorithm and Data Structure Design

- What choices will support:
  - A representation that represents singleton sets with little space?
    - A single field holding the singleton member
  - A representation for which union and intersection are fast?
    - A UnionSet object with fields for its constituent sets (and similar for IntersectionSet)
  - A representation for which testing membership is fast?
    - A hashtable allows expected constant-time membership testing
  - A representation that is well-balanced across all operations?
    - A sorted array would provide logarithmic membership testing and union operations

Implementing Set

- Trivial example: an empty set

```
class EmptySet implements IntSet {
   /** @return the union of this and s */
   IntSet union(IntSet s) {              }



}
```

- Some OO rules and concepts:
  - Must provide method bodies for all the messages in the interface
    - It is an error if we forget one, or change its signature
  - May define additional methods and/or data fields
  - The class is a ***subtype*** of the interfaces it implements

Implementing Set

```
interface IntSet {
    IntSet union(IntSet s);
    boolean contains(int i);
}
```

- Trivial example: an empty se

```
class EmptySet implements IntSet {
    /** @return the union of this and s */
    IntSet union(IntSet s) { return s; }



}
```

error: method contains from interface IntSet is not implemented

- Some OO rules and concepts:
  - Must provide method bodies for all the messages in the interface
    - It is an error if we forget one, or change its signature
  - May define additional methods and/or data fields
  - The class is a ***subtype*** of the interfaces it implements

Implementing Set

```
interface IntSet {
    IntSet union(IntSet s);
    boolean contains(int i);
}
```

- Trivial example: an empty se

```
class EmptySet implements IntSet {
    /** @return the union of this and s */
    IntSet union(IntSet s) { return s; }
    /** @return true if this contains i */
    boolean contains(int i) { return false; }
}
```

- Some OO rules and concepts:
  - Must provide method bodies for all the messages in the interface
    - It is an error if we forget one, or change its signature
  - May define additional methods and/or data fields
  - The class is a ***subtype*** of the interfaces it implements

Typechecking client code

```
interface IntSet {
    IntSet union(IntSet s);
    boolean contains(int element);
}
class EmptySet implements IntSet { … }
```

2. OK to assign an EmptySet to an IntSet, because EmptySet **implements** IntSet

1. The **new** expression has type EmptySet

```
IntSet s = new EmptySet();
boolean f = s.contains(0); // false
```

4. The contains method in IntSet accepts an **int** argument so the actual argument is OK

5. contains() returns a **boolean**, which we can assign safely to f

3. s has type IntSet. We check that IntSet defines a contains method.

Typechecking: What Could Go Wrong?

```
interface IntSet {
    IntSet union(IntSet s);
    boolean contains(int element);
}
class EmptySet implements IntSet { … }
```

```
EmptySet s = new IntSet();

int f = s.contans(“hello”); // false
```

1. Can’t instantiate an interface; its methods are undefined.

2. Can’t assign an IntSet to an EmptySet because IntSet is not a subtype of (i.e. does not implement) EmptySet

3. s has type EmptySet. But EmptySet does not define a contans method

4. Even if we spell contains correctly, the method takes an **int** argument, and String is not a subtype of **int**

5. contains() returns a **boolean**, which is not a subtype of **int** (unlike in C)

Implementing Singleton and Union Sets *(version 1)*

```
class SingletonSet implements IntSet {
    int member;
    boolean contains(int e) { return m
    IntSet union(IntSet otherSet) {
        UnionSet u = new UnionSet();
        u.set1 = this;
        u.set2 = otherSet;
        return u;
    }
}
class UnionSet implements IntSet {
    IntSet set1;
    IntSet set2;
    boolean contains(int e) {
        return set1.c
    }
    // other methods
}
```

class UnionSet is a ***Composite***—an object that groups other objects, while behaving just like the objects it groups. For example, you can make a UnionSet out of UnionSets.

Quality Attribute: Should be able to modify each representation independently

Issue: what if we want to represent unions with an array of **int**s?

Design Guideline [Representation Hiding]: Hiding the representation of an object from other code helps make it easier to modify the representation

Implementing Singleton and Union Sets *(version 2)*

```
class SingletonSet implements IntSet {
    private int member;

    public SingletonSet(int element) { member = element; }

    public boolean contains(int e) { return member == e; }
    public IntSet union(IntSet otherSet) {
        return new UnionSet(this, otherSet);

                                    IntSet {

    private IntSet set2;
    public UnionSet(IntSet s1, IntSet s2) {
        set1 = s1; set2 = s2; }
    // other methods not shown
}
```

A *private* field can't be used from outside the class

A *constructor* method initializes the fields

Allocates memory and calls the constructor of UnionSet

***public* members—i.e. methods and fields—can be accessed from anywhere**

Now we can change the representation of unions without affecting other code

Implementing Singleton and Union Sets *(version 2)*

```
class SingletonSet implements IntSet {
    private int member;

    public SingletonSet(int element) { member = element; }


    public boolean contains(int e) { return
    public IntSet union(IntSet otherSet) {
        return new UnionSet(this, otherSet);
    }
}

SingletonSet s = new SingletonSet(5);
if (s.member <= 5)
    s.member++;
```

Note: all methods in an **interface** are implicitly **public**

Discussion: when is it useful to have a **private** method?

error: cannot access **private** field member from outside class SingletonSet

Implicit Constructors

- If you don’t define a constructor, Java generates one for you
  - It has no return type and is named after the class
    - Just like all constructors
  - It has no arguments
  - Fields (if any) are initialized to default values
    - 0 for numeric values
    - **false** for **boolean** variables
    - **null** for reference (pointer) variables

```
class EmptySet implements IntSet {
    /** This is equivalent to the auto-generated constructor */
    public EmptySet() {}
    public IntSet union(IntSet s) { return s; }
    public boolean contains(int i) { return false; }
}
```

Using Sets Together

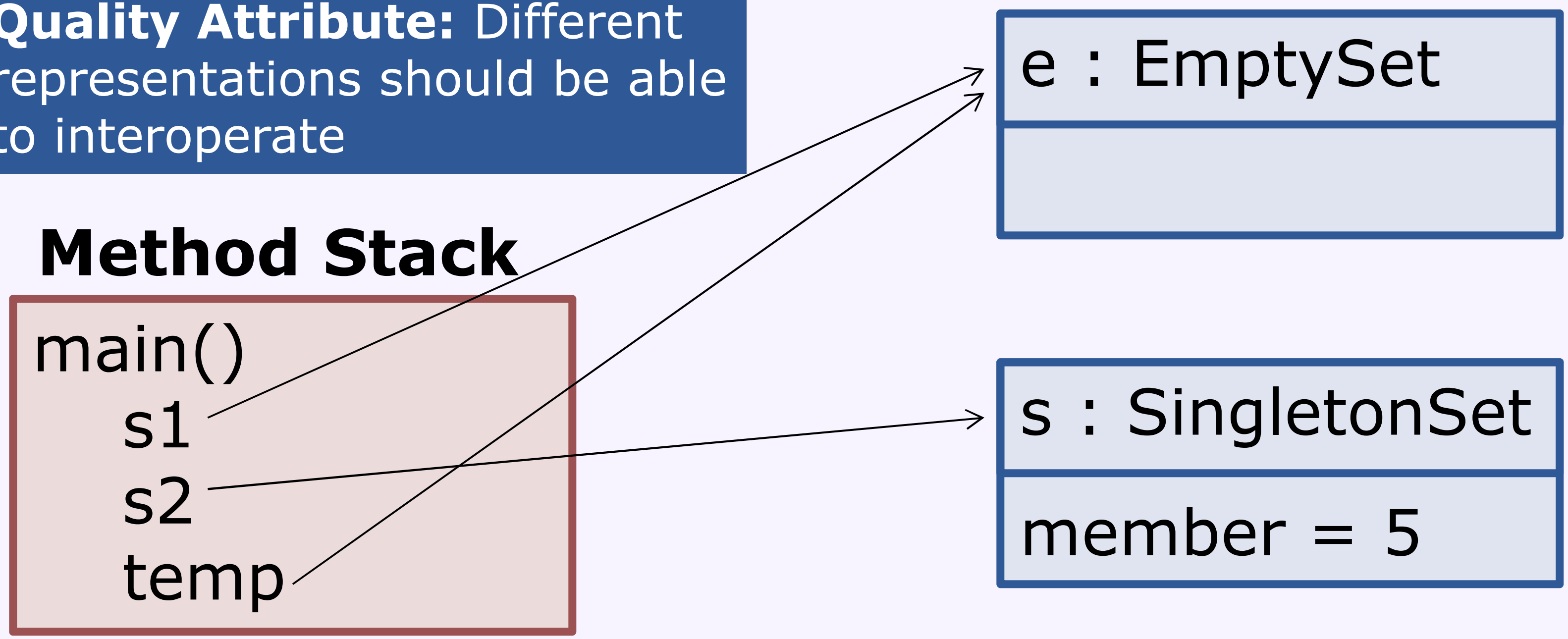
Quality Attribute: Different representations should be able to interoperate

```
IntSet s1 = new EmptySet();
IntSet s2 = new SingletonSet(5);
IntSet temp = s1;
s1 = s2;
s2 = temp;
System.out.println(s1.contains(5));
System.out.println(s2.contains(5));
```

What does this program print?

Using Sets Together

Quality Attribute: Different representations should be able to interoperate

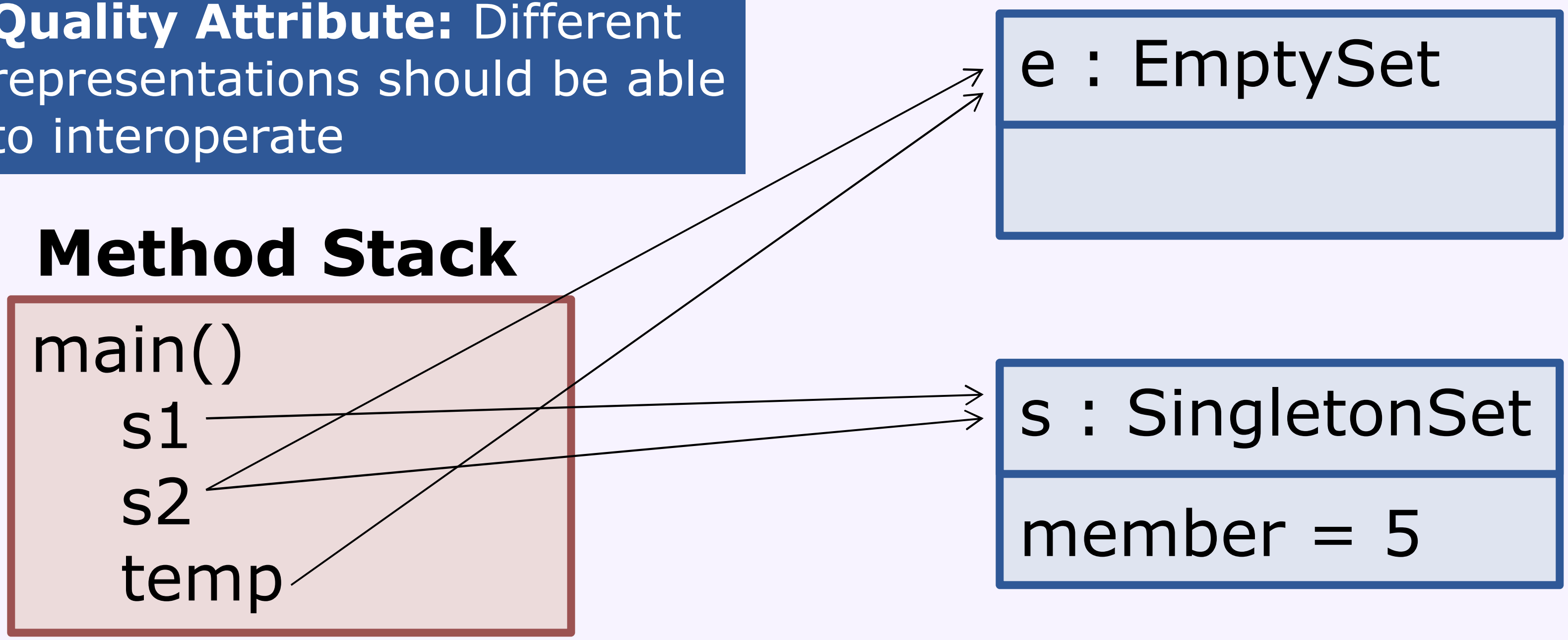



```
IntSet s1 = new EmptySet();
IntSet s2 = new SingletonSet(5);
IntSet temp = s1;
s1 = s2;
s2 = temp;
System.out.println(s1.contains(5));
System.out.println(s2.contains(5));
```

What does this program print?

Using Sets Together

Quality Attribute: Different representations should be able to interoperate




```
IntSet s1 = new EmptySet();
IntSet s2 = new SingletonSet(5);
IntSet temp = s1;
s1 = s2;
s2 = temp;
System.out.println(s1.contains(5));
System.out.println(s2.contains(5));
```

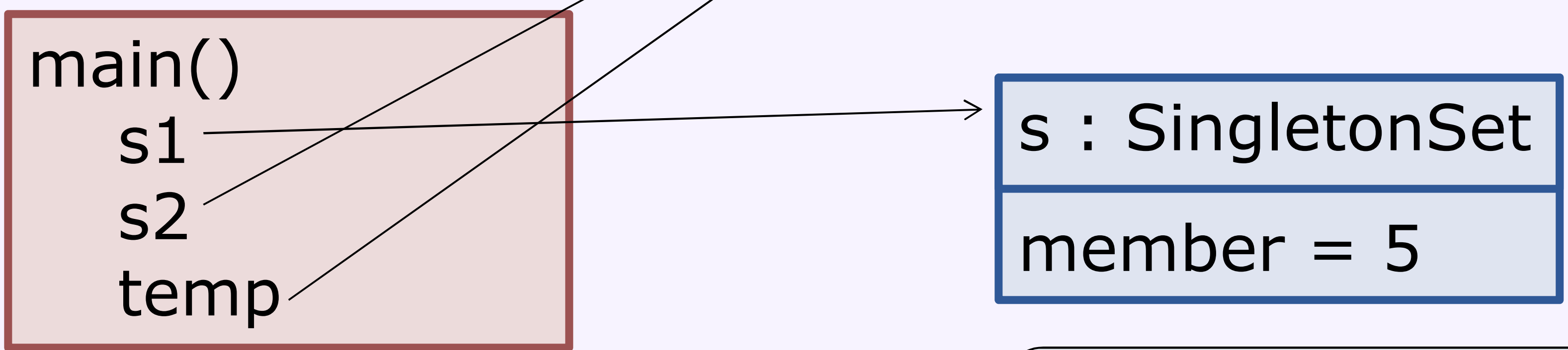
What does this program print?

Using Sets Together

Dynamic Dispatch: determine which method to call based on the run-time class of the object

Polymorphism ("many forms")***:*** Sets can take two forms, and the behavior of a set depends on which form it takes.

Method Stack




```
IntSet s1 = new EmptySet();
IntSet s2 = new SingletonSet(5);
IntSet temp = s1;
s1 = s2;
s2 = temp;
System.out.println(s1.contains(5));
System.out.println(s2.contains(5));
```

s1 points to s.
s is of class SingletonSet.
SingletonSet.contains() is called, printing true

s2 points to e.
e is of class EmptySet.
EmptySet.contains() is called, printing false

Achieving Balanced Performance (version 1)

```
class ArraySet implements IntSet {
    private int members[]; // the array is sorted
    public ArraySet(int ms[], int size) {
        members = new int[size];
        for (int i = 0; i < size; ++i)
            members[i] = ms[i];
    }
    public boolean contains(int eleme
        /* binary search */
    }
    public IntSet union (IntSet s) {
        int ms[] = new int[size + s.size];
        // copy non-duplicate members and s.members into ms
    }
}
```

Why copy the ms array?

Representation hiding requires not sharing objects with the outside

error: s is an IntSet, and IntSet does not have a member size

Achieving Balanced Performance (version 2)

```
class ArraySet implements IntSet {
    private int members[]; // the array is sorted
    public ArraySet(int ms[], int size) {
        members = new int[size];
        for (int i = 0; i < size; ++i)
            members[i] = ms[i];
    }
    public boolean contains(int element) {
        /* binary search */
    }
    public ArraySet union (ArraySet s) {
        int ms[] = new int[size + s.size];
        // copy non-duplicate members and s.members into ms
    }
}
```

We can fix the error by not implementing IntSet

error: ArraySet does not implement union from the IntSet interface – the argument type differs

ArraySet – is this Good Code?

the special **length** field can be used to get the array size

```
class ArraySet {
    private int members[]; // the array is sorted
    private int temp[]; // for performing unions
    public ArraySet union (ArraySet s) {
        if (temp == null || size + s.size > temp.length)
            temp = new int[size + s.size];
        // copy non-duplicate members and s.members into ms
    }
    // other methods not shown
}
```

ArraySet – is this Good Code?

the special **length** field can be used to get the array size

```
class ArraySet {
    private int members[]; // the array is sorted
    private int temp[]; // for performing unions
    public ArraySet union (ArraySet s) {
        if (temp == null || size + s.size > temp.length)
            temp = new int[size + s.size];
        // copy non-duplicate members and s.members into ms
    }
    // other methods not shown
}
```

***cohesion* means the code for one issue is localized. High cohesion makes code easier to understand and modify.**

This code has low cohesion because data structures used in the union algorithm are spread outside the union method.

~~Using Sets Together~~

Quality Attribute: Different representations should be able to interoperate

Quality Attribute: Support a representation that performs well on all operations

Two quality attributes, interoperability and performance, are in conflict

```
IntSet s1 = new EmptySet();
ArraySet s2 = new ArraySet(new int[] { 5
IntSet temp = s2;
ArraySet s3 = s2.union(s1);
IntSet s4 = s1.union(s2);
```

error: ArraySet is not a subtype of IntSet

error: argument s1 of type IntSet is not a subtype of ArraySet

error: argument s2 of type ArraySet is not a subtype of IntSet

An ArraySet Abstract Data Type (ADT)

```
class ArraySet {
    private int members[]; // the arr
    public ArraySet(int ms[], int size
        members = new int[size];
        for (int i = 0; i < size; ++i)
            members[i] = ms[i];
    }
    public boolean contains(int elem
        /* binary search */
    }
    public ArraySet union (ArraySet s
        int ms[] = new int[size + s.size];
        // copy non-duplicate memb
    }
```

ArraySet is an Abstract Data Type. Its binary operations access objects of the same fixed class type (and therefore the same fixed representation).

This has performance advantages but interoperability and extensibility disadvantages.

union is a *binary operation* because it accepts another set.

Typical application programs prioritize interoperability/ extensibility and therefore prefer interface types over class types.

ArraySet Design Alternatives, Part 1

```
class ArraySet implements IntSet {
    public IntSet union (IntSet s) {
        if (s instanceof ArraySet) {
            ArraySet arrSet = (ArraySet) s;
            // optimized code here
        } else {
            // default code here
        }
    }
    // other methods not shown
}
```

instanceof checks at run time whether s is really an ArraySet

A *cast* lets us treat s as an ArraySet. Java checks (again) that the object really is an ArraySet

- Benefits

- Drawbacks

ArraySet Design Alternatives, Part 1

```
class ArraySet implements IntSet {
    public IntSet union (IntSet s) {
        if (s instanceof ArraySet) {
            ArraySet arrSet = (ArraySet) s;
            // optimized code here
        } else {
            // default code here
        }
    }
}
```

instanceof checks at run time whether s is really an ArraySet

A *cast* lets us treat s as
ks
t

Because extensibility is typically a high priority, object-oriented designers avoid using instanceof

How do we avoid case-based reasoning?
Come up with a higher-level method that unifies the cases, and *dispatch* to it.

- Benefits
  - Provides both interoperability and performance
- Drawbacks
  - Two versions of code are harder to maintain
  - Hard to extend – need new **instanceof** cases for each new rep.

ArraySet Design Alternatives, Part 2

```
interface IntSet {
    IntSet union(IntSet s);
    boolean contains(int element);
    int[] getMembers();
}
```

- Benefits

- Drawbacks

ArraySet Design Alternatives, Part 2

```
interface IntSet {
    IntSet union(IntSet s);
    boolean contains(int element);
    int[] getMembers();
}
```

- Benefits
  - Can implement union efficiently for ArraySet
  - ArraySet is an instance of IntSet and interoperates
- Drawbacks
  - Maybe getMembers() is not a method clients should call
  - Maybe getMembers() is hard for other implementations to implement efficiently

Toad’s Takeaways: Design Goals and Process

- Quality attributes such as extensibility and performance affect design
- Following a process can help with being a more effective designer
- Design guidelines that enhance **quality attributes**
  - Hiding an object’s representation makes it easier to **change representations**
  - Making fields private and copying internal arrays (or mutable objects) is one way to **hide representation**
  - Designing for high cohesion makes code **easier to understand and modify**
  - Programming to interfaces rather than class types facilitates **extensibility** and **interoperability**
  - ADTs, instanceof, and casts can be useful, e.g. for **performance**, but compromise **extensibility** and are discouraged in OO. Use dispatch instead!