# Réflexivité et annotations

# Introspection

- Le byte code (binaire) d'une classe contient de nombreuses informations permettant d'examiner, ou modifier les objets au moment de l'exécution d'une application Java
- Les champs et méthodes sont identifiés par leur nom
- Les noms des classes sont générés à partir du package et du nom de la classe
- L'utilitaire java permet d'explorer les classes

# Introspection

- Codage des types dans le bytecode

| Type | Code |
|---|---|
| byte | B |
| char | C |
| double | D |
| float | F |
| int | I |
| long | J |
| short | S |
| boolean | Z |
| T[] | [T |
| package.Foo | Lpackage.Foo ; |

# API de réflexion

- Les classes utilisées pour l'introspection sont contenues dans les packages *java.lang.reflect* et *java.lang*
- Principales classes :
  - `Class`
  - `Field`
  - `Modifier`
  - `Method`
  - `Constructor`
  - `Annotation`

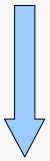
# Classe `Class`

- La classe `java.lang.Class` est le point d'entrée des opérations d'introspection
- La méthode `getClass()` sur un objet permet de récupérer l'instance de `Class` associée à cet objet
  - possibilité d'utiliser la propriété `.class` sur la classe elle-même

# Classe `Class`

- La syntaxe `.class` est le moyen le plus simple pour récupérer une `Class` sur un type primitif
- Chaque wrapper possède un champ `TYPE`
  - `int.class` est équivalent à `Integer.TYPE`

```
System.out.println("int.class => "+int.class);
System.out.println("Integer.TYPE => "+Integer.TYPE);
```



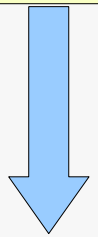
```
int.class => int
Integer.TYPE => int
```

# Classe `Class`

```
List<Contact> contacts = new ArrayList<Contact>();
Contact[] tab = new Contact[5];

Class c1 = contacts.getClass();
Class c2 = contacts.get(0).getClass();
Class c3 = tab.getClass();

System.out.println("Type de ArrayList<Contact> => "+c1.getCanonicalName());
System.out.println("Type d'un élément de ArrayList<Contact> => "+c2.getCanonicalName());
System.out.println("Type de Contact[] => "+c3.getCanonicalName());
System.out.println("Type de int[] => "+int[].class.getCanonicalName());
```



```
Type de ArrayList<Contact> => java.util.ArrayList
Type d'un élément de ArrayList<Contact> =>
org.antislashn.formation.Contact
Type de Contact[] => org.antislashn.formation.Contact[]
Type de int[] => int[]
```

# Classe `Class`

- `Class.classForName("package.Foo")` permet de récupérer l'objet `Class` correspondant à la classe `Foo`.
  - le code statique de la classe est alors exécuté
- L'instanciation de la classe `Foo` peut être effectuée par :
  `Class.classForName("package.Foo").newInstance()`
  - Les instances de tableaux pourront être effectuées par `Array.newInstance(Class<?>, int length)`

# Classe `Class`

```
try {
     Contact c = (Contact) Class.forName("org.antislashn.Contact").newInstance();
} catch (InstantiationException e) {
     e.printStackTrace();
} catch (IllegalAccessException e) {
     e.printStackTrace();
} catch (ClassNotFoundException e) {
     e.printStackTrace();
}


Contact[] t1 = (Contact[]) Array.newInstance(Contact.class, 10);
int[] t2 = (int[]) Array.newInstance(int.class, 10);
```

# Classe `Class`

- Certaines méthode permettent d'obtenir des informations sur la classe

| Méthodes | Desciption |
|---|---|
| Field getField(String name) | renvoie l'objet Field name |
| Field[] getFields() | renvoie l'ensemble des champs |
| Method[] getMethods | renvoie l'ensemble des méthodes |
| Method getMethod | renvoie l'objet Methode name |
| Constructor getConstructor(Class[] paramType) | renvoie l'objet Construtor avec les paramètes définis |
| Constructor[] getConstructors() | renvoie l'ensemble des constructeurs |
| Class[] getInterfaces() | renvoie l'ensemble des interfaces implémentées |
| Class getSuperclass() | renvoie la classe mère |
| Package getPackage() | renvoie l'objet Package de la classe |

# Examens des modificateurs

- Les modificateurs affectent le comportement
  - `public`, `protected`, `private`
  - `abstract`
  - `final`
  - `static`
  - `interface`
- `Class.getModifiers()` permet d'examiner les modificateurs
  - cf. la classe `Modifier` pour aider au décodage

# Codage des modificateurs

| Flag Name | Value | Interpretation |
|---|---|---|
| ACC_PUBLIC | 0x0001 | Declared `public`; may be accessed from outside its package. |
| ACC_FINAL | 0x0010 | Declared `final`; no subclasses allowed. |
| ACC_SUPER | 0x0020 | Treat superclass methods specially when invoked by the *invokespecial* instruction. |
| ACC_INTERFACE | 0x0200 | Is an interface, not a class. |
| ACC_ABSTRACT | 0x0400 | Declared `abstract`; must not be instantiated. |
| ACC_SYNTHETIC | 0x1000 | Declared synthetic; not present in the source code. |
| ACC_ANNOTATION | 0x2000 | Declared as an annotation type. |
| ACC_ENUM | 0x4000 | Declared as an `enum` type. |

source : Oracle

```
public final class Contact {
...
}
```

```
int modifiers = Contact.class.getModifiers();
System.out.println(String.format("%x", modifiers));
```

11

# Examens des types

- **Quelques méthodes de** `Class`
  - `Class.getTypeParameters()` permet de connaître l'existence et le nombre de types paramétrés
  - `Class.getAnnotations()` permet l'examen des annotations
  - reportez-vous à la javadoc pour plus de détails

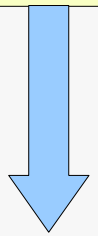
# Classe `Field`

- Représente un attribut de la classe
  - est obtenu par les méthodes sur Class
    - `getFields()`,`getField(String Name)`
    - `getDeclaredFields()`,
    - `getDeclaredField(String Name)`
- `Field` possède des méthodes utilitaires
  - `String getName()`
  - `Class getType()`
  - `int getModifiers()`

# Class `Field`

```
Field[] fields = Contact.class.getFields();
System.out.println("Nb de propriétés publiques = "+fields.length);
for(Field f : fields){
     System.out.println(f.getName() + " => "+f.getType());
}

fields = Contact.class.getDeclaredFields();
System.out.println("Nb de propriétés déclarées = "+fields.length);
for(Field f : fields){
     System.out.println(f.getName() + " => "+f.getType());
}
```



```
Nb de propriétés publiques = 0
Nb de propriétés déclarées = 2
nom => class java.lang.String
age => int
```

# Manipulation d'une propriété

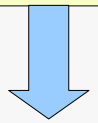
- La classe `Field` possède une méthode de lecture `get(Object obj)`
  - cf. aussi `getBoolean(...)`, `getByte(...)`, etc.
  - la méthode `setAccessible(boolean flag)` permet de rendre un champ privé accessible
    - cf. droits au moment du chargement de la classe

# Manipulation d'une propriété

- La classe `Field` possède une méthode d'écriture `set(Object obj, Object value)`
  - cf. aussi `setBoolean(...)`, `setByte(...)`, etc
  - la méthode `setAccessible(boolean flag)` permet de rendre un champ privé accessible
    - cf. droits au moment du chargement de la classe

# Manipulation d'une propriété

```
Contact toto = new Contact("Toto",12);
try {
     Field nameField = Contact.class.getDeclaredField("nom");
     nameField.setAccessible(true);
     System.out.println(">>> "+nameField.get(toto));
     nameField.set(toto, "Titi");
     System.out.println(">>> "+nameField.get(toto));
} catch (NoSuchFieldException e) {
     e.printStackTrace();
} catch (SecurityException e) {
     e.printStackTrace();
} catch (IllegalArgumentException e) {
     e.printStackTrace();
} catch (IllegalAccessException e) {
     e.printStackTrace();
}
```



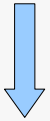
```
>>> Toto
>>> Titi
```

# Class `Method`

- `Class[] getExceptionTypes()` renvoie un tableau de classes `Exception`
- `String getName()` renvoie le nom de la méthode
- `Class getReturnType()` renvoie la classe de la valeur de retour
- `Class[] getParameterTypes()` renvoie un tableau des paramètres
- `int getModifiers()` renvoie les modificateurs

# Class Method

```
Method[] methods = Contact.class.getDeclaredMethods();
for(Method m : methods){
     System.out.println(m.getName() + " - return type : "+m.getReturnType());
}
```



```
toString - return type : class java.lang.String
getNom - return type : class java.lang.String
getAge - return type : int
setAge - return type : void
setNom - return type : void
```

# Appel d'une méthode

- Il faut d'abord retrouver la bonne signature de la méthode auprès de `Class`
  - **`par exemple :`**
    `getMethod(String name, Class[] parameterTypes)`
    - `name` : nom de la méthode
    - `parameterTypes` : tableau contenant les types des paramètres

# Appel d'une méthode

- Puis la méthode invoquée la méthode avec `invoke(Object o, Object... args)`
  - `o` : instance de l'objet sur laquelle la méthode est appelée
    - pas pris en compte sur une méthode statique
  - `args` : liste des arguments
    - tableau dans les versions précédentes de JDK
  - un `Object` est retourné par l'invocation

# Appel d'une méthode

```
Contact titi = new Contact("Titi",13);
Class[] paramType = {String.class};
try {
     System.out.println(">>>> "+titi.getNom());
     Method m = Contact.class.getMethod("setNom", paramType);
     m.invoke(titi, "tata");
     m = Contact.class.getMethod("getNom", new Class[0]);
     System.out.println(">>>> "+m.invoke(titi, new Object[0]));
} catch (NoSuchMethodException e) {
     e.printStackTrace();
} catch (SecurityException e) {
     e.printStackTrace();
} catch (IllegalAccessException e) {
     e.printStackTrace();
} catch (IllegalArgumentException e) {
     e.printStackTrace();
} catch (InvocationTargetException e) {
     e.printStackTrace();
}
```

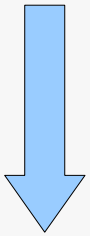
```
>>>> Titi
>>>> tata
```

# Class `Constructor`

- `Class[] getExceptionTypes()` renvoie un tableau de classes `Exception`
- `String getName()` renvoie le nom du constructeur
- `Class[] getParameterTypes()` renvoie un tableau des paramètres
- `int getModifiers()` renvoie les modificateurs

# Class `Constructor`

```
Constructor[] constructors =
Contact.class.getConstructors();
for(Constructor c : constructors){
     System.out.println(c.toGenericString());
}
```



```
public org.antislashn.formation.Contact(java.lang.String,int)
public org.antislashn.formation.Contact(org.antislashn.formation.Contact)
public org.antislashn.formation.Contact()
```

# Les annotations

- Appelées également métadonnées
  - préfixées par `@`
- Les annotations marquent certains éléments du langage
  - classe, propriété, arguments et/ou méthode
- Les annotations peuvent être utilisées par :
  - le compilateur (`SOURCE`)
    - ou autre outil utilisant le source
  - la machine virtuelle, par introspection (`RUNTIME`)
  - par un outil externe (`CLASS`)
    - annotation présente dans le binaire, mais pas utilisable par la JVM

# Les annotations

- Annotations standards
  - `@Deprecated`
    - signale au compilateur que l'élément marqué ne devrait plus être utilisé
  - `@Override`
    - précise au compilateur que la méthode est redéfinie
    - sur JDK 1.5 ne marque pas les méthodes héritées d'interface
  - `@SuppressWarning`
    - indique au compilateur de ne pas afficher certains warnings
    - prend en attribut le nom du warning

# Les méta-annotations

- Ce sont des annotations destinées à marquer d'autres annotations
  - package `java.lang.annotation`
  - `@Documentated`
    - indique à javadoc que l'annotation doit être présente lors de la génération de la documentation
  - `@Inherited`
    - par défaut les annotations ne sont pas héritées
    - force l'héritage de l'annotation sur les éléments marqués

# Les méta-annotations

- `@Retention`
  - politique de rétention de l'annotation
    - `RetentionPolicy.SOURCE`
      - l'annotation n'est pas enregistrée dans le fichier binaire
    - `RetentionPolicy.CLASS`
      - l'annotation est enregistrée dans le binaire mais non utilisée par la JVM
    - `RetentionPolicy.RUNTIME`
      - l'annotation est présente dans le fichier binaire
      - utilisable par la JVM

# Les méta-annotations

- `@Target`
  - limite le type de l'élément sur lequel est utilisée l'annotation

| Valeur | Utilisation |
|---|---|
| `ElementType.ANNOTATION_TYPE` | utilisable sur d'autres annotations |
| `ElementType.CONSTRUCTOR` | utilisable sur les constructeurs |
| `ElementType.FIELD` | utilisable sur les propriétés d'une classe |
| `ElementType.LOCAL_VARIABLE` | utilisable sur les variables locales d'une méthode |
| `ElementType.METHOD` | utilisable sur les méthodes |
| `ElementType.PACKAGE` | utilisable sur les packages |
| `ElementType.PARAMETER` | utilisable sur les paramètres de méthodes ou constructeurs |
| `ElementType.TYPE` | utilisable sur la déclaration d'un type (class, interface, @interface, enum) |

# Les annotations

- En plus des annotations standards et méta-annotations il existe de nombreuses bibliothèques d'annotations
  - pour Hibernate, les EJB3
  - annotations spécifiques serveurs JBoss, WebSphere
- Vous pouvez créer vos propres annotations
  - automatisation de tâches sur les classes
  - utilisation en AOP (Aspect Oriented Programming)

# Les annotations

- Un ensemble d'annotations communes a été défini dans la JSR 250
  - JSR intégrée à Java 6
  - permet de définir des annotations couramment utilisées et d'éviter leur redéfinition par chaque outil
  - ces annotations concernent
    - Java SE dans le `package javax.annotation`
    - Java EE dans le package `javax.annotation.security`
- Ces annotations sont prises en charge par les conteneurs

# Annotations Java SE

- `@Generated` : marque la génération de code
- `@Resource`, `@Resources` : pour l'injection de ressource
  - EJB, JMS, DataSource, …
- `@PostConstruct`, `@PreDestoy` : marquent les méthodes appelées après la construction de l'objet et avant sa destruction

# Annotations Java EE

- `@DeclareRoles`, `@DenyAll`, `@PermitAll`, `@RolesAllowed`, `@RunAs`
- Permettent de gérer l'accès à des classes et/ou des méthodes en fonction de profil

# Exemple d'annotation personnalisée

- Code de l'annotation
  - l'annotation sera conservée dans le binaire
  - l'annotation est visible par la machine virtuelle
  - l'annotation est utilisée uniquement sur les propriétés
  - l'annotation possède deux attributs non obligatoires
    - string et integer

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD})
public @interface Inject {
     String string() default "";
     int integer() default 0;
}
```

# Exemple d'annotation personnalisée

- Utilisation de l'annotation par la classe `Contact`
  - si la classe est utilisée dans un contexte qui ne "comprend pas" l'annotation, elle garde son comportement habituel

```
public final class Contact {
    @Inject(string="Toto") private String nom;
    @Inject(integer=14) private int age;

...
}
```

# Exemple d'annotation personnalisée

- Exemple de code utilisant l'annotation

```
public class Injector {
public static void inject(Object o) throws IllegalArgumentException, IllegalAccessException{
     Field[] fields = o.getClass().getDeclaredFields();

     for(Field f : fields){
          if(f.isAnnotationPresent(Inject.class)){
               Inject inject = f.getAnnotation(Inject.class);
               f.setAccessible(true);
               if(f.getType() == int.class){
                    f.setInt(o, inject.integer());
               }
               else if(f.getType() == String.class){
                    f.set(o, inject.string());
               }
          }
     }
}
}
```

# Exemple d'annotation

- Test de la classe `Injector`

```
public class InjecteurTest {

public static void main(String[] args) throws IllegalArgumentException, IllegalAccessException {
     Contact c = new Contact();
     Injector.inject(c);
     System.out.println(c);
}

}
```

# Classe `Proxy`

- La classe `Proxy` permet la création dynamique d'instances proxy
  - `newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler h)`
  - `loader` : chargeur de classe qui chargera la classe proxy
  - `interfaces` : liste des interfaces que le proxy doit implémenter
  - `h` : gestionnaire d'invocation des méthodes

# Classe `Proxy`

- Le gestionnaire d'invocation de méthode doit implémenter l'interface `java.lang.reflect.InvocationHandler`
  - une seule méthode : `invoke(Object proxy, Method method, Object[] args)`
    - `proxy` :l'instance du proxy sur laquelle la méthode est invoquée
    - `method` : instance de la méthode invoquée sur le proxy
    - `args` : tableau des paramètres

# Classe `Proxy`

- Exemple de classe métier

```java
public interface IProcess {
     public int compute();
}
```

```java
public class TraitementLong implements IProcess {

     @Override
     public int compute() {
          System.out.println("Début TraitementLong.compute()");
          try {
               Thread.sleep(1500);
          } catch (InterruptedException e) {
               e.printStackTrace();
          }
          System.out.println("Début TraitementLong.compute()");
          return 0;
     }

}
```

# Classe `Proxy`

- Exemple de handler

```
public class ProcessInvocationHandler implements InvocationHandler {
     private IProcess process;

     public ProcessInvocationHandler(IProcess process) {
          this.process = process;
     }

     @Override
     public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
          long start = System.currentTimeMillis();
          Object r = method.invoke(process, args);
          long end = System.currentTimeMillis();
          System.out.println("Process time : "+ (end-start));
          return r;
     }

}
```

# Classe `Proxy`

- Exemple d'utilisation

```
public class ProxyTest {

    public static void main(String[] args) {
        //IProcess process = new TraitementLong();
        //int r = process.compute();

        InvocationHandler handler = new ProcessInvocationHandler(new TraitementLong());

        IProcess process = (IProcess) Proxy.newProxyInstance(IProcess.class.getClassLoader(),
                                                             new Class[]{IProcess.class},
                                                             handler);

        int r = process.compute();
        System.out.println("Résultat => "+r);
    }
}
```

```
Début TraitementLong.compute()
Début TraitementLong.compute()
Process time : 1500
Résultat => 0
```