Reflection

What is Metadata?

- Data about data is called as metadata. In other words, metadata refers to data that provides information about other data.
- For example, metadata for a photo might include:
    - The date and time it was taken
    - The location where it was taken
    - The camera used to take it
    - The resolution of the image.
- In the case of digital media, metadata can include information about:
    - the artist
    - title
    - genre and album of a particular song or video.
- In a file management application, metadata such as:
    - file name
    - size
    - creation date and modification date are commonly used to organize and search for files.

Metadata for Interface

- What is the name of interface?
- In which package it is declared?
- Which is the access modifier of interface?
- Which annotations are used on interface?
- Which are the super interfaces of interface?
- Which are the members declared inside interface?

Metadata for Class

- What is the name of class?
- In which package it is declared?
- Which are the modifiers used on class?
- Which are the annotations used on class?
- Which is the super class of class?
- Which are the super interfaces of the class?
- Which are the members declared inside class?

Metadata of Field

- What is the name of the field?
- Which is the type of field?
- Which are the modifiers of field?
- Which annotations has been used on the field?
- Whether field is declared or inherited field?

Metadata of method

- What is the name of method?
- Which are the modifiers used with method?
- Which is the return type of method?
- Which are the parameters of the methods?
- Which exceptions method throws?
- Which annotations has been used on the method.
- Whether method is declared or inerhited method?

Application of Metadata

- Metadata removes the need for native C/C++ header and library files when compiling.
- Integrated Development Environment(IDE) uses metadata to help us write code. Its IntelliSense feature parses metadata to tell us what fields and methods a type offers and in the case of a method, what parameters the method expects.
- Metadata allows an object’s fields to be serialized into a memory block, sent to another machine, and then deserialized, re-creating the object’s state on the remote machine.
- The Garbage Collector uses metadata to keep track of each object's lifecycle, from creation to destruction.

Reflection

- Reflection in Java is a feature that allows a program to examine or modify the behavior of a class, method, or object at runtime.
- It is a relatively advanced feature and should be used only by developers who have a strong grasp of the fundamentals of the language.

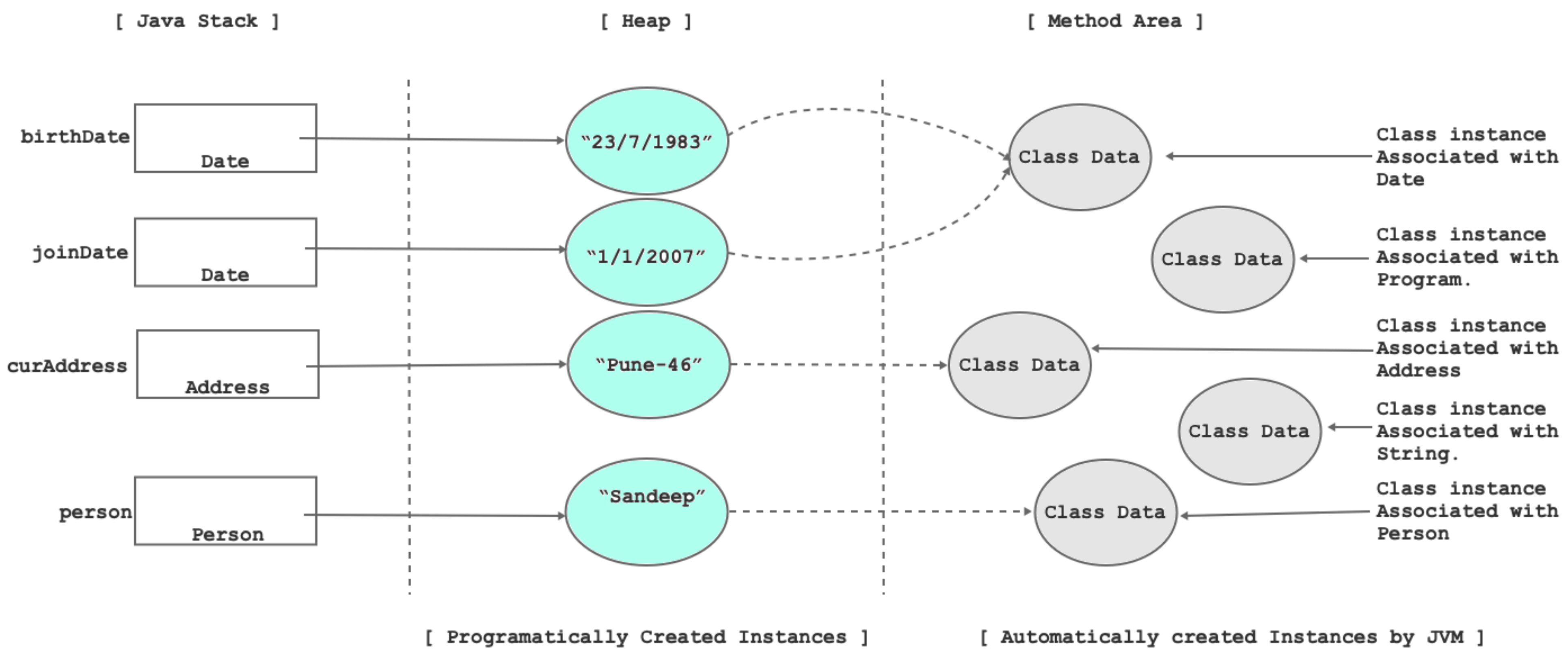
- reflection is a powerful technique and can enable applications to perform operations which would otherwise be impossible.
- using reflection, we can:
  - Obtain information about the class at runtime, such as its name, superclass, implemented interfaces, constructors, methods, and fields.
  - Create new objects of a class dynamically, without knowing the class name at compile time.
  - Access and modify the values of fields in an object, even if they are declared as private.
  - Invoke methods on an object dynamically, without knowing the method names at compile time.

How to use reflection in Java?

- Consider following code:

```java
class Date{
  //TODO: Member definition
}
class Address{
  //TODO: Member definition
}
class Person{
  //TODO: Member definition
}
class Program{
  public static void main(String[] args) {
    Date birthDate = new Date( 23, 7, 1983 );
    Date joinDate = new Date( 1, 1, 2007 );
    Address curAddress = new Address( "Pune-46");
    Person person = new Person("Sandeep", currentAddress, birthDate );
  }
}
```

- When class loader loads Program, String, Date, Address, Person class for execution then it create instance of java.lang.Class per loaded type on Method area. Instance contains metadata of the loaded type.


[Java Stack]
[Heap]
[Method Area]
birthDate
Date
joinDate
Date
curAddress
Address
person
Person
"23/7/1983"
"1/1/2007"
"Pune-46"
"Sandeep"
Class Data
Class Data
Class Data
Class Data
Class Data
Class instance Associated with Date
Class instance Associated with Program.
Class instance Associated with Address
Class instance Associated with String.
Class instance Associated with Person
[Programatically Created Instances]
[Automatically created Instances by JVM]

java.lang.Class class

- Class class is a final class declared in java.lang package.
- The entry point for all reflection operations is java.lang.Class.
- Instances of the class java.lang.Class represent classes and interfaces in a running Java application.
- Class has no public constructor. Instead Class objects are constructed automatically by the Java Virtual Machine.
- **Methods of java.lang.Class:**
  - public static Class<?> forName(String className) throws ClassNotFoundException
  - public Annotation[] getAnnotations()
  - public Annotation[] getDeclaredAnnotations()
  - public ClassLoader getClassLoader()
  - public Constructor<?>[] getConstructors() throws SecurityException
  - public Constructor getConstructor(Class<?>... parameterTypes) throws NoSuchMethodException, SecurityException
  - public Field getDeclaredField(String name) throws NoSuchFieldException, SecurityException
  - public Field[] getDeclaredFields() throws SecurityException
  - public Field getField(String name) throws NoSuchFieldException, SecurityException
  - public Field[] getFields() throws SecurityException
  - public Class<?>[] getInterfaces()

- public Method[] getMethods() throws SecurityException
- public Method[] getDeclaredMethods()throws SecurityException
- public String getName()
- public String getSimpleName()
- public Package getPackage()
- public InputStream getResourceAsStream(String name)
- public String getTypeName()
- public T newInstance() throws InstantiationException, IllegalAccessException

Retrieving Class Objects

- Using **getClass() method**:

```
Integer i = new Integer( 123 );
Class<?> c = i.getClass( ); //OK
```

  - getClass is final method of java.alng.Object class:

```
public final native Class<?> getClass();
```

- Using **.class Syntax**:

```
int i = 123;
Class<?> c = i.getClass( ); //Not OK
Class<?> c = i.class; //OK
Class<?> c = Number.class; //OK
```

  - It is convenient to use .class syntax with primitive type and abstract class.

- Using **Class.forName() method**:
  - If the fully-qualified name of a class is available, it is possible to get the corresponding Class using the static method Class.forName().

```
Class<?> c = Class.forName("org.example.main.Program"); //OK

Class<?> c = Class.forName("[D"); //OK:  same as double[].class

Class<?> c = Class.forName("[[Ljava.lang.String"); //OK:  same as String[][].class
```

- Using **TYPE Field for Primitive Type**:
  - Each of the primitive types and void has a wrapper class in java.lang that is used for boxing of primitive types to reference types. Each wrapper class contains a field named TYPE which is equal to the Class for the primitive type being wrapped.
  - The .class syntax is a more convenient and the preferred way to obtain the Class for a primitive type; however there is another way to acquire the Class.

```
Class<?> c = Double.TYPE;
Class<?> c = Void.TYPE; //OK
Class<?> c = Void.class; //OK
```

Examine Type Metadata

```
import java.lang.reflect.Modifier;

public class Program {
    public static void main(String[] args) {
        Class<?> c = Integer.class;

        String typeName = c.getName(); //Returns the name of the entity represented by this Class object.

        String simpleTypeName = c.getSimpleName(); //Returns the simple name of the underlying class.

        Package pkg = c.getPackage();
        String packageName = pkg.getName(); //Return the name of this package.
        int mod= c.getModifiers();
        String modifiers = Modifier.toString(mod);  //Returns string representation of the set of modifiers.

        Class<?> sc = c.getSuperclass();
        String superClassName = sc.getName();//Returns the name of the super class

        Class<?>[] si = c.getInterfaces();
        StringBuffer sb = new StringBuffer();
        for (Class<?> i : si) {
             sb.append(i.getName());//Returns the name of the super interfaces
        }
    }
}
```

Examine Field Metadata

```
import java.lang.reflect.Field;
import java.lang.reflect.Modifier;

public class Program {
    public static void main(String[] args) {
        Class<?> c = Integer.class;
        Field[] fields = c.getFields(); //Returns an array of public fields that are declared in the class or its
superclasses
        Field[] declaredFields = c.getDeclaredFields(); //Returns an array of all the fields declared in the class
        for (Field field : declaredFields) {
            String modifiers = Modifier.toString(field.getModifiers());
            String typeName = field.getType().getSimpleName();
            String fieldName = field.getName();
            System.out.println( modifiers+" "+typeName+" "+fieldName);
        }
    }
}
```

- getFields() returns an array of public fields that are declared in the class or its superclasses. This includes fields inherited from the superclass or any interface that the class implements. This method does not return any private or protected fields, regardless of whether they are inherited or declared in the class.
- getDeclaredFields() returns an array of all the fields declared in the class. This includes public, private, and protected fields. It does not include any fields inherited from a superclass or an interface.

Examine Method Metadata

```
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;
import java.lang.reflect.Parameter;

public class Program {
    public static void main(String[] args) {
        Class<?> c = Integer.class;
        Method[] methods = c.getMethods();
        Method[] declaredMethods = c.getDeclaredMethods();
        for (Method method : declaredMethods) {
            String modifiers = Modifier.toString( method.getModifiers());
            String returnType = method.getReturnType().getSimpleName();
            String methodName = method.getName();

            StringBuffer paramList = new StringBuffer("( ");
            Parameter[] parameters = method.getParameters();
            for( Parameter parameter : parameters ) {
                String type = parameter.getType().getSimpleName();
                paramList.append(type);
                paramList.append(" ");
                String name = parameter.getName();
                paramList.append(name);
                paramList.append(", ");
            }
            if( parameters.length > 0 )
                paramList.deleteCharAt( paramList.length() - 2 );
            paramList.append(" )");


            StringBuffer exceptionList = new StringBuffer( );
            Class<?>[]  exceptionTypes= method.getExceptionTypes();
            if( exceptionTypes.length > 0 )
                exceptionList.append("throws ");
            for( Class<?> exceptionType : exceptionTypes) {
                exceptionList.append( exceptionType.getSimpleName() );
                exceptionList.append( ", ");
            }
            if( exceptionTypes.length > 0 )
                exceptionList.deleteCharAt( exceptionList.length() - 2 );

            System.out.println( modifiers+" "+ returnType+" "+methodName +""+paramList+""+exceptionList);
        }
    }
}
```

- getMethods(): This method returns an array of Method objects that represent all the public methods of the class (including inherited methods) and the public methods declared in any interfaces implemented by the class.
- getDeclaredMethods(): This method returns an array of Method objects that represent all the methods declared explicitly by the class, including both public and non-public methods. It does not include any inherited methods or the methods declared in any interfaces implemented by the class.

Accessing private fields using Reflection

```
import java.lang.reflect.Field;

class Complex{
    private int real;
    private int imag;
    public Complex() {
        this.real = 10;
        this.imag = 20;
    }
    public int getReal() {
        return this.real;
    }
    public int getImag() {
        return this.imag;
    }
}
public class Program {
    public static void main(String[] args) {
        try {
            Complex complex = new Complex();
            System.out.println("Real Number :   "+complex.getReal());
            System.out.println("Imag Number :   "+complex.getImag());

            Class<?> c = complex.getClass();
            Field field = null;

            field = c.getDeclaredField("real");
            field.setAccessible(true);
            field.setInt(complex, 50);

            field = c.getDeclaredField("imag");
            field.setAccessible(true);
            field.setInt(complex, 60);

            System.out.println("Real Number :   "+complex.getReal());
            System.out.println("Imag Number :   "+complex.getImag());
        } catch (NoSuchFieldException | SecurityException | IllegalArgumentException | IllegalAccessException e) {
            e.printStackTrace();
        }
    }
}
```

Reflection to access and invoke a private constructor of a class.

```
import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;

class Complex {
    private int real;
    private int imag;

    private Complex(int real, int imag) {
        this.real = real;
        this.imag = imag;
    }

    public int getReal() {
        return this.real;
    }

    public int getImag() {
        return this.imag;
    }
}

public class Program {
    public static void main(String[] args) {
        try {
            Class<?> c = Complex.class;
            Constructor<?> constructor = c.getDeclaredConstructor(int.class, int.class);
            constructor.setAccessible(true);
            Complex complex = (Complex) constructor.newInstance(50, 60);
            System.out.println("Real Number :   " + complex.getReal());
            System.out.println("Imag Number :   " + complex.getImag());
        } catch (NoSuchMethodException | SecurityException | InstantiationException | IllegalAccessException
                | IllegalArgumentException | InvocationTargetException e) {
            e.printStackTrace();
        }
    }
}
```

Middleware Application

```
//Calculator.java
public class Calculator {
  public double sum( int num1, float num2, double num3) {
    return num1 + num2 + num3;
  }
  public int sub( int num1, int num2 ) {
    return num1 + num2;
  }
}
```

```
//Convert.java
class Convert{
  public static Object changeType( String type, String value ) {
    switch( type ) {
    case "int":
      return Integer.parseInt(value);
    case "float":
      return Float.parseFloat(value);
    case "double":
      return Double.parseDouble(value);
    }
    return null;
  }
}
```

```
//Program.java
import java.lang.reflect.Method;
import java.lang.reflect.Parameter;
import java.util.Scanner;

public class Program {
  public static void main(String[] args) {
    try( Scanner sc = new Scanner(System.in)){
      System.out.print("Enter F.Q. Class Name :   ");
      String className = sc.nextLine();   //org.example.domain.Calculator
      Class<?> c = Class.forName(className);

      System.out.print("Enter Method Name :   ");
      String methodName = sc.nextLine();  //sum
```

```
        Method[] methods = c.getMethods();
        for (Method method : methods) {
          if( method.getName().equals( methodName ) ) {
            Parameter[] parameters = method.getParameters();
            Object[] arguments = new Object[ method.getParameterCount() ];
            for( int index = 0; index < method.getParameterCount(); ++ index ) {
              String parameterType =  parameters[ index ].getType().getName();
              System.out.print("Enter value for the argument "+parameterType+" Type : ");
              Object argument = Convert.changeType(parameterType, sc.nextLine());
              if( argument != null )
                arguments[ index ] = argument;
            }

            Object result = method.invoke(c.newInstance(), arguments );
            System.out.println("Result  :   "+result);
            break;
          }
        }
      }catch( Exception ex ) {
        ex.printStackTrace();
      }
    }
  }
```

Advantages of Reflection:

- Dynamic class loading: Reflection allows classes to be loaded and instantiated dynamically, which can be useful when the class to be used is not known at compile time.
- Introspection: Reflection allows the properties and methods of a class to be inspected at runtime. This can be useful for building tools like debuggers or IDEs, where we need to be able to examine the structure of a program while it is running.
- Frameworks and libraries: Many Java frameworks and libraries make use of reflection to provide powerful features like dependency injection, ORM, and serialization.
- Testing: Reflection can be useful for testing by allowing access to private methods and fields, and for mocking objects.

Drawbacks of Reflection:

- Performance overhead: Reflection operations are slower than direct method calls because they involve additional runtime checks and lookups.
- Security risks: Reflection can be used to bypass access controls and security measures, allowing malicious code to access private fields and methods or modify the behavior of a program in unexpected ways.
- Complexity and readability: Reflection code can be harder to read and understand, especially for developers who are not familiar with the language's reflection APIs.

Proxy Design Pattern(Assignment)

- The official Java documentation on the Proxy pattern:
  - https://blogs.oracle.com/javamagazine/post/the-proxy-pattern
- Reference: The Gang of Four (GoF) book: The GoF book "Design Patterns: Elements of Reusable Object-Oriented Software".