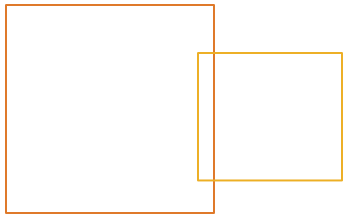
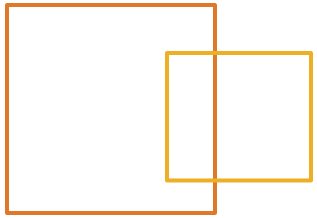
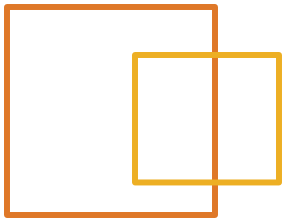
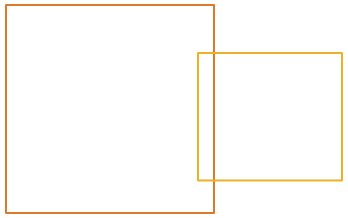
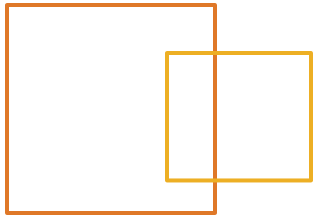
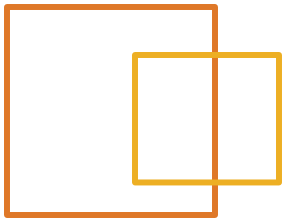





# Fast Track to Java

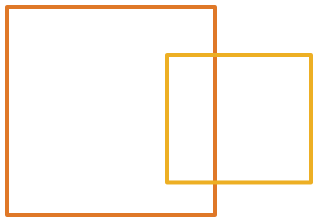
Customized for Starbucks

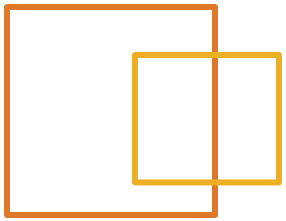
*Delivered by DevelopIntelligence*


# Introduction to Reflection

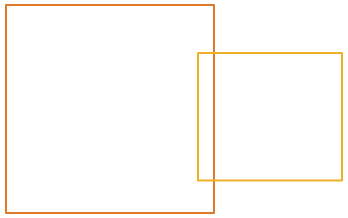
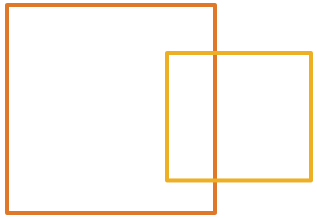
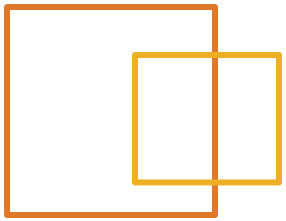
# Objectives





At the end of this module you should be able to:

- Define reflection
- Describe how reflection is implemented in Java
- Use reflection to perform dynamic discovery
- Use reflection to perform dynamic invocation
- Create a basic Java Reflection Example


# What is Reflection?

# What is Reflection? [part 1]

- “Process by which a computer program can be modified in the process of execution”
- Process typically referred to as *reflective programming*

# What is Reflection? [part 2]

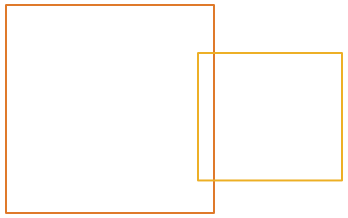
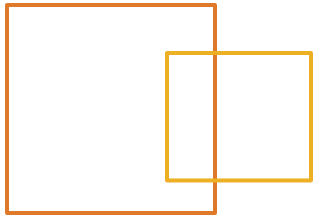
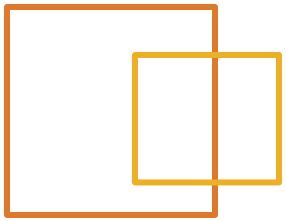
- API - Provides run-time information discovery for classes, fields, methods, and constructors for objects loaded into the VM
- Introduced in 1.1
- Released in concert with JavaBeans and Remote Method Invocation
- Updated to support new language features
- Performance enhancements made in 1.4

# Motivations for using Reflection

- Delegate decisions to run-time instead of compile-time
  - Instantiation of objects
  - Discovery of object's capabilities
  - Modification of capabilities
- Provides more decoupled source-code

# Platform Support for Reflection

- Fundamental support provided by JVM
- Utilizes dynamic binding
  - Known as dynamic class loading
  - Provided through `ClassLoaders`
  - “Translates” .class files into `Class` objects
- Supported by two packages
  - `java.lang`
  - `java.lang.reflect`


# Working with the Reflection API

# Working with the Reflection API

- `Class` object is the starting point
- Accessed through `Class` literal
  - `Class clazz = Math.class;`
  - Abides by platform security rules
- Multiple ways to get `Class` object
- Use dynamic class loading -
  - `Class clazz = Class.forName("java.lang.String");`
  - Get class from an object -
  - `String s = "Hello";`
  - `Class clazz = s.getClass();`
- Once `Class` object is located; can start dynamic discovery process

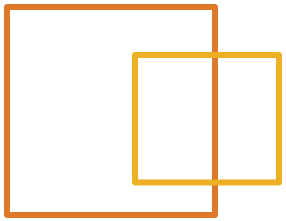
# Dynamic Class loading Example

```
1  package examples.reflection;
2
3  /**...*/
11 public class DynamicClassLoadingExample {
12
13   public static void main(String[] args) {
14     if(args.length == 0) {
15       System.out.println("Please specify a classname");
16       System.exit(0);
17     }
18
19     Class clazz = null;
20     try {
21       clazz = getClasss(args[0]);
22       System.out.println("Class name: " + clazz.getName());
23       System.out.println("Class simple name: " + clazz.getSimpleName());
24     } catch (ClassNotFoundException e) {
25       e.printStackTrace();
26       System.out.println("Could not load: " + args[0]);
27     }
28   }
29
30   private static Class getClasss(String className) throws ClassNotFoundException {
31     Class returnValue = null;
32     returnValue = Class.forName(className);
33     return returnValue;
34   }
35 }
36
```

# Discovery Class Type-Information

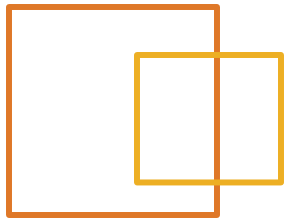
- Can be used to find information about the class
  - `Package` – supports further reflection
  - Modifiers
  - Name – simple or canonical
  - Hierarchy – supports further reflection
  - `getSuperclass : Class`
  - `getInterfaces : Class []`
- Can be used to find annotations

# Discovery Class Contents



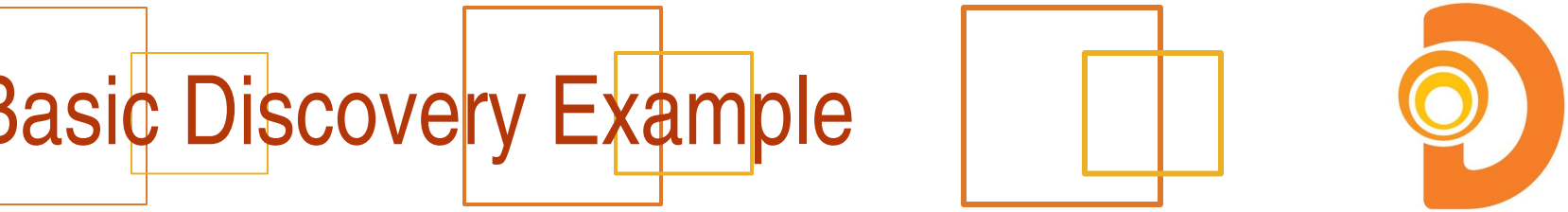
- Can be used to find contents of class
  - `Field` – named, public, or entire set
  - `Constructor` – named, typed, public, or entire set
  - `Method` – named and typed, public, or entire set
- All contents are considered `AccessibleObject`

# Further Discovery



- Most discovered entities support further reflection
- Primitives are handled in special way
  - Denoted by wrapper class
  - Can test to determine if field is primitive
- Included support for
  - Generics
  - Enumerations

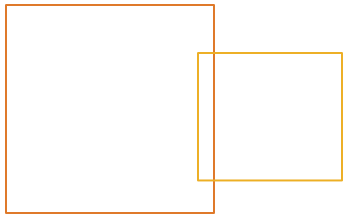
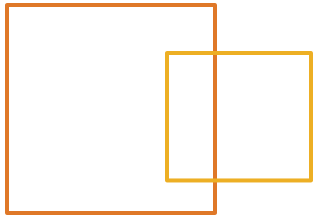
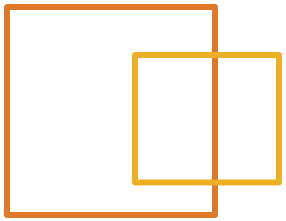
# Basic Discovery Example



```
1   package examples.reflection;
2
3   import java.lang.reflect.Modifier;
4
5   /**...*/
9   public class BasicDiscoveryExample {
10
11      public static void main(String[] args) {
12        if (args.length == 0) {
13          System.out.println("Please specify a classname");
14          System.exit(0);
15        }
16
17        try {
18          //get the class
19          Class clazz = getClasss(args[0]);
20          //some class information
21          printBasicClassInfo(clazz);
22        } catch (ClassNotFoundException e) {
23          e.printStackTrace();
24          System.out.println("Could not load: " + args[0]);
25        }
26      }
27
28      private static Class getClasss(String className)
29                              throws ClassNotFoundException {...}
```

# Basic Discovery Example [cont.]

```
40    /**...*/
51    private static void printBasicClassInfo(Class clazz) {
52      Package containPkg = clazz.getPackage();
53      String className = clazz.getSimpleName();
54      boolean isInterface = clazz.isInterface();
55      boolean isEnum = clazz.isEnum();
56      String typeStructure = (isInterface ? "interface" :
57                              (isEnum ? "enumeration" : "class"));
58
59      String modifiers = Modifier.toString(clazz.getModifiers());
60      String pkgName = containPkg.getName();
61
62      System.out.println("Class simple name: " + className);
63      System.out.println(className + " is a " + typeStructure);
64      System.out.println(className + " is considered " + modifiers);
65      System.out.println(className + " belongs to " + pkgName);
66
67      //parent class
68      Class parent = clazz.getSuperclass();
69      if(parent != null) {
70        System.out.println(className + "'s parent : " + parent.getName());
71        System.out.println("\nParent information");
72        printBasicClassInfo(parent);
73      }
74
75    }
76
77  }
```


# Advanced Reflection

# What is Reflection? [part 3]

- API - Provides run-time invocation and modification on discovered fields, constructors, and methods
- Reflection is not just about discovery; also about execution

# Run-time Modification Using Reflection

- `Class` object is still starting point
- Dynamically discover fields, constructors, or methods
- Perform appropriate invocation

# Instantiation Using Reflection

- May need to convert `Class` object into `Object` object
- Two reflection based mechanisms
  - `newInstance` method on every `Class` object
  - Functions like new operator
  - Relies on public no-argument constructor
  - `newInstance` method on `Constructor` object
  - Discover `Constructor` from class
  - Functions like `new` operator
  - Relies on argument list associated with specific `Constructor` object
- Be sure to handle exceptions

# Field Modification Using Reflection

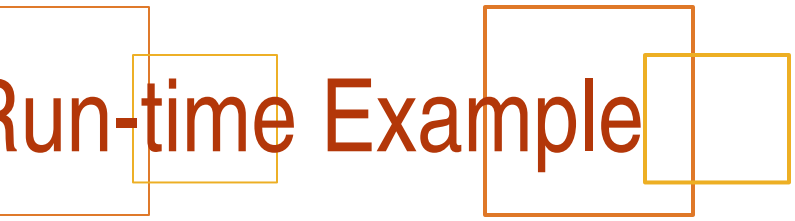
- Field modification requires an object instance
  - Could be `Class` object (static)
  - Or an `Object` object (instance)
- Object can be result of run-time instantiation, but not required
  - Discover `Field` from `Class`
  - Get / set `Field` using `Field`, `Object` instance, and field value
  - `public void set(Object instance, Object value)`
  - `public Object get(Object instance)`
  - `public void setInt(Object instance, int value)`
  - `public int getInt(Object instance)`

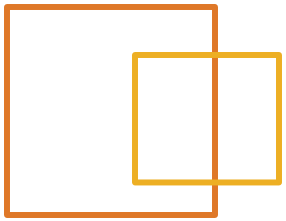
# Method Invocation Using Reflection

- Method invocation requires an object instance
  - Could be `Class` object (static method)
  - Or an `Object` object (instance method)
- `Object` can be result of run-time instantiation
  - Discover method from `Class`
  - Invoke method using `Method`, `Object` instance, and method args

```
public Object invoke(Object instance, Object… args)
```

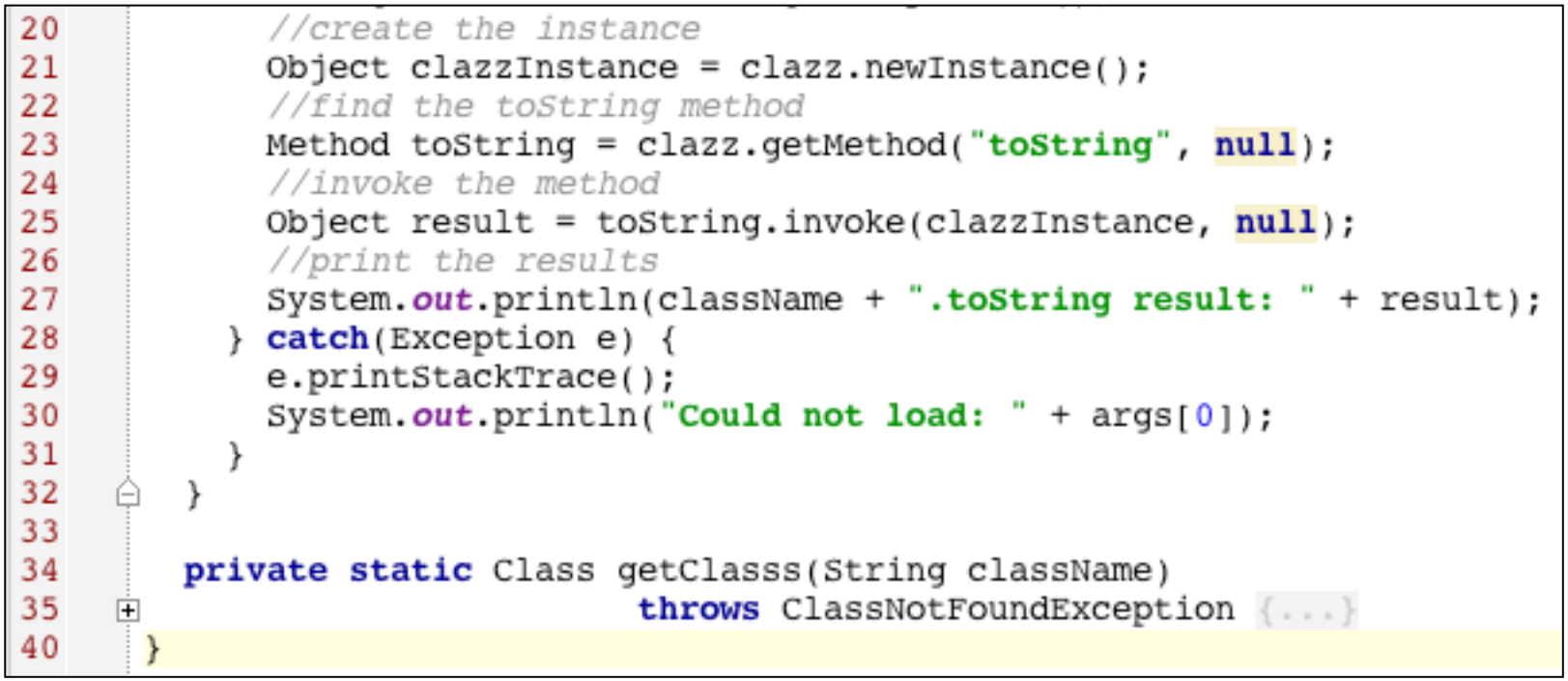
# Run-time Example





```
1   package examples.reflection;
2
3   import java.lang.reflect.Method;
4   /**...*/
8   public class RuntimeInvocationExample {
9
10    public static void main(String[] args) {
11      if (args.length == 0) {
12        System.out.println("Please specify a classname");
13        System.exit(0);
14      }
15
16      try {
17        //get the class
18        Class clazz = getClasss(args[0]);
19        String className = clazz.getSimpleName();
20        //create the instance
```

# Run-time Example [cont.]

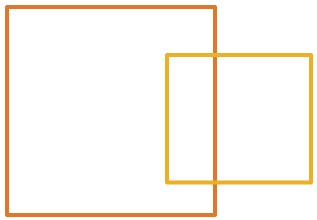


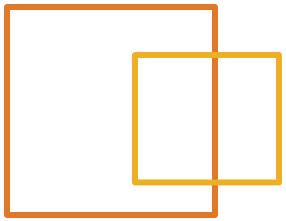
```
20          //create the instance
21          Object clazzInstance = clazz.newInstance();
22          //find the toString method
23          Method toString = clazz.getMethod("toString", null);
24          //invoke the method
25          Object result = toString.invoke(clazzInstance, null);
26          //print the results
27          System.out.println(className + ".toString result: " + result);
28        } catch(Exception e) {
29          e.printStackTrace();
30          System.out.println("Could not load: " + args[0]);
31        }
32    }
33
34    private static Class getClasss(String className)
35                              throws ClassNotFoundException {...}
40  }
```

# Reflection best-practices

- At one point, goal was to use sparingly
- Things have changed
  - Significant performance improvements have helped
- Consider using Reflection to implement design patterns
- Make sure Security Manager is configured to support reflection

# Summary





In this module, we covered:

- Reflection is not a new API
- It is a powerful run-time object interrogation facility
- It supports run-time object execution mechanism