# We can also use *Object* class

```
public class Test {
    public static void main(String args[]){
        try{
            Constructor<Employee> employeeConstructor = Employee.class.getConstructor();
            Employee employeeInstance = employeeConstructor.newInstance();
        }catch(NoSuchMethodException | InstantiationException |
            IllegalAccessException | InvocationTargetException e){
            System.out.println(e);
        }
    }
}
```

```
public class Test {
    public static void main(String args[]){
        try{
            Constructor<Employee> employeeConstructor = Employee.class.getConstructor();
            Object employeeInstance = employeeConstructor.newInstance();
        }catch(NoSuchMethodException | InstantiationException |
            IllegalAccessException | InvocationTargetException e){
            System.out.println(e);
        }
    }
}
```

*These two versions of code compile and run with the same results*

# Example: creating new instance with constructors that use parameters

```java
public class Test {
    public static void main(String args[]){
        try{
            Constructor<Employee> employeeConstructor = Employee.class.getConstructor(String.class, int.class, String.class);
            Employee employeeInstance = employeeConstructor.newInstance("John Smith", 101, "Large Company");
            System.out.println("My employee: "+employeeInstance.toString());
        }catch(NoSuchMethodException | InstantiationException |
            IllegalAccessException | InvocationTargetException e){
            System.out.println(e);
        }
    }
}
```

this constructor takes String, int, and another String

we pass in parameter values

*Same class definition as in previous example*

```java
import java.lang.reflect.*;

class Employee{
    private String name;
    private int id;
    private String employer;

    public Employee(){
        this.name = "";
        this.id = 0;
        this.employer = "";
    }

    public Employee(String name, int id, String employer){
        this.name = name;
        this.id = id;
        this.employer = employer;
    }

    public String getName(){return this.name;}
    public int getID(){return this.id;}
    public String getEmployer(){return this.employer;}

    public void setName(String name){this.name = name;}
    public void setID(int id){this.id = id;}
    public void setEmployer(String employer){this.employer = employer;}

    public String toString(){
        return this.name + " (" + this.id + ") at "+this.employer;
    }
}
```

Output:

```
My employee: John Smith (101) at Large Company
```

# What if we try to use a constructor that does not exist?

Employee class does not have a constructor that takes in two String objects

```
public class Test {
    public static void main(String args[]){
        try{
            Constructor<Employee> employeeConstructor = Employee.class.getConstructor(String.class, String.class);
            Employee employeeInstance = employeeConstructor.newInstance("John Smith", "Large Company");
            System.out.println("My employee: "+employeeInstance.toString());
        }catch(NoSuchMethodException | InstantiationException |
            IllegalAccessException | InvocationTargetException e){
            System.out.println(e);
        }
    }
}
```

This program compiles successfully but upon execution get NoSuchMethodException at run-time:

```
java.lang.NoSuchMethodException: Employee.<init>(java.lang.String, java.lang.String)
```

# What if the constructor exists but the access modifier is not set to public?

- *Constructor.newInstance()* can access non-public constructors

- Why would you have a private constructor in a class?
  - Delegating constructors
    - Public constructors delegate the work to one or more private constructors
  - Using informative method names for object instantiation
    - Use public methods with descriptive names as instantiators which delegate to private constructors
  - Uninstantiable class
    - Often used for collection of common static methods
  - Singleton anti-pattern
    - Singleton – object of which there will only be a single instance
      - This assumption is included into the design of a singleton class
    - Not a recommended technique as maintaining a singleton assumption is unfeasible in practice

# Example: invoking non-public constructor with *Constructor.newInstance()*

```java
import java.lang.reflect.*;

class Employee{
    private String name;
    private int id;
    private String employer;

    private Employee(){
        this.name = "";
        this.id = 0;
        this.employer = "";
    }

    public Employee(String name, int id, String employer){
        this.name = name;
        this.id = id;
        this.employer = employer;
    }

    public String getName(){return this.name;}
    public int getID(){return this.id;}
    public String getEmployer(){return this.employer;}

    public void setName(String name){this.name = name;}
    public void setID(int id){this.id = id;}
    public void setEmployer(String employer){this.employer = employer;}

    public String toString(){
        return this.name + " (" + this.id + ") at "+this.employer;
    }
}
```

this constructor is private

```java
public class Test {
    public static void main(String args[]){
        try{
            Constructor<Employee> employeeConstructor = Employee.class.getDeclaredConstructor();
            employeeConstructor.setAccessible(true);
            Employee employeeInstance = employeeConstructor.newInstance();
        }catch(NoSuchMethodException | InstantiationException |
            IllegalAccessException | InvocationTargetException e){
            System.out.println(e);
        }
    }
}
```

changing access modifier

instantiating a *Constructor* object for constructor without parameters

If we do not change access modifier we will get IllegalAccesssException at run-time:

```
java.lang.IllegalAccessException: Class Test can not access a member of class Employee with modifiers "private"
```

# Differences between *Class.newInstance()* and *Constructor.newInstance()* methods

- Class.newInstance() can only instantiate an object using a no-argument constructor
  - It cannot invoke constructors with parameters
- Thrown exceptions
  - *Class.newInstance()* throws checked and unchecked exceptions thrown by the no-argument constructor
  - *Constructor.newInstance()* always throws InvocationTargetException
- *Constructor.newInstance()* can access private and protected constructors while *Class.newInstance()* cannot
- *Constructor.newInstance()* is a preferred method of object instantiation

# Additional useful methods to know

- *Class* class
  - Class.getClasses()
  - Class.getDeclaredClasses()
  - Class.getEnclosingClass()
  - Class.cast()

- *Method* class
  - Method.getReturnType()
  - Method.getGenericParameterTypes()
  - Method.isVarArgs()
  - Method.getExceptionTypes()

# Example: Java reflection and Arrays

```java
import java.lang.reflect.*;
import java.util.*;

public class Test {
    public static void main(String args[]){
        // create a new array of strings and set values
        String[] myArray = (String[]) Array.newInstance(String.class, 5);

        Array.set(myArray, 0, "Steve");
        Array.set(myArray, 1, "George");
        Array.set(myArray, 2, "Will");
        Array.set(myArray, 3, "Larry");
        Array.set(myArray, 4, "David");
        System.out.println(Arrays.toString(myArray));

        // update a value of an element
        Array.set(myArray, 3, "Joe");
        System.out.println(Arrays.toString(myArray));

        // retrive an element
        String myElem = (String)Array.get(myArray, 1);
        System.out.println(myElem);

        // get class information for myArray using getClass()
        Class myArrayClass = myArray.getClass();
        System.out.println(myArrayClass.toString());
        Method[] arrayMethods = myArrayClass.getMethods();
        System.out.println(Arrays.toString(arrayMethods));
        Class arrayComponentType = myArrayClass.getComponentType();
        System.out.println(arrayComponentType);

        // get class information for myArray using forName()
        try{
            Class myArrayClass2 = Class.forName("[Ljava.lang.String;");
            System.out.println(myArrayClass2.toString());
            Method[] arrayMethods2 = myArrayClass.getMethods();
            System.out.println(Arrays.toString(arrayMethods2));
            Class arrayComponentType2 = myArrayClass2.getComponentType();
            System.out.println(arrayComponentType2);
        }catch(ClassNotFoundException e){
            System.out.println(e);
        }

    }
}
```

Create new array of strings

Output:

```
[Steve, George, Will, Larry, David]
[Steve, George, Will, Joe, David]
George
class [Ljava.lang.String;
[public final void java.lang.Object.wait(long,int) throws java.lang.I
.lang.Object.wait() throws java.lang.InterruptedException, public boo
.hashCode(), public final native java.lang.Class java.lang.Object.get
class java.lang.String
class [Ljava.lang.String;
[public final void java.lang.Object.wait(long,int) throws java.lang.I
.lang.Object.wait() throws java.lang.InterruptedException, public boo
.hashCode(), public final native java.lang.Class java.lang.Object.get
class java.lang.String
```

...

"[L" at the beginning and ";" at the end means an array of objects of a given type

# Example: working with annotations in Java reflection

```java
import java.lang.annotation.*;
import java.lang.reflect.*;

@Documented
@Retention(RetentionPolicy.RUNTIME)
@interface Ownership
{
    String name();
    String company();
    String purpose();
}

@Ownership(name="Yulia Newton", company="self", purpose="CS151")
class Animal {
    public void introduce()
    {
        System.out.println("I am animal");
    }
}

public class Test {
    public static void main(String args[]){
        Annotation[] animalAnnotations = Animal.class.getAnnotations();

        for(Annotation annotation : animalAnnotations){
            if(annotation instanceof Ownership){
                Ownership myAnnotation = (Ownership) annotation;
                System.out.println("Name: " + myAnnotation.name());
                System.out.println("Company: " + myAnnotation.company());
                System.out.println("Purpose: " + myAnnotation.purpose());
            }
        }
    }
}
```

my custom annotation

class *Animal* is annotated with my custom annotation

we obtain the list of all annotations, loop through them, test if we found Ownerhip annotation and then print values associated with that annotation

Output:

```
Name: Yulia Newton
Company: self
Purpose: CS151
```

# Synthetic constructs

- Synthetic constructs are classes, methods, or fields created by Java compiler for internal purposes
- *Java.lang.Class.isSynthetic()* can help test if the entity is a synthetic construct

# Example: testing for synthetic constructs

```java
import java.lang.reflect.*;
enum Gender
{
    MALE, FEMALE, NONBINARY, UNSPECIFIED;
}

public class Test{
    public static void main(String[] args){
        Field[] genderFields = Gender.class.getDeclaredFields();
        for(Field f : genderFields){
            if(f.isSynthetic()){
                System.out.println(f.toString());
            }
        }
    }
}
```

Enums data structure gets converted to a class during compile time, hence during run-time it is a synthetic construct

Internally converted at compile time to:

```java
class Gender
{
    public static final Gender MALE = new Gender();
    public static final Gender FEMALE = new Gender();
    public static final Gender NONBINARY = new Gender();
    public static final Gender UNSPECIFIED = new Gender();
}
```

Output:

```
private static final Gender[] Gender.$VALUES
```

# Class loaders

- Remember class loader subsystem from the introduction?
- Class loaders are responsible for loading classes to the JVM during run-time
  - Class loaders is the reason JVM can be platform independent
    - Except for the bootstrap class loader
  - Java class loaders are Java classes themselves
    - Except for the bootstrap class loader
    - A class loader is an instance of *java.lang.ClassLoader*
- Class loaders load into memory as they become required by the program execution
  - Classes are not loaded into memory unless they are being used by the program
- JRE calls the appropriate class loader
  - Class loaders are a part of JRE

# Class loader types

- Different classes are getting loaded by different class loaders

- Class loaders
  - Bootstrap
  - Extension
  - Application

# Bootstrap class loader

- Written in machine code
- Sometimes called “primordial class loader”
- Responsible for loading JDK internal classes when JVM calls it
- The only class loader that is not a Java class
- Does not have any parents
- Loads classes located in *rt.jar* and other classes located in *$JAVA_HOME/jre/lib*

# Extension class loader

- A child of bootstrap class loader
- Loads extensions of core Java classes from JDK extension library
  - *$JAVA_HOME/lib/ext* (e.g. *jre/lib/ext*), specified by *java.ext.dirs* system property

# Application class loader

- Also called “system class loader”
- Loads classes found in $*CLASSPATH*
- A child of extension class loader

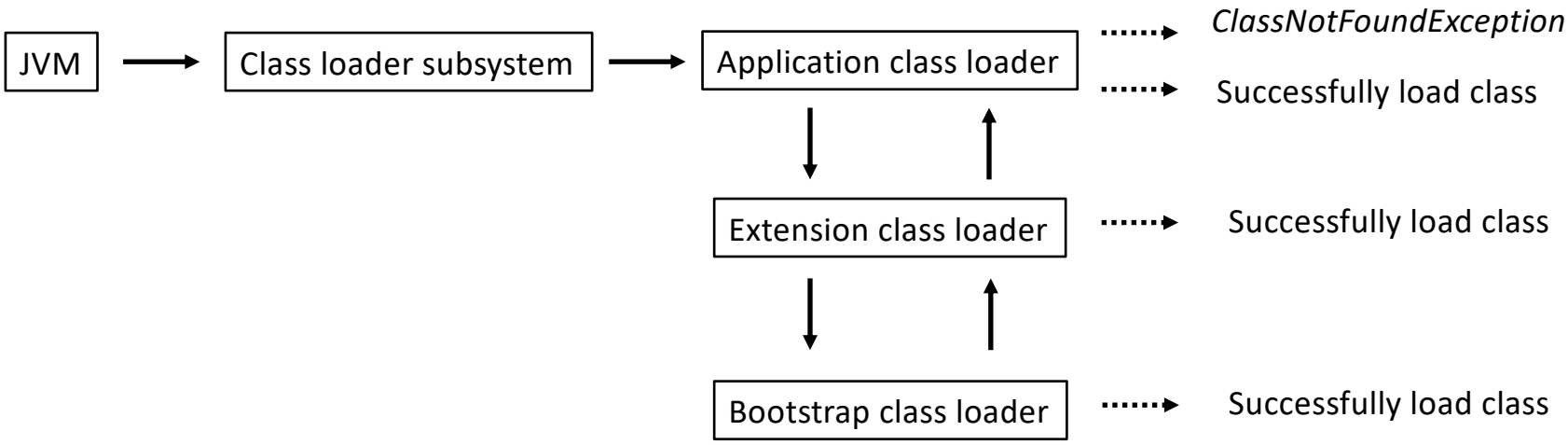
# Class loader delegation model

- *java.lang.ClassLoader.loadClass()* is responsible for loading class definition
  - Based on fully qualified name
- When JVM needs a class it checks if the class is already loaded
- If the class is not already loaded then the class loader delegates the loading to its parent
  - Recursive process
- If the parent does not find the class then the child calls *java.net.URLClassLoader.findClass()* method
- If the class loader is unable to load the class then it throws *NoClassDefFoundError* or *ClassNotFoundException*

# Class loader delegation model (cont'd)

- Delegation Hierarchy Algorithm specifies that
  - If the class is not already loaded, first JVM asks class loader subsystem to load the class
  - Class loader subsystem asks application class loader to load the class
  - If application loader cannot load class it delegates to extension class loader
  - If extension class loader cannot load class it delegates to bootstrap class loader
  - If bootstrap class cannot find class it passes searching back to extension class loader
  - If extension class loader cannot find class it passes searching back to application class loader
  - If application class loader cannot find class it throws *ClassNotFoundException*

# Delegation model diagram



# Class uniqueness principle

- Classes are unique (fully qualified names)
- Classes loaded by parent class loaders are not loaded by children class loaders

- Classes loaded by bootstrap class loader cannot be loaded by extension or application class loaders

# Class visibility principle

- Classes loaded by class loader children are not visible to the parent class loaders
- Classes loaded by the class loader parent are visible to the child class loader

- Classes loaded by extension class loader are visible to application class loader but not visible to bootstrap class loader

# Example: obtaining class loader information

```
public class Test{
    public static void main(String[] args){
        System.out.println("Classloader of Employee class is: "+Employee.class.getClassLoader());
    }
}
```

Output:

```
Classloader of Employee class is: sun.misc.Launcher$AppClassLoader@2a139a55
```

# *NoClassDefFoundError* vs. *ClassNotFoundException*

- Both occur when the class is not found and cannot be loaded
- *ClassNotFoundException*
  - Requested class is not found in the class path
  - Checked exception
  - Exception handling code must be provided for this exception
  - Can occur when one class loader attempts to load a class after it has already been loaded by another class loader
- *NoClassDefFoundError*
  - Occurs when class was present during the compile time but is no longer found in its path at run-time
    - E.g. when there is a dependency on a class that changed its location after compiling

# Custom class loaders

- Sometimes it is useful to write your own custom class loader
  - Subclass of *ClassLoader* class

- Use for custom class loaders:
  - Dynamic class loading at run-time
  - Load classes from non-standard sources (files, network, database, web)
  - Conflict avoidance by constraining class scope to a particular class loader
  - Support different versions of a class in the same JVM
  - Reloading modified classes during run-time
  - Usually more useful for large architectures

# Example: custom class loader

```
class MyCustomClassLoader extends ClassLoader {
    public MyCustomClassLoader(ClassLoader cl){
        super(cl);
    }

    @Override
    public Class findClass(String name) throws ClassNotFoundException {
        System.out.println("Finding Class '" + name + "'");
        return super.findClass(name);
    }

    @Override
    public Class loadClass(String name) throws ClassNotFoundException {
        System.out.println("Loading Class '" + name + "'");
        return super.loadClass(name);
    }
}

public class Test {
    public static void main(String args[]) throws IOException{
        MyCustomClassLoader myCL = new MyCustomClassLoader(Employee.class.getClassLoader());
        try{
            Class employeeClass = myCL.loadClass("Employee");
            Class employeeClass2 = myCL.findClass("Employee");
        }catch(ClassNotFoundException e){
            System.out.println(e);
        }
    }
}
```

Overload loadClass() method to add functionality to it

New custom class loader that is a child of the class loader responsible for loading Employee class

Use class loader's loadClass() method to load Class object for Employee class

Output:

```
Loading Class 'Employee'
Finding Class 'Employee'
```

Same Employee class as in previous examples

```
import java.lang.reflect.*;

class Employee{
    private String name;
    private int id;
    private Gender gender;

    Employee(){
        this.name = "Uknown name";
        this.id = 0;
        this.gender = Gender.UNSPECIFIED;
    }

    Employee(String name, int id, Gender gender){
        this.name = name;
        this.id = id;
        this.gender = gender;
    }

    public String getName(){return this.name;}
    public int getID(){return this.id;}
    public Gender getGender(){return this.gender;}

    public void setName(String name){this.name = name;}
    public void setID(int id){this.id = id;}
    public void setGender(Gender gender){this.gender = gender;}

    @Override
    public String toString(){
        return this.name+" ("+this.id+") is a "+this.gender;
    }
}
```

# Java dynamic proxies

- "A **dynamic proxy** class is a class that implements a list of interfaces specified at runtime such that a method invocation through one of the interfaces on an instance of the class will be encoded and dispatched to another object through a uniform interface." (https://docs.oracle.com/)
  - Proxies acts as an agent between two objects
  - Allow implementing interfaces at run-time
  - Control access to the target object
  - Generates bytecode through Java reflection
- *java.lang.reflect.Proxy* library

# Example: simple example of dynamic proxies

```
import java.lang.reflect.*;

public class Test {
    interface MyInterface {
        void display(String message);
    }

    static class MyImplementor implements MyInterface {
        public void display(String message) {
            System.out.println(message);
        }
    }

    static class Handler implements InvocationHandler {
        private final MyInterface implementor;
        public Handler(MyInterface implementor) {
            this.implementor = implementor;
        }
        public Object invoke(Object proxy, Method method, Object[] args)
          throws IllegalAccessException, IllegalArgumentException, InvocationTargetException {
            return method.invoke(implementor, args);
        }
    }

    public static void main(String[] args){
        MyImplementor implementor = new MyImplementor();
        Handler handler = new Handler(implementor);
        MyInterface i = (MyInterface) Proxy.newProxyInstance(MyInterface.class.getClassLoader(),
                                                             new Class[] { MyInterface.class },
                                                             handler);
        i.display("This is the message to be broadcasted");
    }
}
```

Output:

```
This is the message to be broadcasted
```

# Implementing a proxy

- Proxy behavior is implemented through invocation handler
  - Extends *java.lang.reflect.InvocationHandler*
  - Implement *public Object invoke(Object proxy, Method method, Object[] args)*

- A very useful description of Java proxies
  - https://blog.frankel.ch/the-power-of-proxies-in-java/

# Example: List that cannot be added to

```
public class NoOpAddInvocationHandler implements InvocationHandler {

  private final List proxied;

  public NoOpAddInvocationHandler(List proxied) {
    this.proxied = proxied;
  }

  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    if (method.getName().startsWith("add")) {
      return false;
    }
    return method.invoke(proxied, args);
  }
}
```

```
List proxy = (List) Proxy.newProxyInstance(
  NoOpAddInvocationHandlerTest.class.getClassLoader(),
  new Class[] { List.class },
  new NoOpAddInvocationHandler(list));
```

https://blog.frankel.ch/the-power-of-proxies-in-java/

# Recap

- Java reflection allows modifying class, interface, enum, or method behavior at run-time
- We covered the following reflection topics
  - Testing if a class is an instance of a particular type
  - Obtaining class objects associated with a particular instance of a class
  - Creating new class instances
  - Obtaining inheritance information (superclass and interfaces)
  - Getting class field information
  - Getting class method information
  - Invoking class methods
  - Working with Arrays
  - Working with annotations
  - Synthetic constructs
  - Class loaders
  - Dynamic proxies

# Concluding remarks

- Java reflection is a powerful tool to modify program behavior at run-time
- Reflection allows using outside and third-party classes in your program
- Use sparingly
  - Costly in performance
  - Hard to maintain
  - Code is hard to understand