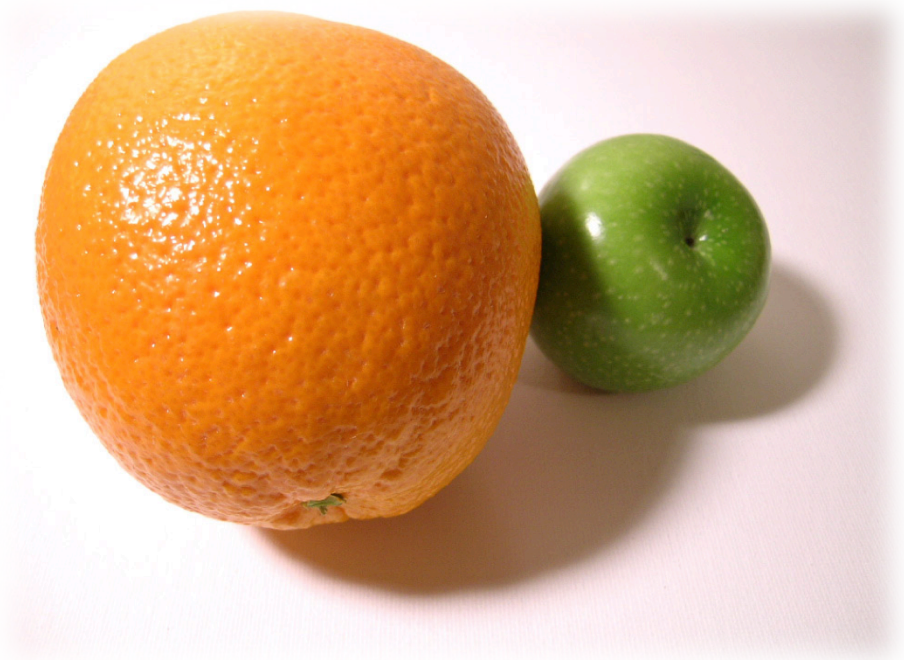
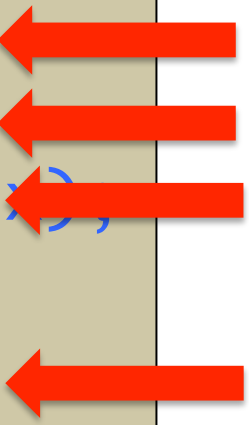
equals() method



The equals method in a super class and its subclass(es)

```
Class SuperOne
private field x
...
public boolean equals(Object other){
   if (other instanceof SuperOne){
      othersuper = (SuperOne)other;
      return (this.x == othersuper.x);
   }
   else return false;
}
```



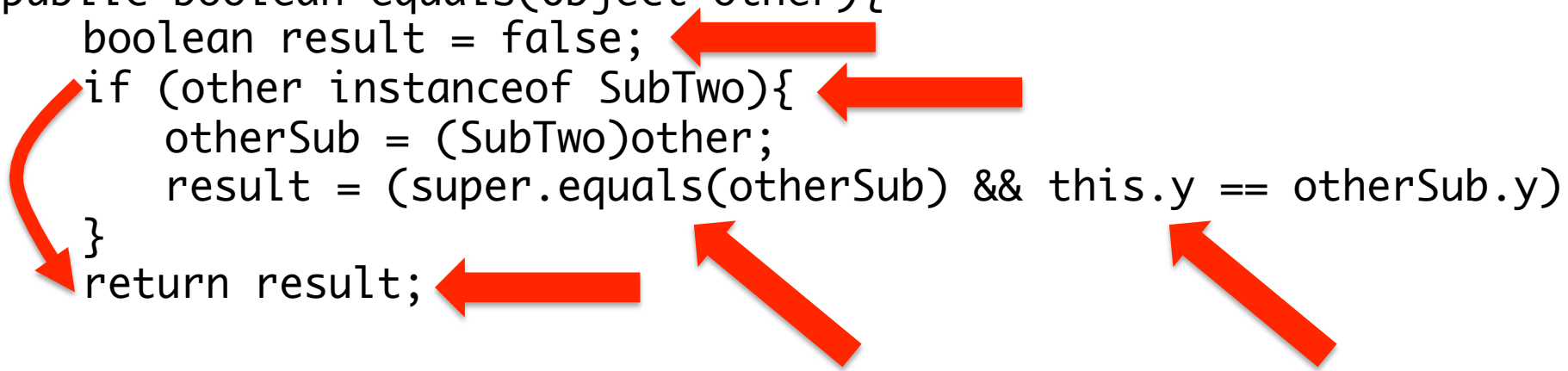
The equals method in a super class and its subclass(es)

```
Class SubTwo extends SuperOne
private field y
...
public boolean equals(Object other){
```

if the object is of type SubTwo
check that they are equal as SuperOne objects
then check any remaining fields or conditions of the SubTwo class in this method

The equals method in a super class and its subclass(es)

```
Class SubTwo extends SuperOne
private field y
...
public boolean equals(Object other){
    boolean result = false;
    if (other instanceof SubTwo){
        otherSub = (SubTwo)other;
        result = (super.equals(otherSub) && this.y == otherSub.y)
    }
    return result;
```



- **How**
- **Why**
- **Why** it works

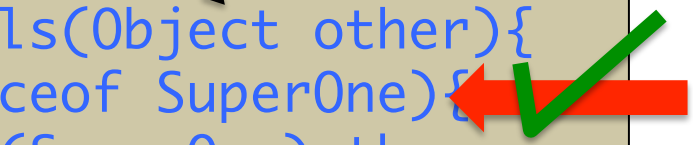
- **How**
  - use `super.equals(`*other object*`)` to call the equals method of the
- **Wh**
  - b
- **Why** works

refactor:
to change the implementation details of code without changing it's behavior (as experienced by users of the class)

Why does this work?

```
result = (super.equals(otherSub)...
```

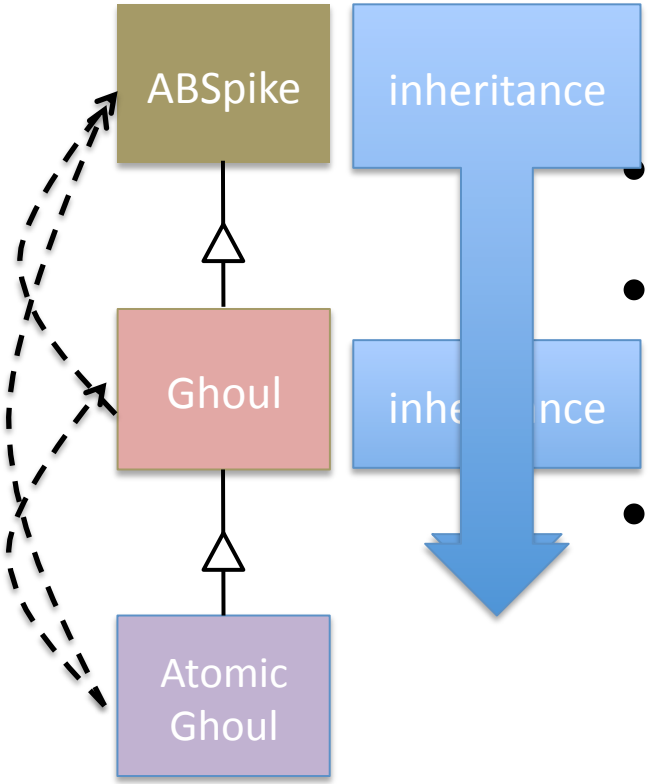
```
Class SuperOne
private field x
...
public boolean equals(Object other){
   if (other instanceof SuperOne){
      othersuper = (SuperOne)other;
      return (this.x == othersuper.x);
   }
   else return false;
}
```



- **How**
  - use `super.equals(`*other object*`)` to call the equals method of the super class
- **Wh**
  - be
- **Why** it works
  - polymorphism

polymorphism:
the ability for an object to take on the identity of any of its ancestor classes

Polymorphism



- An AtomicGhoul is a Ghoul
- An AtomicGhoul is an ABSpike
- A Ghoul is an ABSpike

A few rules on method calls

- a subclass file can call a method from its superclass

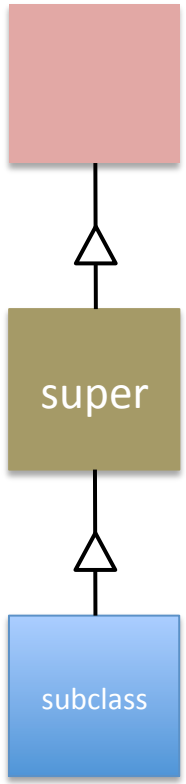
```
setTotal(num, 0.07);
```

- an object in a subclass can invoke a method from its

```
myObject.getTotal();
```

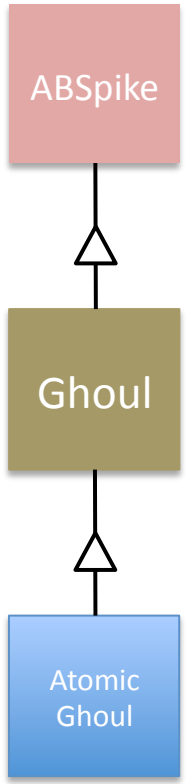
A few rules on method calls

- when an object invokes a method that is inherited, the compiler will *look*
  - in the subclass file
  - then in the super class file one level above
  - then in the super class file two levels above
  - ...until it finds a definition for the method



A few rules on method calls

- when an object calls a method that is overloaded, the compiler will *look* for the method with the matching parameter list
  - in the subclass file
  - then in the super class file one level above
  - then in the super class file two levels above
  - ...until it finds a *match*
  - the return type may also be different between overloaded methods, but that can't be the *only* thing that differs between them

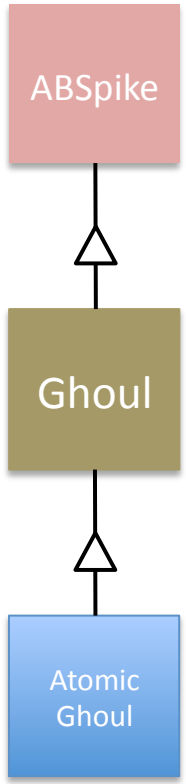


A few rules on method calls

- When an object calls a method that is overridden

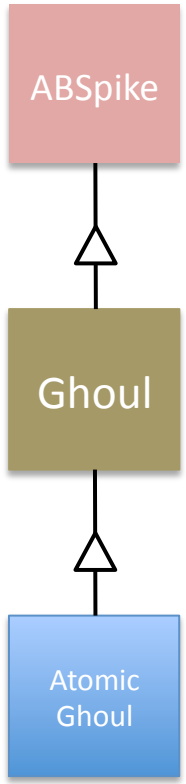
```
Ghoul Igor = new Ghoul();
AtomicGhoul Claw = new AtomicGhoul();
Igor.move(8,2);
Claw.move(-13,9);
```

  - still keep things simple for client files



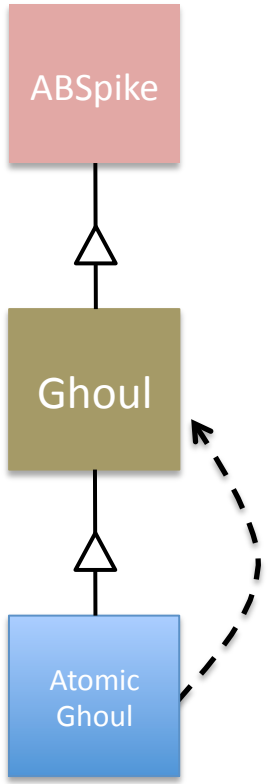
A few rules on method calls

- when an object calls a method that is overriden, the compiler will *look*
  - in the subclass file
  - then in the super class file one level above
  - then in the super class file two levels above
  - ...using the first definition it finds



A few rules on method calls

- If an object wishes to call the super class definition of a method that is overriden in the subclass, this must must be done in the class file first



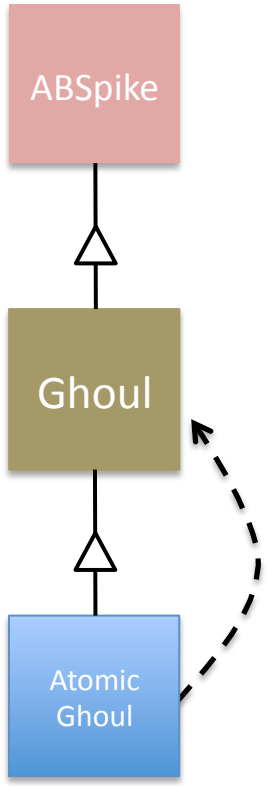
in the super class

```
public String toString(){
   return "StringA";
}
```

in the subclass

```
public String toString(){
   return ("String B");
}
public String aANDb(){
   return (super.toString() + toString());
}

public String onlyA{
   return(super.toString());
}
```



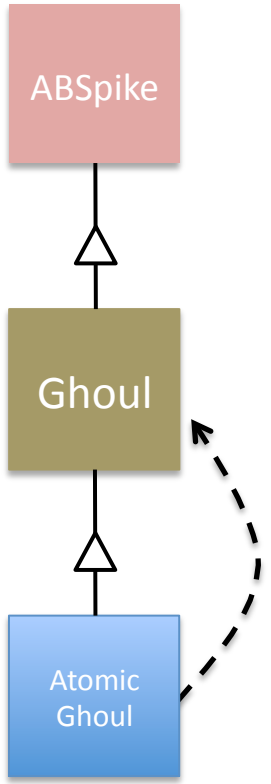
```
in the parent class
public String toString(){
   return "StringA";
}

in the s
public
   ret
}
public
   ret
}

public String onlyA{
   return(super.toString());
}
```

```
SubClass x = new SubClass();
out.println(x); //an implicit call
out.println(x.aANDb());
out.println(x.onlyA();

output:
String B //from the subclass only
StringAStringB //calls both versions
StringA //from the super class only
```



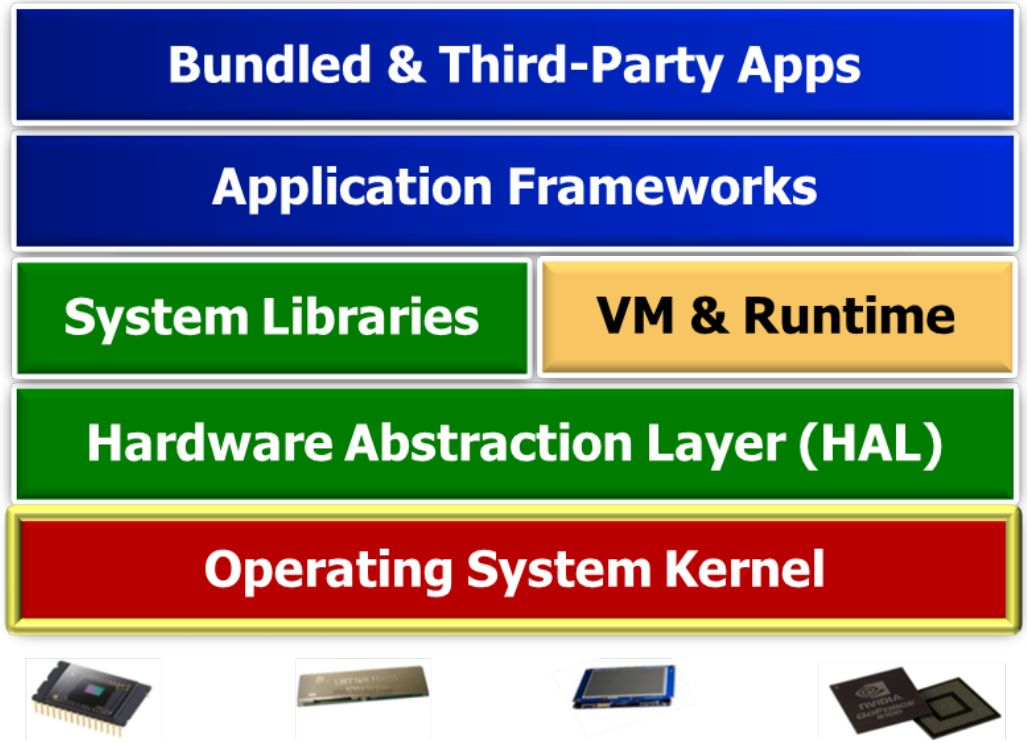

ABSTRACTION
Atomic
Ghoul

Implementing Dynamic Dispatching in Java

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

Implementing Dynamic Dispatching in Java

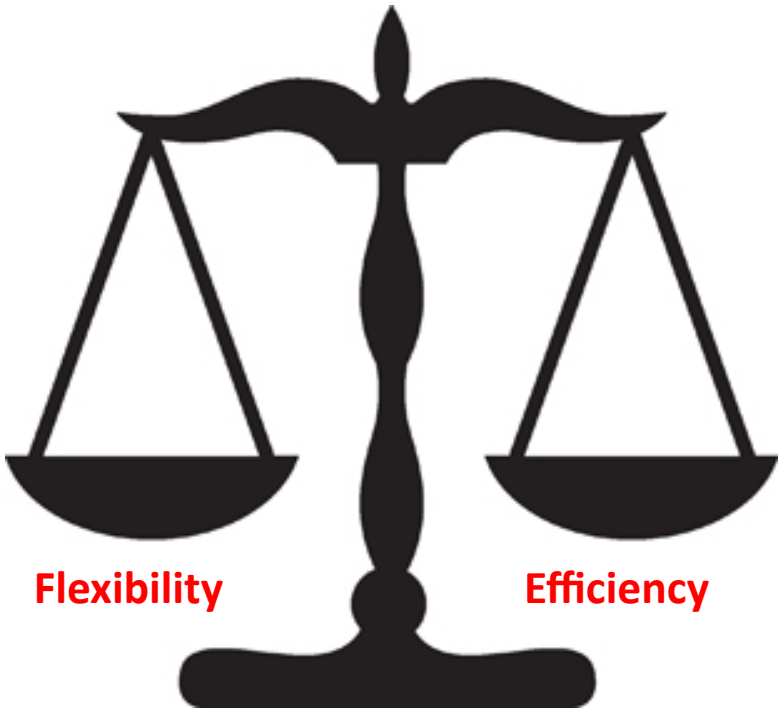
- You needn't know how polymorphism is implemented to use it properly
  - Understanding how polymorphism works will help you become a "full stack developer"



See www.laurencegellert.com/2012/08/what-is-a-full-stack-developer

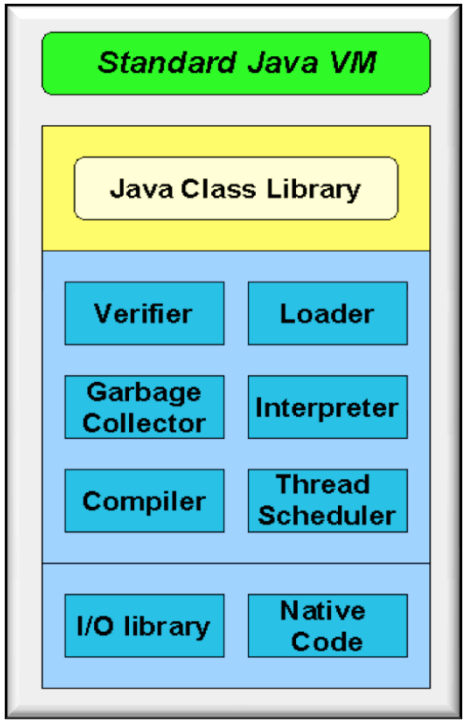
Implementing Dynamic Dispatching in Java

- 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



Implementing Dynamic Dispatching in Java

- Polymorphism is implemented by the Java compiler & Java Virtual Machine



invokevirtual

Operation

Invoke instance method; dispatch based on class

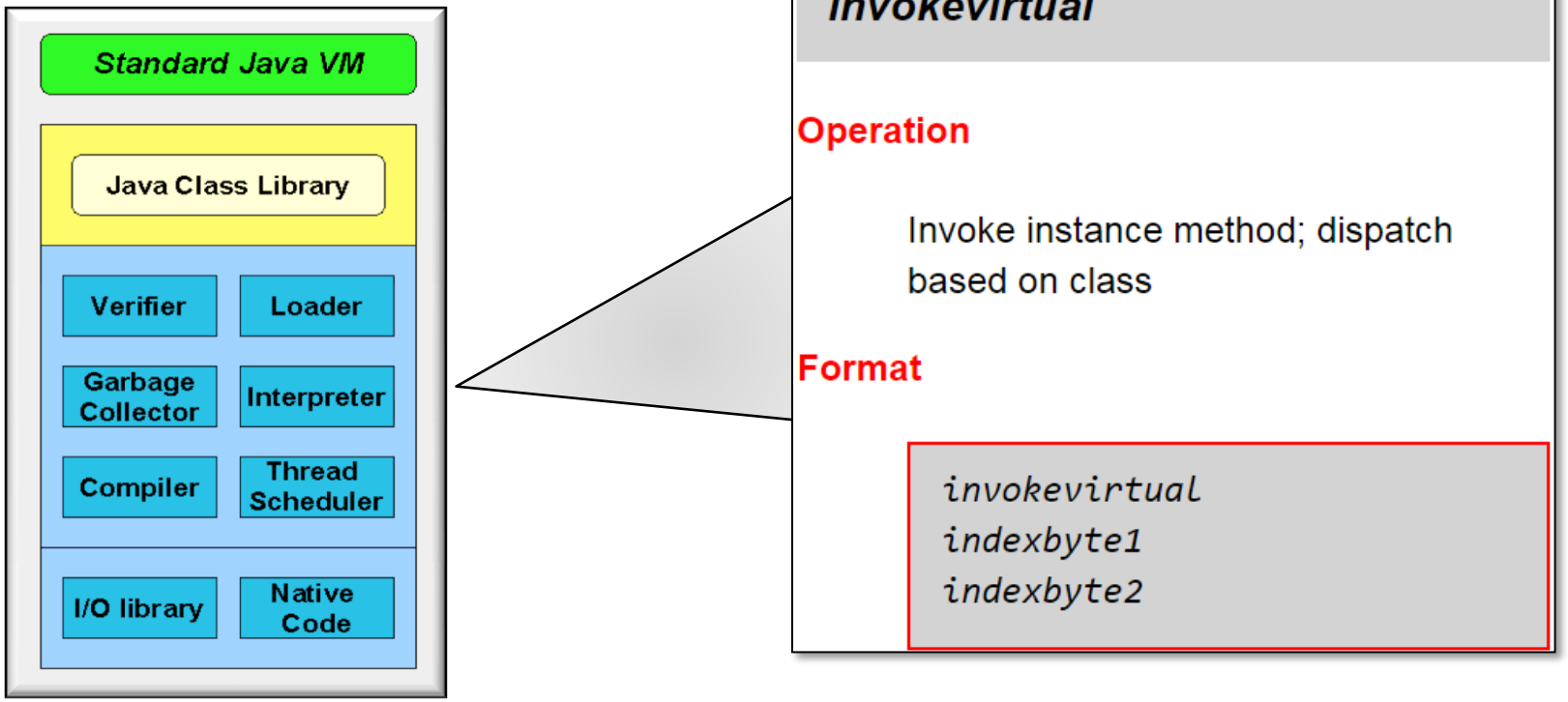
Format

```
invokevirtual
indexbyte1
indexbyte2
```

See docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html#jvms-6.5.invokevirtual

Implementing Dynamic Dispatching in Java

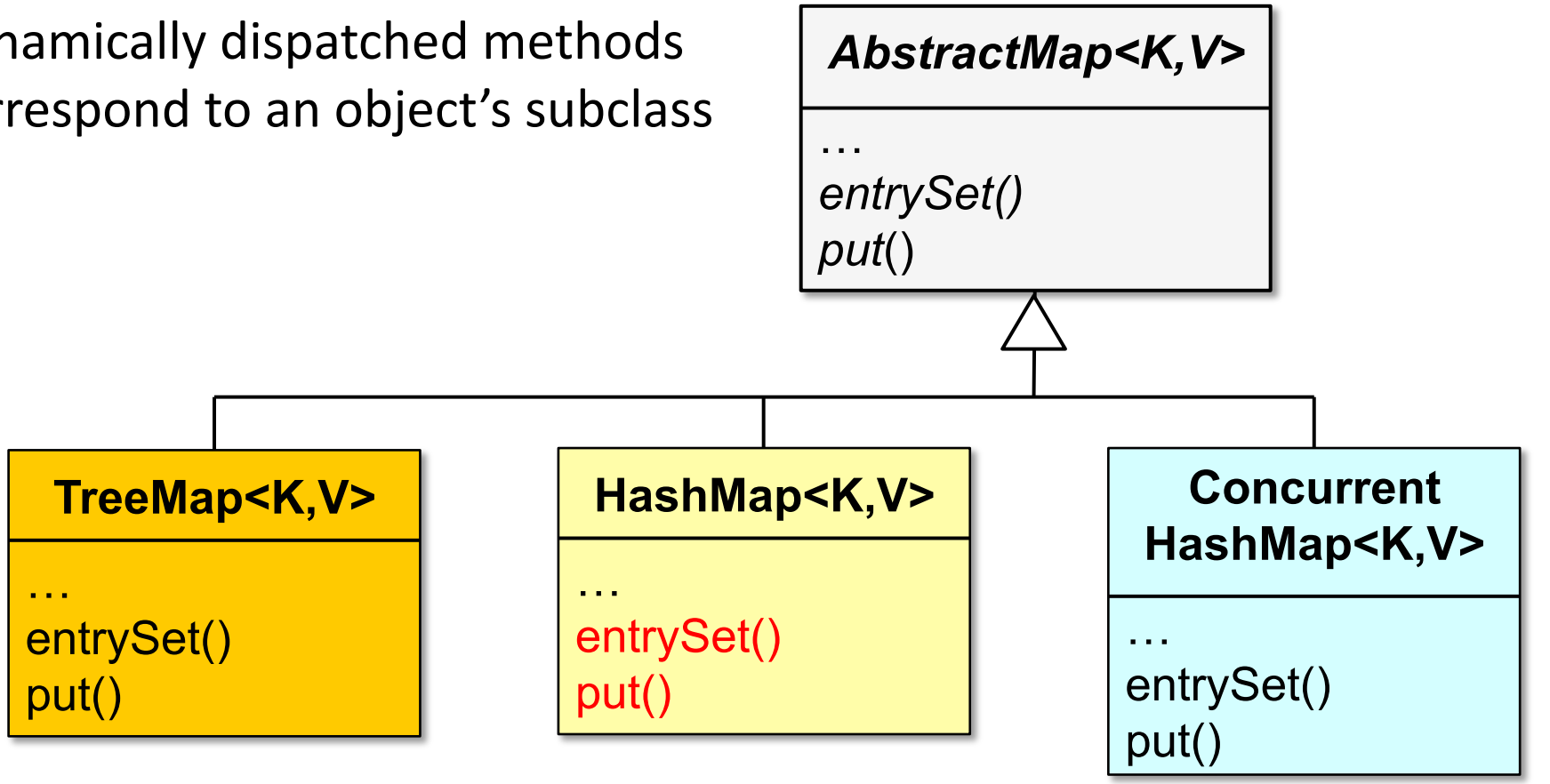
- Polymorphism is implemented by the Java compiler & Java Virtual Machine



See en.wikipedia.org/wiki/Dynamic_dispatch

Implementing Dynamic Dispatching in Java

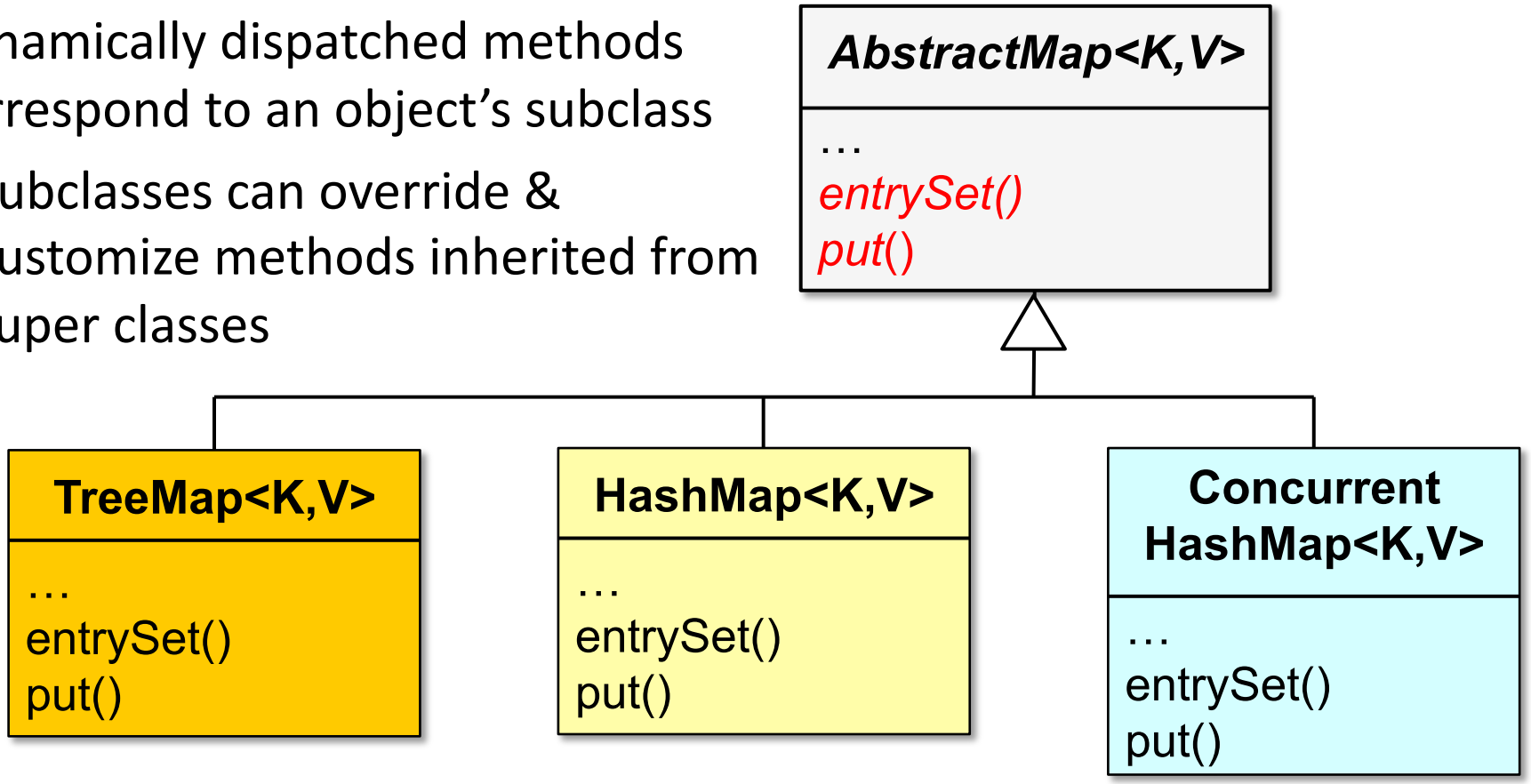
- Dynamically dispatched methods correspond to an object's subclass



See en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)#Subclasses_and_superclasses

Implementing Dynamic Dispatching in Java

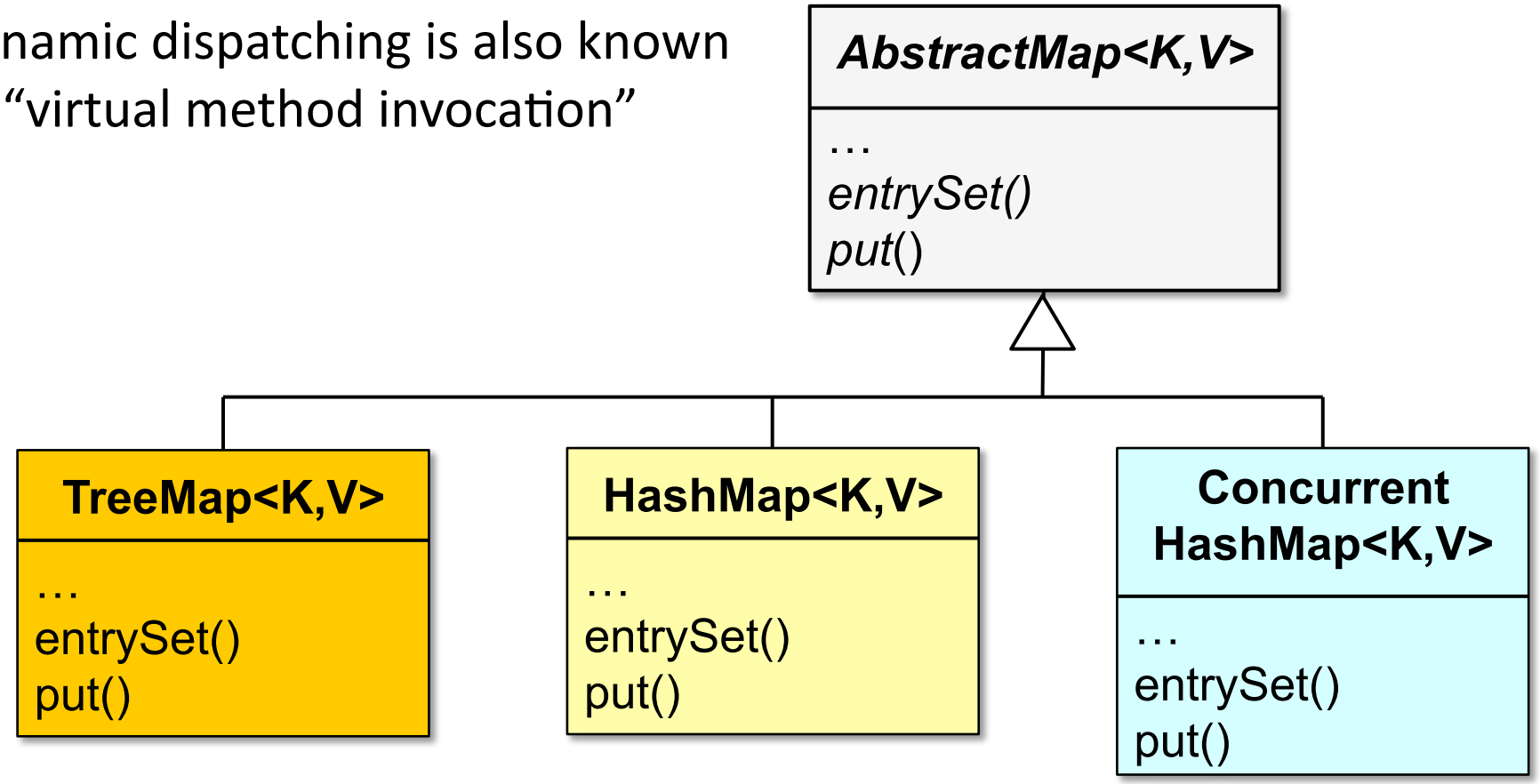
- Dynamically dispatched methods correspond to an object's subclass
  - Subclasses can override & customize methods inherited from super classes



See en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)#Subclasses_and_superclasses

Implementing Dynamic Dispatching in Java

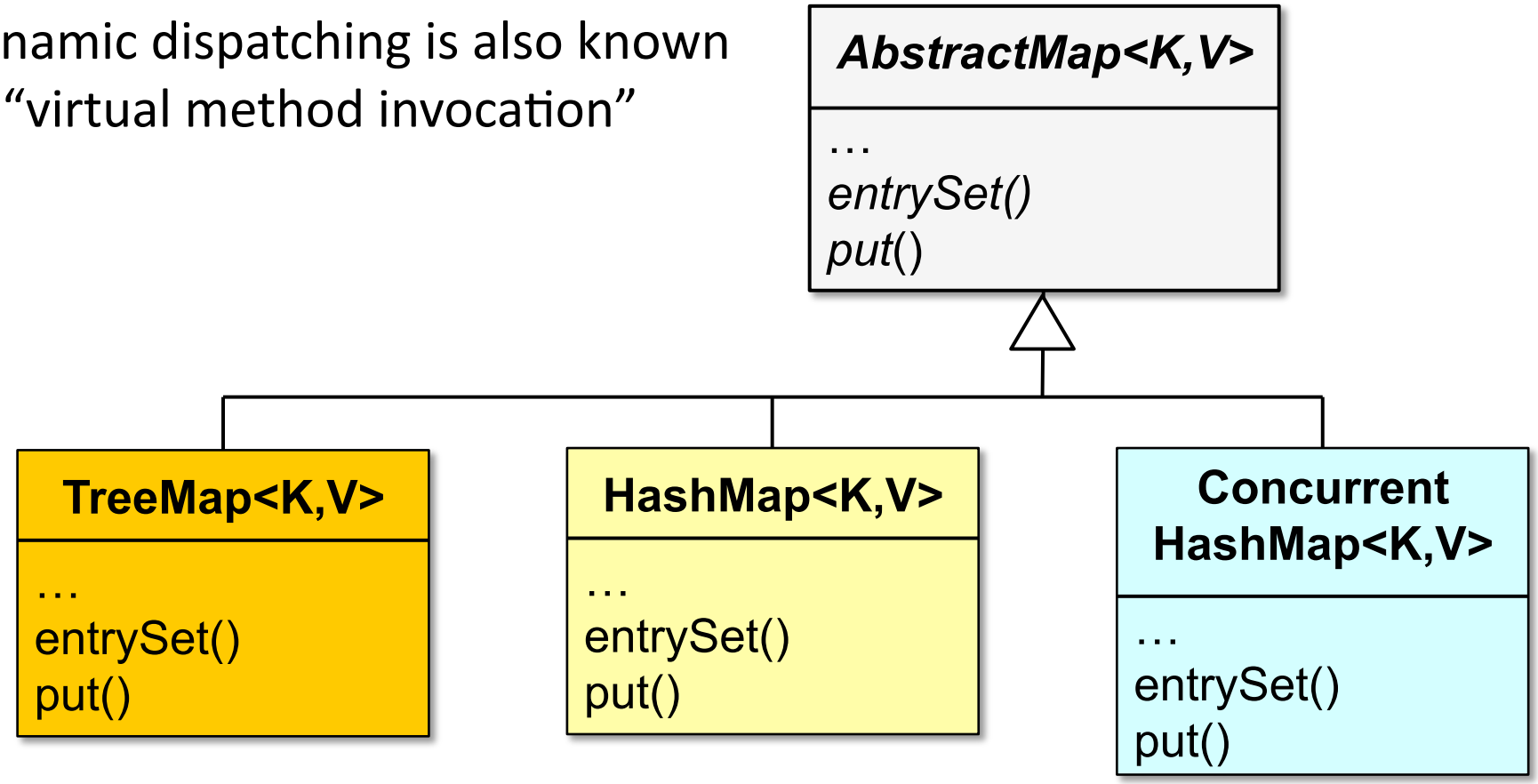
- Dynamic dispatching is also known as "virtual method invocation"



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

Implementing Dynamic Dispatching in Java

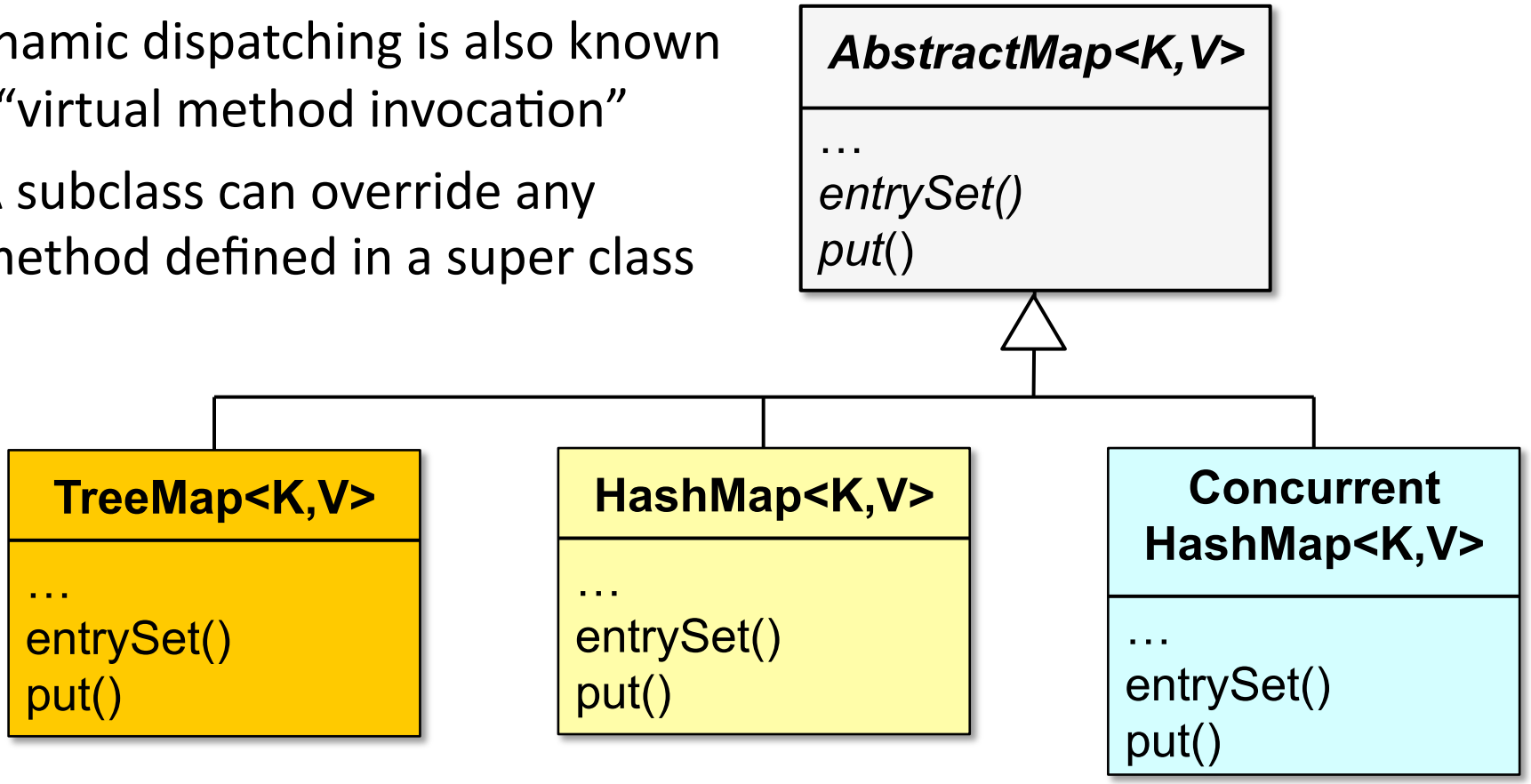
- Dynamic dispatching is also known as "virtual method invocation"



Dynamic dispatching is default mechanism for dispatching Java methods

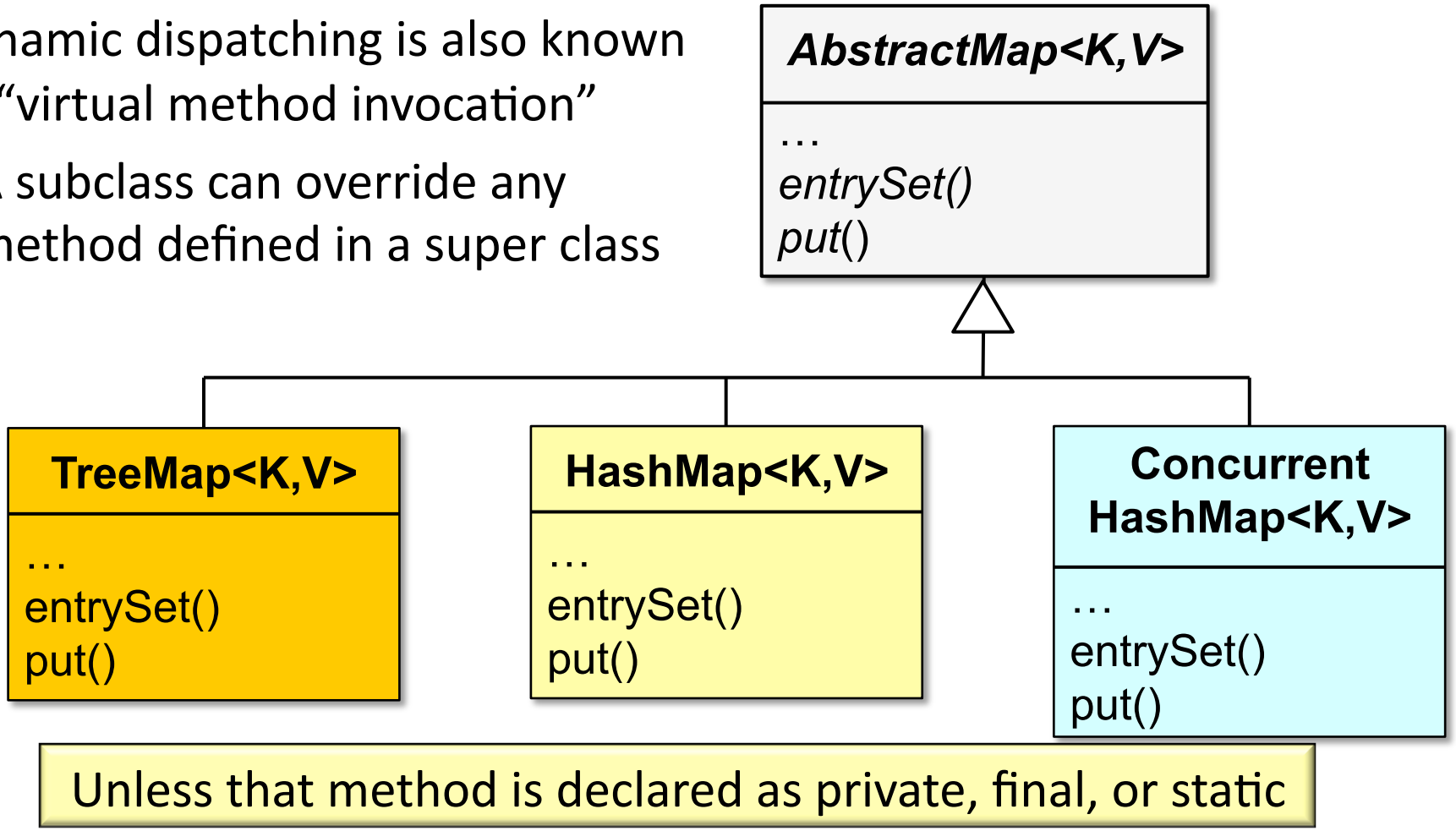
Implementing Dynamic Dispatching in Java

- Dynamic dispatching is also known as "virtual method invocation"
  - A subclass can override any method defined in a super class



Implementing Dynamic Dispatching in Java

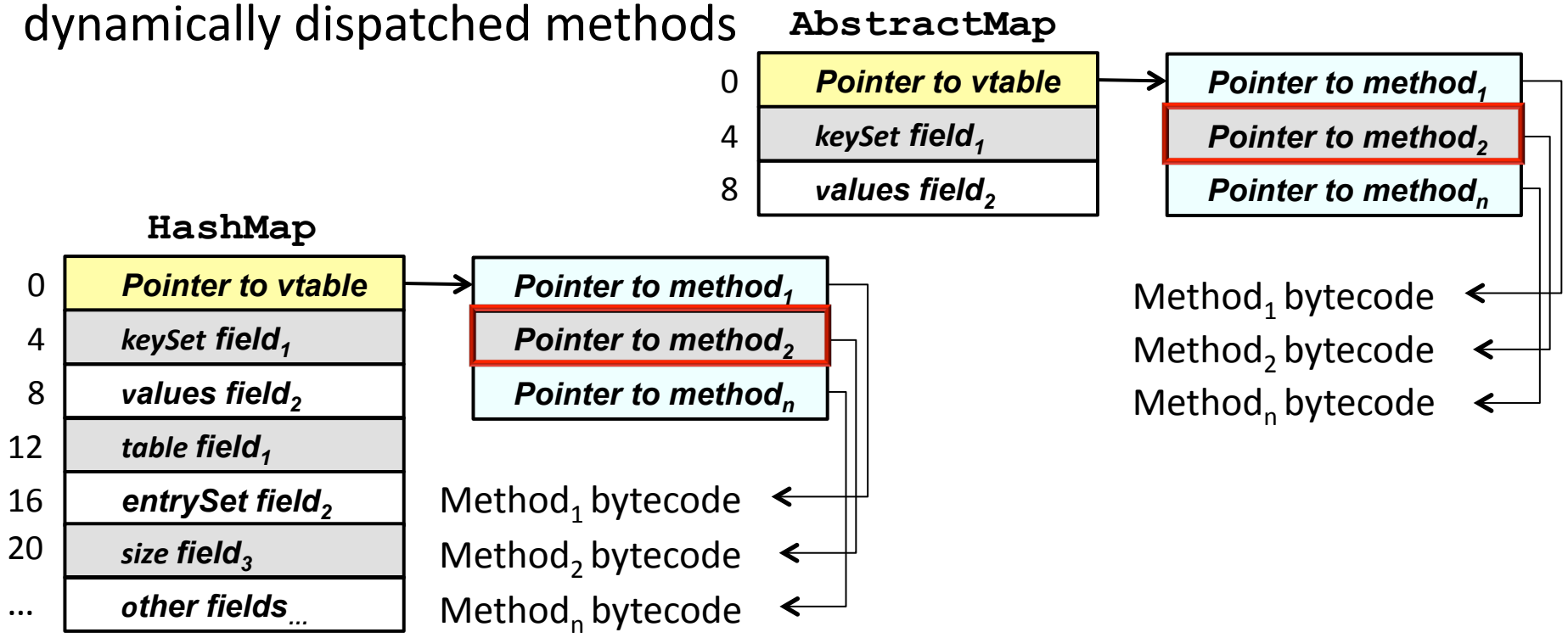
- Dynamic dispatching is also known as "virtual method invocation"
  - A subclass can override any method defined in a super class



Unless that method is declared as private, final, or static

Implementing Dynamic Dispatching in Java

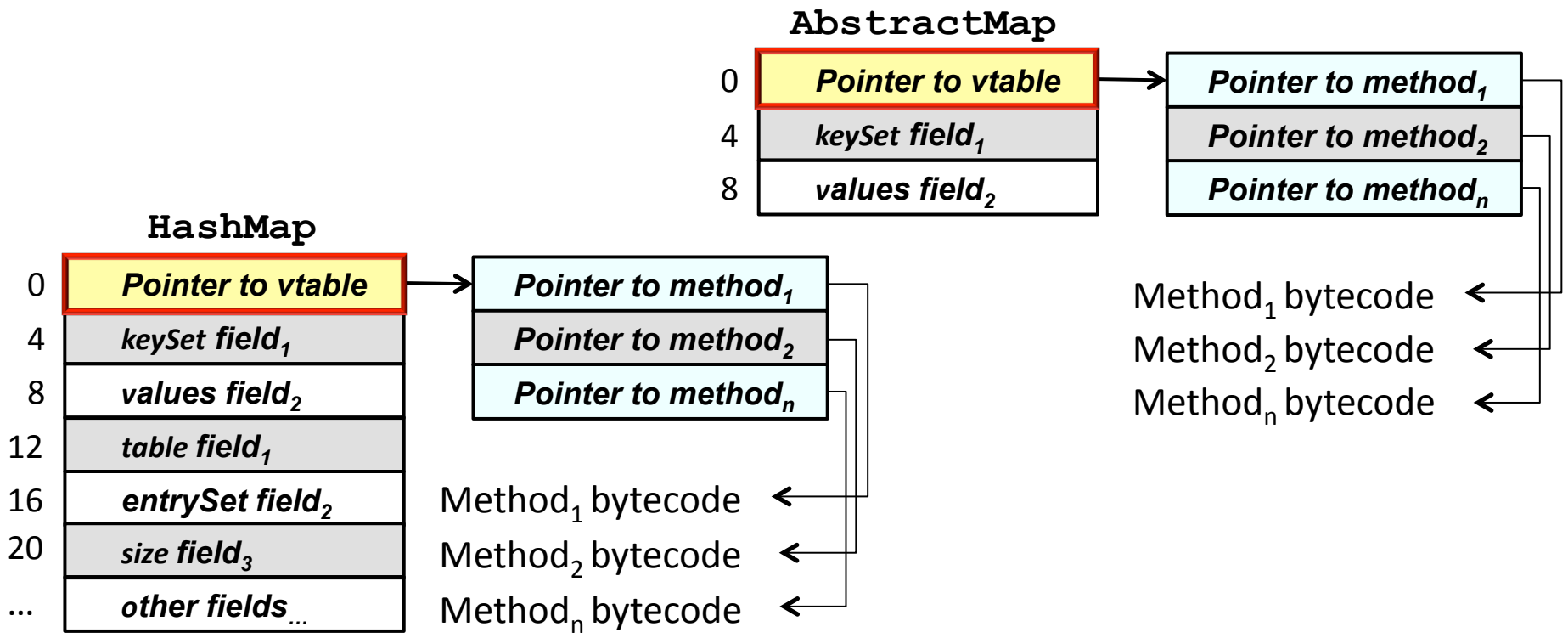
- Each Java class has a virtual table (vtable) that contains addresses of dynamically dispatched methods



See en.wikipedia.org/wiki/Virtual_method_table

Implementing Dynamic Dispatching in Java

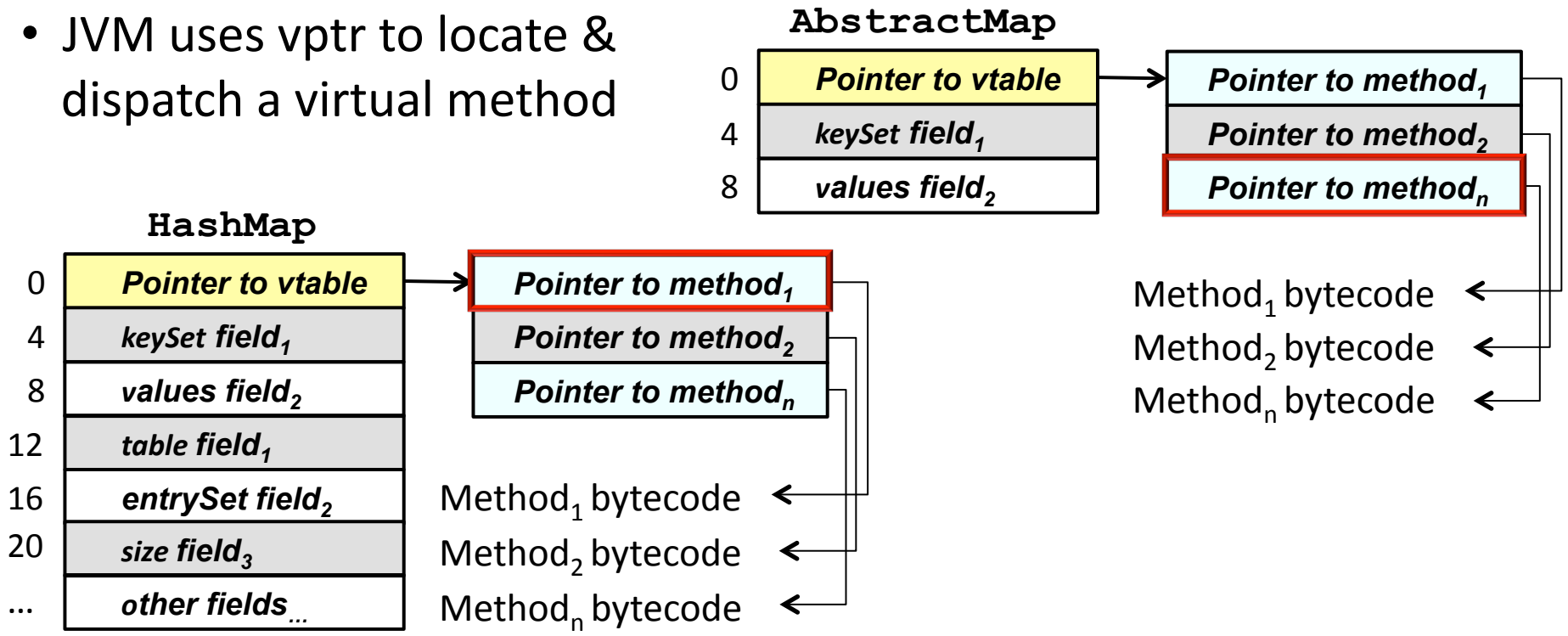
- Each Java object contains a pointer to the vtable (vptr) of its class



See en.wikipedia.org/wiki/Polymorphism_(computer_science)#Subtyping

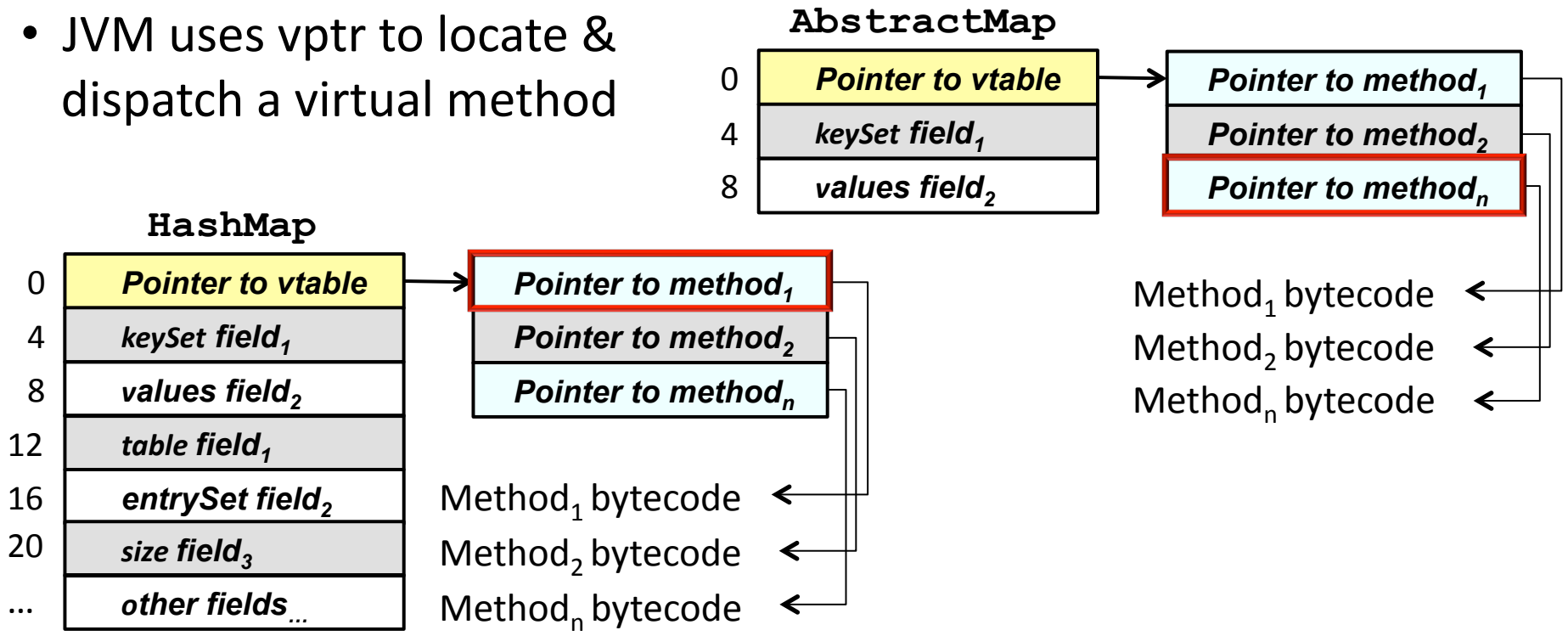
Implementing Dynamic Dispatching in Java

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



Implementing Dynamic Dispatching in Java

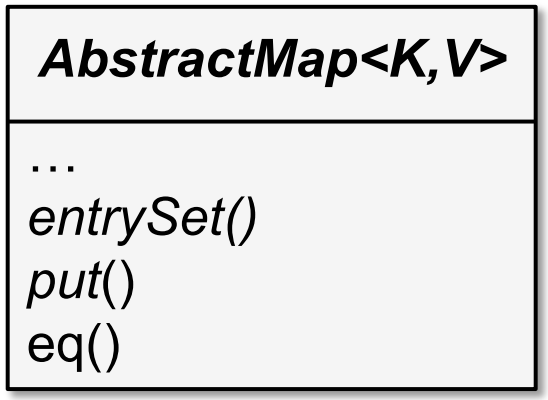
- Each Java object contains a pointer to the vtable (vptr) of its class
  - JVM uses vptr to locate & dispatch a virtual method



See en.wikipedia.org/wiki/Late_binding#Late_binding_in_Java

Implementing Static Dispatching in Java

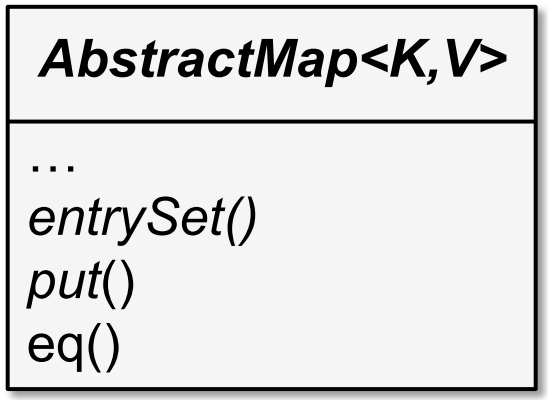
- Java also supports static method dispatching, where implementation of a method is selected at compile-time



See en.wiktionary.org/wiki/static_dispatch

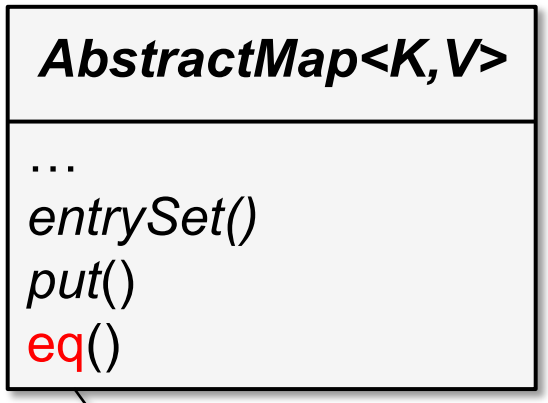
Implementing Static Dispatching in Java

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



Implementing Static Dispatching in Java

- Java also supports static method dispatching, where implementation of a method 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);
}
```

Statically dispatched methods can be implemented & optimized efficiently

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