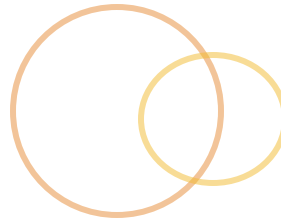
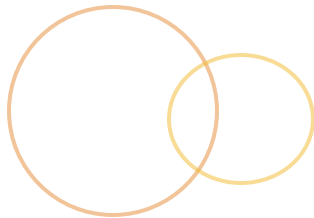
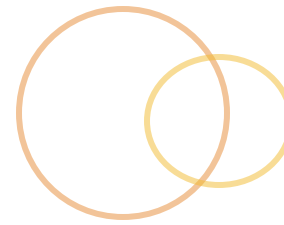


Introduction to Reflection



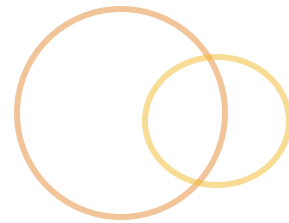
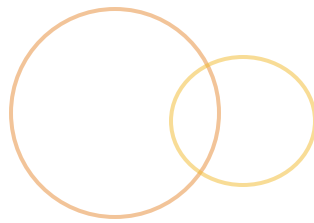
Presentation Topics



In this presentation, we will cover:

- ⦿ What is Reflection?
- ⦿ Working with the Reflection API
- ⦿ Advanced Reflection

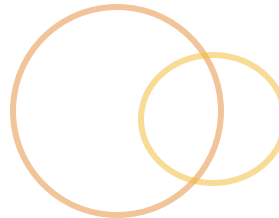
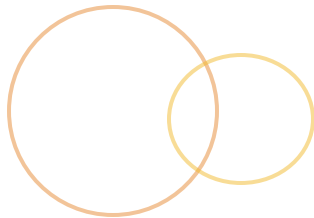
Objectives



When we are done, 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



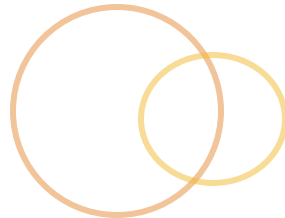
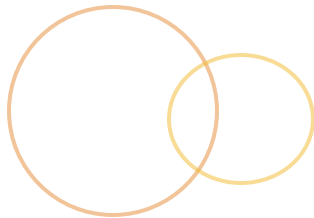
- ⦿ Delegate decisions to run-time instead of compile-time
 - ⦿ Instantiation of objects
 - ⦿ Discovery of object's capabilities
 - ⦿ Modification of capabilities
- ⦿ Provides more decoupled source-code

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 = getClassss(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 getClassss(String className) throws ClassNotFoundException {
31         Class returnValue = null;
32         returnValue = Class.forName(className);
33         return returnValue;
34     }
35 }
36
```

Discovery Class Type-Information



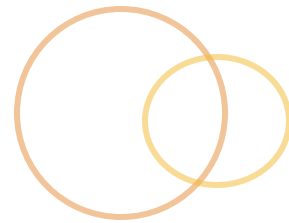
- Can be used to find information about the class
 - Package – supports further reflection
 - Modifiers
 - Name – simple or canonical
 - Hierarchy – supports further reflection
 - `getSuperclass : Class`
 - `getInterfaces : Class []`
- Can be used to find annotations

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



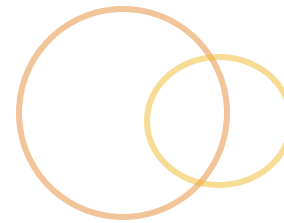
```
1  package examples.reflection;
2
3  import java.lang.reflect.Modifier;
4
5  /** ... */
9  public class BasicDiscoveryExample {
10
11     public static void main(String[] args) {
12         if (args.length == 0) {
13             System.out.println("Please specify a classname");
14             System.exit(0);
15         }
16
17         try {
18             //get the class
19             Class clazz = getClassss(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 getClassss(String className)
29         throws ClassNotFoundException {...}
30 }
```

Basic Discovery Example [cont.]



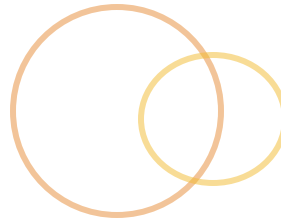
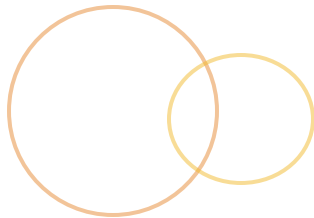
```
40  /**...*/
41  +
42  -
43  private static void printBasicClassInfo(Class clazz) {
44      Package containPkg = clazz.getPackage();
45      String className = clazz.getSimpleName();
46      boolean isInterface = clazz.isInterface();
47      boolean isEnum = clazz.isEnum();
48      String typeStructure = (isInterface ? "interface" :
49                              (isEnum ? "enumeration" : "class"));
50
51      String modifiers = Modifier.toString(clazz.getModifiers());
52      String pkgName = containPkg.getName();
53
54      System.out.println("Class simple name: " + className);
55      System.out.println(className + " is a " + typeStructure);
56      System.out.println(className + " is considered " + modifiers);
57      System.out.println(className + " belongs to " + pkgName);
58
59      //parent class
60      Class parent = clazz.getSuperclass();
61      if(parent != null) {
62          System.out.println(className + "'s parent : " + parent.getName());
63          System.out.println("\nParent information");
64          printBasicClassInfo(parent);
65      }
66  }
67
68  }
69
70  }
```


Reflection Lab 1



- **Description:** Use `BasicDiscoveryExample` as your starting point. Convert `BasicDiscoveryExample` into a replacement for the JDK command-line utility `javap`. `javap` provides a source-code like view of a class, showing class structure, fields, and methods. See `instructions.txt` for more details.
- **Duration:** 30 minutes

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 /**...*/
5
6 public class RuntimeInvocationExample {
7
8     public static void main(String[] args) {
9         if (args.length == 0) {
10             System.out.println("Please specify a classname");
11             System.exit(0);
12         }
13
14         try {
15             //get the class
16             Class clazz = getClasses(args[0]);
17             String className = clazz.getSimpleName();
18             //create the instance
19
20 }
```


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  }
```

New Features in Reflection



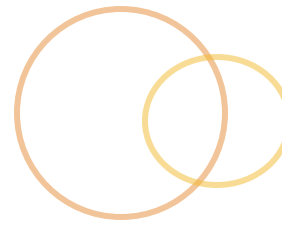
- ① 1.3 – Dynamic proxy support
 - ① Useful for “wrapping” an implementation
 - ① Implementation invocation requests go through `InvocationHandler`
- ① 1.4 – Performance improvements
- ① 1.5 – Support for new language features

Reflection best-practices



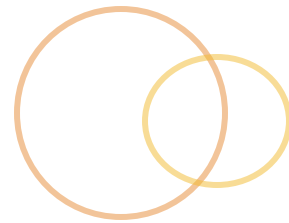
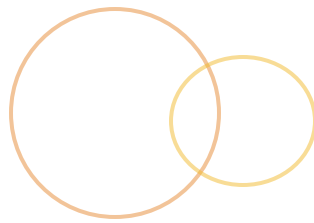
- ⦿ At one point, goal was to use sparingly
- ⦿ Things have changed
 - ⦿ Significant performance improvements have helped
- ⦿ Consider using Reflection to implement design patterns
- ⦿ Make sure Security Manager is configured to support reflection

Reflection Lab 2



- **Description:** Use `RuntimeInvocationExample` as a starting point. The current `RunTimeInvocationExample` uses the `newInstance` method to create an instance of a class. In certain cases, this is fine. However, most classes have constructors that take arguments for their object initialization. Expand `RuntimeInvocationExample` to support constructor-based object instantiation for the primitive wrapper classes and `String`. See `instructions.txt` for more information.
- **Duration:** 30 minutes

Summary



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

Reflection Lab [Optional]



- ⦿ **Description:** Create a basic unit-test framework using the Reflection API. The unit-test framework, `UnitTester` should dynamically load a class passed as a command line argument. Once the class is loaded, the `UnitTester` should inquire the class for its test methods. All test methods should begin with the word `test` and should not take any arguments. They should return either a `true` or `false`. Test methods can generate exceptions if need be. The `UnitTester` should test static methods first and then proceed to instance methods. The `UnitTester` should keep track of the methods that passed (`true`), failed (`false`), or error (exception). Once all the methods have been tested, the `UnitTester` should generate a test report.
- ⦿ **Duration:** 30 minutes.