The Reflection Pattern. Reflective Programming Languages

The Reflection Pattern

- Bibliography:
  - The architectural *Reflection* pattern
    - [POSA1]
  - Reflection in Java:
    - https://dev.java/learn/reflection/
  - Reflection in .Net:
    - https://learn.microsoft.com/en-us/dotnet/fundamentals/reflection/overview

Reflection

*The **Reflection architectural pattern** provides a mechanism for dynamically changing structure and behavior of software systems.*

*In this pattern, an application is split into two parts. **A meta level** provides information about selected system properties and makes the software **self-aware**. A **base level** includes the application logic. Its implementation builds on the meta level. Changes to information kept in the meta level affect subsequent base-level behavior.*

[POSA1]

Meta

- The word “Meta”:
  - Greek: a preposition; "beside“, “beyond"
  - *Used as: About itself* or *self-referential (something that* refers to or analyzes its own structure or category)
    - Metadata: data that defines and describes the characteristics of other data
      - Example: an image stored in a file. Metadata = the header of the file
    - Metamodel: a model of a model
      - A model describes something specific (example: a diagram of a software system).
      - A metamodel describes how those models are constructed, what elements they can contain and how those elements relate.

Reflection – The principle

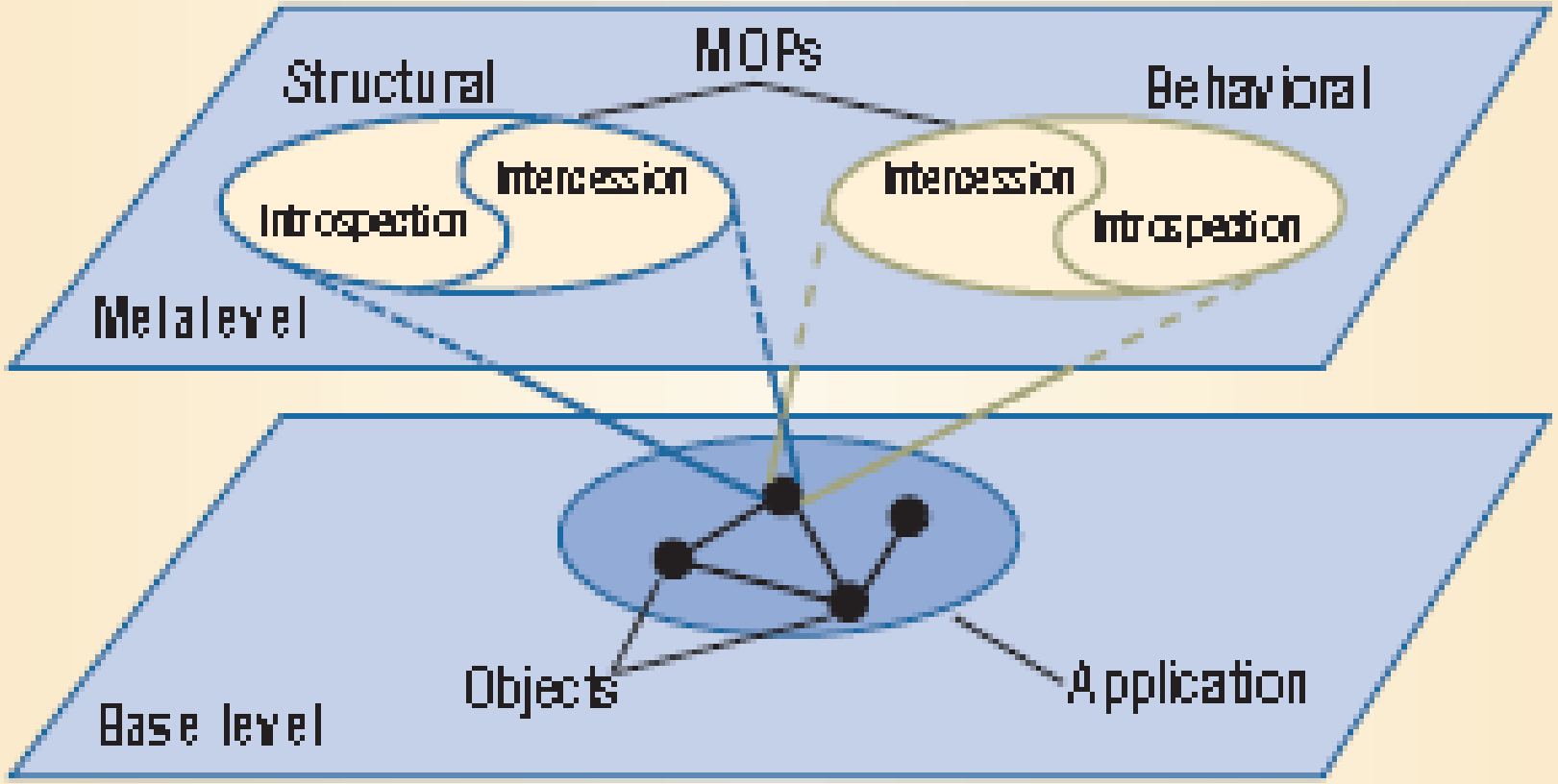
- A precondition for doing changes is to be aware of the current condition => a “mirror” is needed
- A self-aware software system is built on 2 levels:
- **Meta level = the “mirror” reflecting the Base level**
- **Base level**

- There is a protocol that governs the interaction with the MetaLevel: The **Meta Object Protocol**
  - Establishes limits of what manipulation of the meta-level is allowed from the base level
    - an ordinary mirror vs a magic mirror vs how much magic?

Kinds of Reflection

- A taxonomy, according to:
  - How much can be done via reflection:
    - Introspection
    - Intercession/Manipulation/Adaptation
  - What it reflects:
    - Structural reflection
    - Behavioural reflection
- Reflective towers: meta-meta-meta-….

Reflection Taxonomy



Meta level

- **Meta level:**
  - provides a self-representation of the software to give it knowledge of its own structure and behavior
  - It consists of ***metaobjects.*** Metaobjects encapsulate and represent information about the software.

Base level

- **Base level:**
  - The base level defines the application logic.
  - Its implementation uses the metaobjects to remain independent of those aspects that are likely to change.
    - Base-level components may only communicate with each other via a metaobject that implements a specific interaction mechanism. Changing this metaobject changes the way in which base-level components communicate, but without modifying the base-level code.

Meta Object Protocol

- **Meta Object Protocol (MOP)**:
  - An interface for manipulating the metaobjects.
  - It allows clients to request particular changes on the metaobjects.
  - The metaobject protocol itself is responsible for checking the correctness of the change specification, and for performing the change. Every manipulation of metaobjects through the metaobject protocol affects subsequent base-level behavior

Usage of the Reflection Pattern

- Created as an architectural solution for (self)-adaptive systems
- Historically, a buzzword around 2000:
  - Reflective operating systems
  - Reflective middleware
- Biggest application and the most significant one today: Reflective programming languages
- The Implementation of the Reflection feature inside different programming languages follows this architectural pattern

Importance of Reflective Programming Languages

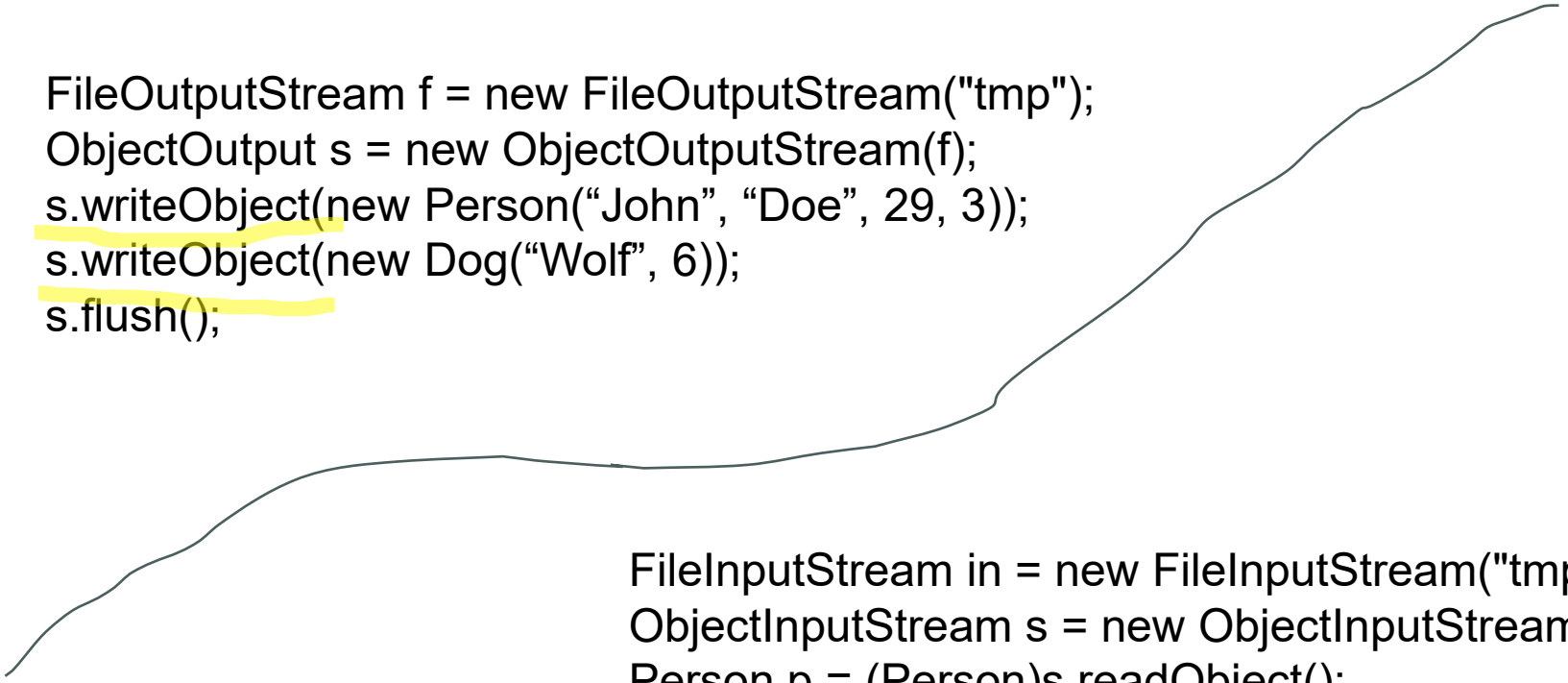
- The reflective features of programming languages are a key enabling technology for many frameworks and tools. They make possible:
  - Object serialization
  - Event systems (e.g., the Greenrobot EventBus)
  - Factory design pattern
  - Stub/skeleton generation for remote method invocation (e.g., Java RMI, WCF)
  - Testing frameworks (e.g., JUnit)
  - Dependency injection frameworks (e.g., Spring)
  - Object-relational mapping (ORM) frameworks

Motivating Example: Object Serialization

- The **Concept of Serialization**: **converting** an object (data + structure in memory) into a format that can be **stored or transmitted**, and later **reconstructed** into an object.
- In principle, a Serializer must offer following operations:
  - WriteObject(Object, file) : saves the values of all attributes of the object
  - ReadObject(file) -> Object : creates object and restores values of all attributes
- Challenge: A Serializer must be able to handle different custom object types, including types that were not known at the time when the Serializer was implemented.
  - The Serializer must be able to “adapt” to unanticipated situations
- **Serialization** is a feature present in many programming languages/programming platforms, such as:
  - Java: https://docs.oracle.com/en/java/javase/20/docs/specs/serialization/index.html
  - .NET: https://learn.microsoft.com/en-us/dotnet/standard/serialization/
  - Python: pickle, marshal, JSON

Example: Java Serialization

```
FileOutputStream f = new FileOutputStream("tmp");
ObjectOutput s = new ObjectOutputStream(f);
s.writeObject(new Person(“John”, “Doe”, 29, 3));
s.writeObject(new Dog(“Wolf”, 6));
s.flush();
```



```
FileInputStream in = new FileInputStream("tmp");
ObjectInputStream s = new ObjectInputStream(in);
Person p = (Person)s.readObject();
Dog d = (Dog)s.readObject();
```

Solving the Serialization Problem with Reflection

- The Serializer must be independent of specific type structures
- Solution with Reflection:
  - Base level: user application (contains objects to be serialized)
  - Meta level: metaobjects that reflect base level objects
- The metaobjects in Java have been designed starting from what was needed in order to be able to implement Serialization and Remote Invocation

Example: Using Reflection to implement Serialization

```
serialize(object):
    if object == null:
        write NULL marker
        return

    typeInfo = object.getClass()

    write typeInfo.getName()

    dataInfoList= typeInfo.getDeclaredFields()

    for each field in dataInfoList:
        value = field.get(object)
        if field is primitive:
            write value
        else:
            serialize(value)    // recursive
```

Example: Using Reflection to implement Deserialization

```
deserialize():

    if read NULL marker:  return null

    read class name
    typeInfo = Class.forName(name)

    create object = typeInfo.newInstance()

    dataInfoList = typeInfo.getDeclaredFields()
    for each field in dataInfoList:
            if field is primitive:
            fieldvalue = read value
        else:
            fieldvalue = deserialize()
        field.set(object, fieldvalue)

    return object
```

Reflection as a feature of a Programming Language

- Reflection (in a programming language):
  - the ability of a program to examine and control its own implementation
  - the process by which a program can observe and modify its own structure and behavior at runtime.
- Reflective programming
  - the programming paradigm driven by reflection
  - is a special case of metaprogramming

RTTI (Run-Time Type Identification)

- Reflection is based on RTTI (Run-Time Type Identification):
  - RTTI: allows programs to discover at runtime and use at runtime types that were not known at their compile time
  - Non-RTTI / Traditional approaches:
    - assume all types are known at compile time
    - Polymorphism in OO languages: is a particular case of very limited RTTI

Kinds of tasks specific to Reflection

- ***Inspection (introspection)****: analyzing objects and types to gather information about their definition and behavior*.
  - Find the run-time type information of an object
  - Find information about a type (supertypes, interfaces, members)
    - Dynamic type discovery
- ***Manipulation (intercession)****: uses the information gained through inspection to change the structure/behavior:*
  - create new instances of new types discovered at runtime
  - dynamically invoke discovered methods
    - Late binding: the types and methods used by a program are not known at compile-time
  - The most one could imagine to do in a reflective language: *restructure types and objects on the fly* !

Case Studies for Reflective Languages

- Java:
  - https://dev.java/learn/reflection/
- .NET:
  - https://learn.microsoft.com/en-us/dotnet/fundamentals/reflection/overview

- Reflective capabilities need special support in language compiler and virtual machine
  - Java: java.lang.reflection
  - .NET: System.Reflection

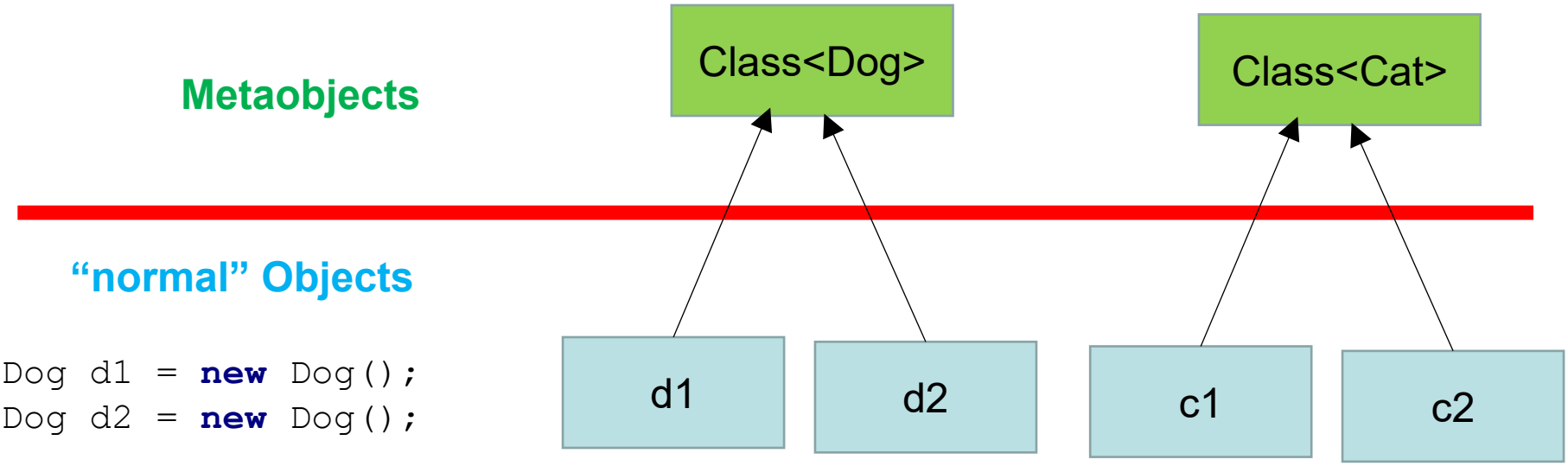
Reflection case study: Reflection in Java

https://dev.java/learn/reflection/

Reflection Entry Point: The Class Class

- `java.lang.reflect`
- Class `java.lang.Class<T>`
  - Class<T> is a generic class in Java that represents metadata about a class or interface at runtime
  - It is ***a Meta-class, its instances are called meta-objects.***
  - It is the entry point for all of the Reflection API
  - You cannot instantiate Class manually (no public constructor).The JVM automatically creates Class objects when types (class, interface, enum, array, primitive) are loaded.
  - Immutable: once created, a Class object cannot be modified
  - One instance per type: for each loaded type, there is exactly one Class object in memory
  - It is a Generic Type Class<T>: T represents the actual type the Class object refers to. If this is unknown, use wildcard Class<?>

Objects and metaobjects

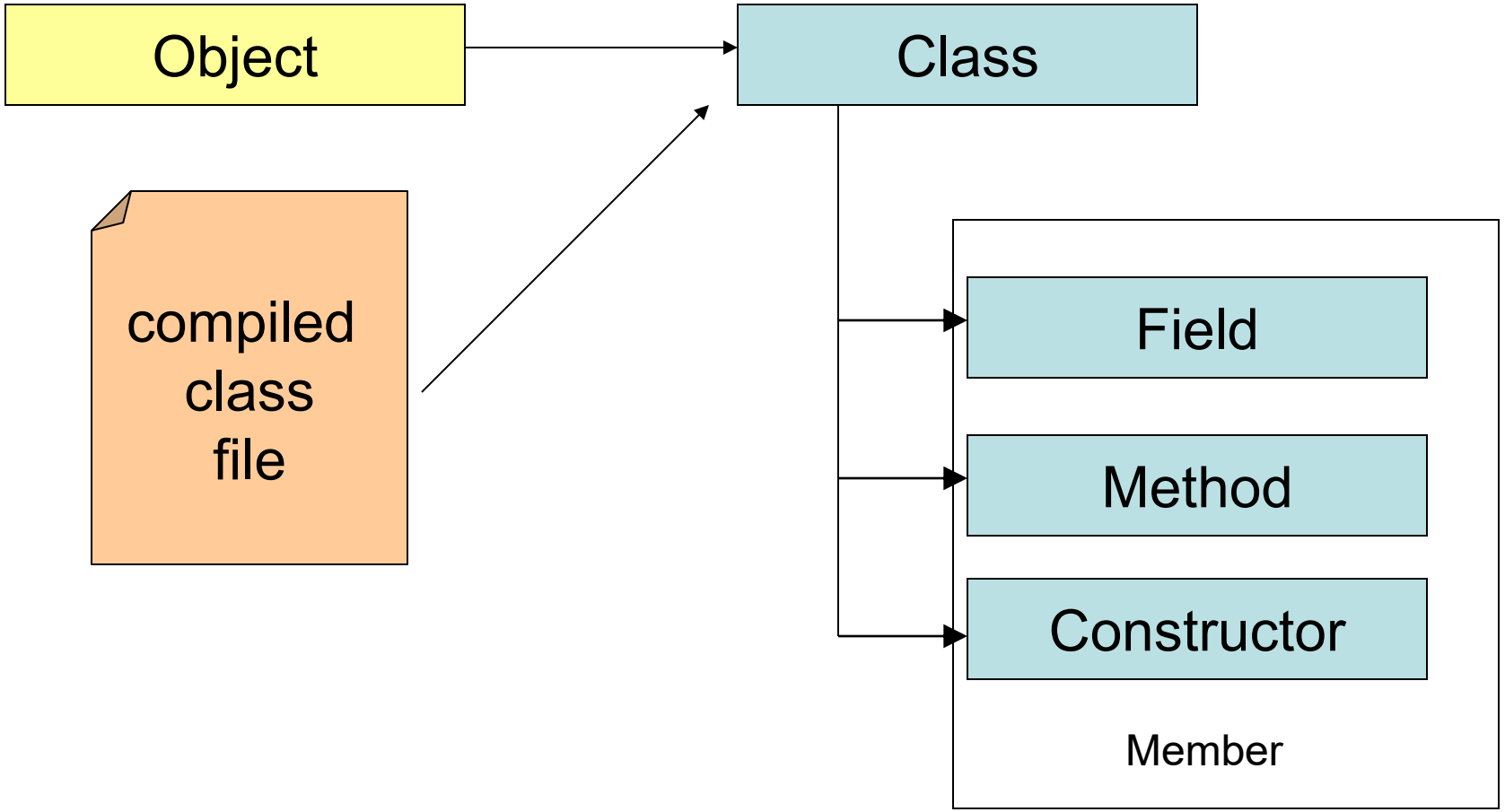



```
Dog d1 = new Dog();
Dog d2 = new Dog();

Cat c1 = new Cat();
Cat c2 = new Cat();
```

The Reflection Logical Flow in Java

Observation: arrows show the *logical flow* ("***obtains from***")



Retrieving a Class object

- Obtaining a Class object is the first step in operations involving Reflection
- Ways to obtain a Class object:
  1. From an available object instance
  2. From an available type
  3. From compiled .class file in a location on classpath
  4. From compiled .class file in a location at a random URL

Retrieving a Class object (1)

- **If there is an object (an instance) of this class available:**

- **Object.getClass():** If an instance of an object is available, then the simplest way to get its Class is to invoke o**`bject.getClass()`**

```
Rectangle r;
…
Class<?> c = r.getClass();
…

Class<?> c= "foo".getClass();
```

Retrieving a Class object (2)

- **If the type is available but there is no instance:**

- **`.class`**: If the 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. This is also the easiest way to obtain the Class for a primitive type.

```
boolean b;
Class<?> c = b.getClass(); // compile-time error
Class<?> c = boolean.class; // correct
```

Retrieving a Class object (3)

- **If the class is available as compiled code in a file on the classpath:**

- **`Class.forName()`:** If the fully-qualified name of a class is available, and *the file containing the compiled code is correctly put on the runtime classpath*, it is possible to get the corresponding Class using the static method Class.forName()

```
Class<?> cString = Class.forName("java.lang.String;");
```

Retrieving a Class object (4)

- **If the class is available as compiled code somewhere at a URL:**
- Each class object is constructed from bytecode by a java classloader
- The default class loader loads from the local file system only, directed by the classpath variable
- Other class loaders may load differently or from other locations. URLClassLoader loads from a URL, that may point to a directory or jar

```
URL[] urls = new URL[]{
             URI.create("file:///home/pack.jar").toURL()};

URLClassLoader cl = new URLClassLoader(urls);
Class<?> c = cl.loadClass("mypack.MyClass");
```

Inspecting a Class

- After we obtain a Class object `myClass`, we can:
- Get the class name

```
String s = myClass.getName() ;
```

- Get the class modifiers

```
int m = myClass.getModifiers() ;
bool isPublic = Modifier.isPublic(m) ;
bool isAbstract = Modifier.isAbstract(m) ;
bool isFinal = Modifier.isFinal(m) ;
```

- Test if it is an interface

```
bool isInterface = myClass.isInterface() ;
```

- Get the interfaces implemented by a class

```
Class<?> [] itfs = myClass.getInterfaces() ;
```

- Get the superclass

```
Class<?> super = myClass.getSuperClass() ;
```

Discovering Class members

- fields, methods, and constructors
- java.lang.reflect.* :
  - Member interface
  - Field class
  - Method class
  - Constructor class

Class Methods for Locating Members

Member	Class API	List of members?	Inherited members ?	Private members ?
Field	getDeclaredField()	no	no	yes
	getField()	no	yes	no
	getDeclaredFields()	yes	no	yes
	getFields()	yes	yes	no
Method	getDeclaredMethod()	no	no	yes
	getMethod()	no	yes	no
	getDeclaredMethods()	yes	no	yes
	getMethods()	yes	yes	no
Constructor	getDeclaredConstructor()	no	N/A[1]	yes
	getConstructor()	no	N/A[1]	no
	getDeclaredConstructors()	yes	N/A[1]	yes
	getConstructors()	yes	N/A[1]	no

Working with Class members

- Members: fields, methods, and constructors
- For each member, the reflection API provides support to retrieve declaration and type information, and operations unique to the member (for example, setting the value of a field or invoking a method),
- java.lang.reflect.* :
  - “Member” interface
  - “Field” class: Fields have a **type** and a **value**. The java.lang.reflect.Field class provides methods for accessing type information and **setting and getting values** of a field on a given object.
  - “Method” class: Methods have **return values**, **parameters** and may throw **exceptions**. The java.lang.reflect.Method class provides methods for accessing type information for return type and parameters and **invoking** the method on a given object.
  - “Constructor” class: The Reflection APIs for constructors are defined in java.lang.reflect.Constructor and are similar to those for methods, with two major exceptions: first, constructors have no return values; second, the invocation of a constructor creates a new instance of an object for a given class.

Example: retrieving **public** fields

```
Class<?> c = Class.forName(“Dtest");

// get all public fields
Field[] publicFields = c.getFields();
for (int i = 0; i < publicFields.length; ++i) {
        String fieldName = publicFields[i].getName();
        Class <?> typeClass = publicFields[i].getType();
        System.out.println("Field: " + fieldName + " of type " +
                                                    typeClass.getName());
        }
```

Example. Classes Btest and Dtest will be used as data

```
public class Btest
{
    public String aPublicString;
    private String aPrivateString;
    public Btest(String aString) {
        // …
    }

    public Btest(String s1, String s2) {
        // …
    }
    private void Op1(String s) {
        // …
    }
    protected String Op2(int x) {
        // …
    }
    public void Op3()          {
        // …
    }
}
```

```
public class Dtest extends Btest
{
    public int aPublicInt;
    private int aPrivateInt;
    public Dtest(int x)
    {
        // …
    }

    private void OpD1(String s) {
        // …
    }

    public String OpD2(int x){
        // …
    }


}
```

Example: retrieving **public** fields

```
Class c = Class.forName(“Dtest");

// get all public fields
Field[] publicFields = c.getFields();
for (int i = 0; i < publicFields.length; ++i) {
        String fieldName = publicFields[i].getName();
        Class<?> typeClass = publicFields[i].getType();
        System.out.println("Field: " + fieldName + " of type " +
                                                    typeClass.getName());
        }
```

```
Field: aPublicInt of type int
Field: aPublicString of type java.lang.String
```

Example: retrieving **declared** fields

```
Class c = Class.forName(“Dtest");

// get all declared fields
Field[] publicFields = c.getDeclaredFields();
for (int i = 0; i < publicFields.length; ++i) {
        String fieldName = publicFields[i].getName();
        Class<?> typeClass = publicFields[i].getType();
        System.out.println("Field: " + fieldName + " of type " +
                                                    typeClass.getName());
        }
```

```
Field: aPublicInt of type int
Field: aPrivateInt of type int
```

Example: retrieving **public** constructors

```
// get all public constructors

Constructor[] ctors = c.getConstructors();
for (int i = 0; i < ctors.length; ++i) {
    System.out.print("Constructor (");
    Class<?>[] params = ctors[i].getParameterTypes();
    for (int k = 0; k < params.length; ++k)
            {
            String paramType = params[k].getName();
            System.out.print(paramType + " ");
            }
    System.out.println(")");
    }
```

```
Constructor (int )
```

Example: retrieving **public** methods

```
//get all public methods

Method[] ms = c.getMethods();
for (int i = 0; i < ms.length; ++i)        {
    String mname = ms[i].getName();
    Class retType = ms[i].getReturnType();
    System.out.print("Method : " + mname + " returns " + retType.getName() + "
    parameters
    Class<?>[] p
    for (int k = 0
            {
            String
            Systen
            }
    System.out.
    }
```

```
Method : OpD2 returns java.lang.String parameters : ( int )
Method : Op3 returns void parameters : ( )
Method : wait returns void parameters : ( )
Method : wait returns void parameters : ( long int )
Method : wait returns void parameters : ( long )
Method : hashCode returns int parameters : ( )
Method : getClass returns java.lang.Class parameters : ( )
Method : equals returns boolean parameters : ( java.lang.Object )
Method : toString returns java.lang.String parameters : ( )
Method : notify returns void parameters : ( )
Method : notifyAll returns void parameters : ( )
```

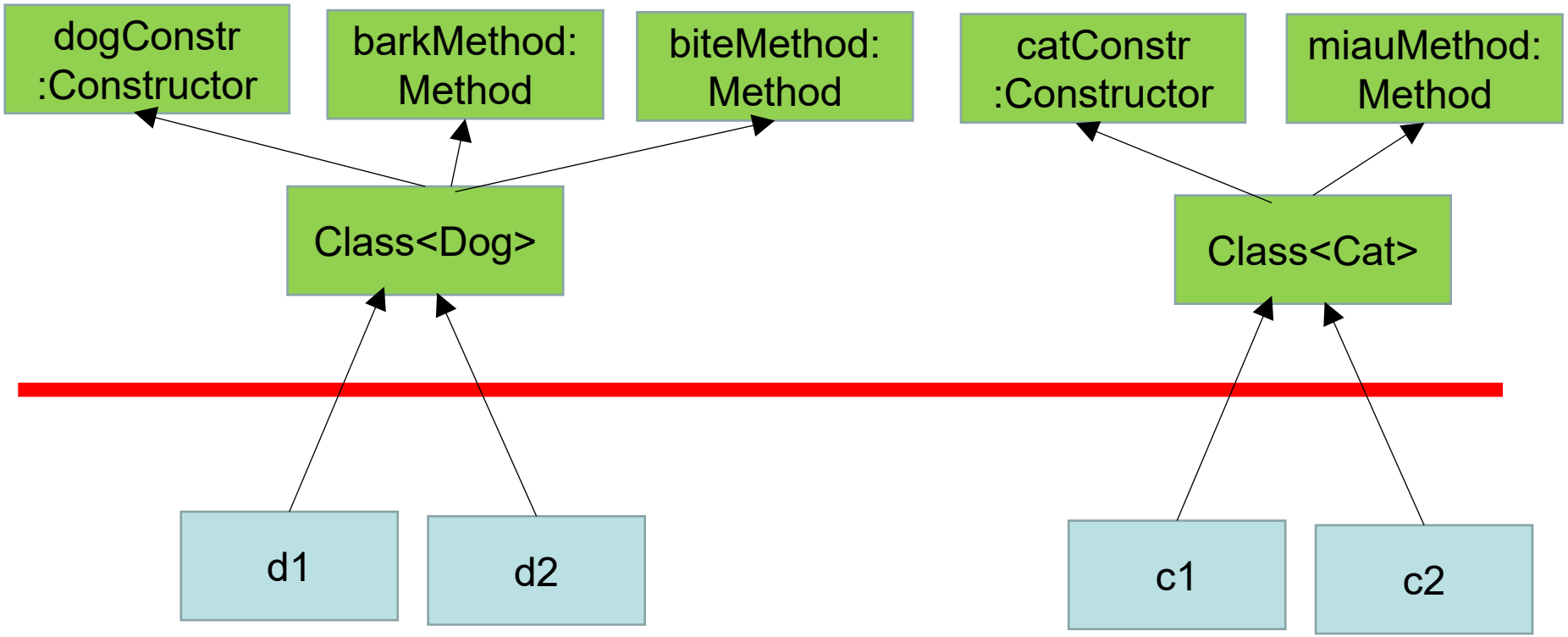
Example: retrieving **declared** methods

```
//get all declared methods

Method[] ms = c.getDeclaredMethods();
for (int i = 0; i < ms.length; ++i)        {
    String mname = ms[i].getName();
    Class retType = ms[i].getReturnType();
    System.out.print("Method : " + mname + " returns " + retType.getName() + "
    parameters : ( ");
    Class[] params = ms[i].getParameterTypes();
    for (int k = 0; k < params.length; ++k)
            {
            String paramType = params[k].getName();
            System.out.print(paramType + " ");
            }
    System.out.println(") ");
    }
```

```
Method : OpD1 returns void parameters : ( java.lang.String )
Method : OpD2 returns java.lang.String parameters : ( int )
```

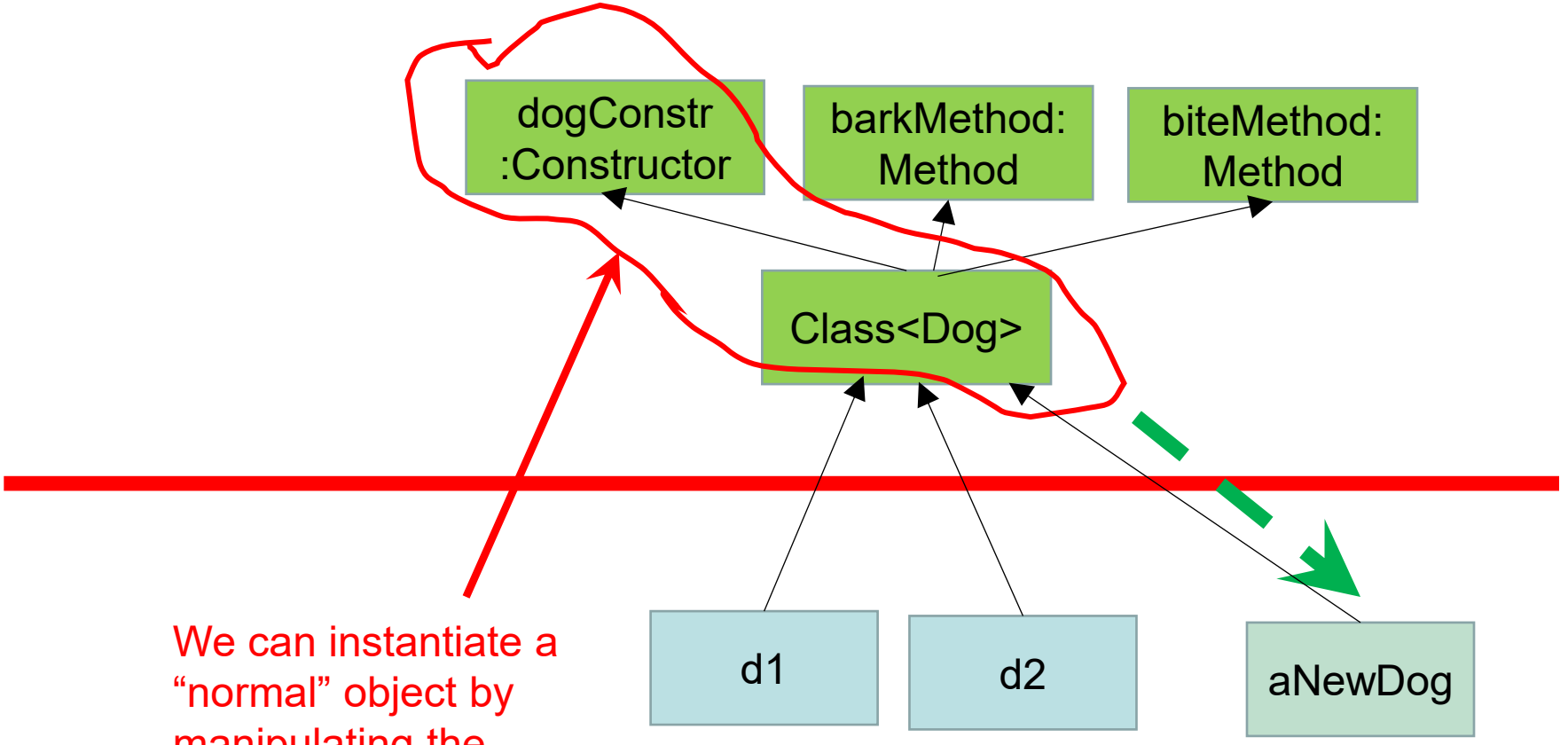
Objects and metaobjects



Using Reflection for Program Manipulation

- Previous examples used Reflection for Introspection only
- Reflection is a powerful tool to:
  - Creating new objects of a type that was not known at compile time
  - Accessing members (accessing fields or invoking methods) that are not known at compile time

Example: Using Reflection for Program Manipulation

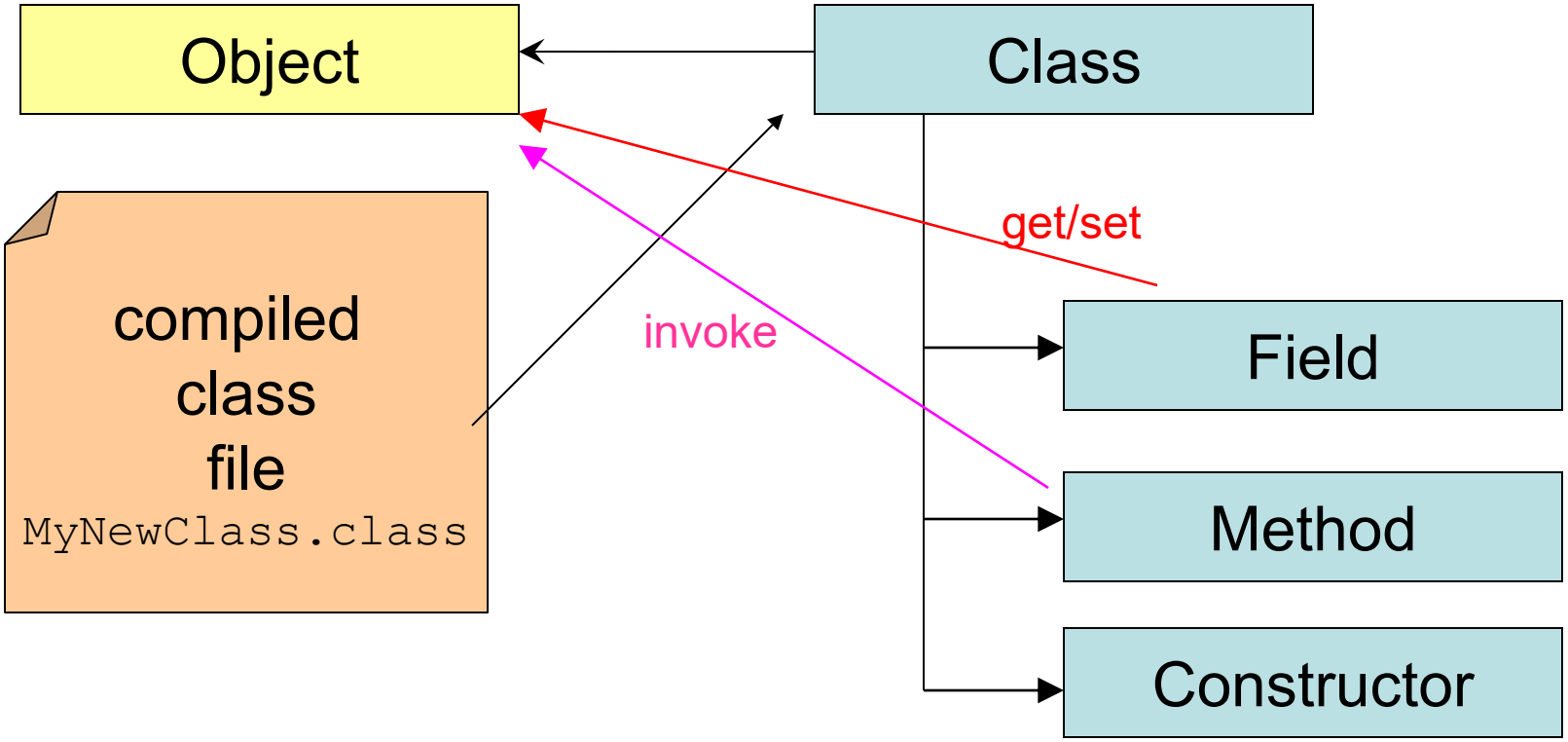


We can instantiate a "normal" object by manipulating the corresponding metaobjects

Object aNewDog of type Dog is created, but **NOT** by explicitly instantiating it with

~~Dog aNewDog = new Dog();~~

Using Reflection for Program Manipulation



Creating new objects

- Using Default Constructors
  - java.lang.reflect.Class.newInstance()

```
Class<?> c = Class.forName("java.awt.Rectangle") ;
Object r = c.newInstance() ; // r is a Rectangle!
```

- Using Constructors with Arguments
  - java.lang.reflect.Constructor.newInstance(Object... initargs)

```
Class<?> c = Class.forName("java.awt.Rectangle") ;
Class<?>[] intArgsClass = new Class<?>[]{ int.class,
  int.class } ;
Constructor ctor = c.getConstructor(intArgsClass) ;

Object[] intArgs = new Object[]{new Integer(12),new
  Integer(24)} ;
Object r = ctor.newInstance(intArgs) ;
```

Example

```
String className = "java.lang.String";

Class<?> c;

try {
 c = Class.forName(className);
 Class<?>[] stringArgsClass = new Class<?>[] { String.class };
 Constructor<?> ctor = c.getConstructor(stringArgsClass);

 Object[] stringArgs = new Object[] { new String("abc") };
 Object something = ctor.newInstance(stringArgs);

 System.out.println(something.getClass().getName());
 System.out.println(something);

 } catch (Exception e) {
       e.printStackTrace();
```

Example: Implementing Factory Pattern. Without Reflection

```
class ShapeFactory {

    public static Shape createShape(String type) {
        if (type == null) return null;

        switch (type.toLowerCase()) {
            case "circle":
                return new Circle();
            case "rectangle":
                return new Rectangle();
            case "triangle":
                return new Triangle();
            default:
                throw new IllegalArgumentException("Unknown shape");
        }
    }
}
```

Example: Implementing Factory Pattern. With Reflection

```
class ShapeFactory {

    public static Shape createShape(String className) {
        try {
            Class<?> clazz = Class.forName(className);
            return (Shape) clazz.getDeclaredConstructor().newInstance();
        } catch (Exception e) {
            throw new RuntimeException("Cannot create shape: " +
                                                    className, e);
        }
    }
}
```

Accessing fields

- Getting Field Values

```
Rectangle r = new Rectangle(12,24) ;
Class<?> c = r.getClass() ;
Field f = c.getField("height") ;
Integer h = (Integer) f.get(r) ;
// equivalent with: h=r.height;
```

- Setting Field Values

```
Rectangle r = new Rectangle(12,24) ;
Class<?> c = r.getClass() ;
Field f = c.getField("width") ;
f.set(r,new Integer(30)) ;
// equivalent with:  r.width=30
```

Invoking methods

```
String s1 = “Hello ” ;
String s2 = “World” ;

Class<?> c = String.class ;
Class<?>[] paramtypes = new Class[] { String.class } ;
Method concatMethod = c.getMethod(“concat”,paramtypes) ;

Object[] args = new Object[] { s2 } ;
String result = (String) concatMethod.invoke(s1,args) ;
// equivalent with result=s1.concat(s2);
```

```
String className = "java.lang.String";
String methodName="length";

Class<?> c;

c = Class.forName(className);

Class<?>[] stringArgsClass = new Class[] { String.class };
Constructor<?> ctor = c.getConstructor(stringArgsClass);

Object[] stringArgs = new Object[] { new String("abc") };
Object something = ctor.newInstance(stringArgs);
System.out.println(something.getClass().getName());

Class<?>[] paramtypes = new Class[] {};
Method lengthMethod = c.getMethod(methodName, paramtypes);

Object[] argsM = new Object[] {};
Object result= lengthMethod.invoke(something, argsM);
System.out.println(result.getClass().getName());
```

Source Code

- Introspection example

  ReflectionDemo.java

- Dynamic instantiation and dynamic method invocation:

  DynamicDemo.java

Example

- User interface implementation
- Contains several visual components (classes) from different sources
- All these components understand a method setColor(Color aColor) but the components share no common supertype of interface;
- Problem: how to invoke this method?

Color Example without Reflection

```
    Object[] components = new Object[10];
    Color color = new Color(0, 128, 255);

    components[0] = new Component1();
    components[1] = new Component2();
    // Add more components as needed

    // Apply color
    for (int i = 0; i < components.length; i++) {
        if (components[i] != null) {
            if (components[i] instanceof Component1) {
                ((Component1) components[i]).setColor(color);
            } else if (components[i] instanceof Component2) {
                ((Component2) components[i]).setColor(color);
            }
            // Add more else-if branches for more comp types
        }
    }
}
```

Color Example with Reflection

```
    Object[] components = new Object[10];
    Color color = new Color(0, 128, 255);

    components[0] = new Component1();
    components[1] = new Component2();
    // Add more components as needed

    // Apply color dynamically
    for (int i = 0; i < components.length; i++) {
        if (components[i] != null) {
            try {
                Method method =
components[i].getClass().getMethod("setColor", Color.class);
                method.invoke(components[i], color);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

Accessible Objects

- Can request that Field, Method, and Constructor objects be “accessible.”
  - Request granted if no security manager, or if the existing security manager allows it
- Can invoke method or access field, even if inaccessible via privacy rules ! -> this is needed i.e. for Serialization
- AccesibleObject Class: the Superclass of Field, Method, and Constructor
- boolean isAccessible( )
  - Gets the value of the accessible flag for this object
- void setAccessible(boolean flag)
  - Sets the accessible flag for this object to the indicated boolean value

Arrays and Reflection

- Arrays have a component type (which is part of the type) and a length (which is not part of the type)
- Identifying Array types: Class.isArray()
- Class.getComponentType(): if the class is an Array, returns the Class object describing the type of array elements, otherwise returns null
- Array.newInstance(Class<?> compType, int size)
- Array.getLength(Object array)
- Array.set(Object array, int index, Object value)
- Array.get(Object array, int index)

Generics and Reflection

In Generics, the type of information is only available at compile-time. Java compiler erases type information during compilation and it's not available at runtime (***type erasure***)

```
List<String> l = new LinkedList<>();
Class<?> cl = l.getClass();
System.out.println("class is "+cl.getName());
System.out.println(cl.getTypeParameters().length);
System.out.println(cl.getTypeParameters()[0]);
```

```
class is java.util.LinkedList
1
E
```

getTypeParameters gets the formal type parameter, not the actual parameter!!

Generics and Reflection (2)

- Still it **is** possible to access generics information at runtime in certain cases:
  - It is possible to access actual type parameters for the generic types of public fields
    - Field.GetGenericType()
  - It is possible to access actual type parameters for generic types that are parameters or return types of methods:
    - Method.getGenericReturnType()
    - Method.getGenericParameterTypes()
  - The getGeneric…() methods return Type objects. If the Type object is actually an instanceof ParameterizedType, then call ParametrizedType.getActualTypeArguments() to get the actual type parameters
  - Type and ParametrizedType have been added later to the Java reflection API (after adding generics)

Generics and Reflection (3)

```
class Subject{
 List<Observer> observers;
 …
}

…

Field field = Class.forName("Subject").getField("observer");

Type genericFieldType = field.getGenericType();

if(genericFieldType instanceof ParameterizedType){
    ParameterizedType aType = (ParameterizedType) genericFieldType;
    Type[] fieldArgTypes = aType.getActualTypeArguments();
    for(Type fieldArgType : fieldArgTypes){
        Class fieldArgClass = (Class) fieldArgType;
        System.out.println("fieldArgClass = " + fieldArgClass);
    }
}
```

Annotations and Reflection

- Annotations are a kind of comment or meta data in Java code.
- https://dev.java/learn/annotations/
- Annotations can be placed over classes, methods or fields; they start with character @
- Examples of built-in annotations: `@Override`, `@Deprecated` …
- Example in Greenrobot: `@Subscribe`
- The annotation can include *elements*, which can be named or unnamed, and there are values for those elements:
  `@Subscribe(threadMode = ThreadMode.POSTING)`

- Annotations can be processed at compile time by pre-compiler tools, or at runtime via Java Reflection (if their lifetime has been defined so)

Annotations and Reflection

- You can get or check the annotations of a class, method or field at runtime (if their lifetime has been defined so)
- java.lang.Annotation

```
Class c =  …
Method m = …
Annotation[] annotations = c.getAnnotations();
Annotation[] annotations = m.getAnnotations();
if (m.isAnnotationPresent(Override.class)) …
```

Examples: Inspecting Annotations

- Greenrobot: uses reflection to inspect the class of a subscriber object. Then it finds declared methods and searches for methods with the annotation @Subscribe.
- register(Object obj):

```
->  Obj.getClass()  -> class.getDeclaredMethods() ->
method.isAnnotationPresent(Subscribe.class)
```

- Junit: uses reflection to look through your classes for methods tagged with the @Test annotation, and will then call them when running the unit test.

Defining Custom Annotations

- Custom annotations: defined just like a Java interface
- The annotation type definition looks similar to an interface definition where the keyword interface is preceded by @

- There are Annotations that apply to other Annotations (meta-annotations).
- @Retention annotation specifies how the marked annotation is stored (their lifetime) : RetentionPolicy.SOURCE , RetentionPolicy.CLASS, RetentionPolicy.RUNTIME
- @Target annotation marks another annotation to restrict what kind of Java elements the annotation can be applied to: ElementType.METHOD , ElementType.TYPE

Example: Defining a Custom Annotation

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation {
}

class MyClass{
   …
   @MyAnnotation
   public void aMethod(){…
   }
```

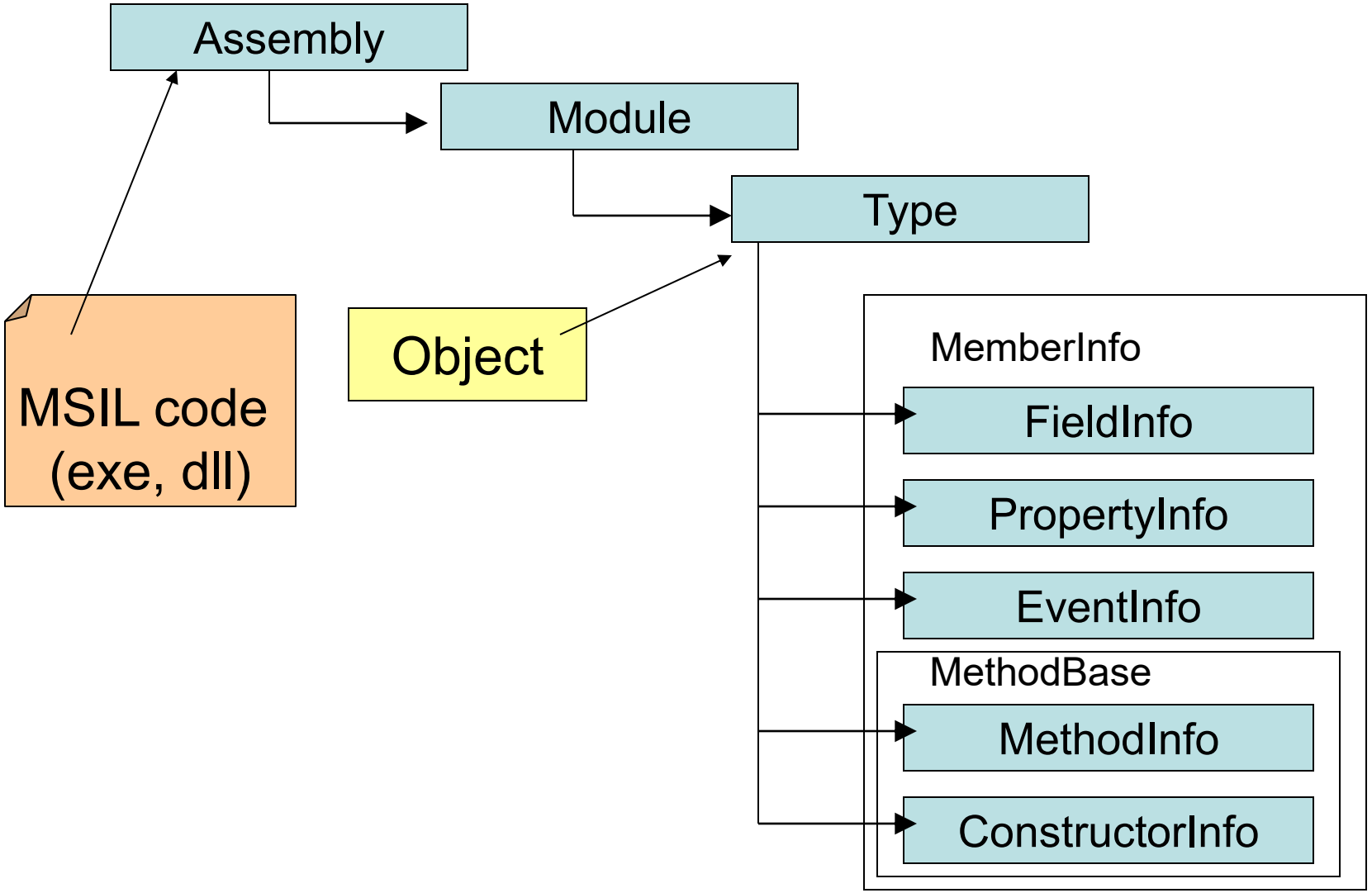
Reflection case study: Reflection in .NET

https://learn.microsoft.com/en-us/dotnet/fundamentals/reflection/overview

Another Reflection case study: Reflection in .NET

- `System.Reflection`
- What can you do with the System.Reflection API:
  - Enumerate modules and types of an assembly;
  - For each type, obtain its base type, implemented interfaces, fields, methods, properties, events
  - Create instances of types, dynamically invoke methods

The Reflection Logical Hierarchy in .NET



Example (C#) : Introspection

```
Assembly a = Assembly.LoadFile(args[0]);
// Find Modules
foreach (Module m in assem.GetModules())   {
            Console.WriteLine(1, "Module: {0}", m);
            // Find Types
            foreach (Type t in m.GetTypes())   {
                Console.WriteLine(2, "Type: {0}", t);
                // Find Members
                foreach (MemberInfo mi in t.GetMembers())
                    Console.WriteLine(3, "{0}: {1}", mi.MemberType, mi);
            }
}
```

Example (C#) : Introspection

```
Assembly a = Assembly.LoadFile(args[0]);
// Find Modules
foreach (Module m in assem.GetModules())   {
            Console.WriteLine(1, "Module: {0}", m);
            // Find Types
            foreach (Type t in m.GetTypes())   {
                Console.WriteLine(2, "Type: {0}", t);
                // Find Members
                foreach (MemberInfo mi in t.GetMembers())
                    Console.WriteLine(3, "{0}: {1}", mi.MemberType, mi);
            }
}
```

Example (C#) : Manipulation

```
Type type = Type.GetType(“Mynamespace.Class1");
object o = Activator.CreateInstance(type);

MethodInfo m=type.GetMethod(“method1”);
m.Invoke(o, new object[]{});

type.InvokeMember(“method1",
BindingFlags.InvokeMethod, null, o, new object[]{});
```

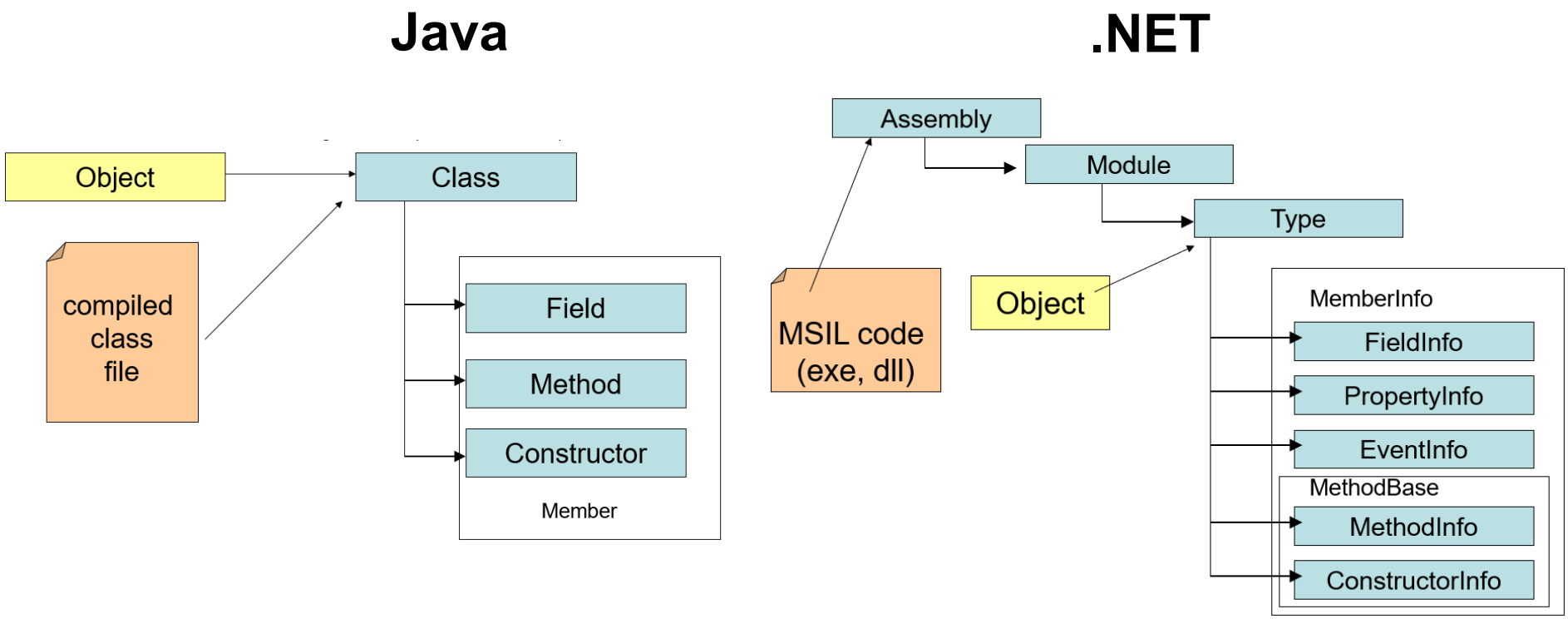
Other Reflection Features

- Arrays: Type.isArray
- Generics: Type.IsGenericType
- Custom Attributes:

```
public class MyAttribute : Attribute
{
    //...
}

public class MyClass
{
   [MyAttribute]
   public v void MyMethod()
    {
        //...
    }
}
```

Reflection Metaobjects



Conclusions

- Reflective capabilities need special support at the levels of language (APIs) and compiler
- Language (API) level:
  - Java: java.lang.reflection
  - .NET: System.Reflection
  - Very similar hierarchy of metaclasses supporting reflection
- Compiler level:
  - The compiler generates metadata describing types (fields, methods, annotations), which is stored in the compiled code.
  - The runtime system (JVM or CLR) uses this metadata to construct metaobjects and provide reflection capabilities.

Uses of Reflection

- Extensibility Features: Plugin & Module Systems
  - Load and integrate external modules or plugins at runtime.
- Framework Development: Widely used in frameworks/libraries for:
  - Serialization
  - Dependency injection: Spring
  - ORM: Hibernate
- Debuggers and Test Tools
  - JUnit

Drawbacks of Reflection

- Performance Overhead
  - Reflective operations are slower because types are resolved dynamically at runtime. This prevents certain optimizations by the JVM, making reflection less efficient than direct (non-reflective) code.
- Security Restrictions
  - Reflection requires special runtime permissions. These may be restricted in secure environments (e.g., when a security manager or similar mechanisms are in place).
- Exposure of Internals
  - Reflection can bypass normal access controls (accessing private fields and methods), which may:
    - Break encapsulation
    - Cause unexpected side effects
    - Reduce portability
    - Lead to issues when the platform or libraries are updated