JAVA REFLECTION

EFTHIMIOS ALEPIS

JAVA REFLECTION – WHAT?

- Reflection is a feature in the Java programming language. It allows an executing Java program to examine or "introspect" upon itself, and manipulate internal properties of the program. (Oracle, January 1998)
- Reflection is commonly used by programs which require the ability to examine or modify the runtime behavior of applications running in the Java virtual machine. This is a **relatively advanced feature** and should be used only by developers who have a strong grasp of the fundamentals of the language. With that caveat in mind, reflection is a powerful technique and can enable applications to perform operations which would otherwise be impossible.

JAVA REFLECTION – WHY?

- **Extensibility Features**
  - An application may make use of external, user-defined classes by creating instances of extensibility objects using their fully-qualified names.
- **Class Browsers and Visual Development Environments (IDEs)**
  - A class browser needs to be able to enumerate the members of classes. Visual development environments can benefit from making use of type information available in reflection to aid the developer in writing correct code.
- **Debuggers and Test Tools**
  - Debuggers need to be able to examine private members on classes. Test harnesses can make use of reflection to systematically call a discoverable set APIs defined on a class, to insure a high level of code coverage in a test suite.

REFLECTION DRAWBACKS

- **Performance Overhead**
  - Because reflection involves types that are dynamically resolved, certain Java virtual machine optimizations can not be performed. Consequently, reflective operations have slower performance than their non-reflective counterparts, and should be avoided in sections of code which are called frequently in performance-sensitive applications.
- **Security Restrictions**
  - Reflection requires a runtime permission which may not be present when running under a security manager. This is in an important consideration for code which has to run in a restricted security context, such as in an Applet.
- **Exposure of Internals**
  - Since reflection allows code to perform operations that would be illegal in non-reflective code, such as accessing private fields and methods, the use of reflection can result in unexpected side-effects, which may render code dysfunctional and may destroy portability. Reflective code breaks abstractions and therefore may change behavior with upgrades of the platform.

SOME NOTES ABOUT CLASSES IN JAVA

- Every type is either a reference or a primitive
- Classes, enums, and arrays as well as interfaces are all reference types
- For every type of object, the Java virtual machine instantiates an immutable instance of java.lang.Class which provides methods to examine the runtime properties of the object including its members and type information
- **Class** class also provides the ability to create new classes and objects
- **Class** is the entry point for all of the Reflection APIs

JAVA.LANG.CLASS CLASS

- The java.lang.Class class performs mainly two tasks:
  - provides methods to get the metadata of a class at run time.
  - provides methods to examine and change the run time behavior of a class.
- Commonly used methods of Class:
  - public String getName()
  - public static Class forName(String className)
  - public Object newInstance()
  - public boolean isInterface()
  - public boolean isArray()
  - public boolean isPrimitive()
  - public Class getSuperclass()
  - public Field[] getDeclaredFields()
  - public Method[] getDeclaredMethods()
  - public Constructor[] getDeclaredConstructors()
  - public Method getDeclaredMethod(String name,Class[] parameterTypes)

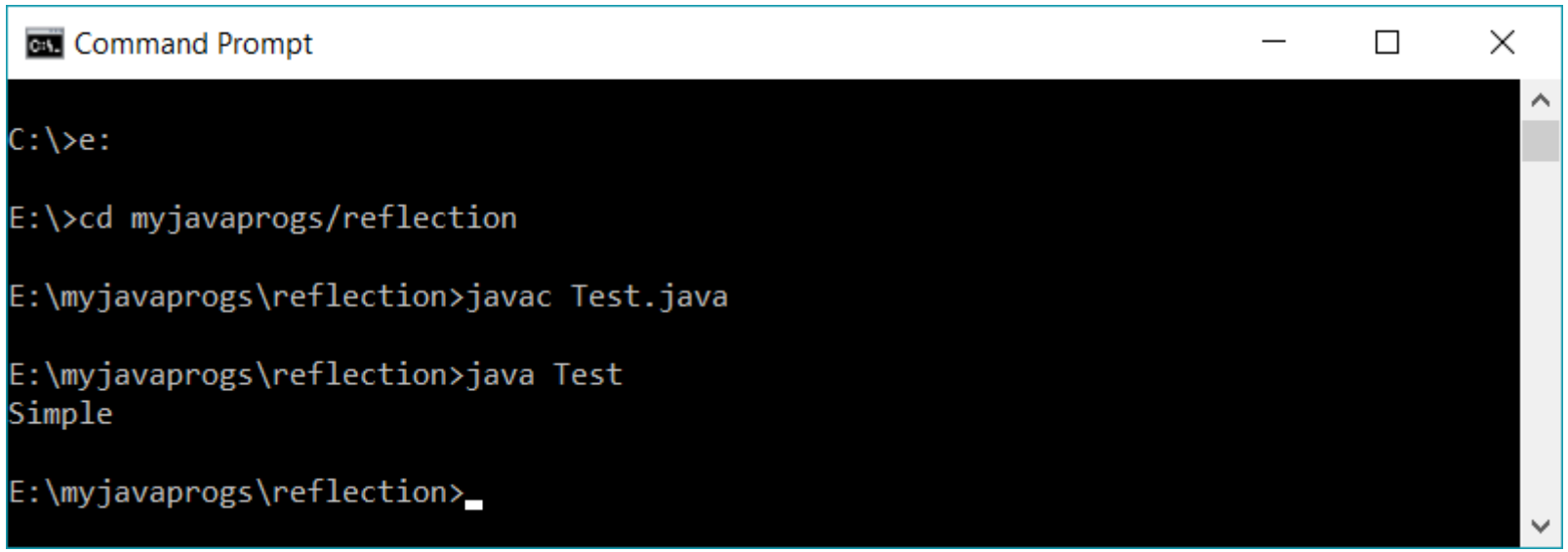
HOW TO RETRIEVE CLASS OBJECTS

- There are several ways to get a Class depending on whether the code has access to an object, the name of class, a type, or an existing Class

1. forName() method of Class class
    1. Is used to load the class dynamically.
    2. Returns the instance of Class class.
    3. It should be used if you know the fully qualified name of class.
    4. Cannot be used for primitive types.
2. getClass() method of Object class
    1. Returns the instance of Class class.
    2. Should be used if you know the type.
    3. It can be used with primitives.
3. .class syntax
    1. If a type is available but there is no instance then it is possible to obtain a Class by appending ".class" to the name of the type.
    2. Can be used for primitive data type also.

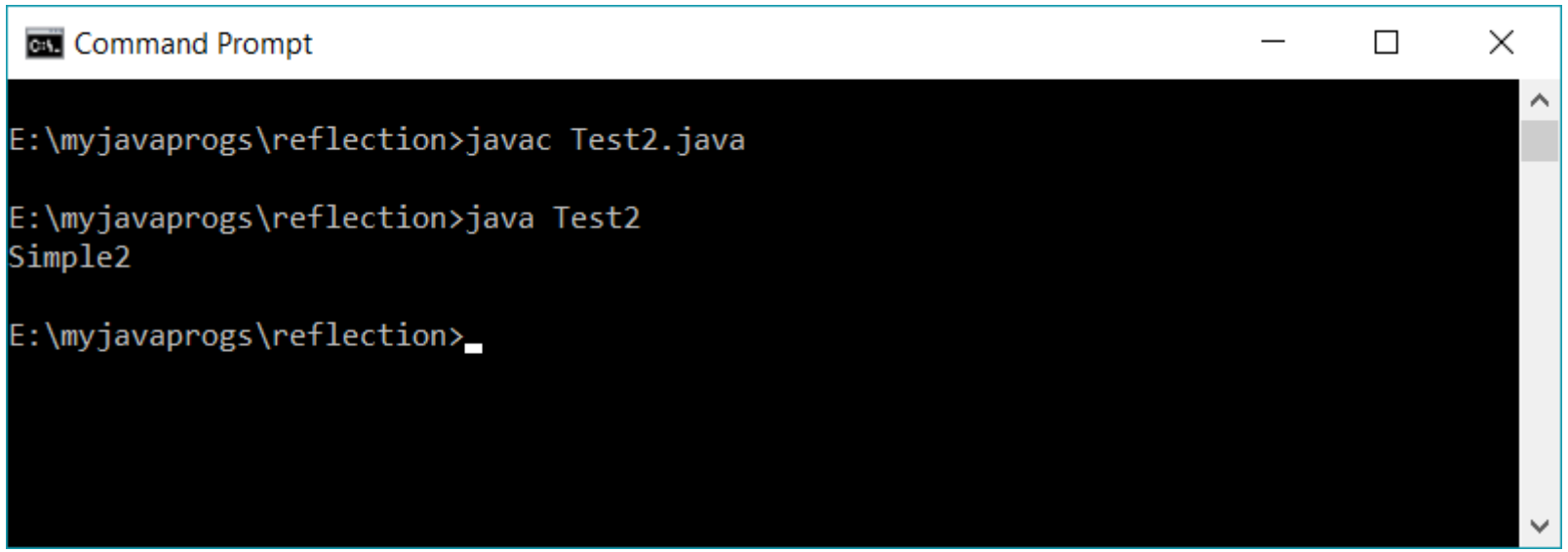
CLASS.FORNAME()

```
class Simple{}

class Test{
 public static void main(String args[]){
     try{
         Class c=Class.forName("Simple");
         System.out.println(c.getName());
     } catch(Exception e){
     }
 }
}
```


Command Prompt
C:\>e:
E:\>cd myjavaprogs/reflection
E:\myjavaprogs\reflection>javac Test.java
E:\myjavaprogs\reflection>java Test
Simple
E:\myjavaprogs\reflection>

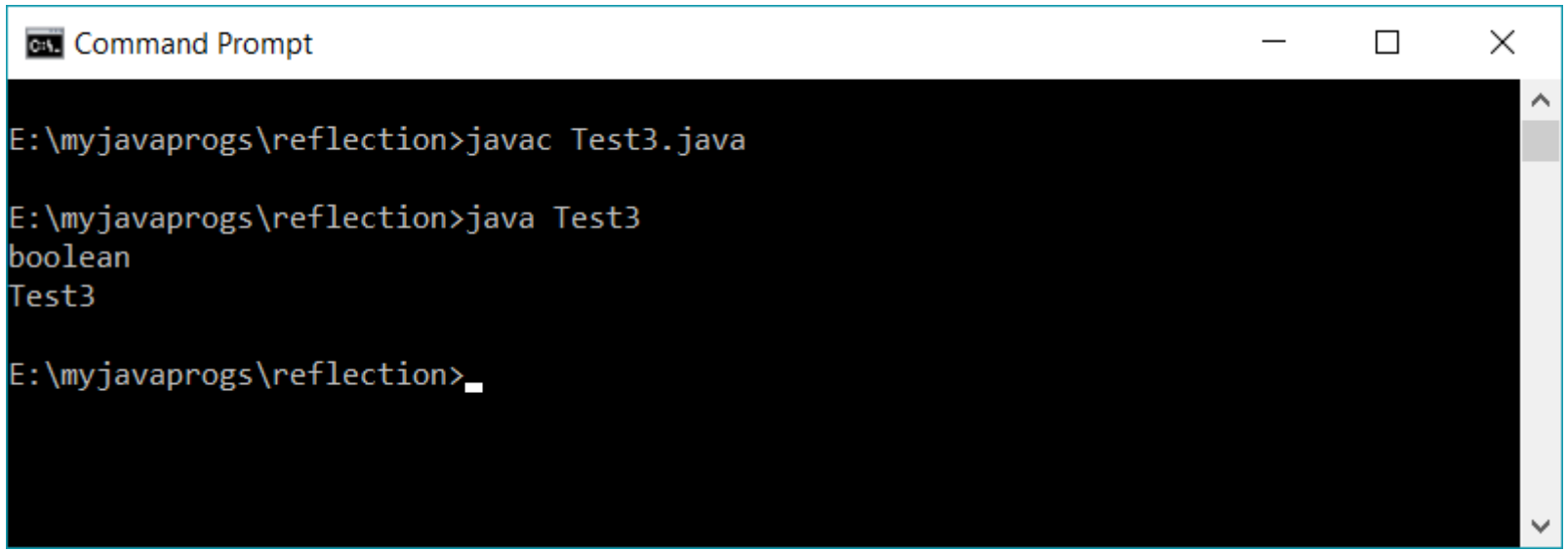
OBJECT.GETCLASS()

```
class Simple2{}

class Test2{
  void printName(Object obj){
  Class c=obj.getClass();
  System.out.println(c.getName());
  }
  public static void main(String args[]){
   Simple2 s=new Simple2();

   Test2 t=new Test2();
   t.printName(s);
 }
}
```


Command Prompt
E:\myjavaprogs\reflection>javac Test2.java
E:\myjavaprogs\reflection>java Test2
Simple2
E:\myjavaprogs\reflection>

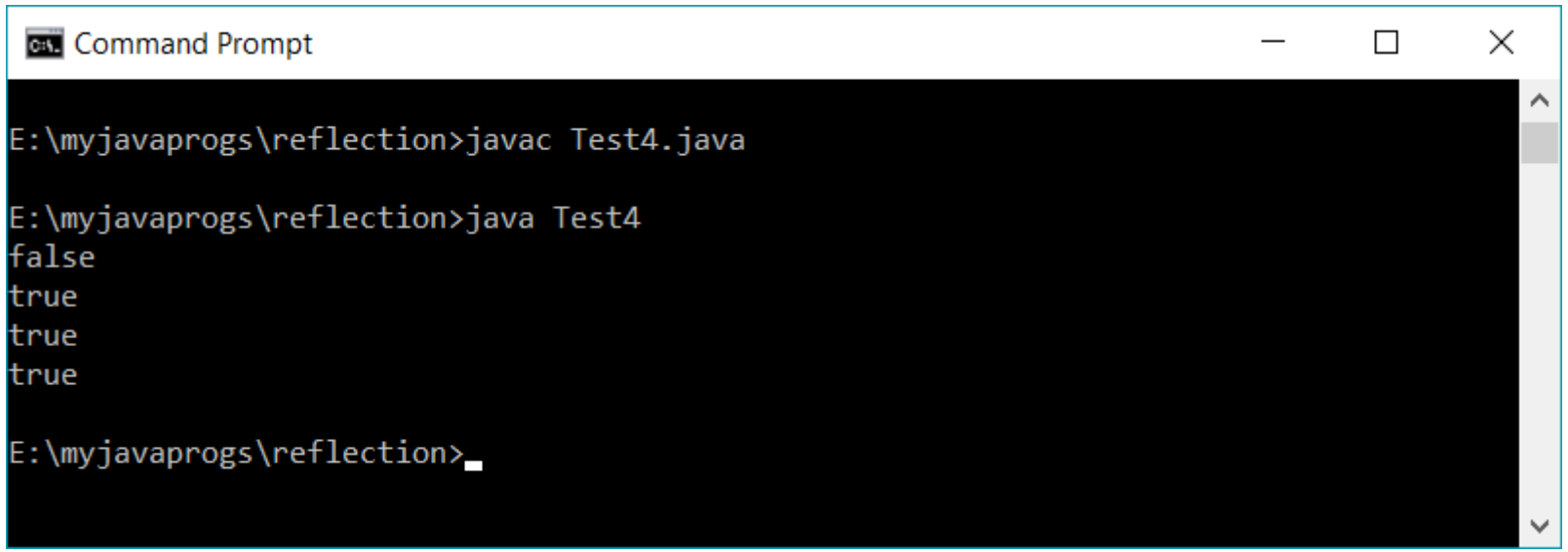
.CLASS SYNTAX

```
class Test3{
  public static void main(String args[]){
   Class c = boolean.class;
   System.out.println(c.getName());

   Class c2 = Test3.class;
   System.out.println(c2.getName());
 }
}
```


Command Prompt
E:\myjavaprogs\reflection>javac Test3.java
E:\myjavaprogs\reflection>java Test3
boolean
Test3
E:\myjavaprogs\reflection>

DETERMINE CLASS TYPES

```
class Simple{}
interface My{}

class Test4{
 public static void main(String args[]){
  try{
   Class c=Class.forName("Simple");
   System.out.println(c.isInterface());
   Class c2=Class.forName("My");
   System.out.println(c2.isInterface());
   byte[] bytes = new byte[1024];
   Class c3 = bytes.getClass();
   System.out.println(c3.isArray());
   Class c4 = boolean.class;
   System.out.println(c4.isPrimitive());
 }catch(Exception e){System.out.println(e);}
 }
}
```


Command Prompt
E:\myjavaprogs\reflection>javac Test4.java
E:\myjavaprogs\reflection>java Test4
false
true
true
true
E:\myjavaprogs\reflection>

OBTAINING CLASS OBJECTS OF ARRAYS

- Class c = int[][][].class;
  - The .class syntax may be used to retrieve a Class corresponding to a multi-dimensional array of a given type.
- Class cDoubleArray = Class.forName("[D");
  - The variable cDoubleArray will contain the Class corresponding to an array of primitive type double.
- Class cStringArray = Class.forName("[[Ljava.lang.String;");
  - The cStringArray variable will contain the Class corresponding to a two-dimensional array of String.
- [Lpacket.to.YourClass; for YourClass[] for general YourClass arrays!

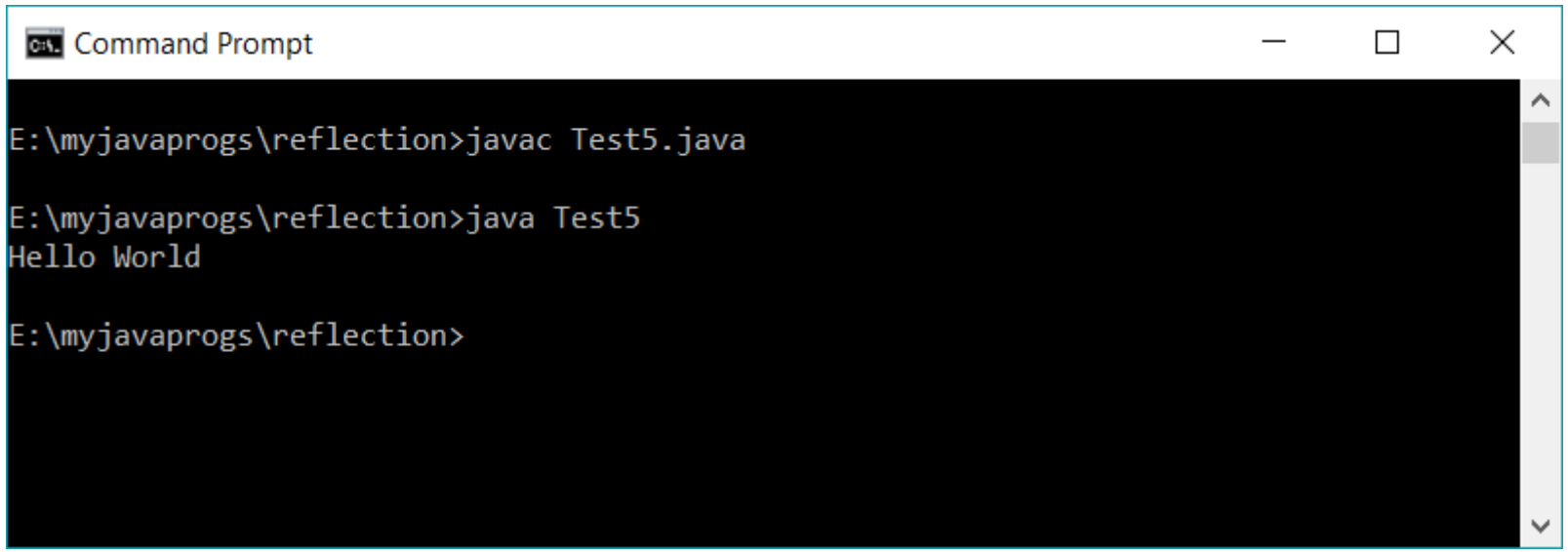
CREATING NEW CLASS INSTANCES

1. Through java.lang.reflect.Constructor.newInstance()
2. Through Class.newInstance()

- ✓ Prefer 1. because:
  - ✓ Class.newInstance() can only invoke the zero-argument constructor, while Constructor.newInstance() may invoke any constructor, regardless of the number of parameters.
  - ✓ Class.newInstance() throws any exception thrown by the constructor, regardless of whether it is checked or unchecked. Constructor.newInstance() always wraps the thrown exception with an InvocationTargetException.
  - ✓ Class.newInstance() requires that the constructor be visible; Constructor.newInstance() may invoke private constructors under certain circumstances.

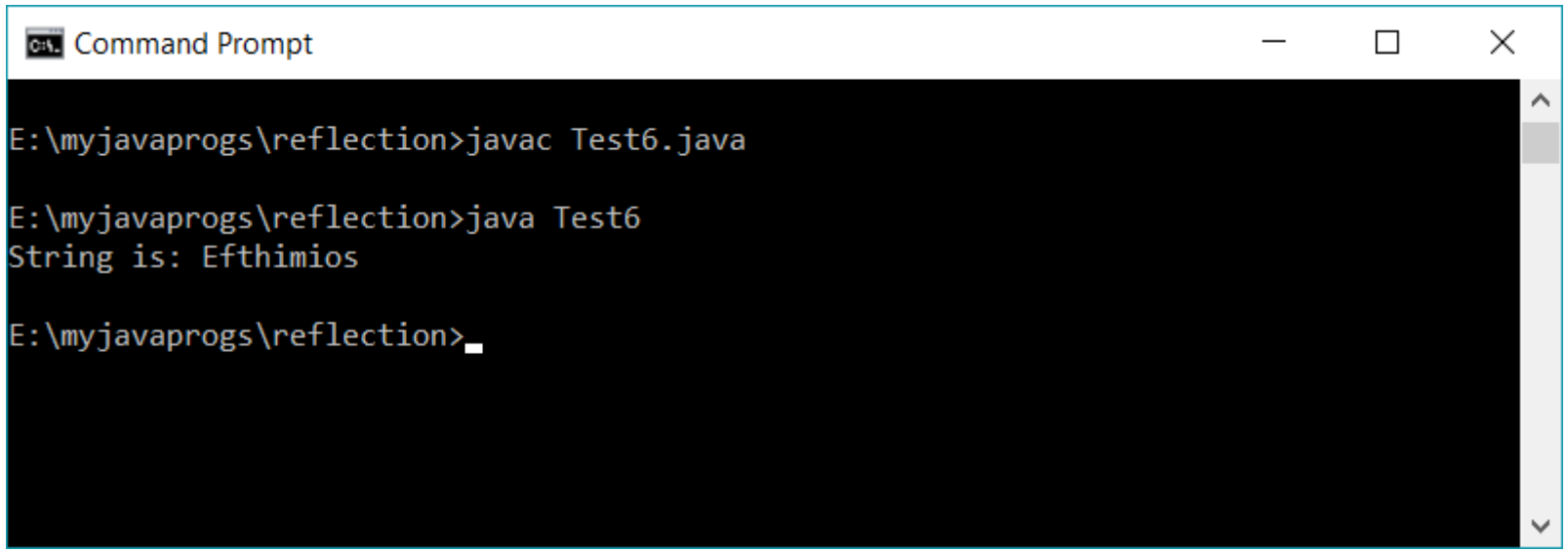
CLASS.NEWINSTANCE()

```
class Simple{
 void message(){System.out.println("Hello World");}
}

class Test5{
 public static void main(String args[]){
  try{
  Class c=Class.forName("Simple");
  Simple s=(Simple)c.newInstance();
  s.message();

  }catch(Exception e){System.out.println(e);}

 }
}
```


Command Prompt
E:\myjavaprogs\reflection>javac Test5.java
E:\myjavaprogs\reflection>java Test5
Hello World
E:\myjavaprogs\reflection>

CONSTRUCTOR.NEWINSTANCE()

```
import java.lang.reflect.Constructor;
class Simple{
 String s;
 public Simple(String s1){
  s=s1;
 }
 void message(){System.out.println("String is: "+s);}
}
class Test6{
 public static void main(String args[]){
  try{
  Constructor<Simple> constructor = Simple.class.getConstructor(String.class);
  Simple s = (Simple)constructor.newInstance("Efthimios");
  s.message();
  }catch(Exception e){System.out.println(e);}
 }
}
```


Command Prompt
E:\myjavaprogs\reflection>javac Test6.java
E:\myjavaprogs\reflection>java Test6
String is: Efthimios
E:\myjavaprogs\reflection>

MEMBER (INTERFACE)

- Member is an interface that reflects identifying information about a single class member (a field or a method) or a constructor
- java.lang.reflect.Member
- Implemented by:
  - Fields
    - java.lang.reflect.Field
  - Methods
    - java.lang.reflect.Method
  - Contstructors
    - java.lang.reflect.Constructor

FIELDS

- Fields have a type and a value
- A Field provides information about, and dynamic access to, a single field of a class or an interface
- The reflected field may be a class (static) field or an instance field
- A Field permits widening conversions to occur during a get or set access operation
- The java.lang.reflect.Field class provides methods for accessing type information and setting and getting values of a field on a given object

OBTAINING FIELD OBJECTS

- A field may be either of **primitive** or **reference** type
- There are eight primitive types: boolean, byte, short, int, long, char, float, and double
- A reference type is anything that is a direct or indirect subclass of java.lang.Object including interfaces, arrays, and enumerated types

```
Class aClass = ...//obtain class object

Field[] fields = aClass.getFields();

Field[] declaredFields = aClass.getDeclaredFields();
```

- getFields() method returns all the accessible public fields declared in the class or inherited from the superclass
- getDeclaredFields() method returns all the fields that appear in the declaration of the class only (not from inherited fields)

FIELD OBJECTS

- If you know the name of the field you want to access:

```
Class aClass = ...//obtain class object

Field field = aClass.getField("someField");
```

- If no field exists with the name given as parameter to the getField() method, a *NoSuchFieldException* is thrown

OBTAIN FIELD NAME

```
Field field = ... //obtain field object
String fieldName = field.getName();
```

OBTAIN FIELD TYPE

```
Field field = ... //obtain field object
Object fieldType = field.getType();
Class genericType = field.getGenericType(); //here we talk about types of Objects too!
```

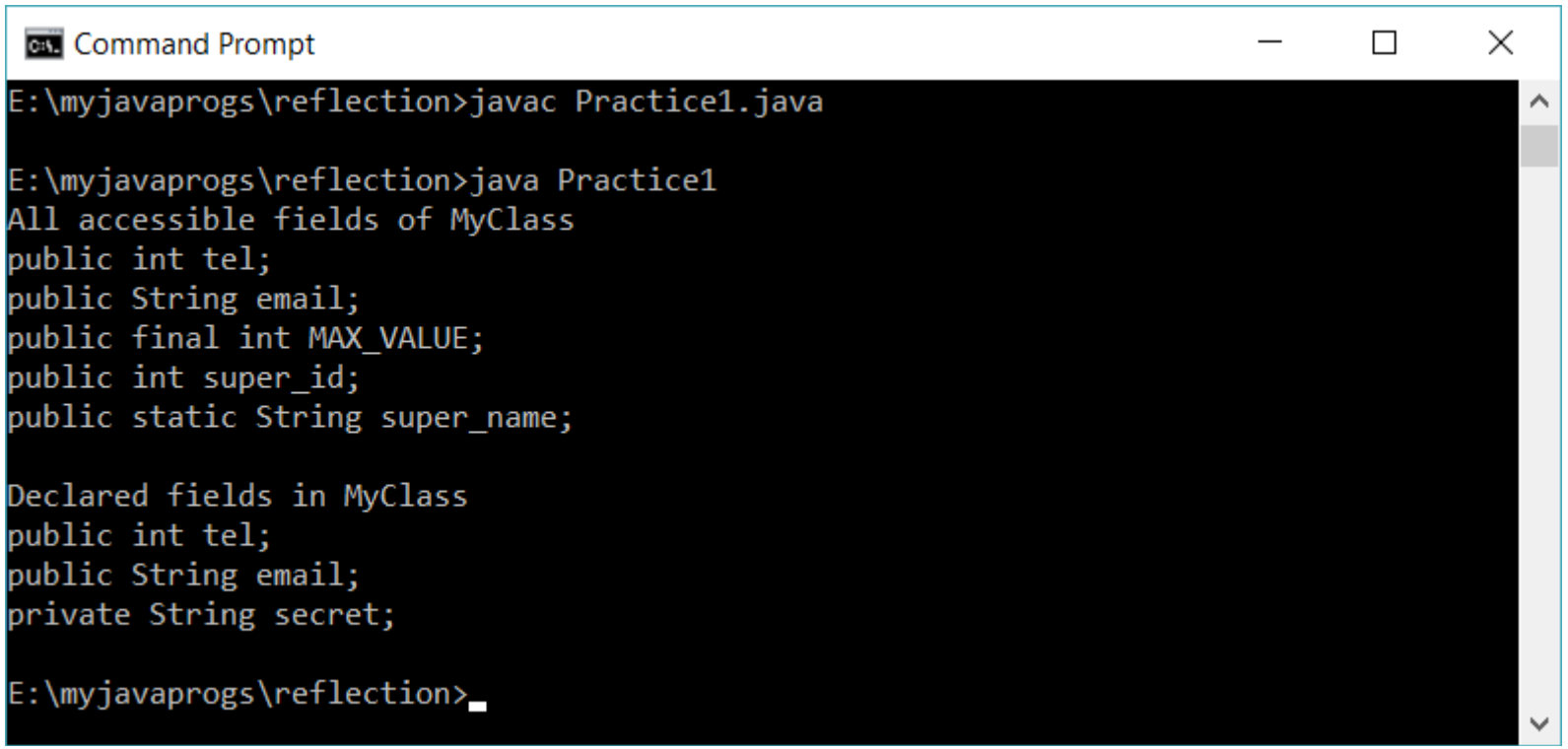
OBTAIN FIELD MODIFIERS

```
Field field = ... //obtain field object
int fieldType = field. getModifiers();
//Be careful here since we need actual names of modifiers not just ints…
String modifiers = Modifier.toString(fieldType);
```

TIME TO PRACTICE

- Write the code that discovers all the available fields for a specific class and print their names, types and modifiers!
- For the "Test" class use the following MyClass:

```
class MySuperClass {
        public final int MAX_VALUE = 100;
        public int super_id;
        public static String super_name;
}
class MyClass extends MySuperClass{
        public int tel;
        public String email;
        private String secret;
}
```


Command Prompt
E:\myjavaprogs\reflection>javac Practice1.java
E:\myjavaprogs\reflection>java Practice1
All accessible fields of MyClass
public int tel;
public String email;
public final int MAX_VALUE;
public int super_id;
public static String super_name;
Declared fields in MyClass
public int tel;
public String email;
private String secret;
E:\myjavaprogs\reflection>

```
public class Practice1 {
  public static void main(String[] args) {
        Class c = MyClass.class;
        Field[] fields = c.getFields();
        Field[] declaredFields = c.getDeclaredFields();
        System.out.println("All accessible fields of " + c.getName());
        for (Field f  : fields){
              Class type = f.getType();
              String name = f.getName();
              String modifier = Modifier.toString(f.getModifiers());
              System.out.println(modifier +" " + type.getSimpleName() + " " + name + ";");
        }
        System.out.println("\nDeclared fields in " + c.getName());
        for (Field f  : declaredFields){
              Class type = f.getType();
              String name = f.getName();
              String modifier = Modifier.toString(f.getModifiers());
              System.out.println(modifier +" " + type.getSimpleName() + " " + name + ";");
        }
  }
}
```

PRACTICE ON METHODS AND CONSTRUCTORS

- ✓ Based on the last 3 slides, practicing on fields, create you own example with two classes (parent and child class) which will also have methods and constructors.
- ✓ Afterwards, write code using reflection to expose them all!