Intro to Java (Page 2)

JUMP June 2019

Draft: 07/03/2019

Table of Contents

Deeper Dive into Methods and Classes

1. Intro
2. Passing Objects to Methods
3. Returning Objects
4. Method Overloading
5. Overloading Constructors
6. Recursion
7. The Concept of Static in Java
8. Nested Block Class
9. The Singleton Class
10. Varargs

Deeper Dive into Methods and Classes - Intro

1. Initially we learned to link data with code to manipulate it
2. Now we will learn to control access to class members
3. We have 2 basic types of class members
    1. Public: can be freely accessed by code defined outside of its class
    2. Private: can be accessed only by other methods defined by its class

Deeper Dive into Methods and Classes - Intro

1. Benefits of Private
    1. Access its data only through a set of well defined methods
    2. Prevent improper values from being assigned
        1. i.e: Range check
    3. Outside code can’t directly set data to any of the private class’ properties
    4. Control how and when data is used in the private class
    5. So with a “black box” class which is why a private class is, you can provide methods to use (to expose) and keep its internal from being modified

Deeper Dive into Methods and Classes - Intro

```
public class Main {


    public static void main(String[] args) {

        // TODO Auto-generated method stub

        Queue test = new Queue(10);


        // use the comments to try this code

        // it will not work

        // these items are private to the Queue class

        // thus they are shielded from being modified

        // directly by outside code

        // must use member functions designed

        // to change the object's data


        // test.q[0] = 99; // wrong!

        // test.putloc = -100; // won't work!

        System.out.println("Just testing that the code compiles and
runs in STS.");

    }

}
```

```
// class Queue demonstrating private properties members
class Queue {

    private char q[]; // this array holds the queue
    private int putloc, getloc; // the put and get indices

    // constructor
    Queue(int size) {
        q = new char[size]; // allocate memory for queue
        putloc = getloc = 0;
    }

    // Put a character into the queue.
    void put(char ch) {
        if(putloc==q.length) {
            System.out.println(" - Queue is full.");
            return;
        }
        q[putloc++] = ch;
    }

    // Get a character from the queue.
    char get() {
        if(getloc == putloc) {
            System.out.println(" - Queue is empty.");
            return (char) 0;
        }
        return q[getloc++];
    }

}
```

The following demonstrates what happens when you try to directly modify a private class member in code, test the lines and see the error.

Deeper Dive into Methods and Classes - Intro

Java's Access Modifiers

1. Public
2. Private
3. ***Protected*** (applies when inheritance is involved)
4. Basically, when using "Private", you can create properties and methods which are not exposed when a class is instantied as an object

Java encourages the concept of setters and getters to ensure that any given variable in a class is only modified via a method.

Of the many benefits of setters are the fact that the data is:

- Checked for type
- Validate for accuracy (range, length, etc..)

Deeper Dive into Methods and Classes - Intro

```
// Public vs private access.
class MyClass {
  private int alpha; // private access
  public int beta; // public access
  int gamma; // default access
  /* Methods to access alpha. It is OK for
a member of a class to access a private
member of the same class. */
  void setAlpha(int a) {
    if(a == 0) {
      alpha = 1;
    } else {
      alpha = a;
    }
  }
  int getAlpha() {
    return alpha;
  }
}
```

```
class AccessDemo {
  public static void main(String args[]) {
    MyClass ob = new MyClass();
    /* Access to alpha is allowed only
through its accessor methods. */
    ob.setAlpha(-99);
    System.out.println("ob.alpha is " +
ob.getAlpha());
    // You cannot access alpha like this:
    // ob.alpha = 10; // Wrong! alpha is
private!
    // These are OK because beta and gamma
are public.
    ob.beta = 88;
    ob.gamma = 99;
  }
}
```

Setters and Getters are simple concepts as you can infer from the above code.

Deeper Dive into Methods and Classes – Passing Objects to Methods

1. Up to now, we've been passing primitive types as parameters to methods
2. Java supports the passing of objects

```
class PassOb {
   public static void main(String args[]) {
      Block ob1 = new Block(10, 2, 5);
      Block ob2 = new Block(10, 2, 5);
      Block ob3 = new Block(4, 5, 5);
      System.out.println("ob1 same dimensions as ob2: " +
ob1.sameBlock(ob2));
      System.out.println("ob1 same dimensions as ob3: " +
ob1.sameBlock(ob3));
      System.out.println("ob1 same volume as ob3: " +
ob1.sameVolume(ob3));
   }
}
```

```
// Objects can be passed to methods.
class Block {
   int a, b, c;
   int volume;

   Block(int i, int j, int k) {
      a = i;
      b = j;
      c = k;
      volume = a * b * c;
   }

   // Return true if ob defines same block.
   boolean sameBlock(Block ob) {
      if((ob.a == a) & (ob.b == b) & (ob.c == c)) {
         return true;
      } else {
         return false;
      }
   }

   // Return true if ob has same volume.
   boolean sameVolume(Block ob) {
      if(ob.volume == volume) {
         return true;
      } else {
         return false;
      }
   }
}
```

Deeper Dive into Methods and Classes – Passing Objects to Methods

Arguments are passed…

As Primitives: by value

As Objects: by reference

Pass by Value

This means that it is a copy of the value itself being passed, the content of the original provider (if it was a primitive variable) will not be affected

Pass by Reference

This means that a pointer to the data is passed, and as such, the content of the original provider isn't being passed but the address where its content reside in memory. This means that when any content within that pointer is modified, it will affect any object which is linked to this area in memory. While the content may content primitive data, only objects are passed or referred to by reference.

Deeper Dive into Methods and Classes – Passing Objects to Methods

Call By Value

```
// Primitive types are passed by value.
class Test {
  /* This method causes no change to
the arguments used in the call. */
  void noChange(int i, int j) {
    i = i + j;
    j = -j;
  }
}
```

```
class CallByValue {
  public static void main(String args[]) {
    Test ob = new Test();
    int a = 15, b = 20;
    System.out.println("a and b before
call: " + a + " " + b);
    ob.noChange(a, b);
    System.out.println("a and b after call:
" + a + " " + b);
  }
}
```

Deeper Dive into Methods and Classes – Passing Objects to Methods

Call By Reference

```
// Objects are passed by reference
class Test {
  int a, b;
  Test(int i, int j) {
    a = i;
    b = j;
  }
  /* Pass an object. Now, ob.a and ob.b
in object used in the call will be
changed. */
  void change(Test ob) {
    ob.a = ob.a + ob.b;
    ob.b = -ob.b;
  }
}
```

```
class PassObRef {
  public static void main(String args[]) {
    Test ob = new Test(15, 20);
    Test ob2 = ob;
    System.out.println("ob.a and ob.b before
call: " + ob.a + " " + ob.b);
    System.out.println("ob2.a and ob2.b
before call: " + ob2.a + " " + ob2.b);
    ob.change(ob);
    System.out.println("ob.a and ob.b after
call: " + ob.a + " " + ob.b);
    System.out.println("ob2.a and ob2.b after
call: " + ob2.a + " " + ob2.b);
  }
}
```

Deeper Dive into Methods and Classes – Returning Objects

Return Methods can be any data type, including class types

```
// Return a programmer-defined object.
class Err {
   String msg; // error message
   int severity; // code indicating severity of error
   Err(String m, int s) {
      msg = m;
      severity = s;
   }
}
class ErrInfo {
   public static void main(String args[]) {
      ErrorInfo err = new ErrorInfo();
      Err e;
      e = err.getErrorInfo(2);
      System.out.println(e.msg + " severity: " +
e.severity);
      e = err.getErrorInfo(19);
      System.out.println(e.msg + " severity: " +
e.severity);
   }
}
```

```
class ErrorInfo {
   String msgs[] = {
      "Output Error",
      "Input Error",
      "Disk Full",
      "Index Out-Of-Bounds"
   };
   int howbad[] = { 3, 3, 2, 4 };

   Err getErrorInfo(int i) {
      if(i >= 0 & i < msgs.length) {
         return new Err(msgs[i], howbad[i]);
      } else {
         return new Err("Invalid Error Code", 0);
      }
   }
}
```

Deeper Dive into Methods and Classes – Method Overloading

Java allows for the ability for methods for share identical name, but have different method signature (aka, the passing of parameters)

With method overloading.. Comes.. Polymorphism.. At least one of the ways..

All you need to do is simply declare different, distinctive versions of the method and the rest is handled by the compiler

Distinction between methods must follow these criteria:

- Return type is different, and/or
- Numbers of parameters passed is different

See live code in STS "MethodOverloading"

Deeper Dive into Methods and Classes – Overloading Constructors

Constructors can be overloaded in the exact same manner as methods.

See live code in STS “OverloadDemo2”

Deeper Dive into Methods and Classes - Recursion

- Recursion is a method that calls itself
- In general term, it is circular programming
- **Pros of recursion** is that code can be kept simple
- **Cons of recursion** is that it may become expensive in memory use and may slow down processing when operations are heavy and recursion is deep
- Recursion must have a mechanism to stop from calling itself or you create an endless loop.

See live code in STS “RecursiveDemo”

Deeper Dive into Methods and Classes – The Concept of Static in Java

1. Static classes are typically use “as-is” without creating any instances
2. Precede the class declaration with the keyword “static”
3. Static classes can have any of its members access without creating a instance of it (no need to create an object)

See live code in STS “StaticDemo”

Deeper Dive into Methods and Classes – Nested Block Class

- Nested Class are in essence classes defined within classes.
- The scope of the nested class reside in its outer scope (parent)
- Typically we have 2 types of nested classes
  - Those that are static
  - Those that are not static
- The main reason for nested class is to break down your code's function in ways which are related
- Within your main class, the nested class provide essential code functionality, which by design, not convention, it is decided to keep together

See live code in STS "NestedBlockClass"

Deeper Dive into Methods and Classes – The Singleton Class

- A Singleton Class is a pattern where a class can only be created once at any point in time within the lifecycle of an application.
- Typically, the pattern is used to store global information, such as state, data queues, etc.
- The difference between a static class and a singleton class is that a static class doesn't need instantiation and it typically provides a bunch of methods as utilities
- The singleton class will be initialized only once, typically at the very beginning of an app and it's object can be used by any class and it will keep state and provide methods to share data across all class members

https://www.geeksforgeeks.org/singleton-class-java/

Deeper Dive into Methods and Classes - Varargs

- Variable Length Arguments aka Varargs
- The purpose is to create methods that will take a range of arguments as opposed to a fixed signature
- The vararg syntax is … (three periods) used in the method's parameter signature
- It will create an array which can be inspected for its content

See live code in STS "VarArgsDemo"

Inheritance

1. 101 Basics
2. Member Access
3. Constructors
4. Using “super” and Multilevel Hierarchy
5. When are constructors called in a class?
6. Superclass references and Subclass Objects
7.
8. Method Overriding
9. Overridden Methods support polymorphism
10. Why bother with overriding methods
11. The Abstract Class
12. Using the “final” keyword
13. The “Object” class

Inheritance – 101 Basics

- Create a base class
- Create subclasses which inherit from a parent class
- Refine the subclass with features which are context sensitive to its nature
- All subclass inherit from their parent class
- All subclass can use initial functionality along with their own unique additions
- Inheritance implementation is done using the “extends” keyword

See live code in STS “InheritanceDemo”

Inheritance – Member Access

1. Inheriting a class doesn’t overrule any “private” declarations
2. While this may seem an impediment in the design of classes, don’t forget we can expose setters and getters.

See live code in STS “InheritanceDemo”

Inheritance - Constructors

1. In a hierarchy, it is possible for both superclasses and subclasses to have their own constructors
2. Each take care of their own constructions as separate
3. Most classes typically have explicit constructors

See live code in STS “ConstructorAndInheritenceDemo”

Inheritance – Using “super” and Multilevel Hierarchy

1. When a subclass needs to invoke a constructor call explicitly, we use the “super” keyword as a call.
2. We can also use super to access class members.

```
// Using super to overcome name hiding.
class A {
   int i;
}
// Create a subclass by extending class A.
class B extends A {
   int i; // this i hides the i in A
   B(int a, int b) {
      super.i = a; // i in A
      i = b; // i in B
   }
   void show() {
      System.out.println("i in superclass: " + super.i);
      System.out.println("i in subclass: " + i);
   }
}
```

```
class UseSuper {
   public static void main(String args[]) {
      B subOb = new B(1, 2);
      subOb.show();
   }
}
```

See more live code in STS “MultilevelInheritence”

Inheritance – When are constructors called in a class?

In a class hierarchy constructors complete their execution in order of derivation from superclass to subclass

```java
// Create a super class.
class A {
   A() {
      System.out.println("Constructing A.");
   }
}

// Create a subclass by extending class A.
class B extends A {
   B() {
      System.out.println("Constructing B.");
   }
}

// Create another subclass by extending B.
class C extends B {
   C() {
      System.out.println("Constructing C.");
   }
}

class OrderOfConstruction {
   public static void main(String args[]) {
      C c = new C();
   }
}
```

Expect this output

```
Constructing A.
Constructing B.
Constructing C.
```

Inheritance – Superclass references and Subclass Objects

Since Java is strongly typed, type compatibility is strictly enforced. A reference variable for one class type in general cannot refer to an object of another type.

```
// This will not compile.
class X {
   int a;
   X(int i) { a = i; }
}

class Y {
   int a;
   Y(int i) { a = i; }
}

class IncompatibleRef {
   public static void main(String args[]) {
      X x = new X(10);
      X x2;
      Y y = new Y(5);
      x2 = x; // OK, both of same type
      x2 = y; // Error, not of same type
   }
}
```

Inheritance – Superclass references and Subclass Objects

The 'exception' to the rule from the preceding slide is when a superclass reference refers to a subclass object

```
/ A superclass reference can refer to a subclass object.
class X {
   int a;
   X(int i) { a = i; }
}

class Y extends X {
   int b;
   Y(int i, int j) {
      super(j);
      b = i;
   }
}

class SupSubRef {
   public static void main(String args[]) {
      X x = new X(10);
      X x2;
      Y y = new Y(5, 6);
      x2 = x; // OK, both of same type
      System.out.println("x2.a: " + x2.a);
      x2 = y; // still Ok because Y is derived from X
      System.out.println("x2.a: " + x2.a);
      // X references know only about X members
      x2.a = 19; // OK
      // x2.b = 27; // Error, X doesn't have a b member
   }
}
```

Inheritance – Method Overriding

When a method in a subclass as the exact same type and signature as the method in the superclass, this is called “Method Override”.

```
// Method overriding.
class A {
   int i, j;
   A(int a, int b) {
      i = a;
      j = b;
   }
   // display i and j
   void show() {
      System.out.println("i and j: " + i + " " + j);
   }
}

class B extends A {
   int k;
   B(int a, int b, int c) {
      super(a, b);
      k = c;
   }
   // display k - this overrides show() in A
   void show() {
      System.out.println("k: " + k);
   }
}
```

```
class Override {
   public static void main(String args[]) {
      B subOb = new B(1, 2, 3);
      subOb.show(); // this calls show() in B
   }
}
```

See live code in STS “MethodOverride”

Inheritance – Overridden Methods support Polymorphism

- Method overriding forms the basis for one of Java's most powerful concepts: "dynamic method dispatch".
- "Dynamic method dispatch" is a mechanism by which a call to an overridden method is resolved at run time rather than at compile time.
- This is essentially is how you achieve polymorphism

```
// Demonstrate dynamic method dispatch.
class Sup {
   void who() {
      System.out.println("who() in Sup");
   }
}
class Sub1 extends Sup {
   void who() {
      System.out.println("who() in Sub1");
   }
}
class Sub2 extends Sup {
   void who() {
      System.out.println("who() in Sub2");
   }
}
class DynDispDemo {
   public static void main(String args[]) {
      Sup superOb = new Sup();
      Sub1 subOb1 = new Sub1();
      Sub2 subOb2 = new Sub2();
      Sup supRef;
      supRef = superOb; supRef.who();
      supRef = subOb1; supRef.who();
      supRef = subOb2; supRef.who();
   }
}
```

Expect this output

```
who() in Sup
who() in Sub1
who() in Sub2
```

Inheritance – Why bother with overriding methods?

1. What does run-time polymorphism give us?
    1. It allows a general class to specify methods that will be common to all its derivatives (subclasses)
    2. It allows for derivatives to fine tune the methods for their specific context
    3. It is Java's "One Interface, Many Methods" implementation of polymorphism
    4. Ensuring a well design plan and strategy of the super/subclass relationship is key in creating an efficient polymorphic class implementation

Inheritance – The Abstract Class

1. The Abstract Class is basically a class which is a generalized form that will be shared by all subclass, but offers no actual code implementation by itself.
2. Basically a class which provides placeholder and an expectation of structure to be enforced

See more live code in STS “AbstractClass”

Inheritance – Using the “final” keyword

1. Overriding and Inheritance is great, until it is not and sometimes we need to prevent these features.
2. “final” is the keyword to use when we do not wish a class to be overridden or to be inherited from.
3. “final” can also be applied at the method level to ensure no methods can be overridden.
4. “final” can also be used with data members, this means they are immutable, they will not change during the lifetime of the execution of your app. In concept, this is a “constant”

Preventing method override

```
class A {
   final void meth() {
      System.out.println("This is a final
method.");
   }
}
class B extends A {
   void meth() { // ERROR! Can't override.
      System.out.println("Illegal!");
   }
}
```

Preventing class override

```
final class A {
// ...
}
// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
// ...
}
```

See more live code in STS “FinalInheritance”

Inheritance – The “Object” class

- The “Object” class is an implicit “superclass” of all other classes in Java.
- Object can refer to any other object.
- Even Arrays are Objects.
- The following table are methods every Object instance as access to.
- We used “finalize()” when we discussed the topic on Java’s Garbage Collection…

Method	Description
clone()	Creates and returns a copy of this object.
equals()	Indicates whether some other object is "equal to" this one.
finalize()	Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
getClass()	Returns the runtime class of an object.
hashCode()	Returns a hash code value for the object.
notify()	Wakes up a single thread that is waiting on this object's monitor.
notifyAll()	Wakes up all threads that are waiting on this object's monitor.
toString()	Returns a string representation of the object.
wait()	Causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.

Packages and Interfaces

1. Intro
2. Packages 101
3. Packages and Member Access
4. Understanding Protected Members
5. Importing Packages
6. Java’s Class Library.. Just a bunch of packages..
7. Interfaces
8. Extending Interfaces
9. Default Interface Methods
10. “static” methods in an interface

Packages and Interfaces - Intro

1. A Java Package is a group of related classes.
   1. Packages help organize your code and provide an extra layer of encapsulation to promote reusability and transport
2. An Interface defines a set of methods that will be implemented by a class
   - You specify what a class will do
   - You do NOT specify HOW it will do it

Packages and Interfaces – Packages 101

- Packages provide:
    - A mechanism by which related pieces of program can be organized as a unit
        - Classes defined in a package must be accessed via their package name
    - Package are part of Java’s access control mechanism
        - Classes defined within a package can be made private to that package and not be accessible to outside code, thus securely implementing an encapsulation mechanism which is tamper proof.
    - In Java a class is really a “namespace” and you cannot have identical namespaces in Java
    - Packages are an extra layer of abstraction and as such classes with same name across packages are not clashing, so packages are a way of partitioning code

Packages and Interfaces – Packages 101

- Defining a Package
  - To create a package we can use the general statement:
    - `package pkg;`
  - Java uses the file system of an operating system (OS) to manage packages
  - Each package is stored in its own directory/subdirectory
  - Package names like everything else in Java are “case-sensitive”.
  - Lowercase is used for naming packages (as a convention).
  - It’s not just that more than one file can be accessible in a package, but also, you can create a hierarchy of packages and you will use the dot “.” operator to access them.
    - `Package alpha.beta.signa.gamma;`

Packages and Interfaces – Packages 101

- Finding packages and a CLASSPATH
  - Since packages are mirrored to a directory path file structure, the Java run-time system will look for packages as follows:
    - By default, Java will use the current working directory
    - If your package is in a subdirectory, it will look for it
    - You can specify a CLASSPATH environment variable to associate the location of a package
    - You can use the –classpath option with java/javac to specify the path to your classes

STS has the ability to create packages using a wizard-like popup menu interface. This is how we will create our packages in this training session.

Packages and Interfaces – Packages 101

In our first set of slides “Java – Page 001.pptx” we had instructions on create a base Java app.

The difference now is that instead of using the ‘default’ package name, we will add our own as required.

Keep your naming convention to lowercase.

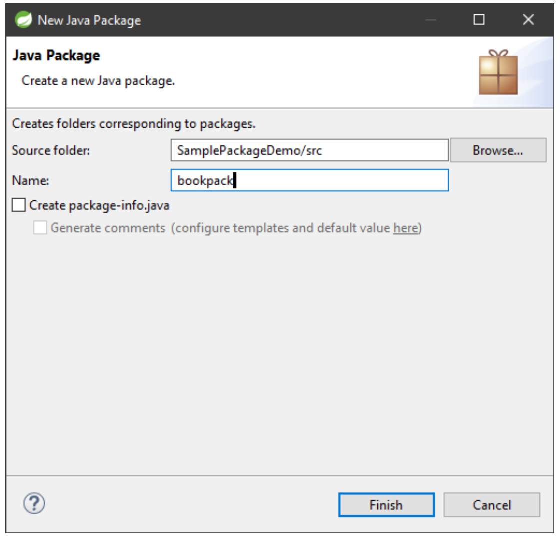
Packages and Interfaces – Packages 101

Let’s create a short package sample.

In STS ensure you create a new Java project and that you will name the package “bookpack”.

All Java files we create will be added to the package, ensure you add your class files correctly.

1) Create a Java Project
2) Create a package (File.. New.. Package..)
    1) You do so by ensuring you have selected the “src” folder, prior to invoking STS’s command to create a new package


New Java Package

Java Package

Create a new Java package.

Creates folders corresponding to packages.

Source folder: SamplePackageDemo/src

Name: bookpack

[ ] Create package-info.java

[ ] Generate comments (configure templates and default value here)

See live code in STS “SamplePackageDemo”

Packages and Interfaces – Packages and Member Access

1. What we have learned so far are the fundamentals of access control using the private and public modifiers.
2. Now that we are tackling the packages topic, we can further the discussion.
3. The visibility of an element is determine based on its access specification and these are: “private”, “public” and “protected”
4. Without an access modifier all contents in a package are available within itself, but never to the outside world.
5. Explicit declaration will ensure complete control of access
6. “protected” access means only available to subclasses

Packages and Interfaces – Packages and Member Access

	Private Member	Default Member	Protected Member	Public Member
Visible within same class	Yes	Yes	Yes	Yes
Visible within same package by subclass	No	Yes	Yes	Yes
Visible within same package by non-subclass	No	Yes	Yes	Yes
Visible within different package by subclass	No	No	Yes	Yes
Visible within different packages by non-subclass	No	No	No	Yes

* Default members are members which are not set with any access control.

Packages and Interfaces – Understanding Protected Members

1. Protected packages means that it’s members are accessible within the package, but cannot be arbitrarily modified by outside code.

See live code in STS “ProtectedPackageDemo”

Packages and Interfaces – Importing Packages

1. The import statement allows you to import an external package into your program.
2. When a package is imported, it can simplify your code by eliminating the need for the package name.
3. You can selectively import members or
4. You can import all members using the * wildcard character
5. Use the dot "." operator to drill down into members

See live code in STS "ProtectedPackageDemo"

Packages and Interfaces – Java’s Class Library.. Just a bunch of packages..

1. Implicit Packages
   1) Default no-name package (discouraged)
   2) java.lang (System is a subset from this package)
2. Other Popular Packages

Subpackage	Description
java.io	Contains I/O classes
java.net	Contains classes for networking support
java.applet	Contains classes for creating applets
java.awt	Contains classes for support of the Abstract Window Tookit (AWT)

Packages and Interfaces - Interfaces

1. Defining a class what it can do and not do it is very useful and we've seen how we can create "abstract" method.
2. The "abstract" method defines the signature, but has no implementation of code.
3. An interface in Java is the separation of implementation vs. definition.
4. The keyword is "interface"
5. To create an interface class, you must provide "bodies" for the methods described by the interface
6. Each class is free to determine the details of its own implementation (the code which executes)
7. To implement an interface a class must use the "implements" keyword
8. You can also create 'variable' reference in an Interface
9. Variables in an Interface can be used as constant.

See live code in STS "InterfaceDemo"

General Form

```
access interface name {
   ret-type method-name1(param-list);
   ret-type method-name2(param-list);
   type var1 = value;
   type var2 = value;
   // ...
   ret-type method-nameN(param-list);
   type varN = value;
}
```

Sample Interface Definition Code

```
public interface Series {
   int getNext(); // return next number in series
   void reset(); // restart
   void setStart(int x); // set starting value
}

/* this is declared as public and as such can be
implemented by code in any package */
```

Interface containing variables as constants

```
public interface Series {
interface IConst {
   int MIN = 0;
   int MAX = 10;
   String ERRORMSG = "Boundary Error";
}
```

Packages and Interfaces – Extending Interfaces

Interfaces like classes can be extended.

```
// One interface can extend another.
interface A {
   void meth1();
   void meth2();
}
// B now includes meth1() and meth2() - it
adds meth3().
interface B extends A {
   void meth3();
}
// This class must implement all of A and B
class MyClass implements B {
   public void meth1() {
      System.out.println("Implement
meth1().");
   }
   public void meth2() {
      System.out.println("Implement
meth2().");
   }
   public void meth3() {
      System.out.println("Implement
meth3().");
   }
}
```

```
class IFExtend {
   public static void main(String args[]) {
      MyClass ob = new MyClass();
      ob.meth1();
      ob.meth2();
      ob.meth3();
   }
}
```

Packages and Interfaces – Default Interface Methods

1. We can create Default Interface Methods
2. This means functioning code in your interface
3. This is new in JDK 8

```
public interface MyIF {
  // This is a "normal" interface method declaration.
  // It does NOT define a default implementation.
  int getUserID();
  // This is a default method. Notice that it provides
  // a default implementation.
  default int getAdminID() {
    return 1;
  }
}}
```

Packages and Interfaces – “static” methods in an interface

1. Interface since JDK 8 can now create static methods as well…

```
public interface MyIF {
// This is a "normal" interface method declaration.
// It does NOT define a default implementation.
  int getUserID();
  // This is a default method. Notice that it provides
  // a default implementation.
  default int getAdminID() {
    return 1;
  }
  // This is a static interface method.
  static int getUniversalID() {
    return 0;
  }
}
```