Overview of Java’s Support for Polymorphism

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Learning Objectives in this Lesson

- Understand what polymorphism is & how it’s supported in Java
Overview of Java’s Support for Polymorphism
Overview of Java’s Support for Polymorphism

- Inheritance is nearly always used in conjunction with polymorphism
Overview of Java’s Support for Polymorphism

- Polymorphism enables transparent customization of methods inherited from a super class

See en.wikipedia.org/wiki/Polymorphism_(computer_science)
Overview of Java’s Support for Polymorphism

• Polymorphism & inheritance are essential to the “open/closed principle”

See en.wikipedia.org/wiki/Open/closed_principle
Overview of Java’s Support for Polymorphism

- Polymorphism & inheritance are essential to the “open/closed principle”
- “A class should be open for extension, but closed for modification”

See en.wikipedia.org/wiki/Open/closed_principle
Overview of Java’s Support for Polymorphism

- Polymorphism & inheritance are essential to the “open/closed principle”
  - “A class should be open for extension, but closed for modification”
- Insulating clients of a class from modifications makes software more robust, flexible, & reusable

See [www.dre.vanderbilt.edu/~schmidt/OCP.pdf](http://www.dre.vanderbilt.edu/~schmidt/OCP.pdf)
Overview of Java’s Support for Polymorphism

The “open/closed principle” can be applied in conjunction with patterns to enable extensions without modifying existing classes or apps.

See en.wikipedia.org/wiki/Factory_method_pattern
Overview of Java’s Support for Polymorphism

- Subclasses defined in accordance with the open/closed principle can define their own custom behaviors that are more suitable for a particular use case.
Overview of Java’s Support for Polymorphism

• Subclasses defined in accordance with the open/closed principle can define their own custom behaviors that are more suitable for a particular use case
• While still reusing structure & functionality from super class
Overview of Java’s Support for Polymorphism

• Subclasses defined in accordance with the open/closed principle can define their own custom behaviors that are more suitable for a particular use case
  • While still reusing structure & functionality from super class
    • e.g., Vector & ArrayList both inherit AbstractList methods
Overview of Java’s Support for Polymorphism

- Subclasses defined in accordance with the open/closed principle can define their own custom behaviors that are more suitable for a particular use case
  - While still reusing structure & functionality from super class
  - e.g., Vector & ArrayList both inherit AbstractList methods
Overview of Java’s Support for Polymorphism

- Subclasses defined in accordance with the open/closed principle can define their own custom behaviors that are more suitable for a particular use case.
  - While still reusing structure & functionality from super class.
  - E.g., Vector & ArrayList both inherit AbstractList methods.

Methods in each subclass emphasize different List properties.

See beginnersbook.com/2013/12/difference-between-arraylist-and-vector-in-java
Method Overriding in Java Polymorphism
Polymorphism in Java occurs when a reference to a super class is used to refer to an object of a subclass.
Polymorphism in Java occurs when a reference to a super class is used to refer to an object of a subclass.

- Subclass methods can override super class methods.

Any non-final & non-static methods can be overridden.
Polymorphism in Java occurs when a reference to a super class is used to refer to an object of a subclass.
Subclass methods can override super class methods.

See [docs.oracle.com/javase/8/docs/api/java/util/AbstractSet.html](http://docs.oracle.com/javase/8/docs/api/java/util/AbstractSet.html)
Method Overriding in Java Polymorphism

- Polymorphism in Java occurs when a reference to a super class is used to refer to an object of a subclass
- Subclass methods can override super class methods

```
AbstractSet<E>
...
iterator()
add(E e)
```

*Returns an iterator that can access each element in a Set*
Polymorphism in Java occurs when a reference to a super class is used to refer to an object of a subclass.

Subclass methods can override super class methods.

Method Overriding in Java Polymorphism

AbstractSet<E>

... iterator()
add(E e)

Associates the specified element to the Set
Polymorphism in Java occurs when a reference to a super class is used to refer to an object of a subclass.

Subclass methods can override super class methods.

Subclasses of AbstractSet can override add(E e) & iterator() differently.
Polymorphism in Java occurs when a reference to a super class is used to refer to an object of a subclass.

Subclass methods can override super class methods.

Subclasses of AbstractSet have different time & space tradeoffs.
Polymorphism in Java occurs when a reference to a super class is used to refer to an object of a subclass. Subclass methods can override super class methods.
Method Overriding in Java Polymorphism

- Polymorphism in Java occurs when a reference to a super class is used to refer to an object of a subclass
- Subclass methods can override super class methods

See [docs.oracle.com/javase/8/docs/api/java/util/HashSet.html](docs.oracle.com/javase/8/docs/api/java/util/HashSet.html)
Method Overriding in Java Polymorphism

- Polymorphism in Java occurs when a reference to a super class is used to refer to an object of a subclass
- Subclass methods can override super class methods

See Java8/ex19/src/main/java/utils/ConcurrentHashSet.java
// Create AbstractSet subclass
AbstractSet<String> absSet = makeSet(setType);
...

// Dispatch add(E e) method
absSet.add(element);

A factory method creates the appropriate concrete set subclass instance
Method Overriding in Java Polymorphism

// Create AbstractSet subclass
AbstractSet<String>
    absSet = makeSet(setType);

... 

// Dispatch add(E e) method
absSet.add(element);

The appropriate method is dispatched at runtime based on the subclass object
Method Overriding in Java Polymorphism

// Create AbstractSet subclass
AbstractSet<String>
    absSet = makeSet(setType);
...

// Dispatch add(E e) method
absSet.add(element);

The appropriate method is dispatched at runtime based on the subclass object
Method Overriding in Java Polymorphism

// Create AbstractSet subclass
AbstractSet<String>
    absSet = makeSet(setType);
...
// Dispatch add(E e) method
absSet.add(element);

The appropriate method is dispatched at runtime based on the subclass object
AbstractSet<E>

// Create AbstractSet subclass
AbstractSet<String>
    absSet = makeSet(setType);

// Dispatch add(E e) method
absSet.add(element);

Example of Inheritance & Polymorphism in Java
Example of Inheritance & Polymorphism in Java

• The implementation of these methods is selected at run-time based on the object’s type

```java
static void main(String[] args) {
    SimpleAbstractSet<String> set = makeSet(...);
    set.put("I");
    set.put("am");
    set.put("Ironman");
    set.put("Ironman");

    for(Iterator<String> it =
        set.iterator();
        it.hasNext();)
        System.out.println
        ("item = " + iter.next());
}
```

See [github.com/douglascraigschmidt/CS891/tree/master/ex/DynamicBinding](http://github.com/douglascraigschmidt/CS891/tree/master/ex/DynamicBinding)
Example of Inheritance & Polymorphism in Java

- The implementation of these methods is selected at run-time based on the object’s type

```java
static void main(String[] args) {
    SimpleAbstractSet<String> set = makeSet(...);
    set.put("I");
    set.put("am");
    set.put("Ironman");
    set.put("Ironman");

    for(Iterator<String> it = set.iterator();
        it.hasNext();)
        System.out.println("item = " + iter.next());
}
```

Factory method creates a HashSet, TreeSet, or ConcurrentHashMap, etc.
Example of Inheritance & Polymorphism in Java

- The implementation of these methods is selected at run-time based on the object’s type

```java
static void main(String[] args) {
    SimpleAbstractSet<String> set = makeSet(...);

    set.put("I");
    set.put("am");
    set.put("Ironman");
    set.put("Ironman");

    for(Iterator<String> it = set.iterator();
        it.hasNext();)
        System.out.println("item = " + iter.next());
}
```

The put() method & iterator that are dispatched are based on the concrete type of the set
Implementing Dynamic & Static Dispatching in Java
Implementing Dynamic Dispatching in Java

• You needn’t know how polymorphism is implemented to use it properly
You needn’t know how polymorphism is implemented to use it properly.

Understanding how polymorphism works will help you become a “full stack developer”.

See www.laurencegellert.com/2012/08/what-is-a-full-stack-developer
Implementing Dynamic Dispatching in Java

- You needn’t know how polymorphism is implemented to use it properly
  - Understanding how polymorphism works will help you become a “full stack developer”
- Also helps you strike a balance between flexibility & efficiency
Polymorphism is implemented by the Java compiler & Java Virtual Machine
Polymorphism is implemented by the Java compiler & Java Virtual Machine

Implementing Dynamic Dispatching in Java

- Dynamically dispatched methods correspond to an object’s subclass

AbstractSet<E>

... iterator()
add(E e)

TreeSet<E>
... iterator()
add(E e)

HashSet<E>
... iterator()
add(E e)

Concurrent HashSet<E>
... iterator()
add(E e)

See en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)#Subclasses_and_super_classes
Implementing Dynamic Dispatching in Java

- Dynamically dispatched methods correspond to an object’s subclass
- Subclasses can override & customize methods inherited from super classes

See en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)#Subclasses_and_super_classes
Dynamic dispatching is also known as “virtual method invocation”

See docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html

```
AbstractSet<E>
... iterator()
add(E e)
```
```
HashSet<E>
... iterator()
add(E e)
```
```
ConcurrentHashSet<E>
... iterator()
add(E e)
```
```
TreeSet<E>
... iterator()
add(E e)
```
Implementing Dynamic Dispatching in Java

- Dynamic dispatching is also known as “virtual method invocation”

Dynamic dispatching is the default method dispatching mechanism in Java.
Dynamic dispatching is also known as “virtual method invocation”

A subclass can override any method defined in a super class.
Dynamic dispatching is also known as “virtual method invocation”

- A subclass can override any method defined in a super class
- Unless that method is declared as private, final, or static
Implementing Dynamic Dispatching in Java

- Each Java class typically has a virtual table (vtable) that contains addresses of dynamically dispatched methods.

```
<table>
<thead>
<tr>
<th>Pointer to vtable</th>
<th>Pointer to method_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>keySet field_1</td>
<td>Pointer to method_2</td>
</tr>
<tr>
<td></td>
<td>Pointer to method_n</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Pointer to vtable</th>
<th>Pointer to method_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>keySet field_1</td>
<td>Pointer to method_2</td>
</tr>
<tr>
<td></td>
<td>Pointer to method_n</td>
</tr>
</tbody>
</table>
```

HashSet

```
<table>
<thead>
<tr>
<th>Pointer to vtable</th>
<th>Pointer to method_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>keySet field_1</td>
<td>Pointer to method_2</td>
</tr>
<tr>
<td></td>
<td>Pointer to method_n</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Pointer to vtable</th>
<th>Pointer to method_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>keySet field_1</td>
<td>Pointer to method_2</td>
</tr>
<tr>
<td></td>
<td>Pointer to method_n</td>
</tr>
</tbody>
</table>
```

AbstractSet

```
<table>
<thead>
<tr>
<th>Pointer to vtable</th>
<th>Pointer to method_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>keySet field_1</td>
<td>Pointer to method_2</td>
</tr>
<tr>
<td></td>
<td>Pointer to method_n</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Pointer to vtable</th>
<th>Pointer to method_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>keySet field_1</td>
<td>Pointer to method_2</td>
</tr>
<tr>
<td></td>
<td>Pointer to method_n</td>
</tr>
</tbody>
</table>
```

See [en.wikipedia.org/wiki/Virtual_method_table](en.wikipedia.org/wiki/Virtual_method_table)
Implementing Dynamic Dispatching in Java

- Each Java object contains a pointer to the vtable (vptr) of its class

See en.wikipedia.org/wiki/Polymorphism_(computer_science)#Subtyping
Implementing Dynamic Dispatching in Java

- Each Java object contains a pointer to the vtable (vptr) of its class.
- The JVM uses the vptr to locate & dynamically dispatch a virtual method.

```
HashSet 0 4 8 12 16 20 ...
| Pointer to vtable | keySet field1 | values field2 | table field1 | iterator field2 | size field3 | other fields...
|-------------------|---------------|---------------|--------------|----------------|-------------|---------------
|
```

```
AbstractSet 0 4 8
<table>
<thead>
<tr>
<th>Pointer to vtable</th>
<th>keySet field1</th>
<th>values field2</th>
</tr>
</thead>
</table>
```

```
Method1 bytecode ← Pointer to method1
Method2 bytecode ← Pointer to method2
Methodn bytecode ← Pointer to methodn
```
Implementing Dynamic Dispatching in Java

- Each Java object contains a pointer to the vtable (vptr) of its class.
- The JVM uses the vptr to locate & dynamically dispatch a virtual method.

```
<table>
<thead>
<tr>
<th>Pointer to vtable</th>
<th>Pointer to method_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>keySet field_1</td>
<td>Pointer to method_2</td>
</tr>
<tr>
<td>values field_2</td>
<td>Pointer to method_n</td>
</tr>
<tr>
<td>table field_1</td>
<td></td>
</tr>
<tr>
<td>iterator field_2</td>
<td></td>
</tr>
<tr>
<td>size field_3</td>
<td></td>
</tr>
<tr>
<td>other fields...</td>
<td></td>
</tr>
</tbody>
</table>
```

See [en.wikipedia.org/wiki/Late_binding#Late_binding_in_Java](en.wikipedia.org/wiki/Late_binding#Late_binding_in_Java)
Java also supports static method dispatching, where a method implementation is selected at compile-time.

AbstractMap<K, V>

...
entrySet()
put(K, V)
eq(Object, Object)

See en.wiktionary.org/wiki/static_dispatch
Statically dispatched methods can be implemented & optimized more efficiently.

Java also supports static method dispatching, where a method implementation is selected at compile-time.

- e.g., Java private, final, & static methods

```
AbstractMap<K, V>

... entrySet() put(K, V) eq(Object, Object)
```
Implementing Static Dispatching in Java

- Java also supports static method dispatching, where a method implementation is selected at compile-time
- e.g., Java private, final, & static methods

```
static boolean eq(Object o1, Object o2) {
    return o1 == null ? o2 == null : o1.equals(o2);
}
```

See docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html
Implementing Static Dispatching in Java

- Statically dispatched methods play an important role in Java apps that value performance more than extensibility.

E.g., apps where the right answer delivered too late becomes the wrong answer.
End of Overview of Java’s Support for Polymorphism