Advanced Java and J2EE 17CS553
MODULE 1

Enumerations:

Versions of Java prior to JDK 5 lacked few features.

Three relatively recent additions to the Java language after JDK 5: enumerations, autoboxing, and annotations (also referred to as metadata)

- An **enumeration** is a list of named constants
- Java enumerations appear similar to enumerations in other languages
- In Java, an enumeration defines a class type - By making enumerations into classes, the capabilities of the enumeration are greatly expanded. For example, in Java, an enumeration can have constructors, methods, and instance variables

Some examples:

- dayOfWeek: SUNDAY, MONDAY, TUESDAY, …
- month: JAN, FEB, MAR, APR, …
- gender: MALE, FEMALE
- title: MR, MRS, MS, DR
- appletState: READY, RUNNING, BLOCKED, DEAD

In the past, enumerations were usually represented as integer values:

public final int SPRING = 0;

public final int SUMMER = 1;

public final int FALL = 2;

public final int WINTER = 3;

Now

enum Season { WINTER, SPRING, SUMMER, FALL }

Example: An enumeration of apple varieties.

enum Apple{ Jonathan, GoldenDel, RedDel, Winesap, Cortland }

The identifiers **Jonathan, GoldenDel**, and so on, are called enumeration constants.

Each is implicitly declared as a public, static final member of Apple.

Type of the enumeration - Apple in this case.

Thus, in the language of Java, these constants are called self-typed, in which “self” refers to the enclosing enumeration.

However, even though enumerations define a class type, you do not instantiate an enum using new.

- Declare and use an enumeration variable

  Apple ap;

Because ap is of type Apple, the only values that it can be assigned (or can contain) are those defined by the enumeration.

For example, this assigns ap the value RedDel:

ap = Apple.RedDel;

Usage:

Two enumeration constants can be compared for equality by using the = = relational operator.

if(ap == Apple.GoldenDel)

Use an enum to control a switch statement.

```
switch(ap)
{
 case Jonathan:
 case Winesap:
}
```

The values( ) and valueOf( ) Methods:

All enumerations automatically contain two predefined methods:

values( ) and valueOf( ).

public static enum-type[ ] values( )

The values( ) method returns an array that contains a list of the enumeration constants

public static enum-type valueOf(String str)

The valueOf( ) method returns the enumeration constant whose value corresponds to the string passed in str.

Sample Program:

```
// Use the built-in enumeration methods.

// An enumeration of apple varieties.
enum Apple {
  Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

class EnumDemo2 {
  public static void main(String args[])
  {
    Apple ap;

    System.out.println("Here are all Apple constants:");

    // use values()
    Apple allapples[] = Apple.values();
    for(Apple a : allapples)
      System.out.println(a);

    System.out.println();

    // use valueOf()
    ap = Apple.valueOf("Winesap");
    System.out.println("ap contains " + ap);

  }
}
```

The output from the program is shown here:

```
Here are all Apple constants:
Jonathan
GoldenDel
RedDel
Winesap
Cortland

ap contains Winesap
```

Java Enumerations Are Class Types:

- Don't instantiate an **enum** using **new**,
- Has much the same capabilities as other classes.
- The fact that **enum** defines a class gives the Java enumeration extraordinary power.
- For example, you can give them constructors, add instance variables and methods, and even implement interfaces.
- It is important to understand that each enumeration constant is an object of its enumeration type.
- Thus, when you define a constructor for an **enum**, the constructor is called when each enumeration constant is created.
- Also, each enumeration constant has its own copy of any instance variables defined by the enumeration.

An Example –

```
// Use an enum constructor, instance variable, and method.
enum Apple {
  Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);

  private int price; // price of each apple

  // Constructor
  Apple(int p) { price = p; }

  int getPrice() { return price; }
}

class EnumDemo3 {
  public static void main(String args[])
  {
    Apple ap;

    // Display price of Winesap.
    System.out.println("Winesap costs " +
                       Apple.Winesap.getPrice() +
                       " cents.\n");

    // Display all apples and prices.
```

```
        System.out.println("All apple prices:");
        for(Apple a : Apple.values())
          System.out.println(a + " costs " + a.getPrice() +
                             " cents.");
      }
    }
```

The output is shown here:

```
Winesap costs 15 cents.

All apple prices:
Jonathan costs 10 cents.
GoldenDel costs 9 cents.
RedDel costs 12 cents.
Winesap costs 15 cents.
Cortland costs 8 cents.
```

- The arguments to the constructor are specified, by putting them inside parentheses after each constant, as shown here:

Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);

- These values are passed to the **p** parameter of **Apple( )**, which then assigns this value to **price**.
- Because each enumeration constant has its own copy of **price**, you can obtain the price of a specified type of apple by calling **getPrice( )**

  Apple.Winesap.getPrice( )

- The preceding example contains only one constructor.
- An **enum** can offer two or more overloaded forms, just as can any other class.

```
// Use an enum constructor.
enum Apple {
  Jonathan(10), GoldenDel(9), RedDel, Winesap(15), Cortland(8);

  private int price; // price of each apple


      // Constructor
      Apple(int p) { price = p; }

      // Overloaded constructor
      Apple() { price = -1; }

      int getPrice() { return price; }
    }
```

Enumerations Inherit Enum:

- All enumerations automatically inherit one class: **java.lang.Enum**.
- String toString() returns the name of this enum constant, as contained in the declaration
- boolean equals(Object other) returns true if the specified object is equal to this enum constant
- int compareTo(E o) compares this enum with the specified object for order; returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object
- static enum-type valueOf(String s) returns the enumerated object whose name is s
- static enum-type[] values() returns an array of the enumeration objects
- final int ordinal( ) obtain a value that indicates an enumeration constant's position in the list of constants. This is called its ordinal value.

Using ordinal method:

```
// An enumeration of apple varieties.
enum Apple {
  Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

class EnumDemo4 {
  public static void main(String args[])
  {
    Apple ap, ap2, ap3;

    // Obtain all ordinal values using ordinal().
    System.out.println("Here are all apple constants" +
                       " and their ordinal values: ");
    for(Apple a : Apple.values())
      System.out.println(a + " " + a.ordinal());
```

```
Here are all apple constants and their ordinal values:
Jonathan 0

 GoldenDel 1
 RedDel 2
 Winesap 3
 Cortland 4
```

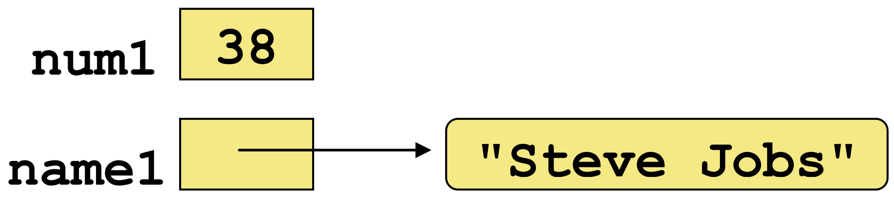


Fig 1: Primitive Types vs Objects to hold basic data types

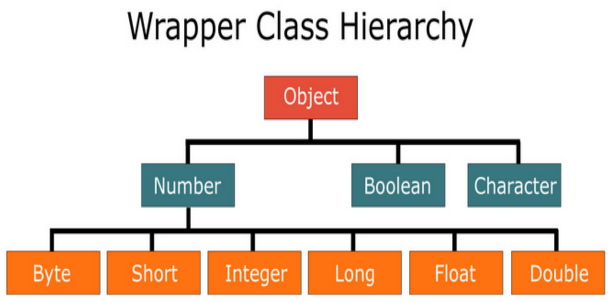
- Note that a primitive variable contains the value itself, but an object variable contains the address of the object, that is, a 'reference' to the object.
- An object reference can be thought of as a pointer to the location of the object. Rather than dealing with arbitrary addresses, we often depict a reference graphically.
- Java uses primitive types (also called simple types), 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 values would add an unacceptable overhead to even the simplest of calculations.
- The primitive types are not part of the object hierarchy, and they do not inherit Object.

Need of Type Wrappers:

- They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method.
- The classes in java.util package handles only objects and hence wrapper classes help in this case also.
- Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.
- An object is needed to support synchronization in multithreading.
- They relate directly to Java's autoboxing feature.

Wrapper Classes:

- Java provides type wrappers, which are classes that encapsulate a primitive type within an object
- The java.lang package contains *wrapper classes* that correspond to each primitive type.



- Wrapper classes also contain static methods that help manage the associated type. For example, the Integer class contains a method to convert an integer stored in a String to an int value:

```
int num;

num = Integer.parseInt(str);

                             Or

float  fltnum;

fltnum = Float.parseFloat(str);
```

Character:

- Character is a wrapper around a char.
- The constructor for Character is

Character(char ch)

Here, ch specifies the character that will be wrapped by the Character object being created.

Example: Character c=new Character('@');

- Beginning with JDK 9, the Character constructor has been deprecated and recommended to use the static method valueOf()to obtain a Character object.
  static Character valueOf(char ch)

  It returns a Character object that wraps ch.

- To obtain the char value contained in a Character object, call charValue(),
  char charValue()

It returns the encapsulated character.

Example: char c1=c.charValue();

Boolean:

- Boolean is a wrapper around boolean values.

Constructors: Boolean(boolean boolValue)

Here, boolValue must be either true or false

Boolean(String boolString)

Here, if boolString contains the string "true" (in uppercase or lowercase), then the new Boolean object will be true. Otherwise, it will be false.

Example: Boolean b=new Boolean(true);

- Beginning with JDK 9, the Boolean constructors have been deprecated and recommended to use the static method valueOf()to obtain a Boolean object.

static Boolean valueOf(boolean boolValue)

static Boolean valueOf(String boolString)

Each returns a Boolean object that wraps the indicated value.

- To obtain a boolean value from a Boolean object, use booleanValue(),

  boolean booleanValue()

It returns the boolean equivalent of the invoking object.

Example: boolean b1=b.booleanValue();

The Numeric Type Wrappers:

- The most commonly used type wrappers that represent numeric values.

  Byte, Short, Integer, Long, Float, and Double.

- All of the numeric type wrappers inherit the abstract class **Number.**
- **Number** declares methods that return the value of an object in each of the different number formats.

  1. byte byteValue() 2. double doubleValue()

  3. float floatValue() 4. int intValue()

  5. long longValue() 6. short shortValue()

- 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.

Example: constructors defined for Integer:

Integer(int num)
Integer(String str)

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

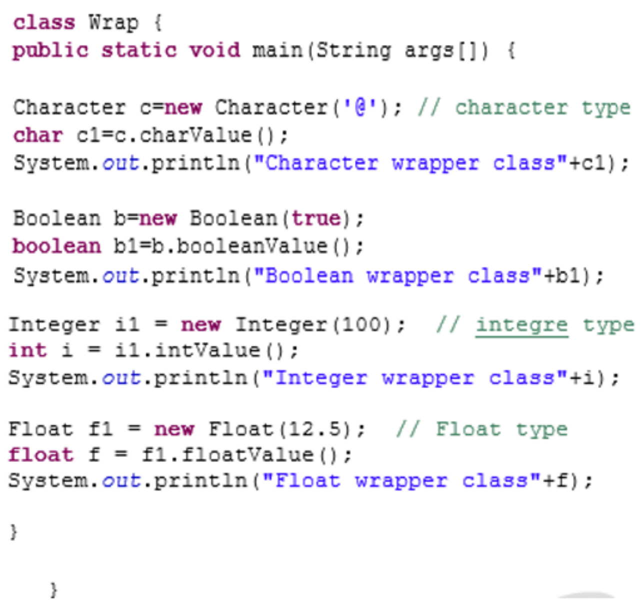
- Beginning with JDK 9, the numeric type-wrapper constructors have been deprecated and it is recommended to one of the valueOf() methods to obtain a wrapper object.

static Integer valueOf(int val)

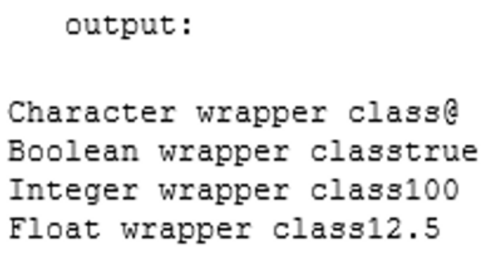
Static Integer valueOf(String valStr) throws NumberFormatException

Example: Integer iOb = Integer.valueOf(100);

- The valueOf() method is a static member of all of the numeric wrapper classes.
- All numeric classes support forms that convert a numeric value or a string into an object.
- 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.

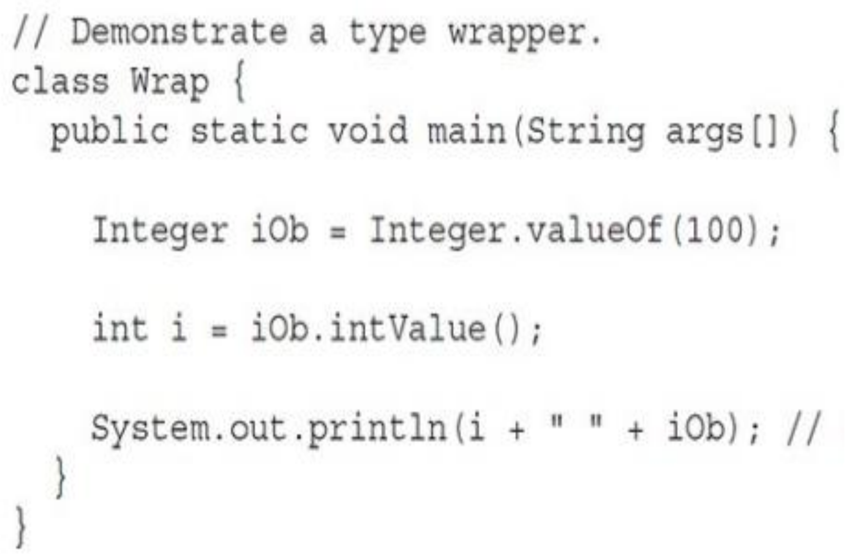


```
class Wrap {
public static void main(String args[]) {

Character c=new Character('@'); // character type
char c1=c.charValue();
System.out.println("Character wrapper class"+c1);

Boolean b=new Boolean(true);
boolean b1=b.booleanValue();
System.out.println("Boolean wrapper class"+b1);

Integer i1 = new Integer(100);  // integre type
int i = i1.intValue();
System.out.println("Integer wrapper class"+i);

Float f1 = new Float(12.5);  // Float type
float f = f1.floatValue();
System.out.println("Float wrapper class"+f);

}

    }
```



```
output:

Character wrapper class@
Boolean wrapper classtrue
Integer wrapper class100
Float wrapper class12.5
```

Program demonstrates how to use a numeric type wrapper to encapsulate a value and then extract that value.



```
// Demonstrate a type wrapper.
class Wrap {
  public static void main(String args[]) {

    Integer iOb = Integer.valueOf(100);

    int i = iOb.intValue();

    System.out.println(i + " " + iOb); //
  }
}
```

Output:
100 100

Summary:

- The process of encapsulating a value within an object is called **boxing**
- The process of extracting a value from a type wrapper is called **unboxing**
- The same general procedure used by the preceding program to box and unbox values has been available for use since the original version of Java.
- However, today, Java provides a more streamlined approach – **Auto boxing & Auto unboxing**

Autoboxing and Unboxing:
Question: There is no call to a method such as charValue() or valueOf(). Is the following code legal?
char a = ‘a’;
Character b = new Character(‘b’);
b = a;
a = b;
char c = new Character(‘c’);
Character d = ‘d’;

Answer: Yes! The code is perfectly valid.

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

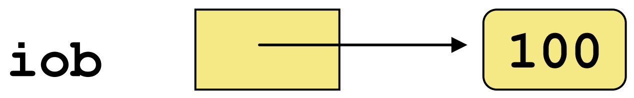
- For example – conversion of int to Integer, long to Long, double to Double etc.

Advantages:

- **Autoboxing** and **auto-unboxing** greatly streamline the coding of several algorithms, removing the tedium of manually boxing and unboxing values.
- They also help prevent errors.
- Moreover, they are very important to **generics**, which operate only on objects.
- Finally, autoboxing makes working with the **Collections Framework** much easier.

Example:
Integer iOb = 100; // autobox an int

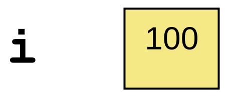
- With autoboxing, it is not necessary to manually construct an object in order to wrap a primitive type.
- You need only assign that value to a type-wrapper reference.
- **Java automatically constructs the object for you**.

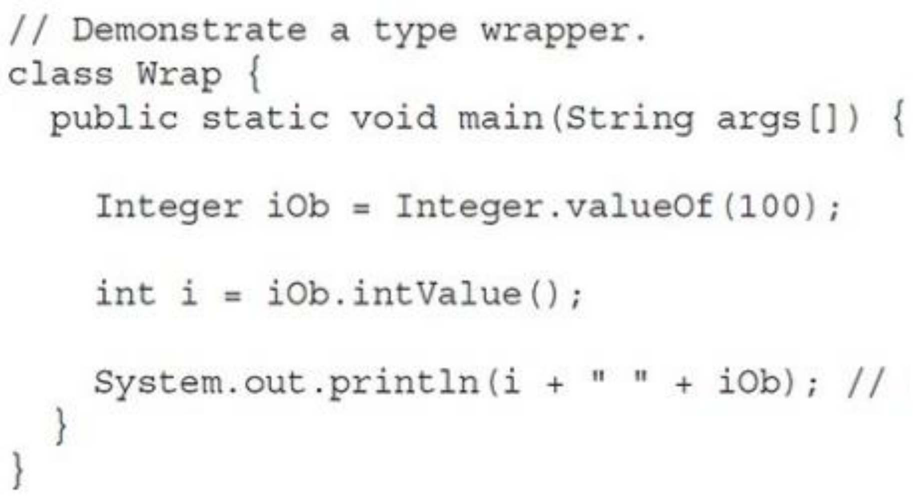


Notice that the object is not explicitly boxed. Java handles this for you, automatically

int i = iOb; // auto-unbox

- To unbox an object, simply assign that object reference to a primitive-type variable .
- Java automatically extracts value from boxed object.
- There is no need to call a method such as intValue( ) or doubleValue( ).





```
// Demonstrate a type wrapper.
class Wrap {
  public static void main(String args[]) {

    Integer iOb = Integer.valueOf(100);

    int i = iOb.intValue();

    System.out.println(i + " " + iOb); //
  }
}
```

```
// Demonstrate autoboxing/unboxing.
class AutoBox {
  public static void main(String args[]) {

    Integer iOb = 100; // autobox an int

    int i = iOb; // auto-unbox

    System.out.println(i + " " + iOb);
  }
}
```

Output:
Displays 100 100

Autoboxing & Methods:
Autoboxing/unboxing takes place with method parameters and return values.

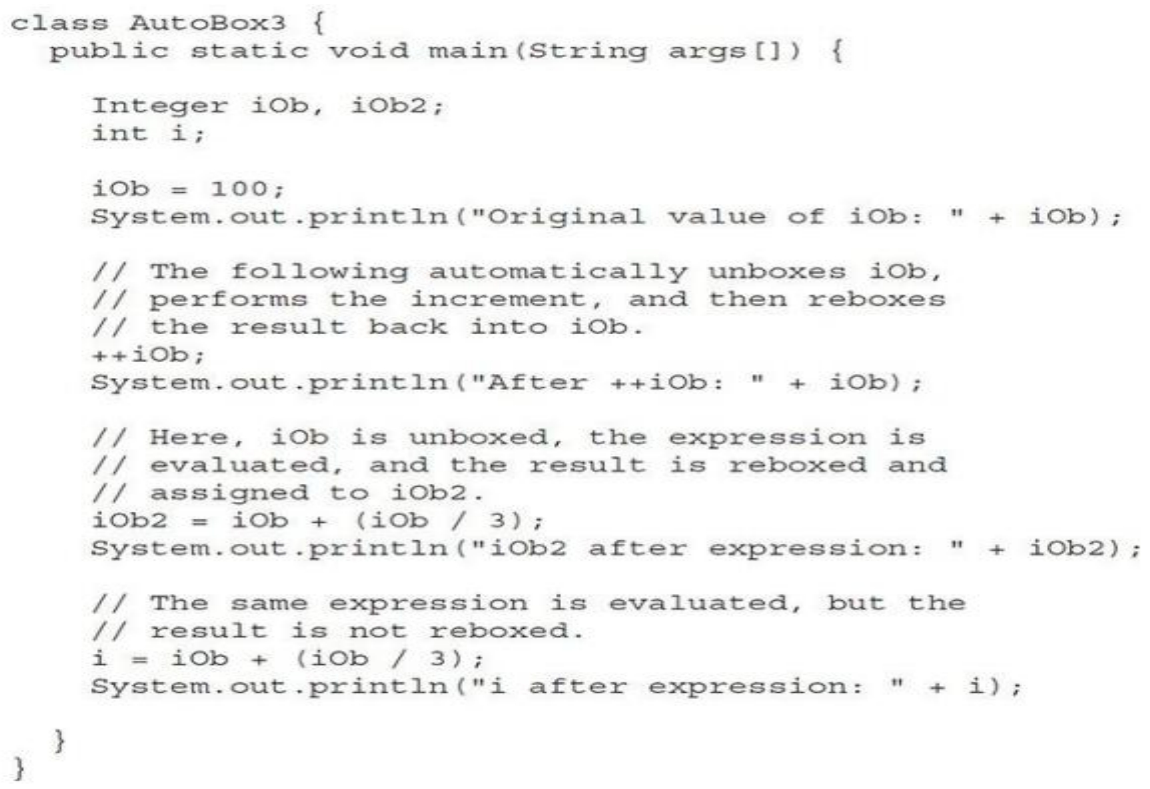
```
static int m(Integer v)
{
     return v;
}
Integer iob= m(100);
```

```
class AutoBox2 {
  // Take an Integer parameter and return
  // an int value;
  static int m(Integer v) {
    return v ; // auto-unbox to int
  }

  public static void main(String args[]) {
    // Pass an int to m() and assign the return value
    // to an Integer.  Here, the argument 100 is autoboxed
    // into an Integer.  The return value is also autoboxed
    // into an Integer.
    Integer iOb = m(100);

    System.out.println(iOb);
  }
}
```

Autoboxing/Unboxing in Expressions:

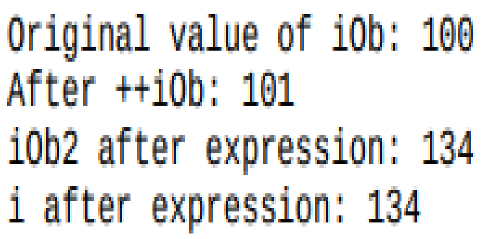
- Within an expression, a numeric object is automatically unboxed.
- The outcome of the expression is reboxed, if necessary.

```
Integer iob1, iob2;
int i;
iob1=100;
++iob1;
iob2= iob1 + (iob1 /3);
i= iob1 + (iob1 / 3);
```



```
class AutoBox3 {
  public static void main(String args[]) {

    Integer iOb, iOb2;
    int i;

    iOb = 100;
    System.out.println("Original value of iOb: " + iOb);

    // The following automatically unboxes iOb,
    // performs the increment, and then reboxes
    // the result back into iOb.
    ++iOb;
    System.out.println("After ++iOb: " + iOb);

    // Here, iOb is unboxed, the expression is
    // evaluated, and the result is reboxed and
    // assigned to iOb2.
    iOb2 = iOb + (iOb / 3);
    System.out.println("iOb2 after expression: " + iOb2);

    // The same expression is evaluated, but the
    // result is not reboxed.
    i = iOb + (iOb / 3);
    System.out.println("i after expression: " + i);

  }
}
```

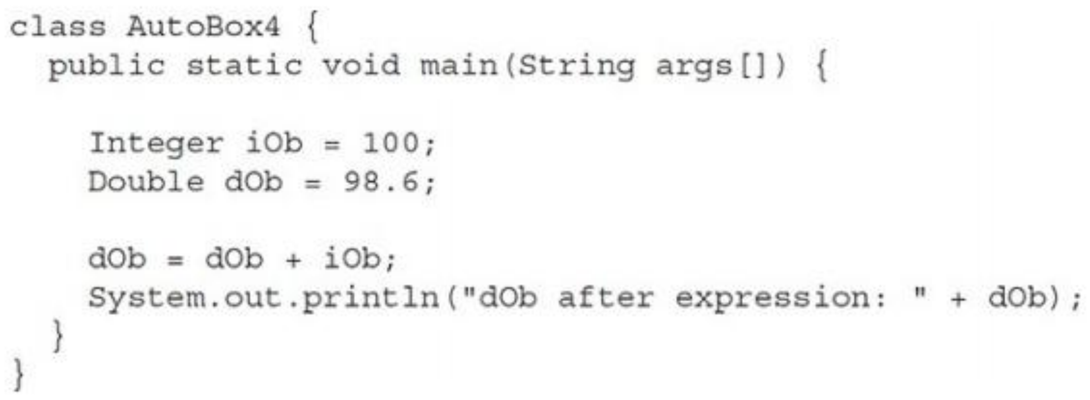
Output:



```
Original value of iOb: 100
After ++iOb: 101
iOb2 after expression: 134
i after expression: 134
```

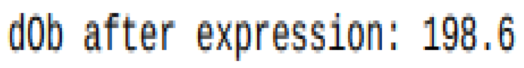
Mix of numeric objects in an expression:

- Auto-unboxing also allows you to mix different types of numeric objects in an expression.

- Once the values are unboxed, the standard type promotions and conversions are applied.
```
Integer iob = 100;
Double dob = 12.6;
dob = iob + dob;
```


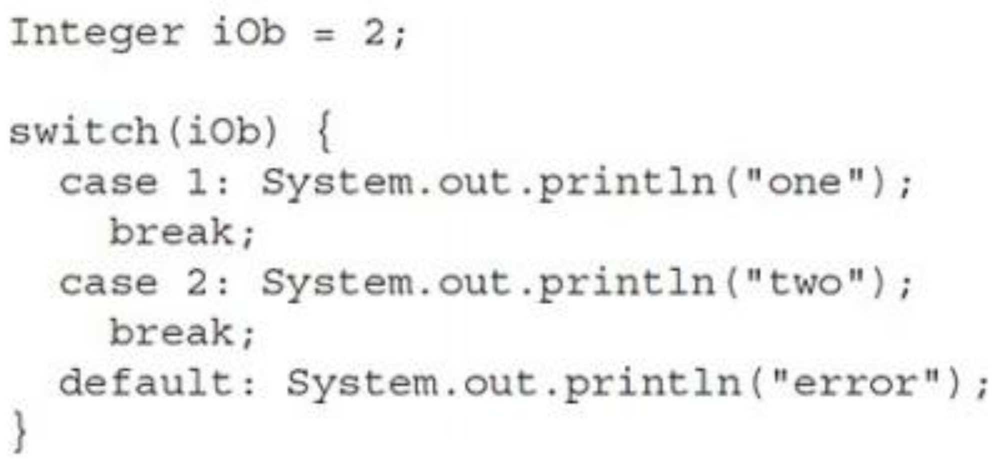
```
class AutoBox4 {
  public static void main(String args[]) {

    Integer iOb = 100;
    Double dOb = 98.6;

    dOb = dOb + iOb;
    System.out.println("dOb after expression: " + dOb);
  }
}
```

Output:



```
dOb after expression: 198.6
```

Integer numeric objects to control a switch statements:



```
Integer iOb = 2;

switch(iOb) {
  case 1: System.out.println("one");
    break;
  case 2: System.out.println("two");
    break;
  default: System.out.println("error");
}
```

Autoboxing/Unboxing Boolean and Character Values:

- Java also supplies wrappers for boolean and char. **Boolean and Character.**
- Autoboxing/unboxing applies to these wrappers, too.
- Because of auto-unboxing, a Boolean object can now also be used to control any of Java's loop statements.
- When a Boolean is used as the conditional expression of a while, for, or do/while, it is automatically unboxed into its boolean equivalent.
  Example:
  Boolean b; while (b) { //..

```
class AutoBox5 {
  public static void main(String args[]) {



    // Autobox/unbox a boolean.
    Boolean b = true;

    // Below, b is auto-unboxed when used in
    // a conditional expression, such as an if.
    if(b) System.out.println("b is true");

    // Autobox/unbox a char.
    Character ch = 'x'; // box a char
    char ch2 = ch; // unbox a char

    System.out.println("ch2 is " + ch2);
  }
}
```

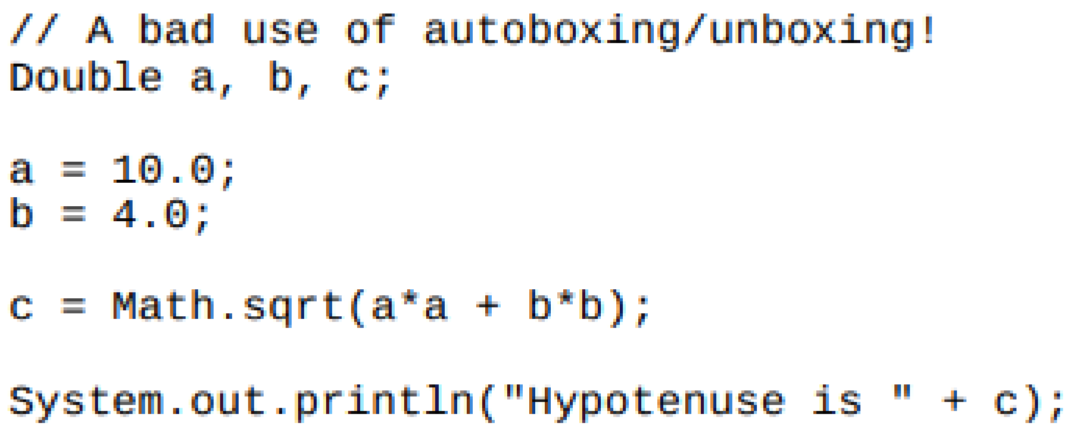
Output:

```
b is true
ch2 is x
```

Autoboxing/Unboxing Helps Prevent Errors:

```
// An error produced by manual unboxing.
class UnboxingError {
  public static void main(String args[]) {

    Integer iOb = 1000; // autobox the value 1000

    int i = iOb.byteValue(); // manually unbox as byte !!!

    System.out.println(i);  // does not display 1000 !
  }
}
```

A Word of Warning:



```
// A bad use of autoboxing/unboxing!
Double a, b, c;

a = 10.0;
b = 4.0;

c = Math.sqrt(a*a + b*b);

System.out.println("Hypotenuse is " + c);
```

- Restrict use of the type wrappers to only those cases in which an object representation of a primitive type is required.
- Each autobox and auto-unbox adds overhead that is not present if the primitive type is used.

Summary:

- Java provides a more streamlined approach – Auto boxing & Auto unboxing - Beginning with JDK 5.
- Autoboxing and auto-unboxing greatly streamline the coding of several algorithms, removing the tedium of manually boxing and unboxing values.
- Makes easier to work with Generics, Collection Framework.

Quiz:

Is it valid?

```
int sum(int a, int b)
 {
   return a + b;
 }
Integer a = sum(new Integer(1), 2);
```

Answer: YES

```
Object Identity(Object x)
{
return x;
}
double c = Identity(new Double(1.0));
```

- Answer: error: incompatible types: Object cannot be converted to double
  double c = Identity(new Double(1.0));

To resolve the issue: double c = (Double)Identity(new Double(1.0));

Annotations (Metadata):

- Since JDK 5, Java has supported 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.
- This information can be used by various tools during both development and deployment.
- The term metadata is also used to refer to this feature, but the term annotation is the most descriptive and more commonly used.

An annotation is created through a mechanism based on the interface.
Example. Here is the declaration for an annotation called MyAnno:

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

- @ that precedes the keyword interface - This tells the compiler that an annotation type is being declared.
- There are two members str( ) and val( ).
- All annotations consist solely of method declarations.
- Does not contain method implementation.
- Java implements these methods.
- The methods act much like fields
- An annotation cannot include an extends clause

More about Annotations:

- All annotation types automatically extend the Annotation interface (Super-interface).
- It is declared within the java.lang.annotation package.
- It overrides hashCode( ), equals( ), and toString( ), which are defined by Object.

It also specifies annotationType( ), which returns a Class object that represents the invoking annotation.

Usage – Foreword:

- 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 all cases, the annotation precedes the rest of the declaration.
- When you apply an annotation, you give values to its members

Example- Usage
MyAnno being applied to a method declaration.

```
// Annotate a method.
```

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

Specifying a Retention Policy:

- A retention policy determines at what point an annotation is discarded.
- Java defines three such policies, which are encapsulated within the java.lang.annotation.RetentionPolicy enumeration
- They are SOURCE, CLASS, and RUNTIME

Retention Policy:

- SOURCE is retained only in the source file and is discarded during compilation.
- CLASS is stored in the .class file during compilation. It is not available through the JVM during run time.
- RUNTIME is stored in the .class file during compilation and is available through the JVM during run time.

Thus, RUNTIME retention offers the greatest annotation persistence.

- A retention policy is specified for an annotation by using one of Java's built-in annotations: @Retention.

@Retention(retention-policy)

- Here, retention-policy must be one of the previously discussed enumeration constants.
- If no retention policy is specified for an annotation, then the default policy of CLASS is used.

Retention policy – Usage

MyAnno uses @Retention to specify the RUNTIME retention policy.
Thus, MyAnno will be available to the JVM during program execution.

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno
{   String str();   int val();
 }
```

Obtaining Annotations at Run Time by Use of Reflection

- Although annotations are designed mostly for use by other development or deployment tools, if they specify a retention policy of RUNTIME, then they can be queried at run time by any Java program through the use of reflection.
- Reflection is the feature that enables information about a class to be obtained at run time.
- The reflection API is contained in the java.lang.reflect package.

Steps to use Reflection:

- The first step to using reflection is to obtain a Class object that represents the class whose annotations you want to obtain

- Class is one of Java's built-in classes and is defined in java.lang.
- One of the easiest is to call getClass( ), which is a method defined by Object final Class<?> getClass( )

It returns the Class object that represents the invoking object
<?> - This is related to Java's generics feature

- If you want to obtain the annotations associated with a specific item declared within a class,
- First obtain an object that represents that item.
- For example, Class supplies (among others) the getMethod( ) - return objects of type Method
- getField( ) - return objects of type Field
- getConstructor( ) – return objects of type Constructor

Example:
An example that obtains the annotations associated with a method

- First obtain a Class object that represents the class
- Then call getMethod( ) on that Class object, specifying the name of the method
- getMethod( ) has this general form:

Method *getMethod*(String *methName*, Class<?> ... *paramTypes*)

1. The name of the method is passed in methName.
2. If the method has arguments, then Class objects representing those types must also be specified by paramTypes
3. If the method can't be found, NoSuchMethodException is thrown.

getAnnotation( )

- From a Class, Method, Field, or Constructor object, you can obtain a specific annotation associated with that object by calling getAnnotation( ).

General form :
<A extends Annotation> getAnnotation(Class<A> annoType)
annoType is a Class object that represents the annotation in which you are interested.
The method returns a reference to the annotation.
Using this reference, you can obtain the values associated with the annotation's members.
The method returns null if the annotation is not found, which will be the case if the annotation does not have RUNTIME retention.

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// An annotation type declaration.
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
  String str();
  int val();
}

class Meta {

  // Annotate a method.
  @MyAnno(str = "Annotation Example", val = 100)


  public static void myMeth() {
    Meta ob = new Meta();

    // Obtain the annotation for this method
    // and display the values of the members.
    try {
      // First, get a Class object that represents
      // this class.
      Class<?> c = ob.getClass();

      // Now, get a Method object that represents
      // this method.
      Method m = c.getMethod("myMeth");

      // Next, get the annotation for this class.
      MyAnno anno = m.getAnnotation(MyAnno.class);

      // Finally, display the values.
      System.out.println(anno.str() + " " + anno.val());
    } catch (NoSuchMethodException exc) {
      System.out.println("Method Not Found.");
    }
  }

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

- In the preceding example, myMeth( ) has no parameters. Thus, when getMethod( ) was called, only the name myMeth was passed.
- To obtain a method that has parameters, you must specify class objects representing the types of those parameters as arguments to getMethod( ).

```
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
  String str();
  int val();
}

class Meta {

  // myMeth now has two arguments.
  @MyAnno(str = "Two Parameters", val = 19)
  public static void myMeth(String str, int i)
  {
    Meta ob = new Meta();

    try {
      Class<?> c = ob.getClass();

      // Here, the parameter types are specified.
      Method m = c.getMethod("myMeth", String.class, int.class);

      MyAnno anno = m.getAnnotation(MyAnno.class);

      System.out.println(anno.str() + " " + anno.val());
    } catch (NoSuchMethodException exc) {
      System.out.println("Method Not Found.");
    }
  }

  public static void main(String args[]) {
    myMeth("test", 10);
  }
}
```

Output:
The output from this version is shown here:

```
   Two Parameters 19
```

myMeth( ) takes a String and an int parameter.
To obtain information about this method,
getMethod( ) must be called as shown here:
Method m = c.getMethod("myMeth", String.class, int.class);
Here, the Class objects representing String and int are passed as additional arguments.

Obtaining Annotations at Run Time by Use of Reflection:

- Although annotations are designed mostly for use by other development or deployment tools, if they specify a retention policy of RUNTIME, then they can be queried at run time by any Java program through the use of reflection.
- Reflection is the feature that enables information about a class to be obtained at run time.
- The reflection API is contained in the java.lang.reflect package.

Steps to use Reflection:

- The first step to using reflection is to obtain a Class object that represents the class whose annotations you want to obtain

- Class is one of Java’s built-in classes and is defined in java.lang.
- One of the easiest is to call getClass( ), which is a method defined by Object

final Class<?> getClass( )
It returns the Class object that represents the invoking object
<?> - This is related to Java’s generics feature

- If you want to obtain the annotations associated with a specific item declared within a class,
- First obtain an object that represents that item.
- For example, Class supplies (among others) the getMethod( ) - return objects of type Method
- getField( ) - return objects of type Field
- getConstructor( ) – return objects of type Constructor

Example:
An example that obtains the annotations associated with a method

- First obtain a Class object that represents the class
- Then call getMethod( ) on that Class object, specifying the name of the method
- getMethod( ) has this general form:

Method *getMethod*(String *methName*, Class<?> ... *paramTypes*)

1. The name of the method is passed in methName.
2. If the method has arguments, then Class objects representing those types must also be specified by paramTypes

If the method can’t be found, NoSuchMethodException is thrown.

getAnnotation( )

- From a Class, Method, Field, or Constructor object, you can obtain a specific annotation associated with that object by calling getAnnotation( ).

General form :
<A extends Annotation> getAnnotation(Class<A> annoType)
annoType is a Class object that represents the annotation in which you are interested.
The method returns a reference to the annotation.
Using this reference, you can obtain the values associated with the annotation’s members.
The method returns null if the annotation is not found, which will be the case if the annotation does not have RUNTIME retention.

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// An annotation type declaration.
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
  String str();
  int val();
}

class Meta {

  // Annotate a method.
  @MyAnno(str = "Annotation Example", val = 100)

  public static void myMeth() {
    Meta ob = new Meta();

    // Obtain the annotation for this method
    // and display the values of the members.
    try {
      // First, get a Class object that represents
      // this class.
      Class<?> c = ob.getClass();

      // Now, get a Method object that represents
      // this method.
      Method m = c.getMethod("myMeth");

      // Next, get the annotation for this class.
      MyAnno anno = m.getAnnotation(MyAnno.class);

      // Finally, display the values.
      System.out.println(anno.str() + " " + anno.val());
    } catch (NoSuchMethodException exc) {
      System.out.println("Method Not Found.");
    }
  }

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

Output:
The output from this version is shown here:

```
  Two Parameters 19
```

myMeth( ) takes a String and an int parameter.
To obtain information about this method,
getMethod( ) must be called as shown here:
Method m = c.getMethod("myMeth", String.class, int.class);
Here, the Class objects representing String and int are passed as additional arguments.

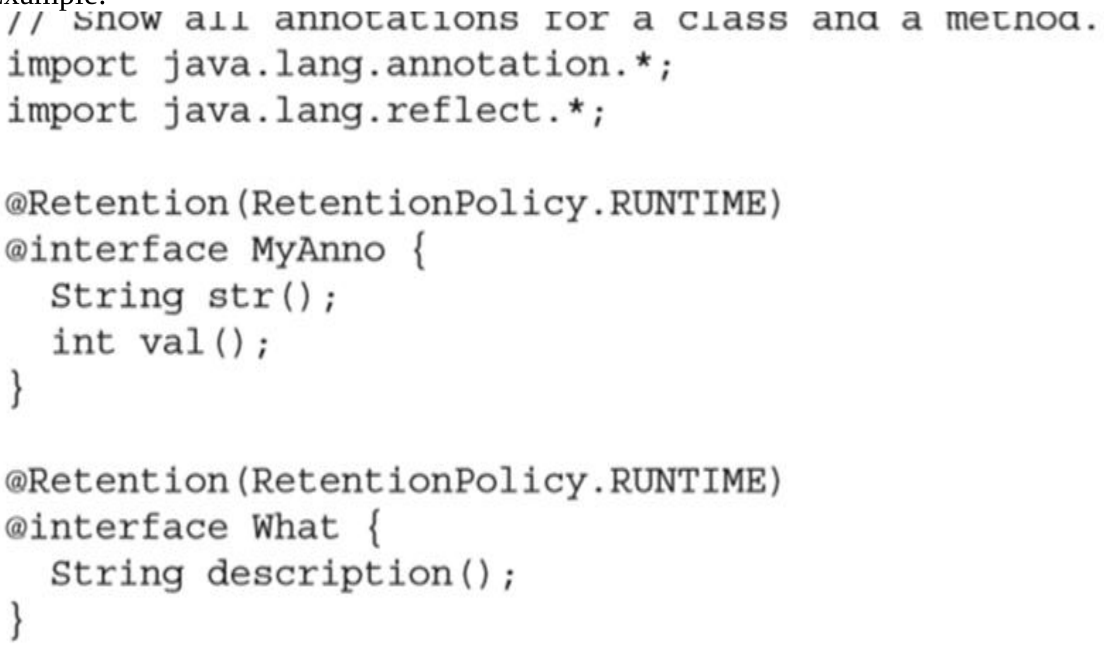
Obtaining All Annotations:

- You can obtain all annotations that have RUNTIME retention that are associated with an item by calling getAnnotations( ) on that item.

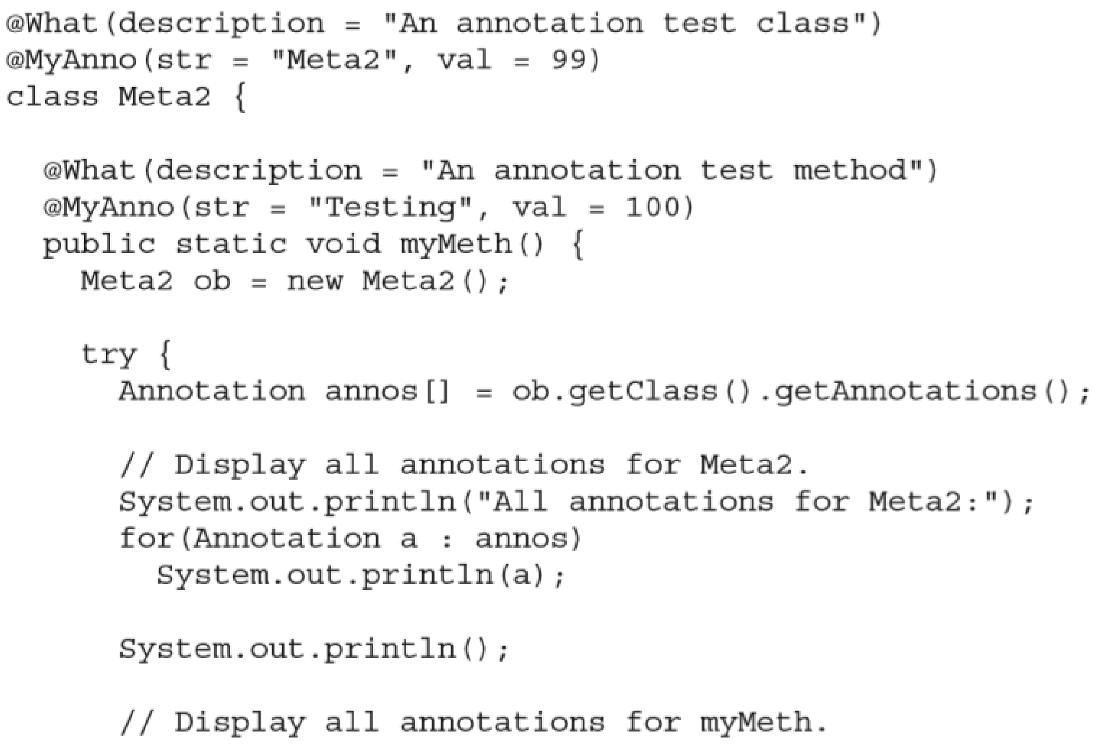
Annotation[ ] getAnnotations( )

- It returns an array of the annotations.
- getAnnotations( ) can be called on objects of type Class, Method, Constructor, and Field, among others

Example:



```
// Show all annotations for a class and a method.
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
  String str();
  int val();
}

@Retention(RetentionPolicy.RUNTIME)
@interface What {
  String description();
}
```



```
@What(description = "An annotation test class")
@MyAnno(str = "Meta2", val = 99)
class Meta2 {

  @What(description = "An annotation test method")
  @MyAnno(str = "Testing", val = 100)
  public static void myMeth() {
    Meta2 ob = new Meta2();

    try {
      Annotation annos[] = ob.getClass().getAnnotations();

      // Display all annotations for Meta2.
      System.out.println("All annotations for Meta2:");
      for(Annotation a : annos)
        System.out.println(a);

      System.out.println();

      // Display all annotations for myMeth.
```

```
          Method m = ob.getClass( ).getMethod("myMeth");
          annos = m.getAnnotations();

          System.out.println("All annotations for myMeth:");
          for(Annotation a : annos)
            System.out.println(a);

        } catch (NoSuchMethodException exc) {
           System.out.println("Method Not Found.");
        }
      }

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

The output is shown here:

```
All annotations for Meta2:
@What(description=An annotation test class)
@MyAnno(str=Meta2, val=99)

All annotations for myMeth:
@What(description=An annotation test method)
@MyAnno(str=Testing, val=100)
```

The AnnotatedElement Interface:

- The methods getAnnotation( ) and getAnnotations( ) are defined by the AnnotatedElement interface, which is defined in java.lang.reflect
- AnnotatedElement defines several other methods.
- Two have been available since JDK 5. The first is getDeclaredAnnotations( ), which has this general form:

Annotation[ ] getDeclaredAnnotations( )

It returns all non-inherited annotations present in the invoking object.

- The second is isAnnotationPresent( ), which has this general form:

boolean isAnnotationPresent(Class<? extends Annotation> annoType)

- It returns true if the annotation specified by annoType is associated with the invoking object. It returns false otherwise.

Using Default Values:

- You can give annotation members default values that will be used if no value is specified when the annotation is applied

type member( ) default value ;

- Here, value must be of a type compatible with type.

// An annotation type declaration that includes defaults.
@Retention(RetentionPolicy.RUNTIME)
 @interface MyAnno
{
String str() default "Testing";

int val() default 9000;
}

Marker Annotations:

- A marker annotation is a special kind of annotation that contains no members.
- Its sole purpose is to mark an item.
- Thus, its presence as an annotation is sufficient.
- The best way to determine if a marker annotation is present is to use the method isAnnotationPresent( ), which is defined by the AnnotatedElement interface.
- Because a marker interface contains no members, simply determining whether it is present or absent is sufficient.

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// A marker annotation.
@Retention(RetentionPolicy.RUNTIME)
@interface MyMarker { }

class Marker {

  // Annotate a method using a marker.
  // Notice that no ( ) is needed.
  @MyMarker
  public static void myMeth() {
    Marker ob = new Marker();

    try {
      Method m = ob.getClass().getMethod("myMeth");

      // Determine if the annotation is present.
      if(m.isAnnotationPresent(MyMarker.class))
        System.out.println("MyMarker is present.");

    } catch (NoSuchMethodException exc) {
      System.out.println("Method Not Found.");
    }
  }



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

- The **output**, shown here, confirms that @MyMarker is present:

MyMarker is present.

- In the program, notice that you do not need to follow @MyMarker with parentheses when it is applied.
- Thus, @MyMarker is applied simply by using its name, like this: @MyMarker
- It is not wrong to supply an empty set of parentheses, but they are not needed.

Single-Member Annotations:

- A single-member annotation contains only one member.
- It works like a normal annotation except that it allows a shorthand form of specifying the value of the member.
- When only one member is present, you can simply specify the value for that member when the annotation is applied—you don't need to specify the name of the member.
- However, in order to use this shorthand, the name of the member must be value.

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// A single-member annotation.
@Retention(RetentionPolicy.RUNTIME)
@interface MySingle {
  int value(); // this variable name must be value
}

class Single {

  // Annotate a method using a single-member annotation.
  @MySingle(100)
  public static void myMeth() {
    Single ob = new Single();

    try {
      Method m = ob.getClass().getMethod("myMeth");

      MySingle anno = m.getAnnotation(MySingle.class);

      System.out.println(anno.value()); // displays 100

    } catch (NoSuchMethodException exc) {
       System.out.println("Method Not Found.");
    }
  }

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

Output:

- As expected, this program displays the value 100. In the program, @MySingle is used to annotate myMeth( ), as shown here:

@MySingle(100)

- Notice that value = need not be specified.
- You can use the single-value syntax when applying an annotation that has other members, but those other members must all have default values.

For example, here the value xyz is added, with a default value of zero:

@interface SomeAnno

```
{
int value();
int xyz() default 0;
}
```

Single-Member Annotations:

- In cases in which you want to use the default for xyz, you can apply @SomeAnno, as shown next, by simply specifying the value of value by using the single-member syntax.
- @SomeAnno(88)
- In this case, xyz defaults to zero, and value gets the value 88. Of course, to specify a different value for xyz requires that both members be explicitly named, as shown here:
- @SomeAnno(value = 88, xyz = 99)
- Remember, whenever you are using a single-member annotation, the name of that member must be value.

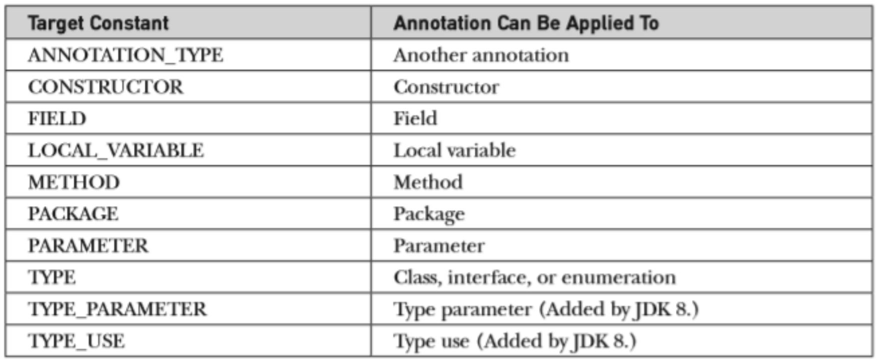
The Built-In Annotations:

- Java defines many built-in annotations. Most are specialized, but nine are general purpose.
- Of these, four are imported from java.lang.annotation:
- @Retention
- @Documented,
- @Target, and @Inherited.
- Five—@Override, @Deprecated, @FunctionalInterface, @SafeVarargs, and @SuppressWarnings

—are included in java.lang

- @Retention- It specifies the retention policy
- @Documented- is a marker interface that tells a tool that an annotation is to be documented
- The @Target annotation specifies the types of items to which an annotation can be applied

@Target takes one argument, which is an array of constants of the ElementType enumeration. This argument specifies the types of declarations to which the annotation can be applied.



Target Constant	Annotation Can Be Applied To
ANNOTATION_TYPE	Another annotation
CONSTRUCTOR	Constructor
FIELD	Field
LOCAL_VARIABLE	Local variable
METHOD	Method
PACKAGE	Package
PARAMETER	Parameter
TYPE	Class, interface, or enumeration
TYPE_PARAMETER	Type parameter (Added by JDK 8.)
TYPE_USE	Type use (Added by JDK 8.)

- You can specify one or more of these values in a @Target annotation. To specify multiple values, you must specify them within a braces-delimited list. For example, to specify that an annotation applies only to fields and local variables, you can use this @Target annotation:

@Target({ ElementType.FIELD,ElementType.LOCAL_VARIABLE })

- @Inherited - causes the annotation for a superclass to be inherited by a subclass
- @Override - annotation that can be used only on methods. A method annotated with @Override must override a method from a superclass. If it doesn't, a compile-time error will result.
- @Deprecated - It indicates that a declaration is obsolete and has been replaced by a newer form.
- @FunctionalInterface - marker annotation added by JDK 8 and designed for use on interfaces. It indicates that the annotated interface is a functional interface. A functional interface is an interface that contains one and only one abstract method. Functional interfaces are used by lambda expressions.
- @SafeVarargs -is a marker annotation that can be applied to methods and constructors. It indicates that no unsafe actions related to a varargs parameter occur
- @SuppressWarnings @SuppressWarnings specifies that one or more warnings that might be issued by the compiler are to be suppressed. The warnings to suppress are specified by name, in string form.

Acknowledgment:

References:

Herbert Schildt: JAVA the Complete Reference, 7th/9th Edition, Tata McGraw Hill, 2007.

Web References:

https://docs.oracle.com/

https://www.geeksforgeeks.org/

-For both content creation and delivery.