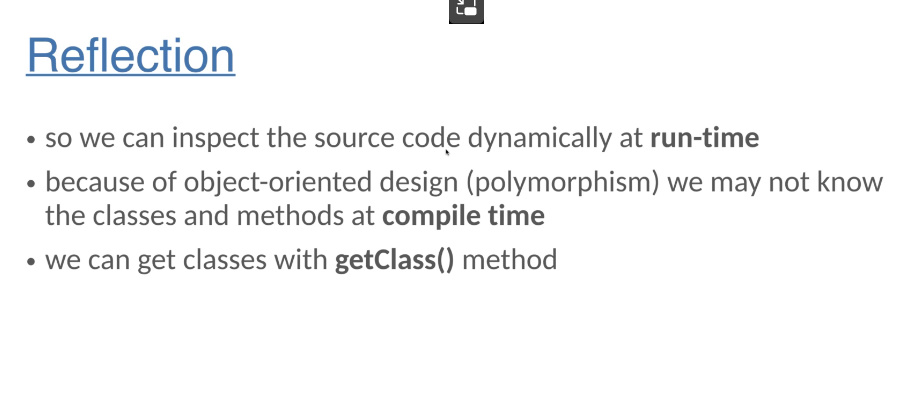
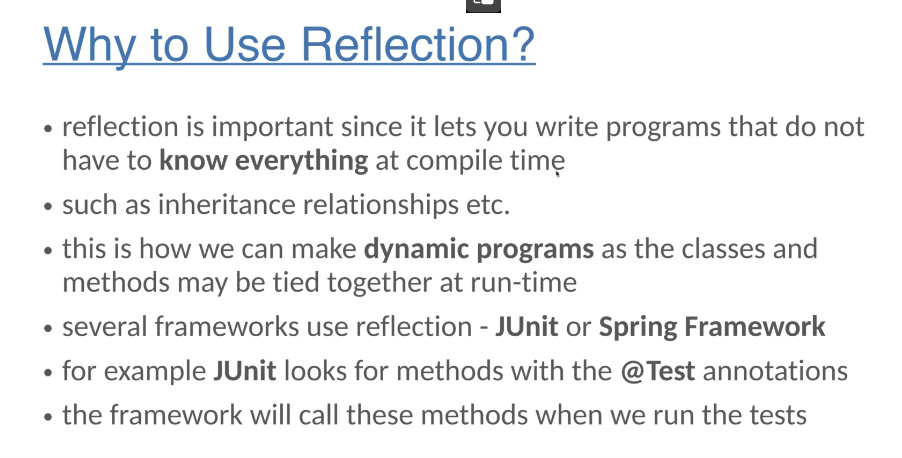
* **Reflection** – abilitatea unui Limbaj de programare de a inspecta si de a chema dinamic metode, clase etc. Sau,pe scurt, este abilitatea sa de a se autoanaliza
* Reflection est un API folosit pentru a examina si modifica comportamentul metodelor, claselor si interfetelor la runtime
* Reflection ofera informatii despre clasa la care un obiect apartine si de asemenea despre metodele acelei clase care pot fi executate folosind obiectul
* obiectele de tip Class sunt construite la Runtime de JVM cand incarca clasele noastre
* Spring folosete foarte mult Reflection



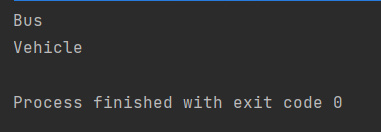


**Get class**

* In Java exista o clasa numita Class
* nu are constructor public
* Este o clasa care poate oferi detalii despre alta clasa sau interfata, ca numele,superclasa sa,interfatele ce le implementeaza,fields si methods
* Fiecare clasa poate folosi .class
* **.class** va returna un obiect de tip Class care va reprezenta propriu zis acea clasa care a apelat-o si va contine toate informatiile necesare despre ea.
* De ex fie ca am creat noi clasa Parent. Deci, Parent.class va returna un obiect de tip Class cu metadate despre clasa Parent
* .class poate fi folosita si pentru interfate
* Datele primitive(int,double,fload...) si void tot sunt obiecte de tip Class.
* Pentru a lua referinta obiectului clasei, folosim Class<clasa>
* public class Test {  
   public static void main(String[] args) {  
   Class <Shop> shopClass = Shop.class;  
   System.*out*.println(shopClass.getName());  
   }  
    
  }  
  class Shop{  
    
  }

.**getClass()**

* Object are o metoda .getClass(), care returneaza un obiect de tip Class a clasei a carui instanta este obiectul ce o cheama.
* Face exact ceea ce face si .class, doar ca .getClass() e folosit de obiecte
* E folositor deoarece de multe ori din cauza la Object Oriented Design si Inheritance nu putem sti de ce tip este un anumit obiect, caci s-ar putea ca variabila ce are referinta la el sa fie de tip parinte.
* public class Test {  
   public static void main(String[] args) {  
   *check*(new Bus());  
   *check*(new Vehicle());  
   }  
   public static void check(Car car){  
   Class obj = car.getClass();  
   System.*out*.println(obj.getName());  
   }  
    
  }  
  interface Car{ }  
  class Vehicle implements Car{ }  
  class Bus implements Car { }



Deci, se returneaza clasa propriu zisa la referinta pastrata.

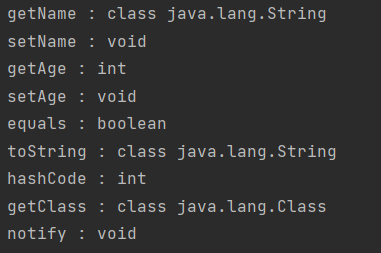
**Metode ale lui Class**

* .getName() – returneaza numele clasei + package unde se afla
* static .forName(“package.className”) – metoda data e statica si returneaza obiectul pentru clasa specificata intre “”

Class ref = Class.*forName*("com.java.test.MyClass");  
System.*out*.println(ref.getName());

* .getPackage() – returneaza package in care se gaseste clasa
* getFields() – returneaza un array de Field, adica Field[] cu toate fields publice din acea clasa/interfata + cele implementate/extinse
* public class Test {  
   public static void main(String[] args) {  
   Class <Person> obj = Person.class;  
   Field[] fields = obj.getFields();  
    
   for(Field f : fields)  
   System.*out*.println(f.getName());  
   }  
    
    
  }  
  class Person{  
   public String name;  
   public int age;  
  }
* getMethods() – returneaza un array de Method, adica Method[] cu toate metodele **publice** din clasa/interfata + cele implementate/extinse

public class Test {  
 public static void main(String[] args) {  
 Class obj = Person.class;  
 Method[] methods = obj.getMethods();  
  
 for(Method m : methods)  
 System.*out*.println(m.getName() + " : " + m.getReturnType());  
 }  
  
  
}  
class Person{  
 public String name;  
 public int age;  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public int getAge() {  
 return age;  
 }  
  
 public void setAge(int age) {  
 this.age = age;  
 }  
}



* getDeclaredFields() – returneaza toate filedurile, indiferent de accesibilitatea lor
* getDeclaredMethods() – returneaza toate metodele, indiferent de accesbilitatea lor
* getConstructor() – returneaza constructorul public
* getDeclaredConstructor() – returneaza constructorul indiferent de accesibilitatea sa. Asa putem instantia chiar si cu un constructor private
* public class Test {  
   public static void main(String[] args) throws Exception {  
   Class<Person> obj = Person.class;  
   Constructor<Person> constructor = obj.getDeclaredConstructor();  
   constructor.setAccessible(true);  
   Person person = constructor.newInstance();  
   System.*out*.println(person);  
   }  
    
  }  
  class Person{  
   private Person() { }  
   @Override  
   public String toString() {  
   return "Person{}";  
   }  
  }
* getSuperClass() – returneaza obiectul super clasei
* public class Test {  
   public static void main(String[] args) throws Exception {  
   Class c = Employee.class;  
   System.*out*.println(c.getSuperclass().getName());  
   }  
    
  }  
  class Person{  
  }  
  class Employee extends Person{  
    
  }
* getInterfaces() - returneaza o Class[] cu toate interfetele implementate.
* Class c = LinkedList.class;  
  Class[] interfaces = c.getInterfaces();  
  for(Class i : interfaces)  
   System.*out*.println(i.getName());

**Fields**

* Clasa Field ofera informatii despre un field dintr-o clasa/interfata
* .getName() – returneaza numele fieldului. Atentie! Va merge doar daca acel field e public
* **.setAccesible(true)**

**Methods**

* Clasa Method ofera informatii despre metoda unei clase/interfete
* .getName()
* getReturnType()
* **.setAccesible(true)**
* isAnnotationPresent(Annotation.class) – returneaza true daca metoda contine anotatia data, false daca nu
* public class Test {  
   public static void main(String[] args) throws Exception {  
   Class c = Person.class;  
   Method[] methods = c.getMethods();  
    
   if(methods[0].isAnnotationPresent(Annotation.class))  
   System.*out*.println(methods[0].getName());  
   }  
    
  }  
    
  @Target(ElementType.*METHOD*)  
  @Retention(RetentionPolicy.*RUNTIME*)  
  @interface Annotation{  
   public String name();  
  }  
  class Person{  
   @Annotation(name="Hi")  
   public void test() { }  
  }

**Constructors**

* Clasa Constructor<> ofera informatii despre constructorul unei clase
* .newInstance() – creaza o instanta cu acel constructor. **Anume de metoda asta se foloeste Spring pentru a implementa Singletop Design Pattern**
* **.setAccesible(true) – facem accesarea constructorului, daca e private, posibila**