

Reflection in Java

By: Zachary Cava

A decorative graphic consisting of several horizontal lines of varying lengths and colors (teal and white) extending from the right side of the slide.

What exactly is a class?

- It's a collection of different things, such as:
 - Fields
 - Methods
 - Constructors
- We define these different things with names, types, parameters, values, expressions, etc while programming, but in reflection all of this already exists.

Programming vs Reflecting

- We use reflection to manipulate things that already exist and are set in a specific way.
- But unlike programming, we are not tied to these defined states
- We have the ability to dynamically change what things are, regardless of how they were defined!
- More specifically, we are modifying the state of objects at runtime.

What do you mean Runtime?

- Normally you program something like this:
 - Write/Modify the class, methods, etc
 - Compile it
 - Run it
- If you want to make any changes you have to recompile and rerun that class.

What do you mean Runtime?

- With reflection, we can manipulate a class without ever recompiling it:
 - Write/Modify the class, methods, etc
 - Compile it
 - Run it
 - Modify the class state here!
- It is important to note that **another** class is the one doing the modification.

Uses of Reflection

- Some common uses of reflection:
 - To load and use classes unknown at compile time, but have set methods.
 - Example: The Critters assignment
 - Test programs by forcing specific states
 - By debuggers to inspect running programs
 - Malicious things
 - Hacking

Programming Reflection

- To program with reflection, we must put on our meta-thinking caps.
- We are going to modify classes from classes with classes!
- To do this we have a great set of classes in the following package:
 - `java.lang.reflect.*;`

```
Java.lang.reflect.*
```

Some classes we will go over, (there are more):

- Method
 - Describes a method for a class and gives access to it.
- Field
 - Describes a field for a class, its type, name, etc.
- Constructor<T>
 - Provides information about constructors and the ability to execute a constructor and get a new class instance

Java.lang.reflect.*

- **AccessibleObject**
 - Describes the accessibility of an object, i.e. its view public, private, protected, default.
- **Array**
 - A special class created just for reflecting with Arrays, since Arrays are such odd objects in Java we must use this class to manipulate them.

So where do we start?

- To start manipulating a class we must first get a hold of that class's "blueprint".
 - Using the `java.lang.Class` class
- There are two ways to do this, if the class is already loaded:
 - `Class<? extends Object> theClass = ClassName.class;`
- Or if we need to cause it to load:
 - `Class theClass = Class.forName("class.package");`
- We won't use this second one, its rather complex

So where do we start?

- So now we have the definition of a class.
- This is like the blueprint to the entire thing, it lists where everything is and how to get to it.
- It is important to point out that this class has information that pertains to the structure of the class, not specific instance information, but hold that thought for a little later.
- For now lets look at how to get some information from the class

The Parts of the Class

- Fields
- Methods
- Constructors
- Miscellaneous

Getting those sweet fields

- There are two ways to get class fields:
 - `getFields()` ;
 - Returns an array of Field objects, specifically all the fields that are public for this class and its super classes.
 - `getDeclaredFields()` ;
 - Returns an array of Field objects, regardless of view.
- Optionally if you know the field name:
 - `getField(String name)` ;
 - Returns a Field with the given name

The Parts of the Class

- Fields
- Methods
- Constructors
- Miscellaneous

Calling all methods, report for duty

- Like Fields there are two ways to get Methods
 - `getMethods () ;`
 - Returns all the public methods for this class and any it inherits from super classes.
 - `getDeclaredMethods () ;`
 - Returns all the methods for this class only regardless of view.
- Like Fields you can also get a specific method, but it takes more information.

Calling all methods, report for duty

- To get a specific method you call
 - `getMethod(String name, Class<?>... parameterTypes);`
- The name parameter is pretty straight forward, but does `Class<?>...` mean?
- This means you can pass any number of `Class<?>` parameters after the name.
- The `Class<?>` parameters you pass reference the types of parameters the method takes.

Calling all methods, report for duty

- For example, say we have this method:
 - `public int doSomething(String stuff, int times, int max){}`
- If we were trying to get this specific method we would have to call `getMethod` like this:
 - `getMethod("doSomething", String.class, int.class, int.class);`
- We are directly passing the types, and this is because the reflection will use the method “fingerprints” to track it down and return it to us.

The Parts of the Class

- Fields
- Methods
- **Constructors**
- **Miscellaneous**

Building blocks

- To get the constructors we have the methods:
 - `getConstructors()`
 - Returns all public constructors for the class
 - `getDeclaredConstructors()`
 - Returns all constructors for the class, regardless of view
- We can again get specific constructors with:
 - `getConstructor(Class<?>... parameterTypes);`
 - Returns the constructor that takes the given parameters

The Parts of the Class

- Fields
- Methods
- Constructors
- **Miscellaneous**

The others

- For this session we will only focus on variables and methods, but there are a number of other useful methods:
 - `getEnclosingMethod()`
 - Gets the method that declared an anonymous class
 - `getName()`
 - Returns the class name
 - `newInstance()`
 - Creates a new instance of the class

The Classes of Reflection

- Field
- Method
- Constructor
- ????????????????

The Field Class

- Some useful methods:

- `get(Object obj)`

- Gets the value of this field in the given object

- `getPrimitiveType(Object obj)`

- `set(Object obj, Object value)`

- Sets the value of this field in the given object, if possible

- `setPrimitiveType(Object obj, PrimitiveType value)`

- `getType()`

- Returns the type of this field

The Field Class

- You may have noticed the two methods *getPrimitiveType(...)* and *setPrimitiveType(...)*
- Here *PrimitiveType* is replaced with a real primitive type, so if a field represents an `int` you would say, `getInt()` or `setInt()`.
- This is done because primitive types are not classes and so we need a special way to get and set them

The Field Class

- The first parameter to all of those methods was `Object obj`
- This parameter is a specific instance of the class.
 - a constructed version of the class
- Like I mentioned before the Field object represents a generic version of a field for a class, it holds no value, its just a blueprint as to where it would be in the class.
- To get a value we must provide a class that has been constructed already.

The Field Class

- Don't forget we can have two types of fields, static/non-static
- If we want to get the value of a static field, we can pass null as the Object obj parameter.

The Classes of Reflection

- Field
- Method
- Constructor
- ??????????????

The Method Class

- Some useful methods

- `getName()`

- Gets the methods name

- `getReturnType()`

- Gets the type of variable returned by this method

- `getParameterTypes()`

- Returns an array of parameters in the order the method takes them

- `invoke(Object obj, Object... args)`

- Runs this method on the given object, with parameters.

The Method Class

- The main method of this class that we will use is `invoke(Object obj, Object... params)`
- The first parameter is exactly like the Field class methods, it is an instantiated class with this method that we can invoke.
- The second parameter means we can pass as many parameters as necessary to call this method, usually we will have to use the result of `getParameterTypes()` in order to fill those in.

The Classes of Reflection

- Field
- Method
- Constructor
- ??????????????

The Constructor Class

- Some useful methods

- `getParameterTypes()`

- Returns an array of parameter types that this constructor takes

- `newInstance(Object... initargs)`

- Creates a new class that this constructor is from using the given parameters as arguments.

The Constructor Class

- Only two methods? Well yes, we only have an hour to work with here! And the others are not as interesting.
- The method we are most concerned with is `newInstance(Object... initArgs)`
 - This is similar to `invoke(...)` for methods except we don't pass an already instantiated object because we are making a new one!
 - Like methods we will probably call `getParameterTypes()` first.

Overview

- Lets take a step back and look at all this information
- We can get a class blueprint and it's a class of type `Class` from `java.lang.Class`
- For reflection we use classes like `Field`, `Method`, and `Constructor` to reference pieces of the class
 - These are generic versions and we must pass them constructed versions (except for constructors)
 - From each of these reflection classes we have the ability to manipulate instances of classes.

Lets try it out!

- Whats the fun in learning something without trying it out?

- **Lets go!!**

Lets try it out

- So it turned out what we learned works pretty well for everything with a public visibility.
- But what about those private, protected, and default views?
- Java kept throwing an `IllegalAccessException`, we just don't have permissions to edit those.
- Well not to worry we can get permission!

The Classes of Reflection

- Field
- Method
- Constructor
- ??????????????

The Classes of Reflection

- Field
- Method
- Constructor
- **AccessibleObject!**

The AccessibleObject

- The accessible object is a superclass that Field, Method, and Constructor extend
 - **How convenient!**
- But what does it do?
- It controls access to variables by checking the accessibility of a field, method, or constructor anytime you try to get, set, or invoke one.

The AccessibleObject

- Some *very* useful methods:

- `isAccessible()`

- Tells whether or not the object can be accessed based on its view type

- A public field, method, or constructor will return true

- The other types will return false.

- `setAccessible(boolean flag)`

- This will override the accessibility setting to whatever is passed in, true or false

Overriding Accessibility

- So how can we use this?
- Well suppose we have a Field object that references a field in our class that was declared like this:
 - `private String secretMessage;`
- Well as we have seen we get an Exception, but we can avoid it by overriding the accessibility
 - `theField.setAccessible(true);`

Overriding Accessibility

- Now before you start the triangle pyramid of evil, note:
 - It is possible to prevent use of `setAccessible()`
 - You do this using a `SecurityManager` to prevent access to variables
 - Stuarts `CritterMain` does this for tournaments.

Applying Reflection

- Now that we have learned a little bit of reflection and have some tools under our belt, lets try it out.
- You can download the `ATM.class` from the course website
- To run it you will need to go to the command line, navigate to where you downloaded the file and then type
 - `java ATM`

The Secure Financial Corporation

- An area where security is extremely important is Banking
- We trust that banks keep all of our transactions secure and money safe
- Lets suppose we were just hired to check the security of Secure Financial Corporation's new Java powered ATM
- We will need to use reflection to try and leverage an attack against the machine.

The Secure Financial Corporation

- The company has decided it would be more secure for the card to verify that an ATM is valid by having cards that can execute methods.
- In particular every card must have a swipe method that takes in an ATM object that the card can use to validate is a real ATM.
- The ATM has a method `applyDecryption()` that the card must call to determine if the ATM has the proper credentials (Security Session Tie-in!)

The Secure Financial Corporation

- The card must pass an encrypted code to `applyDecryption()` which will return a decrypted code. The card can then use this code to make sure the ATM has the appropriate private keys. If it does then the swipe method returns a Data object for ATM with info.
- That would be all well and good for a secure system right?
- That way cards don't give out information to bad systems!

The Secure Financial Corporation

- Well its nice in theory, but it gives us a built ATM object!
- And as we have just learned with Reflection, we can get all the framework we want, but we need an instantiated version of the class to do real damage.
- Lets see what we can do!

Arrays

- If you wish to manipulate arrays with Reflection you must use the `java.lang.reflect.Array` class, you cannot use the `Field` class
- This is because Java does not handle Arrays in the same way it handles Objects or Primitives

Arrays

- Useful Methods

- `get(Object array, int index)`

- Gets the value from the array at the given index

- `getPrimitiveType(Object array, int index)`

- `set(Object array, int index, Object value)`

- Sets the value in the array at the index to the given value

- `setPrimitiveType(Object array, int index, PrimitiveType value)`

Arrays

- Just like the Field class, the *PrimitiveType* is replaced by an actual primitive type and you must use this type of placement when accessing a primitive array
- But there are a couple more methods that are unique to this class

Arrays

- **Unique Methods**

- `getLength(Object array)`

- Returns the length of the given array

- `newInstance(Class<?> componentType, int... dimensions)`

- Creates a new array of the given type and with the given dimensions

- `newInstance(Class<?> componentType, int length)`

- Creates a new array of the given type and with the given length

Critters

- So the last example we will look at is using Reflection to “win” Critters.

That's all folks!

- While there are many more things that make up Reflection and even more things you can do with Reflection, that is the extent of this lecture.
- I will post a secondary ATM that does not pass an ATM object to the swipe method, can you find the secret message and decode it?
- Hint: You can get a copy of the instantiated frames by calling `JFrame.getInstances()`, ATM instantiates a Frame.