Intro to Java (Page 5)

JUMP June 2019

Draft: 07/11/2019

Table of Contents

Enumerations

1. Intro
2. Java Enumerations are Class Types
3. The “**values()**” and “**valueOf()**” methods
4. Constructors, Methods, Instance Variables
5. Enumerators inherit “Enum”

Enumerations - Intro

- Enumerations were added with JDK5
- An enumeration is a list of constant that define a new data type
- An object of an enumeration is bound/constrained to hold values that are defined by the list
- Conceptually, we are always using enumerations in our daily lives
    - Days of the Week: Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday
    - Coins: penny, nickel, dime, quarter, half-dollar and dollar
- Enumerations in programming are useful when you must define a set of values that represent a collection of items.
- We use them for all kinds of reasons, for example, various states or status
    - Success, waiting, failed, retrying..
- In the past we would define these variables as “final”, but with enumerations we now have access to a much more structured approach.

Enumerations - Intro

- A simple example of an implementation of an enumeration:

```
// An enumeration of transportation.
enum Transport {
      CAR, TRUCK, AIRPLANE, TRAIN, BOAT
}
```

- Once an enumeration is defined you can them like a data type
- Although an enumeration is a class, you never use the "new" keyword to instantiate them.

```
Transport airplane = Transport.AIRPLANE;
```

- And then you can use AirPlane as any other variable to compare, switch, etc…

```
if(airplane == Transport.TRAIN) { … }
switch(airplane) {
            case Transport.CAR:
            // …
            case Transport.TRUCK:
            // …
}
```

See live code in STS "EnumDemo"

Enumerations - Java Enumerations are Class Types

The “**values()**” and “**valueOf()**” methods are predefined with all enums

Enums are class types!

- But you don’t instantiate the with “new”
- We will see how we can give them constructors, instance variables, methods, etc…

See live code in STS “EnumDemo2”

Enumerations - Java Enumerations are Class Types

Each enumeration constant is an object of type “**enumeration**”.

Therefore, an enumeration can define Constructors, Methods, Instance Variables

See live code in STS “EnumDemo3”

Enumerations - Java Enumerations are Class Types

- You can't inherit a superclass when declaring an enum
- All enumerations automatically inherit one: java.lang.Enum.
- This class defines several methods that are available for use by all enumerations.
- Two that you may occasionally employ: **ordinal( )** and **compareTo( ).**

See live code in STS "EnumDemo4"

"**ordinal( )**" method obtains a value that indicates an enumeration constant's position in the list of constants. This is called its ordinal value.

- Ordinal values begin at zero.
- Thus, in the Transport enumeration, CAR has an ordinal value of zero, TRUCK has an ordinal value of 1, AIRPLANE has an ordinal value of 2, and so on.

```
final int ordinal( )
```

"**compareTo( )**" allows you to compare the ordinal value of two constants of the same enumeration:

```
final int compareTo(enum-type e)
```

Return value:

- If the invoking constant has an ordinal value less than e's, then **compareTo( )** returns a negative value.
- If the two ordinal values are the same, then zero is returned.
- If the invoking constant has an ordinal value greater than e's, then a positive value is returned.

Autoboxing

1. Intro
2. Type Wrappers
3. 101
4. Methods
5. In Expressions
6. Advice in usage

Autoboxing - Intro

- **autoboxing**: a feature to help turn primitive types into their object wrapper
- **autounboxing**: a feature to help turn object type wrappers into primitive types
- Was introduced with JDK 5.
- Will be a pivotal feature when working with Java generics
- Autoboxing/AutoUnboxing is directly related to Java's Type Wrappers.

Autoboxing – Type Wrappers

- Java uses primitive types, such as **int** or **double**, to hold the basic data types supported by the language.
- Primitive types, rather than objects, are used for these quantities for the sake of performance.
- Using objects for these basic types would add an unacceptable overhead to even the simplest of calculations.
- Thus, the primitive types are not part of the object hierarchy, and they do not inherit **Object**.

Autoboxing – Type Wrappers

Why use an Object representation?

- passing a value by reference
- most standard data structures in Java operate on objects, not primitive
- Type Wrappers are classes that encapsulate a primitive type within an object
- Each Type Wrapper offers a wide array of methods allowing you to fully integrate primitive type into the Java Object hierarchy

```
byte byteValue( )
double doubleValue( )
float floatValue( )
int intValue( )
long longValue( )
short shortValue( )
```

Autoboxing – Type Wrappers

For example, doubleValue( ) returns the value of an object as a double, floatValue( ) returns the value as a float, and so on. These methods are implemented by each of the numeric type wrappers.

All of the numeric type wrappers define constructors that allow an object to be constructed from a given value, or a string representation of that value. The also return the value of the object based on the data type they represent.

Here are the constructors defined for Integer and Double:

```
Integer(int num)

Integer(String str) throws
NumberFormatException

Double(double num)

Double(String str) throws
NumberFormatException
```

If str does not contain a valid numeric value, then a NumberFormatException is thrown.

All of the type wrappers override toString( ). It returns the human-readable form of the value contained within the wrapper. This allows you to output the value by passing a type wrapper object to println( ), for example, without having to convert it into its primitive type.

See live code in STS “TypeWrappers”

Autoboxing - 101

- **Autoboxing** is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed.

  There is no need to explicitly construct an object.

- **Autounboxing** is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed.

  There is no need to call a method such as **intValue( )** or **doubleValue( )**.

See live code in STS "AutoBoxing101"

Autoboxing - Methods

autoboxing/unboxing might occur when an argument is passed to a method or when a value is returned by a method

```
// This method has an Integer parameter.
static void m(Integer v) {
        System.out.println("m() received " + v);
}
// This method returns an int.
static int m2() {
        return 10;
}
// This method returns an Integer.
static Integer m3() {
        return 99; // autoboxing 99 into an Integer.
}
```

See live code in STS "AutoBoxingMethods"

Autoboxing – In Expressions

- Autoboxing and autounboxing take place whenever a conversion into an object or from an object is required and it applies to expressions as well.
- Within an expression, values are automatically unboxed on-the-fly.

See live code in STS “AutoBoxingExpressions”

Autoboxing – Advice on usage

Avoid abusing of autoboxing/unautoboxing.

```
// A bad use of autoboxing/unboxing!
   Double a, b, c;
   a = 10.2;
   b = 11.4;
   c = 9.8;
   Double avg = (a + b + c) / 3;
```

- In the above example, objects of type Double hold values, which are then averaged and the result assigned to another Double object.

- Although this code is technically correct and does, in fact, work properly, it is a very bad use of autoboxing/unboxing.
- It is far less efficient than the equivalent code written using the primitive type double.
- There is an overhead to consider when using the autoboxing/autounboxing features. That overhead translates into much larger CPU cycle to perform operations.
- Autoboxing/unboxing was not added to Java as a "back door" way of eliminating the primitive types.

Static Import

- Java supports an expanded use of the import keyword.
- By following import with the keyword static, an import statement can be used to import the static members of a class or interface.
- This is called static import.
- When using static import, it is possible to refer to static members directly by their names, without having to qualify them with the name of their class.
- This simplifies and shortens the syntax required to use a static member.

Static Import

- The following example located in the file "StaticImport" is a program computes the solutions to a quadratic equation, which has this form:

  `ax2 + bx + c = 0`

- The program uses two static methods from Java’s built-in math class Math, which is part of java.lang. The first is Math.pow( ), which returns a value raised to a specified power. The second is Math.sqrt( ), which returns the square root of its argument.

The use of the wildcard character “*” allows you to import all static members. Here’s what it would look like if we wanted to importatnt all members from Math.

```
import static java.lang.Math.*;
```

See live code in STS “NoStaticImport”

See live code in STS “StaticImport”

Annotations

- Java provides a feature that enables you to embed supplemental information into a source file.
- This information, called an annotation, does not change the actions of a program.
- However, this information can be used by various tools, during both development and deployment. For example, an annotation might be processed by a source-code generator, by the compiler, or by a deployment tool.
- The term metadata is also used to refer to this feature, but the term annotation is the most descriptive, and more commonly used.
- Annotation is a large and sophisticated topic, We are providing an overview to familiarized yourself with the concept and as we progress in the training sessions, we will use annotations on an as-required basis.

Annotations

- An annotation is created through a mechanism based on the interface. Here is a simple example:

```
// A simple annotation type.
@interface MyAnno {
    String str();
    int val();
}
```

- This declares an annotation called MyAnno. Notice the "@" that precedes the keyword interface.
- This tells the compiler that an annotation type is being declared. Next, notice the two members str() and val(). All annotations consist solely of method declarations. However, you don’t provide bodies for these methods. Instead, Java implements these methods. Moreover, the methods act much like fields.
- All annotation types automatically extend the Annotation interface. Thus, Annotation is a super-interface of all annotations. It is declared within the java.lang.annotation package.
- Originally, annotations were used to annotate only declarations. In this usage, any type of declaration can have an annotation associated with it.
- For example, classes, methods, fields, parameters, and enum constants can be annotated.
- Even an annotation can be annotated. In such cases, the annotation precedes the rest of the declaration.
- Beginning with JDK 8, you can also annotate a type use, such as a cast or a method return type.

Annotations

- When you apply an annotation, you give values to its members. For example, here is an example of MyAnno being applied to a method:

```
// Annotate a method.
@MyAnno(str = "Annotation Example", val = 100)
public static void myMeth() { // ... }
```

- This annotation is linked with the method myMeth( ). Look closely at the annotation syntax.
- The name of the annotation, preceded by an @, is followed by a parenthesized list of member initializations. To give a member a value, that member's name is assigned a value. Therefore, in the example, the string "Annotation Example" is assigned to the str member of MyAnno.
- Notice that no parentheses follow str in this assignment. When an annotation member is given a value, only its name is used. Thus, annotation members look like fields in this context.
- Annotations that don't have parameters are called marker annotations. These are specified without passing any arguments

See live code in STS "Annotation"

https://www.geeksforgeeks.org/annotations-in-java/

Generics

1. 101
2. Bounded Types
3. Wildcard Arguments
4. Bounded Wildcards
5. Generic Methods
6. Generic Constructors
7. Generic Interfaces
8. Raw Types and Legacy Code
9. Type Inference with the Diamond Operator
10. Erasure
11. Ambiguity Errors
12. Some Generic Restrictions

Generics - 101

- Generics, introduced in J2SE 5.0 has impacted the entire Java language.
- Generics add a completely new syntax element and caused changes to many of the classes and methods in the core API.
- The inclusion of generics fundamentally reshaped the character of Java.
- The term generics means parameterized types.
- Parameterized types enable you to create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter.
- A class, interface, or method that operates on a type parameter is called generic, as in generic class or generic method.
- A principal advantage of generic code is that it will automatically work with the type of data passed to its type parameter.
- Many algorithms are logically the same no matter what type of data they are being applied to.

Generics - 101

- For example, a Quicksort is the same whether it is sorting items of type Integer, String, Object, or Thread.
- With generics, you can define an algorithm once, independently of any specific type of data, and then apply that algorithm to a wide variety of data types without any additional effort.
- Prior to J2SE 5.0, we could create generalized classes, interfaces, and methods by operating through references of type Object.
- Because Object is the superclass of all other classes, an Object reference can refer to any type of object.
- Problem is that we could not do so with type safety because casts were needed to explicitly convert from Object to the actual type of data being operated upon.
- Side effect was to accidentally create type mismatches.
- Generics add the type safety capability that was lacking because they make these casts automatic and implicit.
- Generics expand your ability to reuse code and let you do so safely and reliably.

See live code in STS “GenericsExample01”

Generics - 101

Generics Work Only with Reference Types

When declaring an instance of a generic type, the type argument passed to the type parameter must be a reference type. You cannot use a primitive type, such as int or char. For example, with Gen, it is possible to pass any class type to T, but you cannot pass a primitive type to T. Therefore, the following declaration is illegal:

```
Gen<int> intOb = new Gen<int>(53); // Error, can't use primitive type
```

Of course, not being able to specify a primitive type is not a serious restriction because you can use the type wrappers (as the preceding example did) to encapsulate a primitive type. Further, Java’s autoboxing and auto-unboxing mechanism makes the use of the type wrapper transparent.

Generics - 101

Generic Types Differ Based on Their Type Arguments

A key point to understand about generic types is that a reference of one specific version of a generic type is not type-compatible with another version of the same generic type. For example, assuming the program just shown, the following line of code is in error and will not compile:

```
iOb = strOb; // Wrong!
```

Even though both iOb and strOb are of type Gen<T>, they are references to different types because their type arguments differ. This is part of the way that generics add type safety and prevent errors.

Generics - 101

Generic Class with 2 Types

You can declare more than one type parameter in a generic type. To specify two or more type parameters, simply use a comma-separated list. For example, the following **TwoGen** class is a variation of the **Gen** class that has two type parameters.

See live code in STS "GenericsExample02"

Generics - 101

General Form of a Generic class

```
class class-name<type-param-list> {
      // ...
}


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

Generics – Bounded Types

- In the preceding examples, the type parameters could be replaced by any class type.
- Sometimes it is useful to limit the types that can be passed to a type parameter.
- For example, assume that you want to create a generic class that stores a numeric value and is capable of performing various mathematical functions, such as computing the reciprocal or obtaining the fractional component.
- Furthermore, you want to use the class to compute these quantities for any type of number, including integers, floats, and doubles.
- Thus, you want to specify the type of the numbers generically, using a type parameter. To create such a class, you might try something like the following to the right.

```
// NumericFns attempts (unsuccessfully) to create
// a generic class that can compute various
// numeric functions, such as the reciprocal or the
// fractional component, given any type of number.
class NumericFns<T> {
    T num;
    // Pass the constructor a reference to
    // a numeric object.
    NumericFns(T n) {
        num = n;
    }
    // Return the reciprocal.
    double reciprocal() {
        return 1 / num.doubleValue(); // Error!
    }
    // Return the fractional component.
    double fraction() {
        return num.doubleValue() - num.intValue(); // Error!
    }
    // ...
}
```

See live code in STS "GenericsBoundedType"

Generics – Wildcard Arguments

- As useful as type safety is, sometimes it can get in the way of perfectly acceptable constructs.
- Given the NumericFns class shown before, assume you want to add a method called absEqual( ) that returns true if two NumericFns objects contain numbers whose absolute values are the same.
- You also want this method to be able to work properly no matter what type of number each object holds.
- If one object contains the Double value 1.25 and the other object contains the Float value –1.25, then absEqual( ) would return true.
- One way to implement absEqual( ) is to pass it a NumericFns argument, and then compare the absolute value of that argument against the absolute value of the invoking object, returning true only if the values are the same. For example, you want to be able to call absEqual( ), as shown here:

```
NumericFns<Double> dOb = new NumericFns<Double>(1.25);
NumericFns<Float> fOb = new NumericFns<Float>(-1.25);
if(dOb.absEqual(fOb)) {
   System.out.println("Absolute values are the same.");
} else {
   System.out.println("Absolute values differ.");
}
```

Generics – Wildcard Arguments

- Trouble with absEquals() starts as soon as you try to declare a parameter of type NumericFns. What type do you specify for NumericFns' type parameter? At first, you might think of a solution like this, in which T is used as the type parameter:

```
// This won't work!
// Determine if the absolute values of two objects are the same.
boolean absEqual(NumericFns<T> ob) {
   if(Math.abs(num.doubleValue()) == Math.abs(ob.num.doubleValue()) {
      return true;
   }
   return false;
}
```

- Here, the standard method Math.abs( ) is used to obtain the absolute value of each number, and then the values are compared. The trouble with this attempt is that it will work only with other NumericFns objects whose type is the same as the invoking object. For example, if the
- invoking object is of type NumericFns<Integer>, then the parameter ob must also be of type NumericFns<Integer>. It can't be used to compare an object of type NumericFns<Double>, for example. Therefore, this approach does not yield a general (i.e., generic) solution.

Generics – Wildcard Arguments

- To create a generic absEqual( ) method, you must use another feature of Java generics: the wildcard argument. The wildcard argument is specified by the ?, and it represents an unknown type. Using a wildcard, here is one way to write the absEqual( ) method:

```
// Determine if the absolute values of two
// objects are the same.
boolean absEqual(NumericFns<?> ob) {
  if(Math.abs(num.doubleValue()) == Math.abs(ob.num.doubleValue())) {
    return true;
  }
  return false;
}
```

See live code in STS “GenericsWildcard”

Generics – Bounded Wildcards

1. Wildcard arguments can be bounded in much the same way that a type parameter can be bounded.

2. A bounded wildcard is especially important when you are creating a method that is designed to operate only on objects that are subclasses of a specific superclass.

```
class A {
  // ...
}
class B extends A {
  // ...
}
class C extends A {
  // ...
}
// Note that D does NOT extend A.
class D {
  // ...
}
```

Generics – Bounded Wildcards

```
// A simple generic class.
class Gen<T> {
  T ob;
  Gen(T o) {
    ob = o;
  }
}
```

- Gen takes one type parameter, which specifies the type of object stored in ob. Because T is unbounded, the type of T is unrestricted. That is, T can be of any class type.
- Assume we need to create a method that takes as an argument any type of Gen object so long as its type parameter is A or a subclass of A.
- We will need a method that operates only on objects of Gen<type>, where type is either A or a subclass of A. To accomplish this, you must use a bounded wildcard.
- Here is a method called test( ) that accepts as an argument only Gen objects whose type parameter is A or a subclass of A.

```
// Here, the ? will match A
// or any class type
// that extends A.
static void test(Gen<?
extends A> o) {
  // ...
}
```

See live code in STS “GenericsBoundedWildcard”

Generics – Generic Methods

- Methods inside a generic class can make use of a class' type parameter and are, therefore, automatically generic relative to the type parameter.

- So it is possible to declare a generic method that uses one or more type parameters of its own. Furthermore, it is possible to create a generic method that is enclosed within a nongeneric class.

See live code in STS "GenericsMethods"

Generics – Generic Constructors

A constructor can be generic, even if its class is not.

See live code in STS “GenericConstructors”

Generics – Generic Interfaces

- An interface can be generic.
- Generic interfaces are specified just like generic classes.

See live code in STS “GenericsInterfaces”

Generics - Raw Types and Legacy Code

- Support for generics did not exist prior to JDK 5, therefore Java had to provide a transition mechanism path from old, pre-generics code.
- Pre-generics legacy code remains both functional and compatible with generics.
- The net effect is that you can expect both pre-generics code to work with generics, and generic code to work with pre-generics code.
- In order to provide this mechanism, Java allows a generic class to be used without any type arguments.
- This creates a raw type for the class.
- This raw type is compatible with legacy code, which has no knowledge of generics.
- The main drawback to using the raw type is that the type safety of generics is lost.

See live code in STS “GenericsRawTypeLegacy”

Generics – Type Inference with the Diamond Operator

1. A short form syntax is available since JDK 7

Sample Class

```
class TwoGen<T, V> {
   T ob1;
   V ob2;
   // Pass the constructor
a reference to
   // an object of type T.
   TwoGen(T o1, V o2) {
      ob1 = o1;
      ob2 = o2;
   }
   // ...
}
```

Prior to JDK 7

```
TwoGen<Integer, String> tgOb = new TwoGen<Integer, String>(42, "testing");
```

JDK 7+

```
TwoGen<Integer, String> tgOb = new TwoGen<>(42, "testing");
```

Generics - Erasure

- It is rarely necessary for the programmer to know the details about how the Java compiler transforms your source code into object code.
- In the case of generics, some general understanding of the process is important because it explains why the generic features work as they do—and why their behavior is sometimes a bit surprising.
- An important constraint that governed the way generics were added to Java was the need for compatibility with previous versions of Java.
- Simply put: generic code had to be compatible with preexisting, nongeneric code.
- Thus, any changes to the syntax of the Java language, or to the JVM, had to avoid breaking older code.
- The way Java implements generics while satisfying this constraint is through the use of erasure.

Generics - Erasure

Erasure 101

- When your Java code is compiled, all generic type information is removed (erased).
- This implies that the JVM is
    1. replacing type parameters with their bound type, which is Object if no explicit bound is specified, and
    2. applying the appropriate casts (as determined by the type arguments) to maintain type compatibility with the types specified by the type arguments.
- The compiler also enforces this type compatibility.
- This approach to generics means that no type parameters exist at run time. They are simply a source-code mechanism.

Generics – Ambiguity Errors

- The inclusion of generics gives rise to a new type of error that you must guard against: ambiguity.

- Ambiguity errors occur when erasure causes two seemingly distinct generic declarations to resolve to the same erased type, causing a conflict. Here is an example that involves method overloading:

```
// Ambiguity caused by erasure on
// overloaded methods.
class MyGenClass<T, V> {
   T ob1;
   V ob2;
   // ...
   // These two overloaded methods
   // are ambiguous
   // and will not compile.
   void set(T o) {
      ob1 = o;
   }
   void set(V o) {
      ob2 = o;
   }
}
```

These 2 methods are ambiguous as they both have the same signature

Solution: change the name/signature to provide distinction

Generics – Some Generic Restrictions

There are a few restrictions that you need to keep in mind when using generics. They involve creating objects of a type parameter, static members, exceptions, and arrays.

- Type Parameters Can’t Be Instantiated
- Restrictions on Static Members
- Generic Array Restrictions
- Generic Exception Restriction

Generics – Some Generic Restrictions

Type Parameters Can’t Be Instantiated

It is not possible to create an instance of a type parameter. For example, consider this class:

```
// Can't create an instance of T.
class Gen<T> {
      T ob;
      Gen() {
            ob = new T(); // Illegal!!!
      }
}
```

Here, it is illegal to attempt to create an instance of T. The reason should be easy to understand: the compiler has no way to know what type of object to create. T is simply a placeholder.

Generics – Some Generic Restrictions

Restrictions on Static Members

No static member can use a type parameter declared by the enclosing class. For example, both of the static members of this class are illegal:

```
class Wrong<T> {
        // Wrong, no static variables of type T.
        static T ob;
        // Wrong, no static method can use T.
        static T getob() {
                return ob;
        }
}
```

Although you can't declare static members that use a type parameter declared by the enclosing class, you can declare static generic methods, which define their own type parameters, as was done earlier in this chapter.

Generics – Some Generic Restrictions

Generic Array Restrictions

There are two important generics restrictions that apply to arrays.

1. you cannot instantiate an array whose element type is a type parameter.
2. you cannot create an array of type-specific generic references.

```
// Generics and arrays.
class Gen<T extends Number>
{
  T ob;
  T vals[]; // OK
  Gen(T o, T[] nums) {
    ob = o;
    // This statement is
illegal.
    // vals = new T[10]; //
can't create an array of T
    // But, this statement
is OK.
    vals = nums; // OK to
assign reference to existent
array
  }
}
```

```
class GenArrays {
  public static void main(String
args[]) {
    Integer n[] = { 1, 2, 3, 4, 5 };
    Gen<Integer> iOb = new
Gen<Integer>(50, n);
    // Can't create an array of
type-specific generic references.
    // Gen<Integer> gens[] = new
Gen<Integer>[10]; // Wrong!
    // This is OK.
    Gen<?> gens[] = new Gen<?>[10];
// OK
  }
}
```

Generics – Some Generic Restrictions

Generic Array Restrictions (from previous)

As the program shows, it’s valid to declare a reference to an array of type T, as this line does:

```
T vals[]; // OK
```

But, you cannot instantiate an array of T, as this commented-out line attempts:

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

The reason you can’t create an array of T is that there is no way for the compiler to know what type of array to actually create. However, you can pass a reference to a type-compatible array to Gen( ) when an object is created and assign that reference to vals, as the program does in this line:

```
vals = nums; // OK to assign reference to existent array
```

This works because the array passed to Gen() has a known type, which will be the same type as T at the time of object creation. Inside main( ), notice that you can’t declare an array of references to a specific generic type. That is, this line

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

won’t compile.

Generics – Some Generic Restrictions

Generic Exception Restriction

- A generic class cannot extend **Throwable**. This means that you cannot create generic exception classes.