Java Programming

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

Today’s Lecture

- Interfaces
- Lambda expressions
- Functional interfaces
- User-defined classes and Iterable

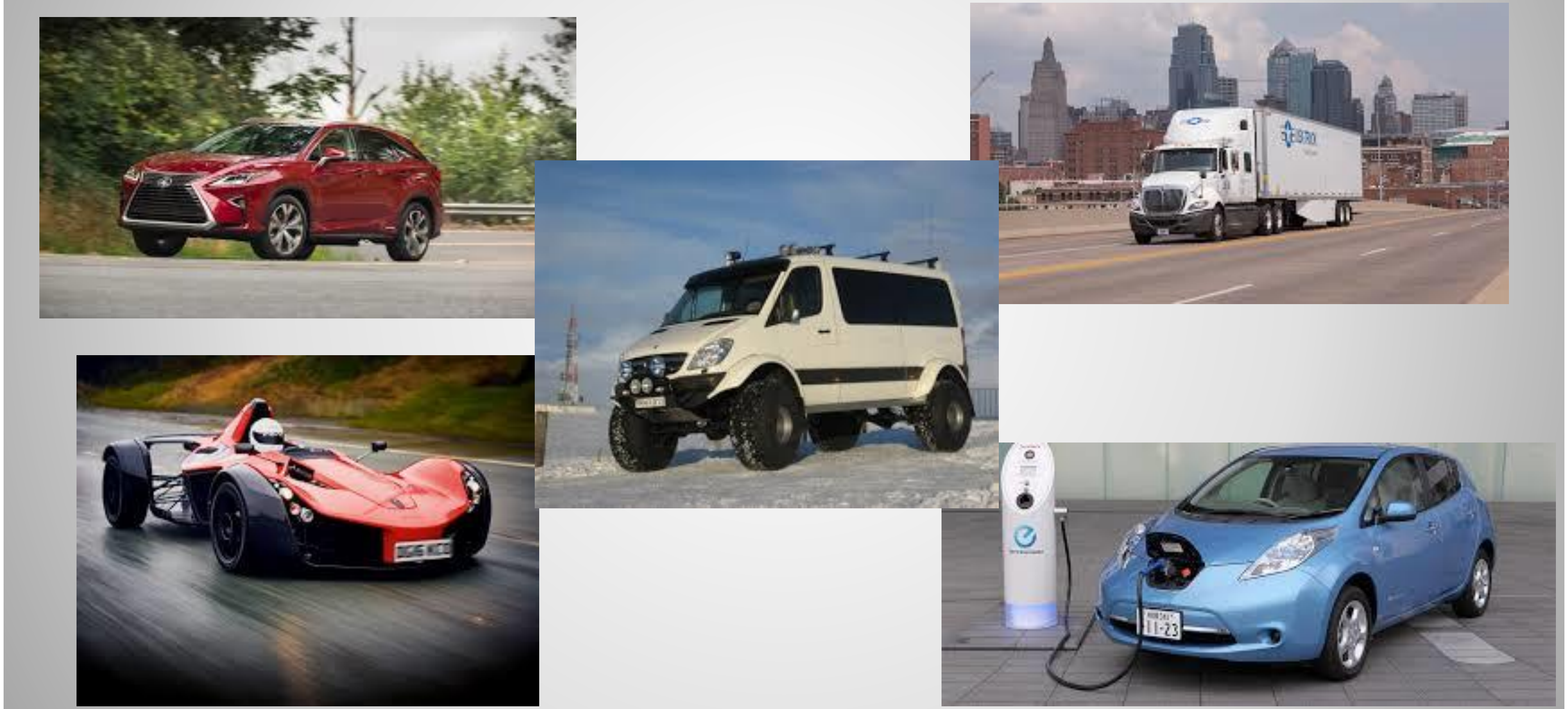
Interfaces

- **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

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…

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

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()
        }
}
```

- 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

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)

- 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

Interfaces

- We can design methods that take interface references.

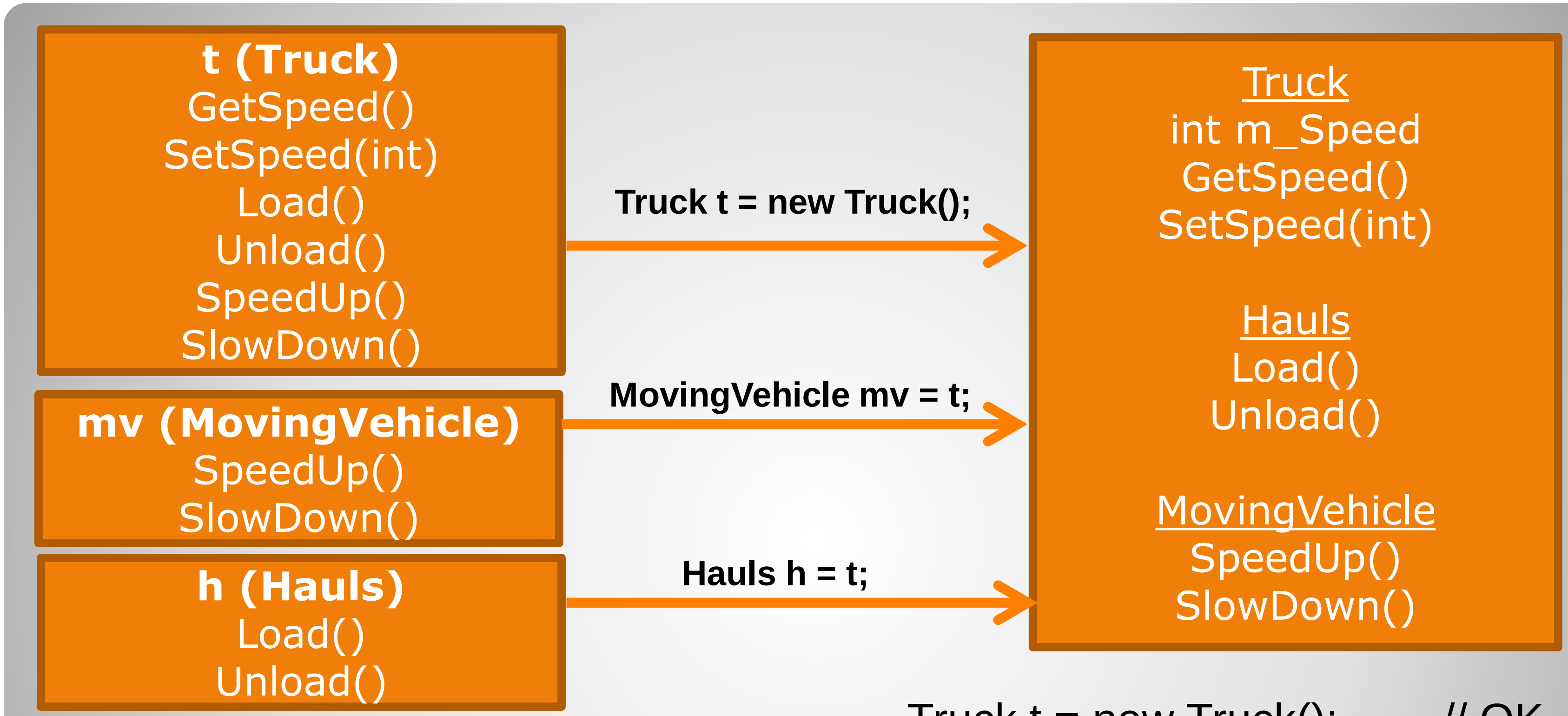
```
Car c = new Car();
Truck t = new Truck();

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



t (Truck)
GetSpeed()
SetSpeed(int)
Load()
Unload()
SpeedUp()
SlowDown()

mv (MovingVehicle)
SpeedUp()
SlowDown()

h (Hauls)
Load()
Unload()

Truck t = new Truck();

MovingVehicle mv = t;

Hauls h = t;

Truck
int m_Speed
GetSpeed()
SetSpeed(int)

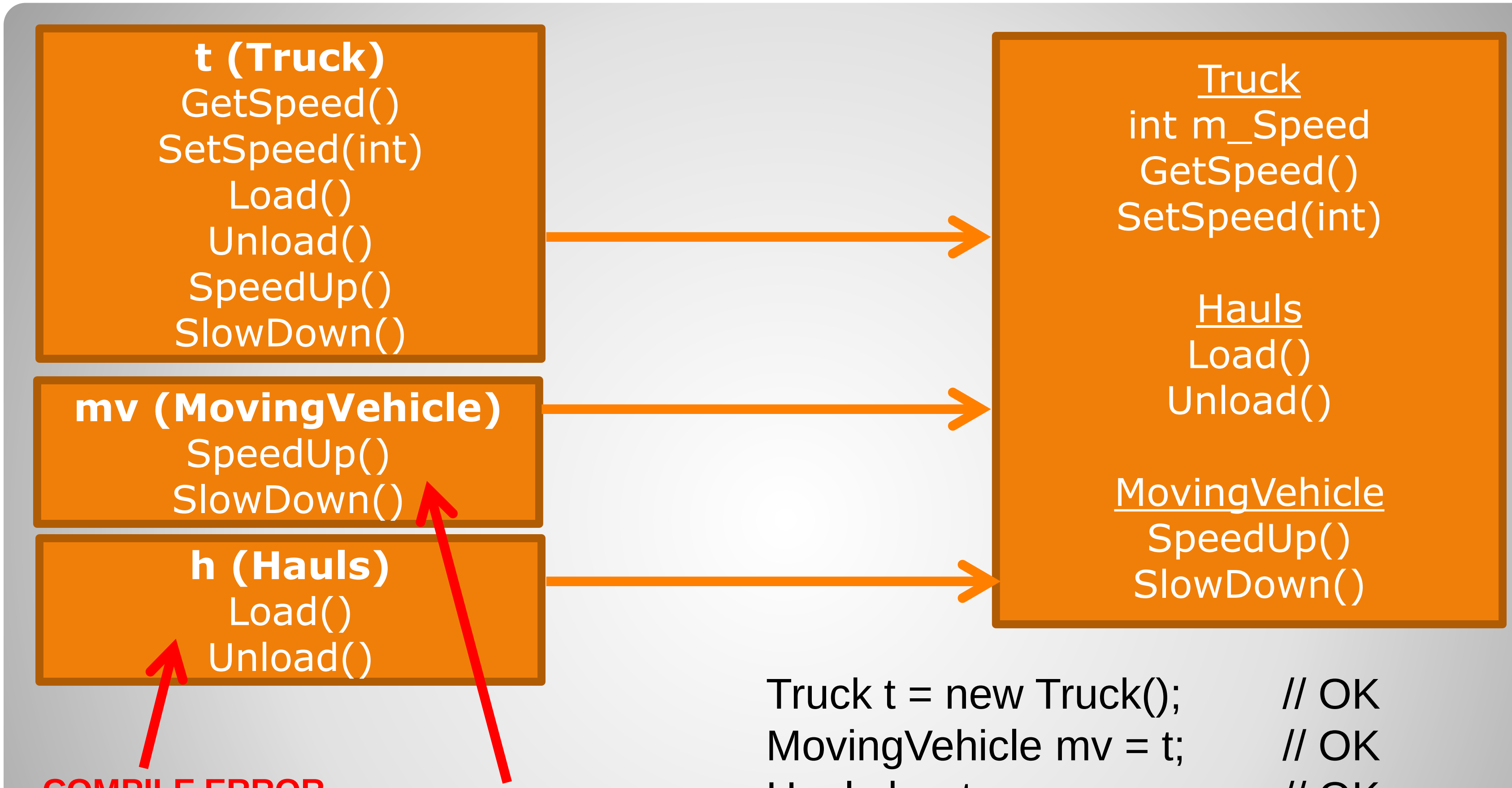
Hauls
Load()
Unload()

MovingVehicle
SpeedUp()
SlowDown()

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

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

Interfaces

Interfaces



t (Truck)
GetSpeed()
SetSpeed(int)
Load()
Unload()
SpeedUp()
SlowDown()

mv (MovingVehicle)
SpeedUp()
SlowDown()

h (Hauls)
Load()
Unload()

Truck
int m_Speed
GetSpeed()
SetSpeed(int)

Hauls
Load()
Unload()

MovingVehicle
SpeedUp()
SlowDown()

COMPILE ERROR
Hauls does not have SlowDown()

COMPILE ERROR
MovingVehicle does not have SetSpeed(int)

```
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
```

- 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

Interfaces

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

- For example:

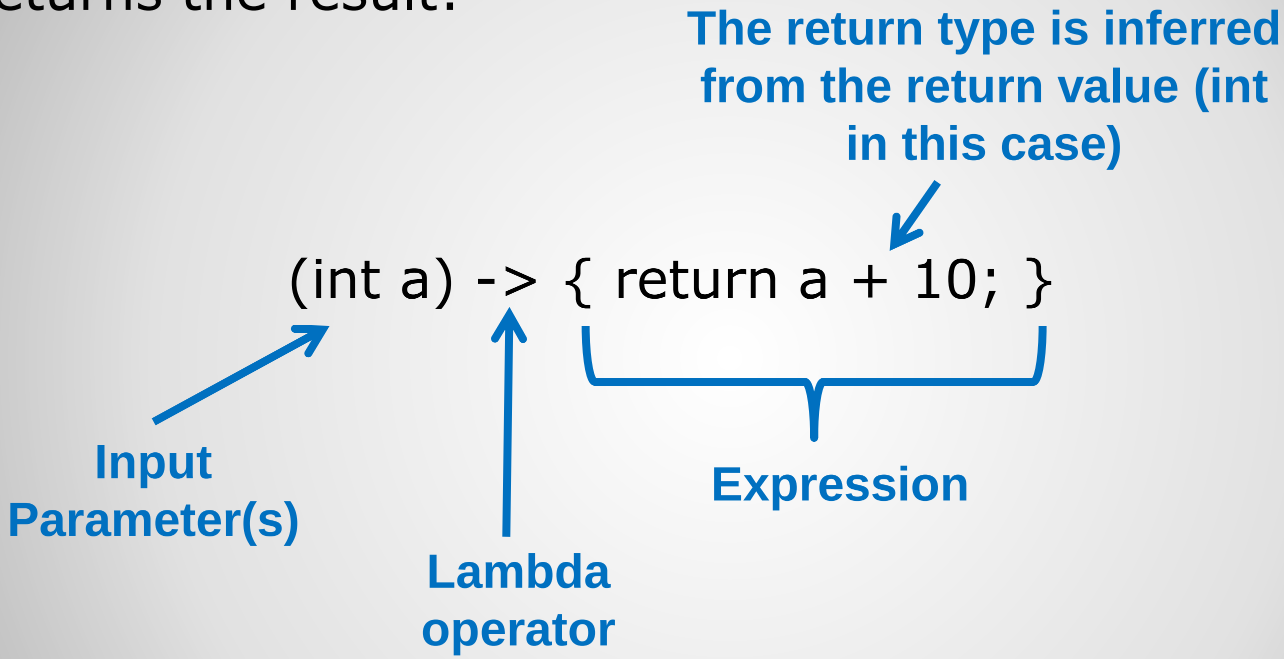
```
interface X {  // X interface methods here… }
interface Y {  // Y interface methods here… }

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**

- Now we will cover lambda expressions and functional interfaces…

Lambda Expressions and Functional Interfaces

- A **lambda expression** is an **anonymous method.**
- Here is a lambda expression that adds 10 to a number and returns the result:



The return type is inferred from the return value (int in this case)

```
(int a) -> { return a + 10; }
```

Input Parameter(s)

Lambda operator

Expression

Lambda Expression

Lambda Expression

You can do the following with lambda expressions:

- Pass a lambda expression to a method as a parameter
- Assign a lambda expression to a variable
- Return a lambda expression from a method

Lambda Expression Syntax

- Syntax for lambda expressions:

```
(int a) -> { return a + 10; }
```

You can omit the parameter data types if you want (it will figure out the type based on how it is used)

```
(a) -> { return a + 10; }
```

You can omit the braces and return if there is only one statement in the body

```
(a) -> a + 10;
```

You can omit the parameter parenthesis if there is only one parameter

```
a -> a + 10;
```

You can omit variables if there are no parameters

```
() -> System.out.println("No parameters in lambda");
```

Functional Inteface

- An interface with only one abstract method.

```
interface MyFunctionalInterface
{
   int square(int x);
}
```

Contains only ONE method

Functional Interface

Functional Interface and Lambda

- The example below declares an instance of the functional interface and populates it using a lambda expression.

```
interface MyFunctionalInterface
{
   int square(int x);
}
```

```
MyFunctionalInterface mfi;
```

Declare a variable for the functional inteface

```
mfi = (int x) -> { return x * x; };
```

Assign a lambda expression to the functional interface variable

```
int result;
result = mfi.square(3);
```

Call the method on the square method on the functional interface

Pass Functional Interface to Method

- The example below passes a functional interface to a method which then uses it.

```
void TestMethod(MyFunctionalInterface x)
{
   int result;
   result = x.square(3);
   System.out.println(result);
}
```

Call the method using the parameter (MyFunctionalInterface is defined on the previous slide)

```
MyFunctionalInterface mfi;
mfi = (int x) -> { return x * x; }
TestMethod(mfi);
```

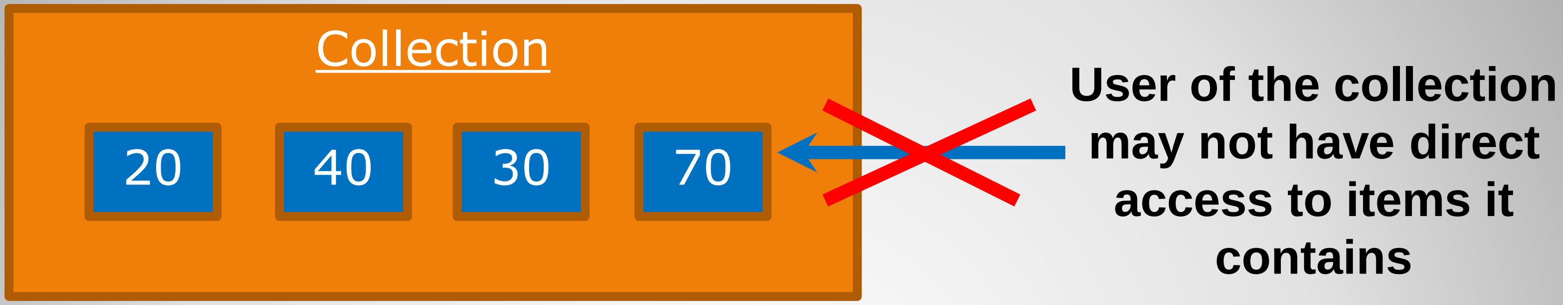
Pass in the functional interface variable as a parameter to TestMethod

User-defined Classes and Iterable

- Now we will cover how to use the Iterable interface on a user-defined class…

Review - Iterators

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



User of the collection may not have direct access to items it contains

- 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

- 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…

Making a Class Usable in for-each

- 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…

```
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 item data type

Collection (an array in this case)

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

- 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

MyCollection – All Code

```
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
    }
}
```

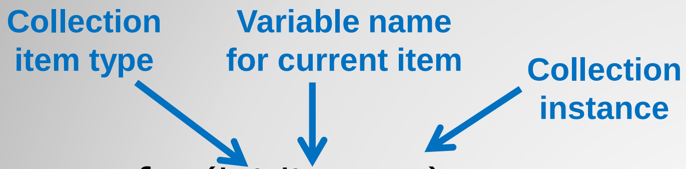
MyCollection implements Iterable<Integer>

MyIterator inner class implements Iterator<Integer>

Making a Class Usable in for-each

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

```
MyCollection c = new MyCollection();
```



Collection item type

Variable name for current item

Collection instance

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

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.**

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 Slides

End of Slides