Java Programming

Wrapper Classes, I/O Streams, Annotation

Module 10

Agenda

1 Wrapper Classes

2 IO Streams

3 Reading & printing to console

4 Writing & reading from file

5 Object Serialization

Agenda (Contd.).

6 What is annotation

7 Annotation used by the compiler

8 Categories of an Annotation

Objectives

- At the end of this session, you will be able to:
  - Describe the need for wrapper classes
  - Define wrapper classes
  - Understand Autoboxing & Unboxing
  - Understand cloning
  - Understand stream
  - Define Byte streams and Character streams
  - Define the predefined stream objects defined in the System class, namely in, out, and err

Objectives (Contd.).

- Highlight the preference for **Character** over **Byte** streams
- Reading & writing operations for console & file
- Implement object serialization with the help of the **ObjectInputStream** and the **ObjectOutputStream**
- Define Annotation
- Understand the Annotation used by compiler
- Differentiate between categories of annotation
- Explore the advantages using annotation

Wrapper Classes

java.util

- Some of the most important utility classes are provided in java.util package

- They are: Vector, Dictionary, Hashtable, StringTokenizer and Enumeration

- The java.util package hosts frequently used
  - data structures like Stack, LinkedList, HashMap etc
  - functionalities like sort, binary search etc

Wrapper Classes

- Heterogeneous values can be stored in structures like **vectors**, or **hashtables**

- However, only objects can be stored in vectors and hashtables

- **Primitive data types cannot be stored directly in vectors and hashtables,** and hence have to be converted to objects

- We have to wrap the primitive data types in a corresponding object, and give them an object representation

Wrapper Classes (Contd.).

- Definition: The process of converting the primitive data types into objects is called wrapping

- To declare an integer ‘i’ holding the value 10, you write   int i = 10;

- The object representation of integer ‘i’ holding the value 10 will be:

  Integer iref = new Integer(i);

- Here, class Integer is the wrapper class wrapping a primitive data type i

The Integer Class

- Class **Integer** is a wrapper for values of type **int**
- **Integer** objects can be constructed with a **int** value, or a string containing a int value
- The constructors for Integer are shown here:

  Integer(int num)

  Integer(String str) throws NumberFormatException
- Some methods of the **Integer** class:

  static int parseInt(String str) throws NumberFormatException

  int intValue( ) returns the value of the invoking object as a **int** value

The Character Class

- Character class is a wrapper class for character data types. The constructor for Character is:
  - **Character(char c)**
  - Here, c specifies the character to be wrapped by the Character object

- After a Character object is created, you can retrieve the primitive character value from it using:
  - **char charValue( )**

The Boolean Class

- The Boolean class is a wrapper around boolean values

- It has the following constructors:
  - **Boolean (boolean bValue)**
    - Here, bValue can be either true or false
  - **Boolean (String str)**
    - The object created by this constructor will have the value true or false depending upon the string value in str – “true” or “false”
    - The value of str can be in upper case or lower case

The Float Class

- Class **Float** is a wrapper for floating-point values of type **float**

- **Float** objects can be constructed with a **float** value, or a string containing a floating-point value

- The constructors for float are shown here:

  Float(float num)

  Float(String str) throws NumberFormatException

- Some methods of the **Float** class:

  static Float valueOf(String str) throws NumberFormatException

  float floatValue( ) returns the value of the invoking object as a **float** value

The Double Class

- Class **Double** is a wrapper for floating-point values of type **double**

- **Double** objects can be constructed with a **double** value, or a string containing a floating-point value

- The constructors for double are shown here:

  Double(double num)

  Double(String str) throws NumberFormatException

- Some methods of the **Double** class:

  static Double valueOf(String str) throws NumberFormatException

  double doubleValue( ) returns the value of the invoking object as a **double** value

The Long Class

- Class **Long** is a wrapper for values of type **long**
- **Long** objects can be constructed with a **long** value, or a string containing a long value
- The constructors for long are shown here:

  Long(long num)

  Long(String str) throws NumberFormatException
- Some methods of the **Long** class:

  static Long valueOf(String str) throws NumberFormatException

  long longValue( ) returns the value of the invoking object as a **long** value

The Short Class

- Class **Short** is a wrapper for values of type **short**

- **Short** objects can be constructed with a **short** value, or a string containing a long value

- The constructors for short are shown here:

  Short(short num)

  Short(String str) throws NumberFormatException

- Some methods of the **Short** class:

  static Short valueOf(String str) throws NumberFormatException

  short shortValue( ) returns the value of the invoking object as a **short** value

The Byte Class

- Class **Byte** is a wrapper for values of type **byte**

- **Byte** objects can be constructed with a **byte** value, or a string containing a long value

- The constructors for byte are shown here:

  Byte(byte num)

  Byte(String str) throws NumberFormatException

- Some methods of the **Byte** class:

  static Byte valueOf(String str) throws NumberFormatException

  byte byteValue( ) returns the value of the invoking object as a **byte** value

AutoBoxing & UnBoxing

- Java 5.0 introduced automatic conversion between a primitive type and the corresponding wrapper class

- During assignment , the automatic transformation of primitive type to corresponding wrapper type is known as **autoboxing**

- Primitive types ---------------------------→ wrapper type

  (autoboxing)

- E. g.

```
Integer i1=10;
```

- During assignment , the automatic transformation of wrapper type into their primitive equivalent is known as **Unboxing**

- wrapper type ---------------------------→ primitive type

  (unboxing)

- E. g.

```
 int i=0;
i=new Integer(10);
```

AutoBoxing & UnBoxing (Contd.).

- Autoboxing also works with comparison

```
int a = 10;

Integer b = 10;

System.out.println(a==b);
```

AutoBoxing & UnBoxing (Contd.).

- Boxing/Unboxing of Character value :

```
public class MyClass {
  public static void main(String ab[]) {
  Boolean boolean b1 = true;

  if (b1){
  System.out.println("b is true");
  }

  Character chr = ‘a’;     // box a char
  char chr1 = chr;            // unbox a char

  System.out.println("chr1 is " + chr1);
  }
  }
```

AutoBoxing & UnBoxing (Contd.).

- Boxing conversion converts values of primitive type to corresponding values of reference type. But the primitive types can not be widened/ Narrowed to the Wrapper classes and vice versa. For example,

```
byte  b     = 12;

Integer I1 = 90;        //Constant integer value
Integer I2 = (int)b;    //Cast to int type

Long    L1 = 90;         //compile error because 90
                               is integer value
Long    L2 = (Long)90;  //can not cast integer
                        value to Long wrapper class

Long    L3 = 90L;
Long    L4 = (long)90;
```

AutoBoxing & UnBoxing (Contd.).

- Autoboxing and unboxing also apply to methods calls. For example, you can pass an argument of type int to a method that has a formal parameter of type Integer

E.g.

```
   class Sample
   {
   void m1(Integer i1)
   { System.out.println(“int value=“+i1); }
}
class E
{
public static void main(String a[])
{
  Sample s1=new Sample();
  s1.m1(10);
}
}
```

AutoBoxing & UnBoxing (Contd.).

- When invoking a method from multiple overloading methods, For the matching method process, the Java compiler will prefer the order of primitive types (Widening Primitive Conversion), wrapper class (Boxing Conversion), and var-args
- For example,

```
class Sample
 {
    public void m1(Long x, Long y) {
        System.out.println("m1(Long x, Long y)");
    }
    public void m1(long x, long y) {
        System.out.println("m1(long x, long y)");
    }

    public static void main(String[] args) {
        long x, y;
        x = y = 0;
        Sample s = new Sample();
        s.m1(x, y);
  Long l1=10L;
  Long l2=20L;
  s.m1(l1,l2); }
 }
```

Quiz

1. Name some of the utility classes defined within java.util package and specify their uses
2. List the corresponding wrapper classes for each of the primitive data types in Java
3. What do you mean by wrapping?
4. How do you convert an Integer type of object to its primitive type?

Quiz (Contd.).

5. Name the two constructors defined within the Boolean class and what type of values can you pass to each of these constructors
6. While creating an object of type Double what values you can pass to the constructors of this class
7. What is autoboxing & unboxing?

The Cloneable Interface

- When you make a copy of an object reference:
  - The original and copy are references to the same object
  - This means a change to either variable also affect the other
- The clone( ) method:
  - is a protected member of Object,
  - can only be invoked on an object that implements Cloneable
- Object cloning performs a bit-by-bit copy

Example on cloning

```
class XYZ implements Cloneable {
      int a;
      double b;
      XYZ cloneTest() {
         try {
         return (XYZ) super.clone();
         }
         catch(CloneNotSupportedException e) {
         System.out.println("Cloning Not
 Allowed");
         return this;
         }
      }
}
```

Contd..

Example on cloning (Contd.).

```
class CloneDemo1 {
    public static void main(String args[])  {
  XYZ x1 = new XYZ();
  XYZ x2;
  x1.a = 10;
  x1.b = 20;
  x2 = x1.cloneTest(); // cloning x1
  System.out.println("x1 : " + x1.a + " " + x1.b);
  System.out.println("x2 : " + x2.a + " " + x2.b);
  x1.a = 100;
  x1.b = 200;
  System.out.println("x1 : " + x1.a + " " + x1.b);
  System.out.println("x2 : " + x2.a + " " + x2.b);
    }
}
```

Output :

```
x1 : 10 20.0
x2 : 10 20.0
x1 : 100 200.0
x2 : 10 20.0
```

I/O Streams

I/O Streams

- Java programs perform I/O through streams. A stream is:
  - an abstraction that either produces or consumes information
  - linked to a physical device by the Java I/O system

- All streams behave similarly, even if the actual physical devices to which they are linked differ.

- Thus the same I/O classes can be applied to any kind of device as they abstract the difference between different I/O devices.

I/O Streams (Contd.).

- Java’s stream classes are defined in the **java.io** package.

- Java 2 defines two types of streams:
  - **byte streams**
  - **character streams**

- Byte streams:
  - provide a convenient means for handling input and output of bytes
  - are used for reading or writing binary data

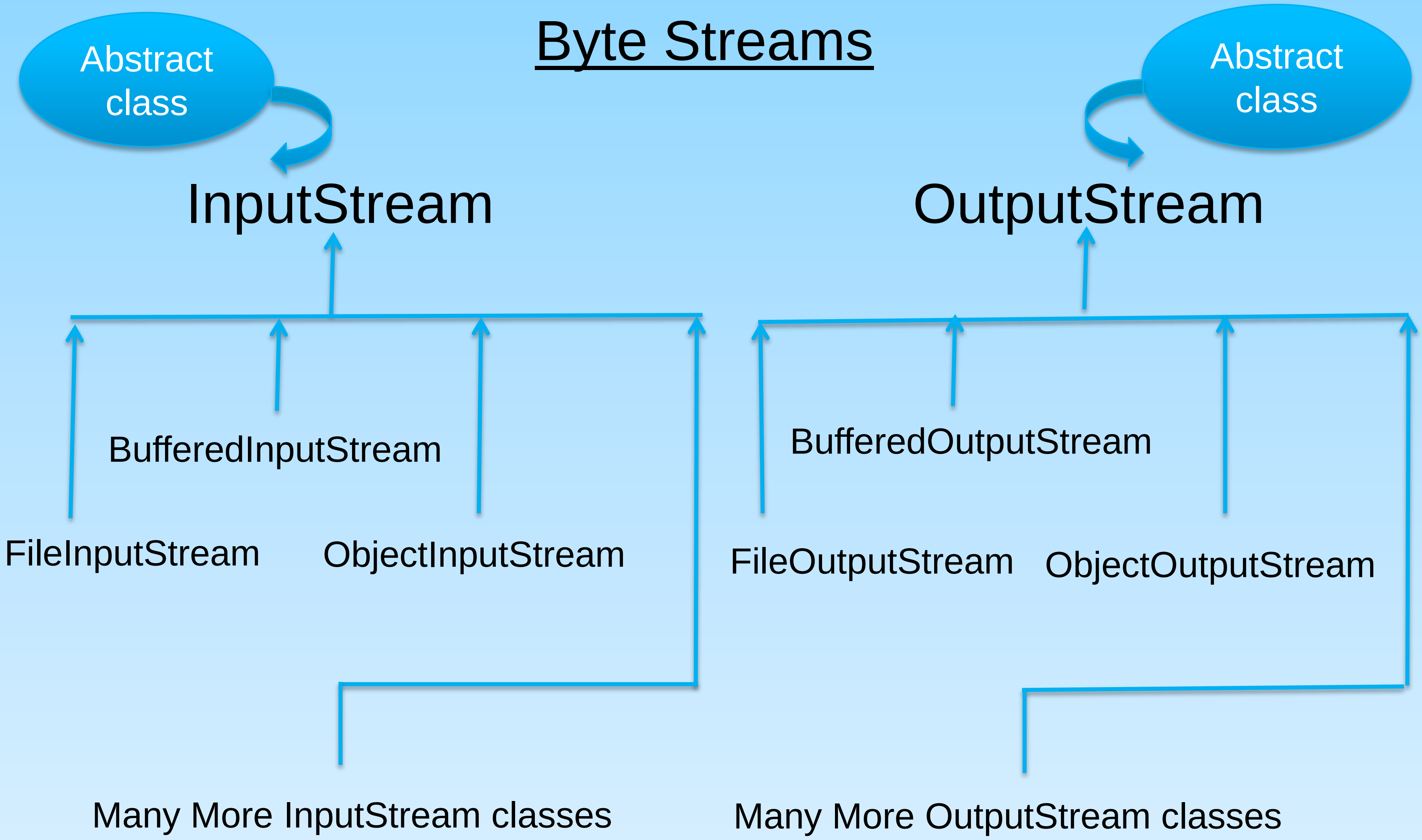
- Character streams:
  - provide a convenient means for handling input and output of characters
  - use **Unicode**, and, therefore, can be internationalized

The Predefined Streams

- System class of the java.lang package contains three predefined stream variables, in, out, and err.
- These variables are declared as public and static within System:
  - System.out refers to the standard output stream which is the console.
  - System.in refers to standard input, which is the keyboard by default.
  - System.err refers to the standard error stream, which also is the console by default.

I/O Streams hierarchy

Byte Streams



Abstract class

InputStream

BufferedInputStream

FileInputStream

ObjectInputStream

Many More InputStream classes

Abstract class

OutputStream

BufferedOutputStream

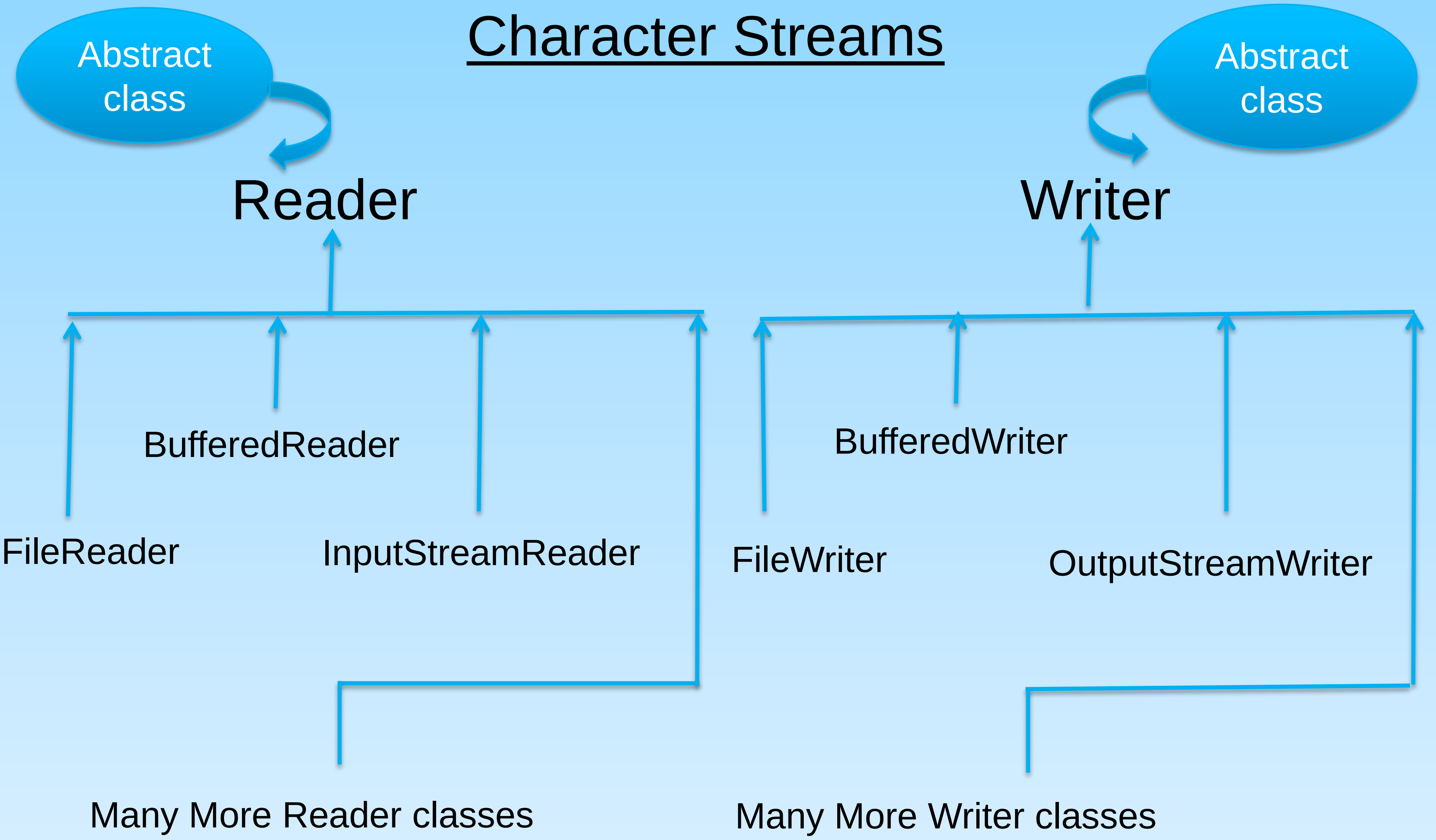
FileOutputStream

ObjectOutputStream

Many More OutputStream classes

I/O Streams hierarchy (Contd.).

Character Streams



Abstract class

Reader

BufferedReader

FileReader

InputStreamReader

Many More Reader classes

Abstract class

Writer

BufferedWriter

FileWriter

OutputStreamWriter

Many More Writer classes

Byte Stream classes

BufferedInputStream
BufferedOutputStream

To read & write data into buffer

FileInputStream
FIleOutputStream

To read & write data into file

ObjectInputStream
ObjectOutputStream

To read & write object into
secondary device (serialization)

Character Stream classes

BufferedReader

BufferedWriter

To read & write data into buffer

FileReader

FIleWriter

To read & write data into file

InputStreamReader

OutputStreamWriter

Bridge from character stream to byte stream

Reading & Printing to Console

Reading Console Input - Stream Wrapping

- The preferred method of reading console input in Java 2 is to use a character stream

- InputStreamReader class acts as a bridge between byte and character streams

- Console input is accomplished by reading from System.in

- To get a character-based stream, you wrap System.in in a BufferedReader object

Reading Characters

```
package m10.io;
import java.io.*;

public class BRRead{

   public static void main (String args[ ]) throws
   IOException {
            char c;
            BufferedReader br = new BufferedReader(new
                              InputStreamReader(System.in));
            System.out.println(“Enter Characters, ‘q’ to
   quit”);
            do {
                          c = (char) br.read( );
                                  System.out.println( c );
                  }while (c != ‘q’);
        }
}
```

Reading Strings

```
package m10.io;
import java.io.*;

public class BRReadLine{

     public static void main (String args[]) throws
   IOException {
        String str;
        BufferedReader br = new BufferedReader(new
                 InputStreamReader(System.in));
        System.out.println(“Enter Characters, ‘stop’
   to quit”);
             do {
                 str = br.readLine( );
                     System.out.println ( str );
             }while (!str.equals( “stop”));
   }
}
```

Writing Console Output

- **print( )** and **println( )** are console output methods defined in PrintStream class

- **System.out** is a byte stream used to write bytes

- The **write()** method in PrintStream can be used to write to the console

Writing Console Output (Contd.).

```
class WriteDemo {
  public static void main (String args[ ])
   {
      int b;
      b = ‘A’;
      System.out.write( b);
      System.out.write(‘\n’);
   }
}
```

Writing & Reading From File

Reading & Writing to File using FileReader & FileWriter

```
package m10.io;
import java.io.*;

public class Copy {

public static void main(String[] args) throws
  IOException {

   File inputFile = new File(”Source.txt");
   File outputFile = new File(”Target.txt");
   FileReader in = new FileReader(inputFile);
   FileWriter out = new FileWriter(outputFile);
   int c;
   while ((c = in.read()) != -1)
        out.write(c);

   in.close();
   out.close();
  }
}
```

Copy image

```
import java.io.*;

class CopyFile{

    public static void main(String args[])
    throws IOException{
    int i;
    FileInputStream fin;
    FileOutputStream fout;

    try{
      fin = new FileInputStream(args[0]);
    }
    catch(FileNotFoundException e){
      System.out.println("File Not Found");
      return;
    }
```

Copy image (Contd.).

```
try{
          fout = new FileOutputStream(args[1]);
   }
   catch(IOException e){
         System.out.println("Error Opening Output File");
         return;
   }
try{
          do {
                 i=fin.read();
                 if(i!=-1)
                 fout.write(i);
          } while (i!=-1);
   }
   catch (IOException e){
          System.out.println("File Error");
   }
   fin.close();
   fout.close();
  }
}
```

Object Serialization

Serialization

- Object serialization is the process of saving an object's state to a sequence of bytes (on disk), as well as the process of rebuilding those bytes into a live object at some future time

- The Java Serialization API provides a standard mechanism to handle object serialization

- You can only serialize the objects of a class that implements Serializable interface

Serializing Objects

- How to Write to an ObjectOutputStream

```
FileOutputStream out = new FileOutputStream("theTime");
ObjectOutputStream s = new ObjectOutputStream(out);
s.writeObject("Today");
s.writeObject(new Date());
s.flush();
```

- How to Read from an ObjectOutputStream

```
FileInputStream in = new FileInputStream("theTime");
ObjectInputStream s = new ObjectInputStream(in);
String today = (String)s.readObject();
Date date = (Date)s.readObject();
```

Object Serialization

```
package m10.io;
import java.io.*;

public class MyClass implements Serializable {
  String s;
  int i;
  double d;
  public MyClass(String s, int i, double d) {
    this.s = s;
    this.i = i;
    this.d = d;
  }
  public String toString() {
    return “s=“ + s + “; i=“ + i + “; d=“ + d;
  }
}
```

Object Serialization (Contd.).

```
public class SerializationDemo {

public static void main(String args[]) {
   try {
         MyClass object1 = new MyClass(“Hello”, -7, 2.7e10);
                System.out.println(“object1; “ + object1);
                FileOutputStream fos = new
  FileOutputStream(“serial”);
         ObjectOutputStream oos = new ObjectOutputStream(fos);
         oos.writeObject(object1);
         oos.flush();
         oos.close();
   }
   catch(Exception e) {
       System.out.println(“Exception during serialization:“+ e);
         System.exit(0);
   }
```

Object Serialization (Contd.).

```
// Object Deserialization
   try {
         MyClass object2;
         FileInputStream fis = new
 FileInputStream(“serial”);
         ObjectInputStream ois = new
 ObjectInputSream(fis);
         object2 = (MyClass)ois.readObject();
         ois.close();
         System.out.println(“object2: “ + object2);
   }
   catch(Exception e) {
         System.out.println(“Exception during
 deserialization: “ + e);
         System.exit(0);
   }
 }
}
```

Match the following

- Match the streams with the appropriate phrases in column B

Column A	Column B
1. FileWriter	Byte stream for reading from file
2. FileInputStream	Character stream for reading from file
3. FileOutputStream	Character stream for writing to a file
4. FileReader	Byte stream for writing to a file

Quiz

- What are streams ?
- Which are the different types of streams?
- What is significance of System.in and System.out
- How console input and output is achieved in Java
- How to copy text and image files in Java
- What is object serialization
- What are the streams that are used for achieving object serialization
- What type of object is returned by the method readObject()
- What is the significance of the interface serializable?

What is an Annotation?

What is an Annotation?

- Annotation is a new feature added in J2SE 5.0 (Tiger)
- Annotations are used to add meta-data to the Java Elements
- Annotations leads to a declarative programming style where the programmer says what should be done and the tools emit the code for it
- It is a mechanism for associating a meta-tag with program elements and allowing the compiler or the VM to extract program behaviors
- It is a special form of metadata that can be added to any program element
  - Classes, methods, variables, parameters and Packages can be annotated

Simple Example

- The annotation definition

```
@interface author
{
String value() default "Patrick Norton";
}
```

- Defines an "author" annotation
- Has one string attribute (value)

- The usage

```
@author(value="Sriram")
public void calculateEMI()
{
}
```

- Adds author annotation as modifier to the method calculateEMI

Annotations used by the Compiler

Annotations used by the Compiler

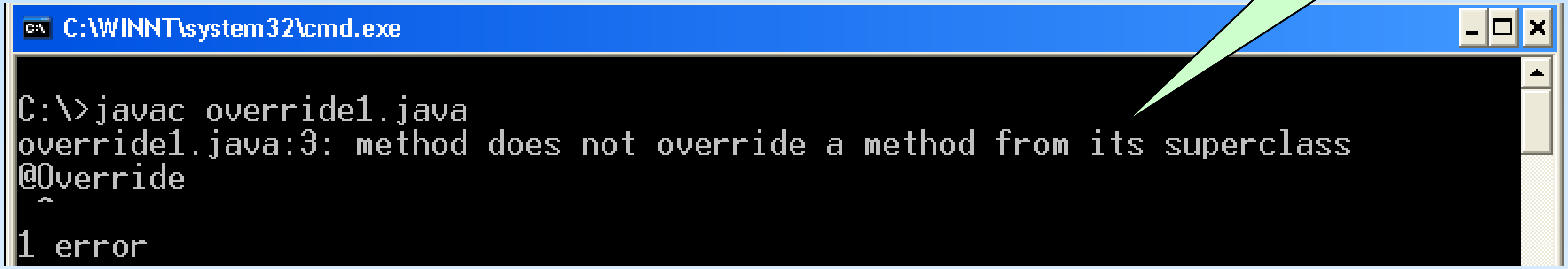
- There are three annotation types that are predefined by the language specification:
  - @Override
  - @Deprecated
  - @SuppressWarnings
  - These are examples of **Simple annotations**
  - Simple annotations are annotations that can be used only in code
  - They cannot be used to create custom annotation types

@Override

- Used to check if the function is an override
- It produces a compilation error if the method does not exist in the parent class

```
class override1
{
@Override
public String tostring()
{
return "Example of Override annotation";
}
}
```

Output



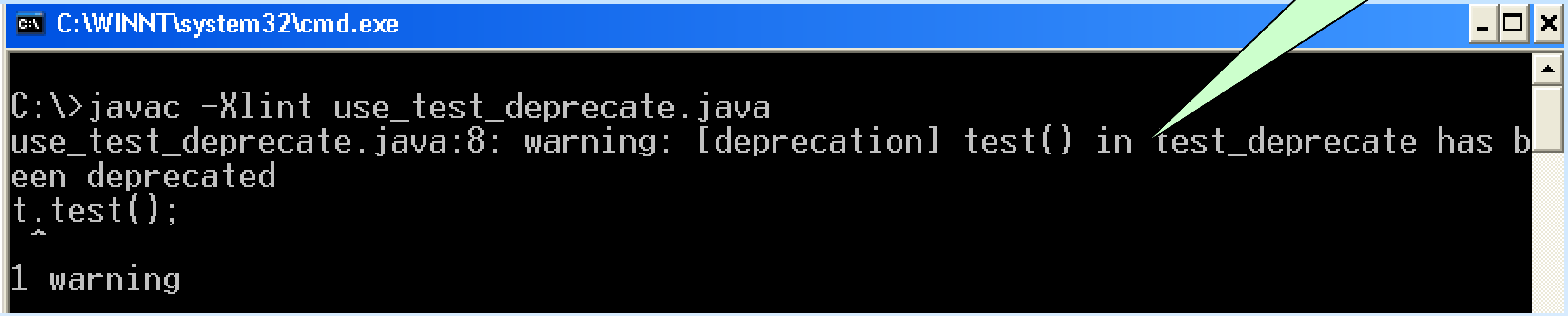
```
C:\>javac override1.java
override1.java:3: method does not override a method from its superclass
@Override
 ^
1 error
```

@Deprecated

- Used to mark a method obsolete
- It produces a warning if the function is used

```
class test_deprecate
{
@Deprecated
void test()
{
System.out.println("Testing
 deprecation");
}
```

```
class use_test_deprecate
{
public static void main(String ae[])
{
test_deprecate t=new test_deprecate();
t.test();
}
}
```



Output

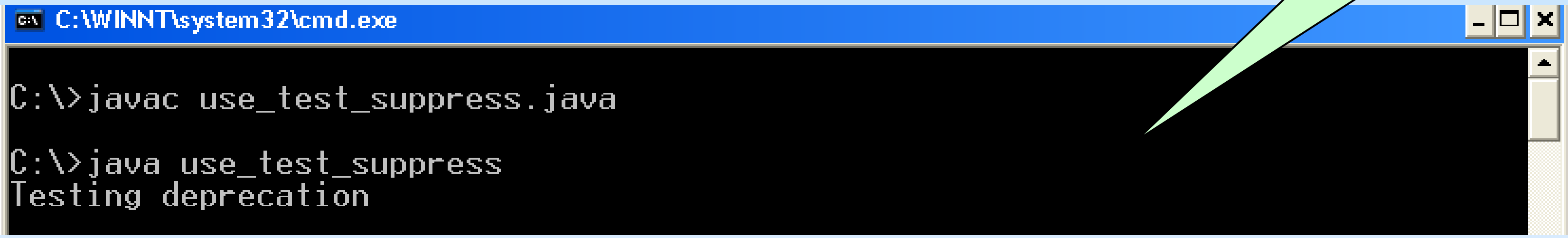
```
C:\>javac -Xlint use_test_deprecate.java
use_test_deprecate.java:8: warning: [deprecation] test() in test_deprecate has b
een deprecated
t.test();
 ^
1 warning
```

@SuppressWarnings

- Used to instruct the compiler to suppress the warnings specified in the annotation parameters

```
class test_deprecate
{
@Deprecated
void test()
{
System.out.println
("Testing deprecation");
}
```

```
class use_test_suppress
{
@SuppressWarnings({"deprecation"})
public static void main(String ae[])
{
test_deprecate t=new test_deprecate();
t.test();
} }
```



```
C:\>javac use_test_suppress.java

C:\>java use_test_suppress
Testing deprecation
```

Output

Review Questions

1. Which of the following keyword is used to create an user defined annotation?

a. class

b. enum

c. interface

d. None of the above

2. Which of the following annotation will be used only in case of Inheritance?

a. @Deprecated

b. @SuppressWarnings

c. @Override

Categories of an Annotation

Categories of Annotation – Marker Annotation

- Marker Annotation
  - Contains only the name
  - Does not contain any other element
  - Creation

```
public @interface marker
{
}
```

  - Usage

```
@marker
public void sampleMethod
{
}
```

Categories of Annotation – Single value Annotation

- Single value Annotation
  - They provide a single piece of data
  - Can be provided as a data value pair or can use shortcut and provide the value within quotation marks
  - Creation

```
@interface author
{
String value() default "Patrick Norton";
}
```

  - Usage

```
@author("Sriram")
public void calculateEMI()
{
}
```

(or)

```
@author(value="Sriram")
public void calculateEMI()
{
}
```

Categories of Annotation – Multi value Annotation

- Multi value/ Full value Annotation
  - They can have multiple data members
  - We have to pass value to all the data members
- Creation

```
@interface calldetails
{
String severity() default "Medium";
String personAssigned();
int no_of_escalations();
String date();
}
```

Categories of Annotation – Multi value Annotation (Contd.).

- Usage

```
class itimhelpdesk
{
@calldetails(severity="High",personAssigned="GovindSamy",no_of_escalations=0,date="1-2-2012")
public void logCall()
{
}
}
```

Review Questions

1. Which of the following annotation is an example of Marker Annotation?

a. @Deprecated

b. @SuppressWarnings

c. @Override

2. Which of the following is an example of Multi value Annotation?

a. @Deprecated

b. @SuppressWarnings

c. @Override

Meta Annotations

- They are used to annotate the annotation type declaration
- There are 4 types of Meta annotations
  - Target
  - Retention
  - Documented
  - Inherited

Meta Annotations - Target

- **Target**
  - It is used to specify which element of the class to be annotated
  - **@Target(ElementType.TYPE)** - can be applied to any element of a class
  - **@Target(ElementType.FIELD)** - can be applied to field or property
  - **@Target(ElementType.PARAMETER)** - can be applied to the parameters of a method
  - **@Target(ElementType.LOCAL_VARIABLE)** - can be applied to local variables
  - **@Target(ElementType.METHOD)** - can be applied to method level annotation
  - **@Target(ElementType.CONSTRUCTOR)** - can be applied to constructors
  - **@Target(ElementType.ANNOTATION_TYPE)** - used to specifiy that the declared type itself is an annotation type

Meta Annotations – Target (Contd.).

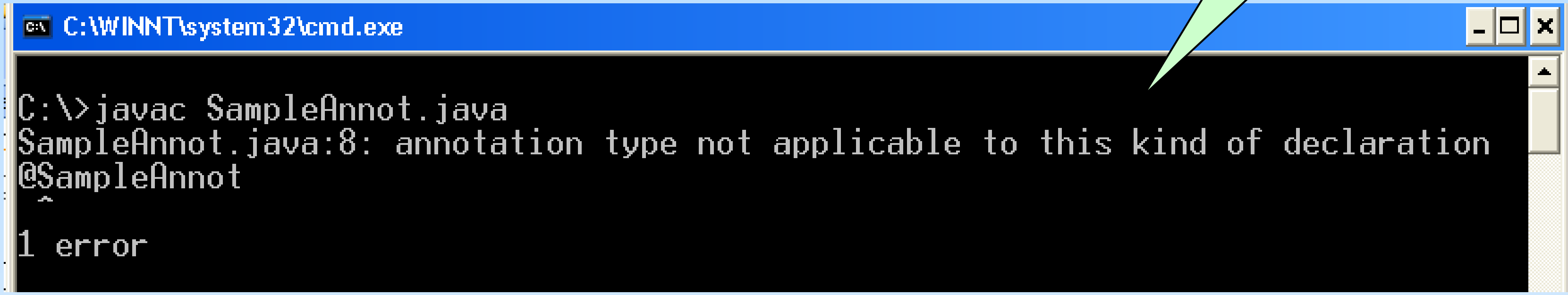
- Example

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;
@Target(ElementType.METHOD)
public @interface SampleAnnot{


}
```

```
@SampleAnnot
class Example
{
}
```

Output



```
C:\>javac SampleAnnot.java
SampleAnnot.java:8: annotation type not applicable to this kind of declaration
@SampleAnnot
 ^
1 error
```

Meta Annotations - Retention

- Used to indicate how long annotations of this type are to be retained
- There are 3 values
- **RetentionPolicy.SOURCE**
  - These annotations will be retained only at the source level and will be ignored by the compiler
- **RetentionPolicy.CLASS**
  - These annotations will be by retained by the compiler at compile time, but will be ignored by the VM
- **RetentionPolicy.RUNTIME**
  - These annotations will be retained by the VM so they can be read at run-time

Meta Annotations – Retention (Contd.).

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)
@interface author
{
String value() default "Patrick Norton";
}
```

Meta Annotations - Documented

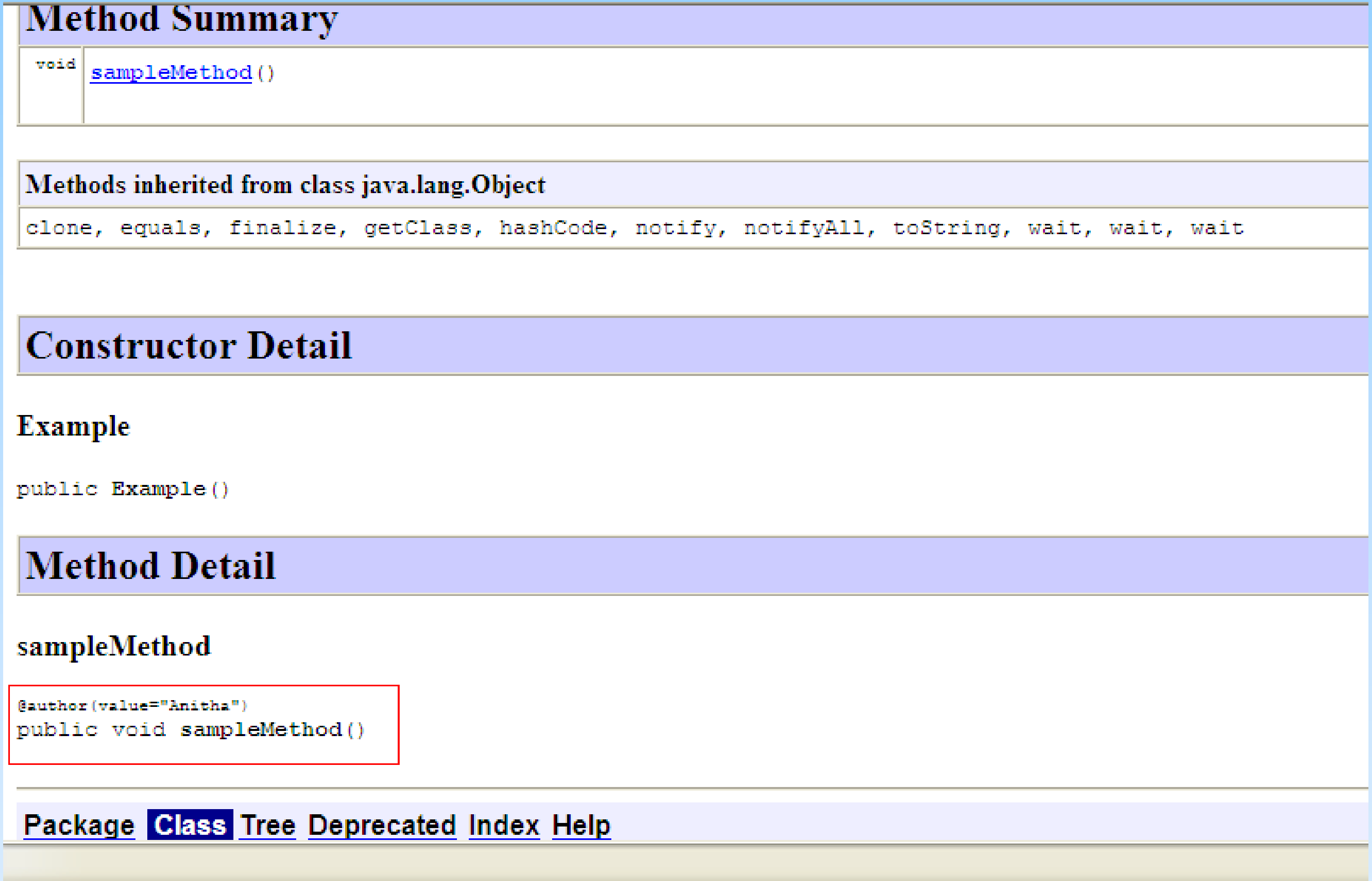
- Used to inform that an annotation with this type should be documented by the javadoc tool

```
import java.lang.annotation.Documented;
import java.lang.annotation.Target;
import java.lang.annotation.ElementType;

@Documented
@Target({ElementType.METHOD })
@interface author
 {
String value();
}
```

```
public class Example
{
@author("Anitha")
public void sampleMethod()
{
}
}
```

Meta Annotations – Documented (Contd.).



Method Summary

void	sampleMethod()

Methods inherited from class java.lang.Object

```
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait
```

Constructor Detail

Example

```
public Example()
```

Method Detail

sampleMethod

```
@author(value="Anitha")
public void sampleMethod()
```

Package Class Tree Deprecated Index Help

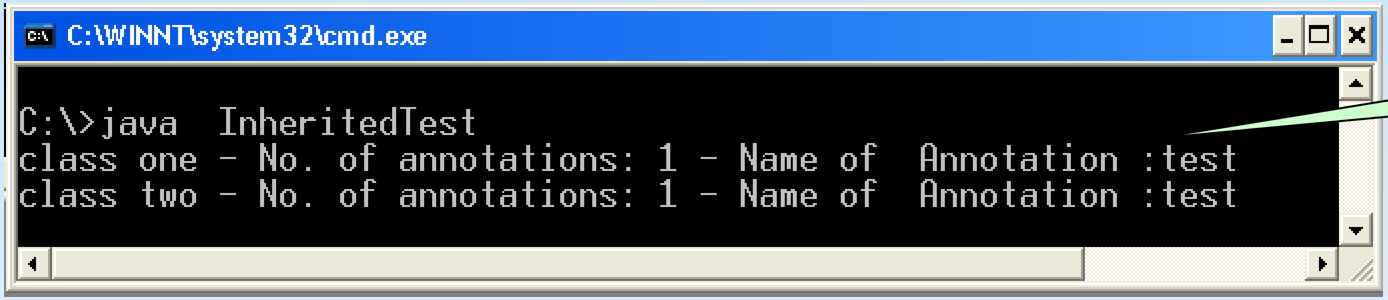
Meta Annotations - Inherited

- A child class inherits the annotation which is marked with @Inherited annotation

```
Inherit.java
---------------
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@interface TestAnnotation{
 String value();
 }
@TestAnnotation("test")
class one { }
class two extends one
{ }
```

Meta Annotations – Inherited (Contd.).

```
import java.lang.annotation.Annotation;
public class InheritedTest {
public static void main(String[] args) {
Class[] classes = {one.class, two.class};
for (Class classObj : classes) {
System.out.print(classObj+" - ");
Annotation[] annotations = classObj.getAnnotations();
System.out.print("No. of annotations: " +
  annotations.length);
for (Annotation annotation : annotations) {
TestAnnotation t = (TestAnnotation)annotation;
System.out.println(" - Name of  Annotation :"+t.value());
  }}}}
```



```
C:\>java  InheritedTest
class one - No. of annotations: 1 - Name of  Annotation :test
class two - No. of annotations: 1 - Name of  Annotation :test
```

Output

Advantages with Annotation

- Annotations helps to shift the responsibility of writing boilerplate code from the programmer to the Compiler or other tools
- The resulting code is less error prone
- Provides information to the compiler
  - It can be used by the compiler to detect errors or suppress warnings
- Compiler-time and deployment-time processing
  - Software tools can process annotation information to generate code, XML files, and so forth
- Runtime processing
  - Some annotations are available to be examined at runtime

Summary

- In this module, you were able to:
  - Describe the need for wrapper classes
  - Define wrapper classes
  - Understand Autoboxing & Unboxing
  - Understand cloning
  - Understand stream
  - Define Byte streams and Character streams
  - Define the predefined stream objects defined in the System class, namely in, out, and err

Summary (Contd.).

- Highlight the preference for **Character** over **Byte** streams
- Reading & writing operations for console & file
- Implement object serialization with the help of the **ObjectInputStream** and the **ObjectOutputStream**
- Defination of Annotation
- Annotation used by compiler
- Categories of annotation
- Advantages using annotation

References

1. Oracle (2012). *Annotations.* Retrieved on Mar 25, 2012, from, http://docs.oracle.com/javase/tutorial/java/javaOO/annotations.html

2. Oracle (2012). *IO Streams.* Retrieved on Mar 25, 2012, from, http://java.sun.com/developer/technicalArticles/Streams/ProgIOStreams/

3. Schildt, H. Java: The Complete Reference. J2SETM. Ed 5. New Delhi: McGraw Hill-Osborne, 2005.

4. Oracle (2012). *Autoboxing.* Retrieved on Feb 25, 2012, from,
5. http://docs.oracle.com/javase/1.5.0/docs/guide/language/autoboxing.html

Thank you