Java Programming

Arthur Hoskey, Ph.D.
Farmingdale State College
Computer Systems Department

- Polymorphism
- Interfaces

Today's Lecture

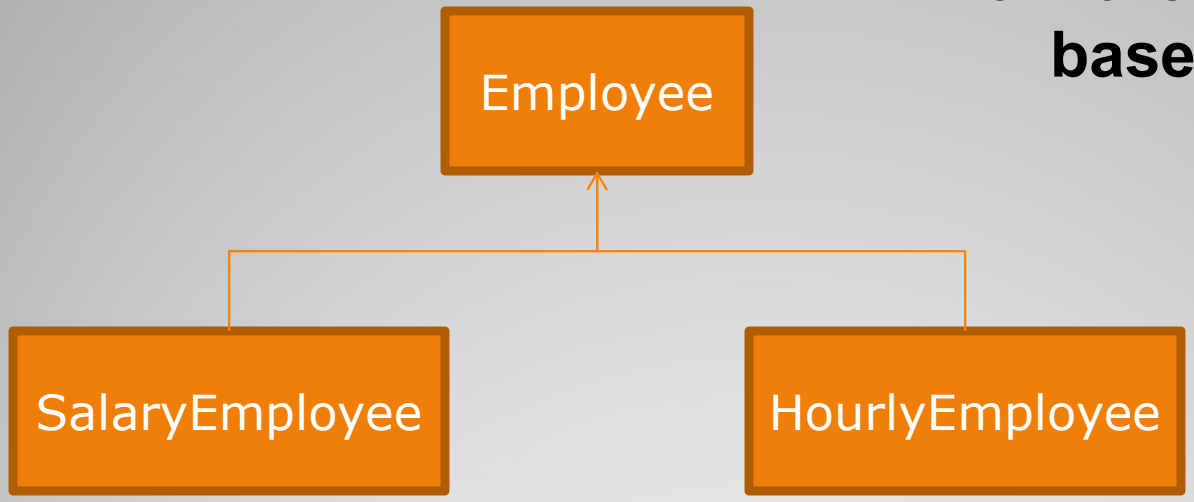
Inheritance - REVIEW

- What is inheritance?
- A form of code reuse.
- Create a new class from an existing class.
- Use an existing class as a “base” for the new class.
- The new class adds on to the existing class.

Inheritance - REVIEW

- **Inheritance**: **"is-a"** relationship
- A derived class "is-a" type of the base class.
- **"A dog is an animal"**
- Base classes are more general than derived classes.

Derive new classes from the Employee base class.



Inheritance - REVIEW

```
public class Employee
{
        private int m_Id;

        public Employee(int newId)
        { m_Id = newId; }

        public int GetId()
        { return m_Id; }

        public void SetId(int newId)
        { m_Id = newId; }
}
```

Employee is base class. There are no changes to the Employee class.

The SalaryEmployee class will just add to it *without changing it*.

Inheritance - REVIEW

```
class SalaryEmployee extends Employee
{
        private double m_YearlySalary;

        public SalaryEmployee(double newSal)
        { m_YearlySalary = newSal; }

        public double GetYearlySalary()
        { return m_YearlySalary; }

        public void SetYearlySalary(double newSal)
        { m_YearlySalary = newSal; }
}
```

Inheritance - REVIEW

- Overload
  - ***Same method name*** BUT ***different signature***.

- Override
  - ***Same method name*** AND ***same signature***
  - Different implementation between classes in the inheritance hierarchy.
  - One implementation of the method is in the base class and the other is in the derived class.

Overloading Vs Overriding

Overload (different signature)	Override (same signature) Base Vs Derived
Employee::Show()	Employee::Show()
Employee::Show(int num)	SalaryEmpolyee::Show()
Employee::Show(string name)	HourlyEmployee::Show()

OVERLOAD
Each implementation of Show() has a DIFFERENT signature.

OVERRIDE
ALL implementations of Show() have the SAME signature. Each implementation of Show() differs between the base and derived class.

Overloading Vs Overriding

- **Polymorphism** means **"many forms"** in Greek.

- **IMPORTANT!!!**
  Overloading AND overriding are two examples of ***polymorphism*** in programming.

- There were many different forms of the Show() method on the previous slide.

Polymorphism

- Write programs that process objects that share the same base class in a class hierarchy.
- Create an abstract base class that other classes can derive from.
- The abstract base class defines the common behavior that we care about.
- ***Put common behavior in the abstract base class.***
- ***Program to the common behavior.***

Polymorphism

- An abstract class is like a “template”.
- In MS Word you have different templates for different types of documents.
- You use the template as a starting off point for your document.

Polymorphism

- Similarly, in programming you can ***use a base class as a jumping off point*** for your other classes.

- ***Design your code to handle the functionality defined in the base class.***

Polymorphism

- Now revisit the employee salary example.
- We will design the main method so that it deals with Employee instances.
- We will do the following:
  - Change Employee to abstract.
  - Define common behavior in base class (Employee).
  - Override common behavior in derived classes.

Polymorphism

```
public abstract class Employee
{
   protected double salary;

   public Employee(double newSalary)
   { salary = newSalary; }

   public double GetSalary()
   { return salary; }

   public void SetSalary(double newSalary)
   { salary = newSalary; }

   public abstract void ShowWeeklySalary(); // Derived classes
}                                           //MUST override this
```

Polymorphism

```
public class HourlyEmployee extends Employee
{
   public HourlyEmployee(double newSalary)
   {
         super(newSalary);
   }

   // OVERRIDE Employee::ShowWeeklySalary()
   @Override
   public void ShowWeeklySalary()
   {
         double weeklySalary = salary * 40;

         System.out.printf("Hourly Rate   = $%.2f\n", salary);
         System.out.printf("Weekly Salary = $%.2f\n",weeklySalary);
   }
}
```

@Override will cause a compile error to appear if the method being overrided does not exist on a base class (helps with spelling mistakes).

@Override is NOT required. Program will run fine without it.

Polymorphism

```
public class SalaryEmployee extends Employee
{
   public SalaryEmployee(double newSalary)
   {
         super(newSalary);
   }

   // OVERRIDE Employee::ShowWeeklySalary()
   @Override
   public void ShowWeeklySalary()
   {
         double weeklySalary = salary / 52.0;

         System.out.printf("Yearly Rate   = $%.2f\n", salary);
         System.out.printf("Weekly Salary = $%.2f\n",weeklySalary);
   }
}
```

Polymorphism

```
public static void main(String[] args)
{
        //Employee e = new Employee(30); // NOT ALLOWED.
                                          // Employee is abstract!!!

        Employee e1 = new SalaryEmployee(52000);
        Employee e2 = new HourlyEmployee(20);

        System.out.println("Weekly Salary Report");
        System.out.println("--------------------");

        e1.ShowWeeklySalary();
        e2.ShowWeeklySalary();
}
```

Polymorphism

- How does the computer know which version of ShowWeeklySalary() to call?

Polymorphism

- Answer:
  The underlying type determines which version of the method to call.

```
Employee e1 = new SalaryEmployee(52000);
Employee e2 = new HourlyEmployee(20);

// Calls SalaryEmployee::ShowWeeklySalary()
e1.ShowWeeklySalary();

// Calls HourlyEmployee::ShowWeeklySalary()
e2.ShowWeeklySalary();
```

Polymorphism

- Show the Employee example code running…

Polymorphism

- What does the following code cause to happen?

```
class B {
        private int salary;
        public B()  // Base constructor
        { salary = 0; }
}

class D extends B {
        @Override
        public D() // Derived constructor
        {    }
}
public class Driver {
        public static void main(String[] args) {
                D d = new D();
        }
}
```

Can you override the base class constructor???

Polymorphism

- What does the following code cause to happen?

```
class B {
        private int salary;
        public B()  // Base constructor
        { salary = 0; }
}

class D extends B {
        @Override
        public D() // Derived constructor
        {    }
}
public class Driver {
        public static void main(String[] args) {
                D d = new D();
        }
}
```

Can you override the base class constructor???

NO. Cannot override base class constructor!

1. **To override you need to use the same name and parameter list.**
2. **Only one of the overridden methods in the inheritance hierarchy runs. We always need both constructors to run no matter what.**

Polymorphism

Interfaces

- **Defines a set of behaviors.**
- Classes **implement** interfaces.
- If a class implements an interface it guarantees that the methods in the interface will be implemented.
- Cannot call new on an interface but you can declare interface type variables.
- For example…

Interfaces

- Each of these vehicles can speed up and slow down (common behaviors).
- They may do it differently internally but they all can speed up and slow down.

Interfaces

```
public interface MovingVehicle {
        public void SpeedUp();
        public void SlowDown();
}
```

- Interfaces specify behaviors but not implementations (no code for the methods).
- Classes will implement interfaces (give implementations for the methods).

- If an object implements the MovingVehicle interface then you know that it has SpeedUp() and SlowDown() methods defined.
- For example…

Interfaces

Car implements the MovingVehicle interface

```
public class Car implements MovingVehicle
{
        private int m_Speed;

        public int GetSpeed() { return m_Speed; }
        public void SetSpeed(int speed) {m_Speed = speed;}

        public void SpeedUp() {
                // Code for SpeedUp
        }

        public void SlowDown() {
                // Code for SlowDown()
        }
}
```

Methods on Car (NOT FROM interface)

Methods on Car (FROM MovingVehicle)

Interfaces

```
public class Airplane implements MovingVehicle
{
        private int m_Speed;

        public int GetSpeed() { return m_Speed; }
        public void SetSpeed(int speed) {m_Speed = speed;}

        pubilc void SpeedUp() {
                // Code for SpeedUp()
        }

        public void SlowDown() {
                // Code for SlowDown()
        }
}
```

Interfaces

- If a class declares that it implements an interface then it ***MUST implement ALL methods in the interface.***

- For example, it would be an error if the Car class only implemented the SpeedUp() method but not the SlowDown() method.

Interfaces

- A class can implement more than one interface.
- There is no limit to the number of interfaces that a class can implement.
- For example…

Interfaces

- Here is another interface:

```
public interface Hauls
{
        public void Load();
        public void Unload();
}
```

Interfaces

```
public class Truck implements MovingVehicle, Hauls {
        private int m_Speed;

        public int GetSpeed() { return m_Speed; }
        public void SetSpeed(int speed) {m_Speed = speed;}

        public void SpeedUp()
        { // Code for SpeedUp()   }

         public void SlowDown()
        { // Code for SlowDown()   }

         public void Load()
        {  // Code for Load()  }

         public void UnLoad()
        {  // Code for Unload()  }
}
```

Must implement ALL methods of ALL interfaces it implements

Methods on Truck (FROM MovingVehicle)

Methods on Truck (FROM Hauls)

Interfaces

- If a class implements an interface I know that I can call the methods defined in the interface on that class.

- **Car** must have **SpeedUp()** and **SlowDown()** since it implements MovingVehicle.

- **Truck** must have **SpeedUp()** and **SlowDown()** since it implements MovingVehicle.

Interfaces

- We can design methods that take interface references.

```
Car c = new Car();
Truck t = new Truck();

TestVehicle(c);
TestVehicle(t);

void TestVehicle(MovingVehicle x)
{
   x.SpeedUp();
   x.SpeedUp();
   x.SlowDown();
}
```

Car implements MovingVehicle so it can be passed in

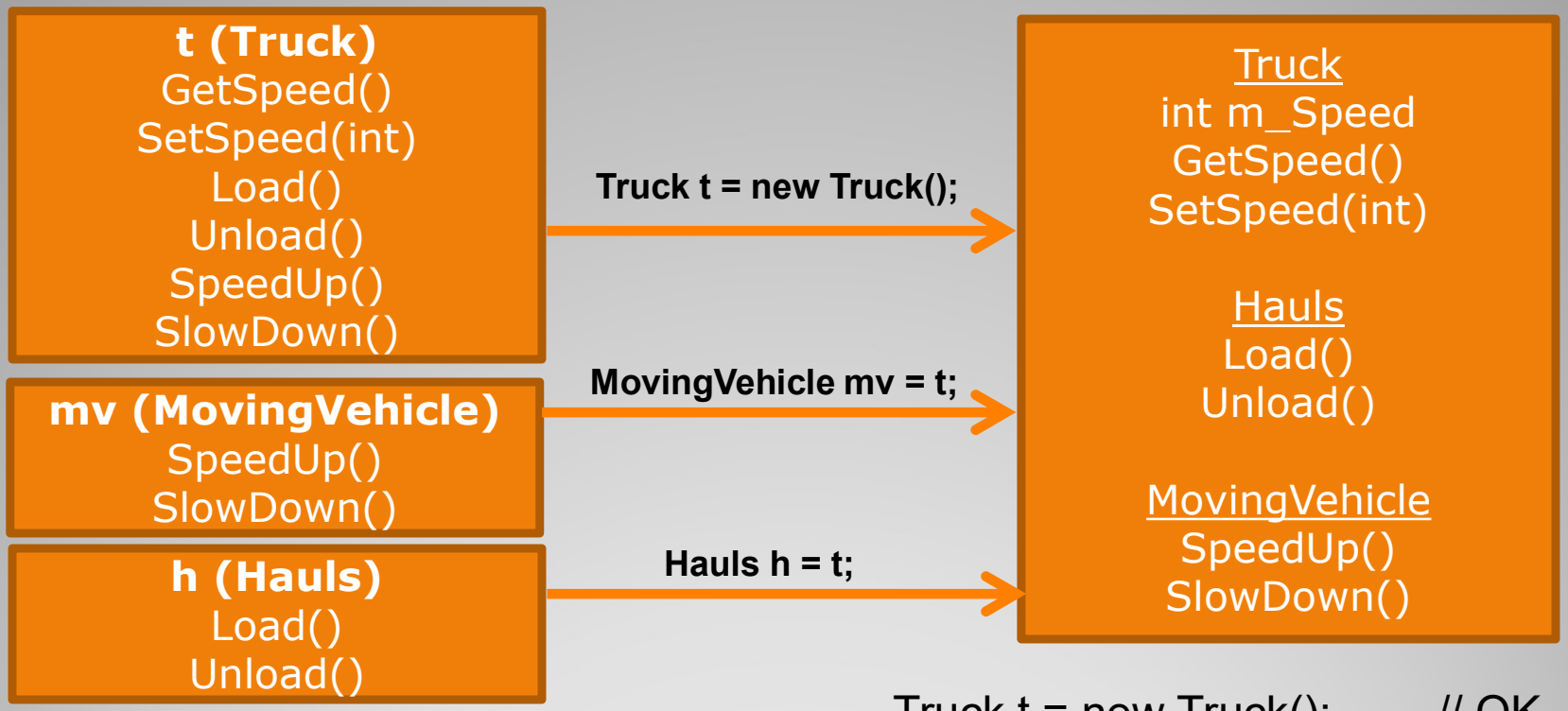
Truck implements MovingVehicle so it can be passed in

TestVehicle takes a MovingVehicle as a parameter. Any class that implements MovingVehicle can be passed as a parameter.

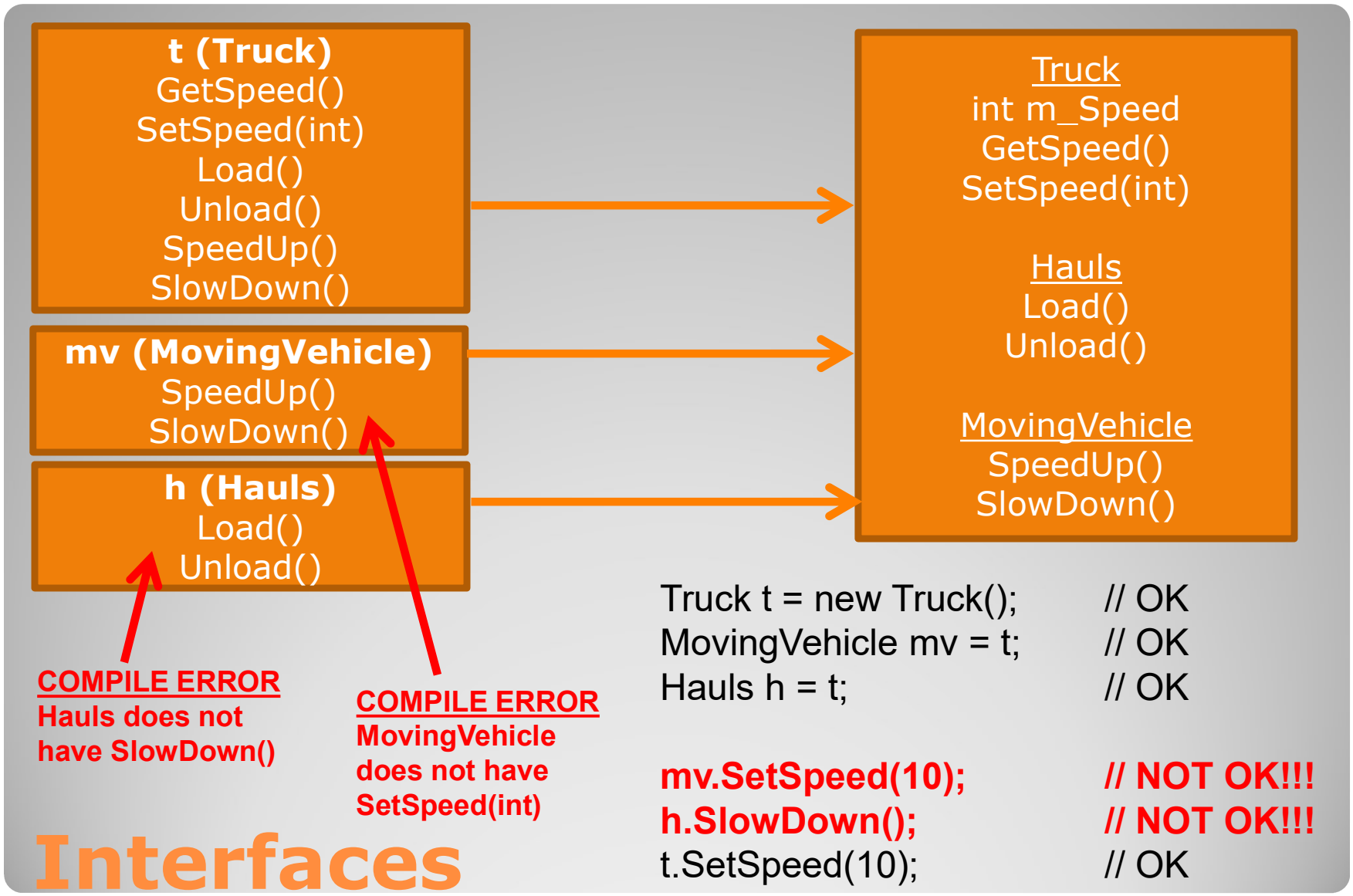
Call methods on the interface

Interfaces

Interfaces




```
Truck t = new Truck();        // OK
MovingVehicle mv = t;         // OK
Hauls h = t;                  // OK

mv.SpeedUp();     // OK
h.Load();         // OK
t.SetSpeed(10);   // OK
```




```
Truck t = new Truck();          // OK
MovingVehicle mv = t;           // OK
Hauls h = t;                    // OK

mv.SetSpeed(10);                // NOT OK!!!
h.SlowDown();                   // NOT OK!!!
t.SetSpeed(10);                 // OK
```

Interfaces

- Can only call methods on an interface reference that the interface has in its definition.
- The interface reference itself has to know the method exists (in interface definition) to be able to call it.

```
Truck t = new Truck();      // OK
MovingVehicle mv = t;       // OK
Hauls h = t;            // OK

mv.SetSpeed(10);        // NOT OK!!!
h.SlowDown();           // NOT OK!!!
t.SetSpeed(10);         // OK
```

Interfaces

- Classes are allowed to both derive from another class and implement an interface.

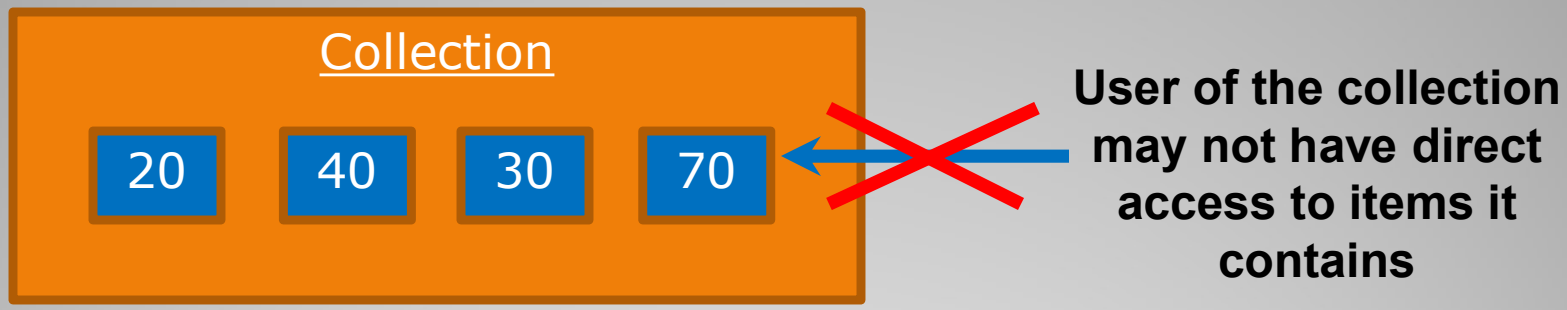
- For example:

```
interface X {  // X interface methods here… }
interface Y {  // Y interface methods here… }

class B { // Class B members here… }

class D extends B implements X, Y
{
        // Class D members here…
}
```

Derives from B and implements X and Y

Interfaces

- Here is a collection with data (could be an array):



- Users of the collection may or may not have direct access to the items of the collection.
- There needs to be a way to "visit" each item of the collection while not having direct access to it.
- That is what an iterator is for.

Review - Iterators

Review - Iterators

- Iterators are helper classes that have access to the items of the collection.
- An iterator points at one item of the class.
- In general, you can do the following with an iterator:
  - Get the data at that item.
  - Go to the next item in the collection.
  - Remove the item from that collection.
- For example…

- You can design a class so that it is usable in the header of a for-each.

- Do the following:
  1. Implement the Iterable interface.
  2. Add an inner class that implements the Iterator interface.

- For example…

Making a Class Usable in for-each

1. Implement the **Iterable** interface on collection class…

Collection item data type

```
public class MyCollection implements Iterable<Integer> {
        private int[] data = { 10, 20, 30 };

        @Override
        public Iterator iterator() {
                // iterator code goes here…
        }

        public class MyIterator implements Iterator<Integer> {
            // MyIterator code goes here…
        }
}
```

Collection (an array in this case)

The one and only method of the Iterable interface. Should return an Iterator instance "pointing" into the collection.

Note: If the collection contains something other than Integer use that type instead. For example:

```
public class MyCollection implements Iterable<Employee> {
…
}
```

Making a Class Usable in for-each

2. Create an **Iterator inner class**…

An inner class has access to the outer classes member variables

```
public class MyCollection implements Iterable<Integer> {
    private int[] data = { 10, 20, 30 };
    @Override public Iterator<Integer> iterator() { // iterator code goes here… }

    public class MyIterator implements Iterator<Integer> {
        int index = 0;

        @Override
        public boolean hasNext() { … }

        @Override
        public Integer next() { … }

        @Override
        public void remove() { … }
    }

}
```

Store the index of the element the iterator is "pointing" at

Is there another element after the current element?

Go to the next element of the collection

Remove the current element from the collection

Making a Class Usable in for-each

- **Iterator class implements hasNext()…**

```
@Override
public boolean hasNext()  {
        if (index < data.length)
                return true;

        return false;
}
```

Make sure the index is "pointing" at a valid element

- **Iterator class implements next()…**

```
@Override
public Integer next() {
        Integer item = Integer.valueOf(data[index]);
        index++;
        return item;
}
```

Create an Integer instance wrapper to hold the primitive piece of data

Go to next element

Return the item

Note: There is no need to use a wrapper class if the data is already a reference type

Making a Class Usable in for-each

- MyCollection implements the iterator() method…

```
public class MyCollection implements Iterable<Integer> {
        private int[] data = { 10, 20, 30 };

        @Override
        public Iterator<Integer> iterator() {
                return new MyIterator();
        }

        public class MyIterator implements Iterator<Integer> {
                // MyIterator members (on previous slides)…
        }

}
```

Return an instance of a class that implements the interface Iterator

Create a instance new instance of MyIterator (it implements the Iterator interface).

Making a Class Usable in for-each

```
public class MyCollection implements Iterable<Integer> {
    private int[] data = { 10, 20, 30 };

    @Override public Iterator<Integer> iterator() {  return new MyIterator(); }

    public class MyIterator implements Iterator<Integer> {
        int index = 0;

        @Override public boolean hasNext() {
            if (index < data.length) return true;
            return false;
        }

        @Override  public Integer next() {
            Integer item = Integer.valueOf(data[index]);
            index++;
            return item;
        }

        @Override public void remove() { } // Optional
    }
}
```

MyCollection implements Iterable<Integer>

MyIterator inner class implements Iterator<Integer>

MyCollection – All Code

- **Using your collection class in a for-each…**

```
MyCollection c = new MyCollection();

for (int item : c)
{
    System.out.println("Item is: " + item);
}
```

Collection item type

Variable name for current item

Collection instance

The for expects the collection to implement the Iterable interface:

1. **for will automatically call the iterator() method on the collection (c in this case).**
2. **The iterator it receives will have next() and hasNext() called on it automatically.**

Making a Class Usable in for-each

Iterator Interface Methods

Modifier and Type	Method	Description
boolean	hasNext()	Returns true if the iteration has more elements.
E	next()	Returns the next element in the iteration.
default void	remove()	Removes from the underlying collection the last element returned by this iterator (optional operation).

Note: E is the type of elements returned by the iterator. In the following example E would be Integer:

E would be Integer

```
public class MyCollection implements Iterable<Integer>
{
}
```

Taken from:
http://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html

Iterable Interface Methods

Modifier and Type	Method	Description
Iterator<T>	iterator()	Returns an iterator over a set of elements of type T.

Taken from:
http://docs.oracle.com/javase/7/docs/api/java/lang/Iterable.html

- End of Presentation

End of Presentation

- End of Slides

End of Slides