CSE-2102
Object Oriented Programming

Dr. Muhammad Ibrahim
Assistant Professor
Dept. of Computer Science and Engineering
University of Dhaka

Topics

Generics

- Syntax
- Benefit
- Multiple data types
- Bounded types

Generics

- Can be defined as parameterized types.
- Using it we can define class, interface, methods that take the type of the data to operate on as parameter.
    - So it is possible to define a single class that works for different types of data.
- A class, interface or method that operates on a generic data type is called generic class, generic interface and generic method respectively.

Example of a Generic Class

```
class Gen<T> {
    T ob;
    Gen(T o) {
        ob = o;
    }
    T getob() {
        return ob;
    }
    void showType() {
        System.out.println("Type
        of T is " +
        b.getClass().getName());
    }
}
```

```
Gen<Integer> iOb = new Gen<Integer>(88);

Gen<String> strOb = new Gen<String>("Test");
```

Details

- `class Gen<T> {`
  - T is called the type parameter
  - `Gen` is a generic class, which is also called a parameterized type.
- `T ob;`
  - Declare an object of type T
  - T is a placeholder for the actual type that will be specified when a `Gen` object is created.
  - Thus, `ob` will be an object of the type passed to T.
- `Gen(T o) {   ob = o;   }`
  - T can be passed as parameter
- `T getob() {   return ob;   }`
  - T can be used as return type

Details (Contd.)

- The `getClass()` method is defined by `Object` and is thus a member of all class types (because all classes inherit `Object`).
    - It returns a `Class` object that corresponds to the type of the class of the object on which it is called.
    - The `Class` object defines the `getName()` method, which returns a string representation of the class name.
- `Gen<Integer> iOb;`
    - A version of `Gen` for Integers is created.
    - Integer is a type argument that is passed to `Gen`'s type parameter, T.

A Few Points

- Generic class mechanism doesn’t work with primitive types.
    - `Gen<int> intOb = new Gen<int>(53);` // Error, can't use primitive type.
- Generic objects are not type-compatible with each other, i.e., a reference variable of one type of a generic is not type-compatible with a reference variable of another type of the same generic class.
    - `Gen<Integer> intOb = new Gen<Integer>(53);`
    - `Gen<String> strOb = new Gen<String>(“text”);`
    - `iOb = strOb;` //ERROR, even though both have Gen<T>
- Autoboxing and auto-unboxing work frequently with generics
    - `Gen<Integer> intOb = new Gen<Integer>(53);` is equivalent to `Gen<Integer> intOb = new Gen<Integer>(new Integer(53));`
    - Similarly, `int v = iOb.getob();` is equivalent to `int v = iOb.getob().intValue();`

Generics and Type Safety

- Since `Object` is a superclass of all classes, and since a superclass reference variable can refer to any subclass object, we can already use variables of type `Object` in a class to make the class accept all types of data (such as `Integer, Double` etc.)
  - However, type safety cannot be achieved using this scheme.
- Without generics we need to explicitly cast to produce type-safe code.
- With generics we no longer need to cast explicitly - all casts are automatic and implicit.
- So what exactly is type safety? Let's wait a few slides to get to the answer!

So What’s the Benefit of Using Generic?

- Using `Object` would apparently do the same job as using `Gen` class.
- However, explicit type-cast would be needed.
- Take a look at the next slide…

An Alternative to Generic Class?

```
class Gen {
    Object ob;
    Gen(Object o) {
        ob = o;
    }
    Object getob() {
        return ob;
    }
    void showType() {
        System.out.println("Type of T is " +
        ob.getClass().getName());
    }
}
```

```
Gen iOb = new Gen(88);
Gen strOb = new Gen("Test");
```

However, we can no longer write:

```
int v = iOb.getob();
String s = strOb.getob();
```

Instead, we must write:

```
int v = (int)iOb.getob();
String s =
(String)strOb.getob();
```

Benefits of Generics

- Thus, generics relieves the programmers from a lot of explicit type-casting.
- Moreover, generics alleviates some runtime errors.
  - Using the previous scheme of using `Object`, the following compiles but is conceptually wrong:
    - `iOb = strOb;`
    - `v = (Integer) iOb.getob(); // run-time error! Because iOb is now a String.`
  - Whereas when using generics, `iOb = strOb;` produces compile time error, which is definitely better than runtime error.
- **The bottom line:** although it is always possible to produce a code of generics into a code that uses `Object` class, (1) it will not be type-safe, and (2) it may incur some runtime errors.

Generic Class with Two Type Parameters

```
class TwoGen<T, V> {
    T ob1;
    V ob2;
    TwoGen(T o1, V o2) {
        ob1 = o1;
        ob2 = o2;
    }
    void showTypes() {
        System.out.println("Type of T 
is " + ob1.getClass().getName());
        System.out.println("Type of V 
is " + ob2.getClass().getName());
    }
    T getob1() {
        return ob1;
    }
```

```
    V getob2() {
        return ob2;
    }
}//class TwoGen

class SimpGen {
  public static void main(String 
args[]){
     TwoGen<Integer, String> tgObj =
        new TwoGen<Integer, String>(88,
        "Generics");
     tgObj.showTypes();
     int v = tgObj.getob1();
     System.out.println("value: " + v);
     String str = tgObj.getob2();
     System.out.println("value: " +
str);
    }
```

Generic Class with Two Type Parameters

- Thus, we can define generic classes with as many variables as required.
- In the previous example, it is possible to write:

```
TwoGen<Integer, Integer> Obj = new TwoGen<Integer, Integer>(88, 89);
```

General Form of Generics

- Declaring a generic class: `class class-name<type-param-list> {`
- Declaring a reference to a generic class and instance creation:

```
class-name<type-arg-list> var-name =
      new class-name<type-arg-list>(cons-arg-list);
```

Bounded Types

- Preceding examples showed the scenario where any data type can be used to create specific instances of generic classes.
- While this is allowed in some applications, some other applications warrant that we restrict the creation of instances of generic class to some data types only.
  - Consider an application that returns the average of the values of an array of numbers (integers, floating point numbers etc.).
  - So this application, when using generics, must ensure that only numbers are used to create instances of the generic class.
  - Recall that all numeric classes (`Integer, Double` etc.) are subclasses of class `Number`.

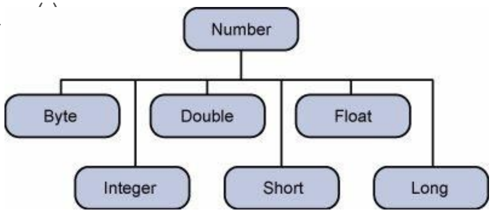
Bounded Types Example

```
class Stats<T> {
    T[] nums;
    Stats(T[] o) {
        nums = o;
    }
    // Return type double in all cases.
   double average() {
     double sum = 0.0;
     for(int i=0; i < nums.length; i++)
            sum += nums[i];
     return sum / nums.length;
    }
}
```

```
class Stats<T> {
    T[] nums;
    Stats(T[] o) {
        nums = o;
    }
    // Return type double in all
cases.
   double average() {
     double sum = 0.0;
     for(int i=0; i < nums.length;
i++)
          sum +=
nums[i].doubleValue();
     return sum / nums.length;
    }
}
```

Bounded Types Example

```
class Stats<T extends Number> {
    T[] nums;
    Stats(T[] o) {
        nums = o;
    }
    // Return type double in all cases.
    double average() {
      double sum = 0.0;
      for(int i=0; i < nums.length;
i++)
           sum +=
nums[i].doubleValu
      return sum /
    }
}//class Stats
```



```
class BoundsDemo {
 public static void main(String
args[]){
   Integer inums[] = { 1, 2, 3, 4, 5 };
   Stats<Integer> iob =
       new Stats<Integer>(inums);
   double v = iob.average();
   System.out.println("iob average is "
                   + v);
   Double dnums[] = { 1.1, 2.2, 3.3,
                   4.4, 5.5 };
   Stats<Double> dob =
       new Stats<Double>(dnums);
   double w = dob.average();
   System.out.println("dob average is "
                   + w);
 }
}//class BoundsDemo
```

Bounded Types Example

Output:

```
Average is 3.0
Average is 3.3



// This will not compile because String is not a subclass of Number.
         String strs[] = { "1", "2", "3", "4", "5" };
         Stats<String> strob = new Stats<String>(strs);
         double x = strob.average();
         System.out.println("strob average is " + v);
```

Having Both Class and Interface as Bound Types

- In addition to using a class type as a bound, we can also use an interface type.
  - In fact, we can specify multiple interfaces as bounds.
- Furthermore, a bound can include both a class type and one or more interfaces.
  - In this case, the class type must be specified first.
- Bottom line: at most one class can be included as bound type, although more than one interface can be used, and both class and interfaces can be used simultaneously.
- When a bound includes an interface type, only type arguments that implement that interface are legal.

Having Both Class and Interface as Bound Types

- When specifying a bound that has a class and an interface, or multiple interfaces, use the `&` operator to connect them.
  - `class Gen<T extends MyClass & MyInterface> { // …`
  - Here, `T` is bounded by a class called `MyClass` and an interface called `MyInterface`.
  - Thus, any type argument passed to `T` must be a subclass of `MyClass` and must implement `MyInterface`.

End of Lecture 17.

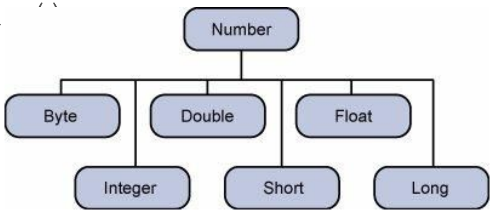
Topics

- Generics
  - Generic class: syntax and benefit
    - Bounded types with classes and interfaces
    - Wildcards, bounded wildcards
  - Generic method
  - Generic interface
  - Inheritance with generics
  - Restrictions on generic array

Bounded Types Example

(From Slide 17)

```
class Stats<T extends Number> {
    T[] nums;
    Stats(T[] o) {
        nums = o;
    }
    // Return type double in all cases.
    double average() {
      double sum = 0.0;
      for(int i=0; i < nums.length;
i++)
           sum +=
nums[i].doubleValu
      return sum /
    }
}//class Stats
```



```
class BoundsDemo {
 public static void main(String
args[]){
   Integer inums[] = { 1, 2, 3, 4, 5 };
   Stats<Integer> iob =
       new Stats<Integer>(inums);
   double v = iob.average();
   System.out.println("iob average is "
                   + v);
   Double dnums[] = { 1.1, 2.2, 3.3,
                   4.4, 5.5 };
   Stats<Double> dob =
       new Stats<Double>(dnums);
   double w = dob.average();
   System.out.println("dob average is "
                   + w);
 }
}//class BoundsDemo
```

Wildcards: Motivation

- Suppose in the previous example, we want to add a method called `sameAvg()` that determines if two arrays have the same average.
- Example:

```
boolean sameAvg(Stats<T> ob){
    if (this.average() == ob.average()) return true;
    else return false;
}
```

- Will it work? (recall from Eg. given in slide 17 that the type of `iob` is `Stats<Integer>`)

```
Integer inums[] = { 1, 2, 3, 4, 5 };
 Double dnums[] = { 1.0, 2.0, 3.0, 4.0, 5.0 };
 System.out.println(iob.sameAvg(dob)); // ERROR
```

Wildcards

- So the problem is, if we use a type parameter T in a generic class, each time it is substituted by a single data type (`Integer` or `Double` etc., but not both at the same time).
- However, oftentimes we require that T takes different data types for the same instance.
- Solution: wildcard

```
boolean sameAvg(Stats<?> ob){
    if (this.average() == ob.average()) return true;
    else return false;
}
```

- Now it is valid: `System.out.println(iob.sameAvg(dob));`

Bounded Wildcards

- Recall the motivation for using bounded types: oftentimes we want to restrict the allowable types for a generic class.
- For a wildcard too, we may want to restrict the allowable types for `<?>` construct.
- It is especially important when a generic involves inheritance.
- Example: next slides…

Example: Bounded Wildcard

```
class TwoD {
   int x, y;
   TwoD(int a, int b) {
     x = a;
     y = b;
   }
}//class ends
// Three-dimensional coordinates.
//class ThreeD extends TwoD {
   int z;
   ThreeD(int a, int b, int c) {
    super(a, b);
     z = c;
   }
}//class ends
```

```
// Four-dimensional coordinates.
//class FourD extends ThreeD {
   int t;
   FourD(int a, int b, int c, int d) {
     super(a, b, c);
     t = d;
   }
}//class ends

// This class holds an array of
//coordinate objects.
class Coords<T extends TwoD> {
   T[] coords;
   Coords(T[] o) { coords = o; }
}//class ends
```

Example: Bounded Wildcard (Contd.)

```
//to show X and Y coords of an object:
static void showXY(Coords<?> c) {
 System.out.println("X Y Coordinates:");
 for(int i=0; i < c.coords.length; i++)
   System.out.println(c.coords[i].x + " 
      " + c.coords[i].y);
   System.out.println();
}
```

Now what if we want to define a method that prints X, Y and Z coordinates?

That is, how can we further restrict the allowable types, on top of the ones already?

```
static void showXYZ(Coords<? extends ThreeD> c) {
 System.out.println("X Y Z Coordinates:");
 for(int i=0; i < c.coords.length; i++)
  System.out.println(c.coords[i].x + " " +
            c.coords[i].y + " " + c.coords[i].z);
 System.out.println();
}
```

Bounded Wildcard

- So the general rule is to use: `<? extends superclass>`
  - where superclass is the name of the class that serves as the upper bound.
  - This is an inclusive clause because the class forming the upper bound (that is, specified by superclass) is also within bounds.
- It is also possible to specify a lower bound for a wildcard by adding a super clause to a wildcard declaration: `<? super subclass>`
  - Here only classes that are superclasses of subclass are acceptable arguments. This is also an inclusive clause.

Topics

- Generics
    - Generic class: syntax and benefit
        - Bounded types with classes and interfaces
        - Wildcards, bounded wildcards
    - Generic method
    - Generic interface
    - Inheritance with generics
    - Restrictions on generic array

Generic Methods

- Methods inside a generic class work on a type parameter, and therefore are automatically generic methods.
- It is possible to declare a method inside a generic class that, in addition to using type parameter, uses some other data types.
- It is also possible to write non-generic methods inside a generic class.
- Equally possible is to define generic methods inside a non-generic class.
- Next slide: GenMethodDemo.java…

Generic Methods

```
class GenMethodDemo {
// Determine if an object is in an array.
   static <T extends Comparable<T>, V extends T> boolean isIn(T x, V[] y)
   {
     for(int i=0; i < y.length; i++)
         if(x.equals(y[i]))
            return true;
         return false;
   }
```

`Comparable` (generic) interface:
https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html

Generic Methods

```
class GenMethodDemo {
// Determine if an object is in an
//array.
    static <T extends Comparable<T>,
            V extends T>
     boolean isIn(T x, V[] y) {
     for(int i=0; i < y.length; i++)
         if(x.equals(y[i]))
              return true;
         return false;
     }
```

`Comparable` interface: https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html

```
public static void main(String args[]) {

  // Use isIn() on Integers.
   Integer nums[] = {1, 2, 3, 4, 5 };
   if(isIn(2, nums))//2 is autoboxed as new Integer(2)
      System.out.println("2 is in nums");
   if(!isIn(7, nums))
       System.out.println("7 is not in nums");
   System.out.println();

   // Use isIn() on Strings.
   String strs[] = { "one", "two", "three",
                      "four", "five" };
   if(isIn("two", strs))
     System.out.println("two is in strs");
   if(!isIn("seven", strs))
    System.out.println("seven is not in strs");

//Will not compile! Types must be compatible.
      // if(isIn("two", nums))
      // System.out.println("two is in strs");
    }
}
```

Generic Methods (Contd.)

```
static <T extends Comparable<T>, V extends T> boolean isIn(T x, V[] y)
```

- The type parameters are declared before the method name.
- `Comparable` is an interface declared in java.lang package and is a generic (and functional) interface.
  - `Comparable` can only be implemented by a class whose objects are comparable by some aspect.
- Thus, T can only take objects that implement the `Comparable` interface with the same type, i.e. T.
- V is upper-bounded by T, so V must be of type T or one of its subclass.
- The method `isIn()` is static, so we don't need any particular object to call it.
  - However, generic methods inside a non-generic class can be non-static as well.

public final class **Integer**
extends Number
implements
Comparable<Integer>

public final class **String**
extends Object
implements
Serializable,
Comparable<String>,
CharSequence

Generic Methods (Contd.)

- `if(isIn(2, nums))`
    - Thus, generic methods inside a non-generic class can be called as usual.
    - 2 (which is autoboxed as new `Integer(2)`) causes the compiler to check if Integer class is implementing `Comparable<Integer>`interface.
    - `nums` causes the compiler to check if `Integer` (for V) class is of type `Integer` (for T) or one of its subclasses.
    - Also allowed form: `GenMethodDemo.<Integer, Integer>isIn(2, nums)`

Recall the main benefit of using generics: preventing type-unsafe codes during compile time!

Generic Methods: Generic Constructors

It is possible to write generic methods as constructors even though the class is not generic.

```
class GenCons {
 private double val;
 <T extends Number> GenCons(T arg) {
  val = arg.doubleValue();
 }
 void showval() {
  System.out.println("val: " + val);
 }
}
class GenConsDemo {
 public static void main(String args[]) {
  GenCons test = new GenCons(100);
  GenCons test2 = new GenCons(123.5F);
  test.showval();
  test2.showval();
 }
}
```

Generic Interfaces

- Generic interfaces are written just like generic classes.
- Next slide: GenInterfaceDemo.java…

```
interface MinMax<T extends Comparable<T>> {
    T min();
    T max();
}
// Now, implement MinMax
class MyClass<T extends Comparable<T>>
               implements MinMax<T> {
    T[] vals;
    MyClass(T[] o) { vals = o; }
    // Return the minimum value in vals.
    public T min() {
        T v = vals[0];
        for(int i=1; i < vals.length; i++)
          if(vals[i].compareTo(v) < 0)
             v = vals[i];
        return v;
    }
    // Return the maximum value in vals.
    public T max() {
        T v = vals[0];
        for(int i=1; i < vals.length; i++)
            if(vals[i].compareTo(v) > 0)
               v = vals[i];
        return v;
    }
}
```

```
class GenInterfaceDemo {

    public static void main(String args[]) {

      Integer inums[] = {3, 6, 2, 8, 6 };
      Character chs[] = {'b', 'r', 'p', 'w'};

      MyClass<Integer> iob =
          new MyClass<Integer>(inums);
      MyClass<Character> cob =
          new MyClass<Character>(chs);

      System.out.println("Max value in inums: "
                          + iob.max());
      System.out.println("Min value in inums: "
                          + iob.min());
      System.out.println("Max value in chs: " +
                          cob.max());
      System.out.println("Min value in chs: " +
                          cob.min());
    }
}
```

Generic Interfaces (Contd.)

- If a class implements a generic interface, must that class be a generic too?
- Answer: not necessarily:
  - `class MyClass implements MinMax<T>` { //Wrong. Here `MyClass` must be generic.
  - `class MyClass implements MinMax<Integer>` { //Valid. Here `MyClass` is not generic.
- In a nutshell, two benefits are offered by generic interface:
  - It can work with different types of data.
  - It can put constraints on types of data to be used with it.

Generic Class Hierarchy: Inheriting Generic Class

- Generic classes can be used in class hierarchies in the same way as a non-generic class.
  - Thus, can be a superclass or subclass.
- However, the required type information must be passed up to all the generic classes in a hierarchy.
  - Much like the scheme of calling of parameterized constructors in the class hierarchies.

```
class Gen<T> {
 T ob;
 Gen(T o) {
  ob = o;
 }
// Return ob.
 T getob() {
  return ob;
 }
}//Gen class ends

// A subclass of Gen.
class Gen2<T> extends Gen<T> {
 Gen2(T o) {
  super(o);
 }
```

Two Restrictions on Generic Array

- Cannot instantiate an array whose element type is a type parameter.

`T[] vals`; // OK

But, you cannot instantiate an array of T:

`vals = new T[10]`; // Wrong, can't create an //array of T

`Integer nums[] = new Integer[10];`
`vals = nums`; // nums is an existent array, so OK to assign reference to existent array

- Cannot create an array of type-specific generic references.

You cannot declare an array of references to a specific generic type:

```
Gen<Integer>[] gens =
  new Gen<Integer>[10]; // Wrong!
```

Misc. Topics

- Anonymous class
- Functional interface
- Lambda expression

Anonymous Class

Based on https://www.programiz.com/java-programming/anonymous-class

- Recall that a class can contain another class known as nested class.
- It is possible to create a nested class without giving any name.
- A nested class that doesn't have any name is known as an anonymous class.
- An anonymous class must be defined inside another class. Hence, it is also known as an anonymous inner class.
- Anonymous classes usually extend subclasses or implement interfaces.

```
class outerClass {
    // defining anonymous class
    object1 = new Type(parameterList) {
         // body of the anonymous class
    };
}
```

Here, Type can be: a superclass that an anonymous class extends, or an interface that an anonymous class implements. The above code creates an object, object1, of an anonymous class at runtime.

Example

```
class Polygon {
   public void display() {
      System.out.println("Inside the Polygon class");
   }
}

class AnonymousDemo {
   public void createClass() {
      // creation of anonymous class extending class Polygon
      Polygon p1 = new Polygon() {
         public void display() {
            System.out.println("Inside an anonymous class.");
         }
      };
      p1.display();
   }
}

class Main {
   public static void main(String[] args) {
       AnonymousDemo an = new AnonymousDemo();
       an.createClass();
   }
}
```

Example

```
interface Polygon {
   public void display();
}

class AnonymousDemo {
   public void createClass() {

      // anonymous class implementing interface
      Polygon p1 = new Polygon() {
         public void display() {
            System.out.println("Inside an anonymous class.");
         }
      };
      p1.display();
   }
}

class Main {
   public static void main(String[] args) {
      AnonymousDemo an = new AnonymousDemo();
      an.createClass();
   }
}
```

Example (https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html#examples-of-anonymous-classes)

```
public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World!");
        Button btn = new Button();
        btn.setText("Say 'Hello World'");
        btn.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                System.out.println("Hello World!");
            }
        });
     //other code… … …
}
```

- In this example, the method invocation `btn.setOnAction` specifies what happens when you select the Say 'Hello World' button.
- This method requires an object of type `EventHandler<ActionEvent>`.
- The `EventHandler<ActionEvent>` interface contains only one method, `handle`.
- Instead of implementing this method with a new class, the example uses an anonymous class expression.
- Notice that this expression is the argument passed to the `btn.setOnAction` method.

Benefit of Anonymous Class

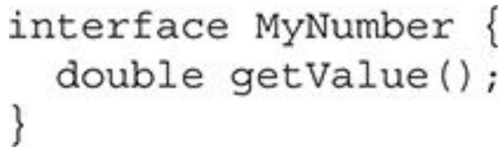
In anonymous classes, objects are created whenever they are required.

That is, objects are created to perform some specific tasks.

Functional Interface

In the previous example, `EventHandler<ActionEvent>` interface contains only one method, `handle`.

A functional interface is an interface that contains one and only one abstract method. Normally, this method specifies the intended purpose of the interface. Thus, a functional interface typically represents a single action.



```
interface MyNumber {
  double getValue();
}
```

Lambda Expression

A lambda expression is, essentially, an anonymous (that is, unnamed) method.

However, this method is not executed on its own.

Instead, it is used to implement a method defined by a functional interface. Thus, a lambda expression results in a form of anonymous class.

Lambda Expression

The new operator, sometimes referred to as the lambda operator or the arrow operator, is –>.

It divides a lambda expression into two parts. The left side specifies any parameters required by the lambda expression. (If no parameters are needed, an empty parameter list is used.) On the right side is the lambda body, which specifies the actions of the lambda expression. The –> can be verbalized as “becomes” or “goes to.”

Lambda Expression

`() -> 123.45` equivalent to `double myMethod() { return 123.45; }`

`(n) -> (n % 2) == 0`

```java
// Demonstrate a lambda expression that takes a parameter.

// Another functional interface.
interface NumericTest {
 boolean test(int n);
}

class LambdaDemo2 {
  public static void main(String[] args)
  {
    // A lambda expression that tests if a number is even.
    NumericTest isEven = (n) -> (n % 2)==0;

    if(isEven.test(10)) System.out.println("10 is even");
    if(!isEven.test(9)) System.out.println("9 is not even");

    // Now, use a lambda expression that tests if a number
    // is non-negative.
    NumericTest isNonNeg = (n) -> n >= 0;

    if(isNonNeg.test(1)) System.out.println("1 is non-negative");
    if(!isNonNeg.test(-1)) System.out.println("-1 is negative");
  }
}
```

Output

```
10 is even
9 is not even
1 is non-negative
-1 is negative
```

Lambda Expression

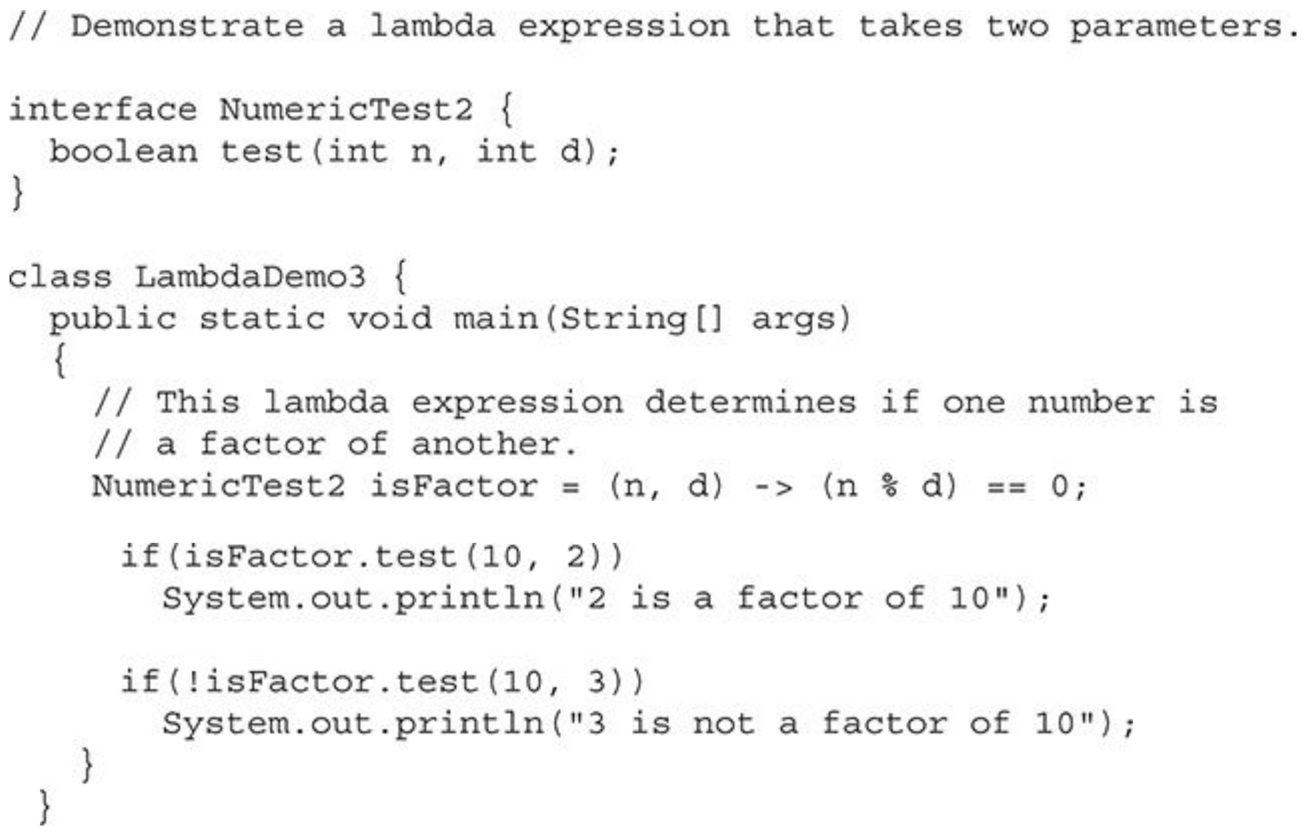
A lambda expression is not executed on its own. Rather, it forms the implementation of the abstract method defined by the functional interface that specifies its target type.

As a result, a lambda expression can be specified only in a context in which a target type is defined. One of these contexts is created when a lambda expression is assigned to a functional interface reference. Other target type contexts include variable initialization, return statements, and method arguments, to name a few.

Lambda Expression

```
interface myNumber{
    double getValue();
}
// Create a reference to a MyNumber instance.
MyNumber myNum;
// Use a lambda in an assignment context.
myNum = () -> 123.45;
// Call getValue(), which is implemented by the previously assigned lambda expression.
System.out.println(myNum.getValue());
```

Lambda Expression



```
// Demonstrate a lambda expression that takes two parameters.

interface NumericTest2 {
  boolean test(int n, int d);
}

class LambdaDemo3 {
  public static void main(String[] args)
  {
    // This lambda expression determines if one number is
    // a factor of another.
    NumericTest2 isFactor = (n, d) -> (n % d) == 0;

     if(isFactor.test(10, 2))
       System.out.println("2 is a factor of 10");

     if(!isFactor.test(10, 3))
       System.out.println("3 is not a factor of 10");
   }
 }
```

Output

```
2 is a factor of 10
3 is not a factor of 10
```

Expression Lambda Vs. Block Lambda

Lambdas that have expression bodies are sometimes called expression lambdas.

Lambdas that have block bodies are sometimes referred to as block lambdas.

```java
// A block lambda that computes the factorial of an int value.

interface NumericFunc {
  int func(int n);
}

class BlockLambdaDemo {
  public static void main(String[] args)
  {

    // This block lambda computes the factorial of an int value.
    NumericFunc factorial = (n) ->  {
      int result = 1;

      for(int i=1; i <= n; i++)
        result = i * result;

      return result;
    };

    System.out.println("The factoral of 3 is " + factorial.func(3));
    System.out.println("The factoral of 5 is " + factorial.func(5));
  }
}
```

Output

```
The factorial of 3 is 6
The factorial of 5 is 120
```

```
// A block lambda that reverses the characters in a string.

interface StringFunc {
  String func(String n);
}

class BlockLambdaDemo2 {
  public static void main(String[] args)
  {

    // This block lambda reverses the characters in a string.
    StringFunc reverse = (str) ->  {
      String result = "";
      int i;

      for(i = str.length()-1; i >= 0; i--)
        result += str.charAt(i);

      return result;
    };

    System.out.println("Lambda reversed is " +
                       reverse.func("Lambda"));
    System.out.println("Expression reversed is " +
                       reverse.func("Expression"));
  }
}
```

Output

```
Lambda reversed is
adbmaL

Expression reversed is
noisserpxE
```

```
// Use a generic functional interface with lambda expressions.

// A generic functional interface.
interface SomeFunc<T> {
  T func(T t);
}

class GenericFunctionalInterfaceDemo {
  public static void main(String[] args)
  {

    // Use a String-based version of SomeFunc.
    SomeFunc<String> reverse = (str) ->  {
      String result = "";
      int i;

      for(i = str.length()-1; i >= 0; i--)
        result += str.charAt(i);

      return result;
    };

    System.out.println("Lambda reversed is " +
                       reverse.func("Lambda"));
    System.out.println("Expression reversed is " +
                       reverse.func("Expression"));

    // Now, use an Integer-based version of SomeFunc.
    SomeFunc<Integer> factorial = (n) ->  {
      int result = 1;

      for(int i=1; i <= n; i++)
        result = i * result;

      return result;
    };

    System.out.println("The factoral of 3 is " + factorial.func(3));
    System.out.println("The factoral of 5 is " + factorial.func(5));
  }
}
```

```
Lambda reversed is adbmaL
Expression reversed is noisserpxE
The factoral of 3 is 6
The factoral of 5 is 120
```

Lambda Expression

A detailed real-life example can be found here: https://mkyong.com/java8/java-8-lambda-comparator-example/

End of Lecture 18.