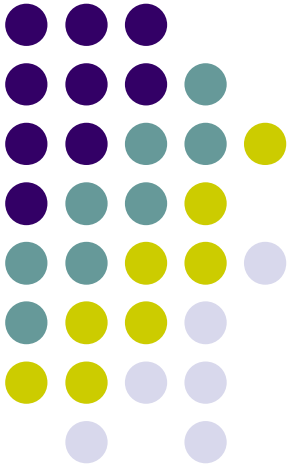
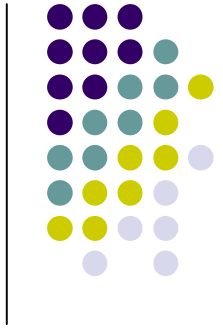
Reflection

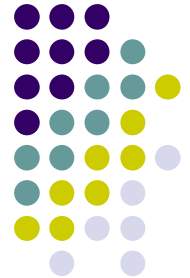


Class



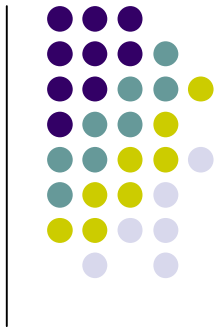
```
class PrivateData {
    private int a = 1;
    public int b;
    public PrivateData(int a) {
        System.out.println("Hello!");
        this.a = a;
        this.b = a;
    }
    private void foo() {
        System.out.println("Hidden! " + a);
    }
    public void boo(int num) {
        System.out.println("Public " + num);
    }
}
```

Class



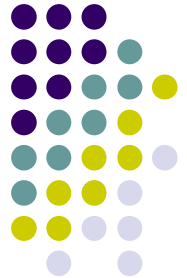
```
class PrivateData2 extends PrivateData {
    private int c = 0;
    public int d = 1;
    public PrivateData2(int a) {
        super(a);
    }
}
```

Class



```
PrivateData pd = new PrivateData(5);
Class<? extends PrivateData> pdClass = pd.getClass();
Class<PrivateData> pdClass2 = PrivateData.class;
Class<Integer> intClass = int.class;
Class<Integer> integerClass = Integer.class;
Class<?> stringClass = null;
try {
    stringClass = Class.forName("java.lang.String");
} catch (ClassNotFoundException e) {
    System.out.println("String class not found - panic!");
}
```

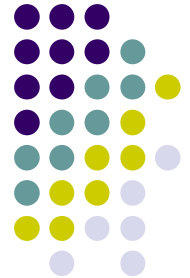
Class.getName()



```
System.out.println(pdClass.getName());
System.out.println(integerClass.getName());
System.out.println(intClass.getName());
System.out.println(stringClass.getName());
```

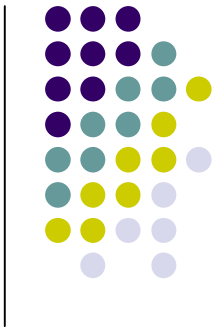
```
Hello!
PrivateData
java.lang.Integer
int
java.lang.String
```

Class.getModifiers()



```
public static void printModifiers(Class<?> c) {
    int mods = c.getModifiers();
    if (Modifier.isPublic(mods)) {
        System.out.println("public");
    }
    if (Modifier.isAbstract(mods)) {
        System.out.println("abstract");
    }
    if (Modifier.isFinal(mods)) {
        System.out.println("final");
    }
}
```

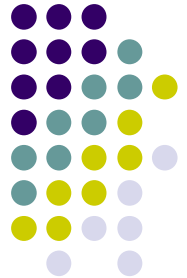
Class.getSuperClass()



```
static void printSuperClasses(Class<?> c, String tabs) {
    if (c == null) {
        return;
    }
    printSuperClasses(c.getSuperclass(), tabs+"   ");
    System.out.println(tabs + c.getName());
}
```

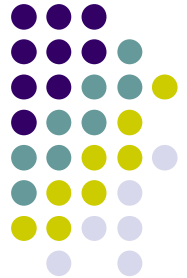
```
          java.lang.Object
   PrivateData
PrivateData2
```

Class.getInterfaces()



```
public static void printInterfaces(Class<?> c) {
    Class<?>[] interfaces = c.getInterfaces();
    for(Class<?> cInterface : interfaces) {
        System.out.println( cInterface.getName() );
    }
}
```

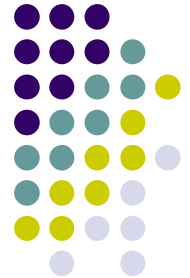
Class.getFields()



```
public static void printFields(Class<?> c) {
    Field[] publicFields = c.getFields();
    for (Field field : publicFields) {
        Class<?> fieldType = field.getType();
        System.out.println("Имя: " + field.getName());
        System.out.println("Тип: " + fieldType.getName());
    }
}
```

```
Имя: d
Тип: int
Имя: b
Тип: int
```

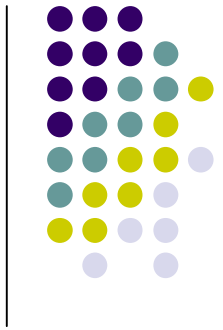
Class. getDeclaredFields()



```
public static void printAllFields(Class<?> c) {
    Field[] publicFields = c.getDeclaredFields();
    for (Field field : publicFields) {
        Class<?> fieldType = field.getType();
        System.out.println("Имя: " + field.getName());
        System.out.println("Тип: " + fieldType.getName());
    }
}
```

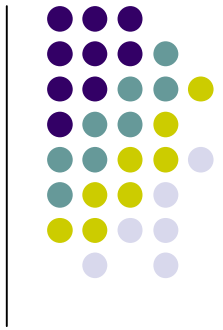
```
Имя: c
Тип: int
Имя: d
Тип: int
```

Field



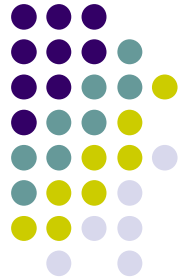
```
PrivateData pd = new PrivateData(5);
try {
    Field field;
    Integer value = 0;
    field = PrivateData.class.getField("b");
    value = (Integer) field.get(pd);
    System.out.println(value);
} catch (NoSuchFieldException | SecurityException |
    IllegalArgumentException | IllegalAccessException
    e) {
    e.printStackTrace();
}
```

Field.set



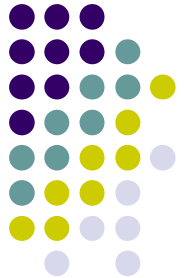
```
try {
    field = PrivateData.class.getDeclaredField("a");
    field.setAccessible(true);
    value = (Integer) field.get(pd);
    field.setInt(pd, value+1);
    value = (Integer) field.get(pd);
} catch (NoSuchFieldException | SecurityException |
    IllegalArgumentException | IllegalAccessException
    e) {
    e.printStackTrace();
}
System.out.println(value);
```

Method



```
Class<?>[] paramTypes = new Class[] { int.class };
Method method;
method = pdClass.getMethod("boo", paramTypes);
Object[] params = new Object[] { new Integer(10) };
method.invoke(pd, params);
```

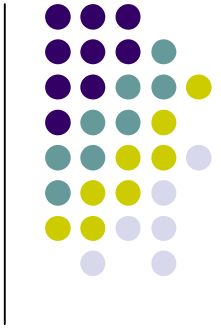
Constructor



```
Class<?>[] paramTypes = new Class[] { int.class };
Constructor<?> aConstrct =
    pdClass.getConstructor(paramTypes);
Object[] params = new Object[] { 5 };
aConstrct.newInstance(params);

Class<?> t = Class.forName("java.lang.String");
Object obj = t.newInstance();
```

ReflectionGenerator

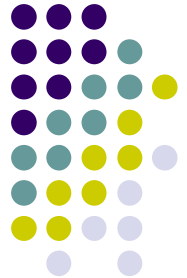


```
interface Generator<T> {
T next();
}

class BasicGenerator<T> implements Generator<T> {
  private Class<T> type;
  public BasicGenerator(Class<T> type){ this.type = type; }
  public T next() {
    try {
      return type.newInstar
    } catch(Exception e) {
      throw new RuntimeE
    }
  }
  public static <T> Gener
    return new BasicGene
  }
}
```

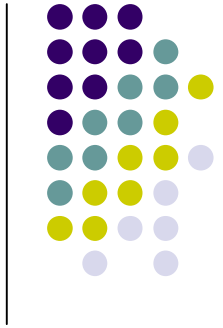
```
public class Main {
   public static void main(String[] args) {
     Generator<CountedObject> gen =
             BasicGenerator.create(CountedObject.class);
     for(int i = 0; i < 5; i++) {
             System.out.println(gen.next());
     }
   }
}
```

GenericArray



```
import java.lang.reflect.Array;
public class GenericArrayWithTypeToken<T> {
  private T[] array;
  public GenericArrayWithTypeToken(Class<T> type, int sz) {
    array = (T[])Array.newInstance(type, sz);
  }
  public void put(int index, T item) { array[index] = item; }
  public T get(int index) { return array[index]; }
  public T[] rep() { return array; }
  public static void main(String[] args) {
    GenericArrayWithTypeToken<Integer> gai = new
     GenericArrayWithTypeToken<Integer>(Integer.class, 10);
    Integer[] ia = gai.rep();
  }
}
```

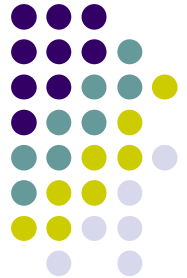
Pets



```
public class Pet extends Individual {
   public Pet(String name) { super(name); }
   public Pet () { super(); }
}
public class Cat extends Pet {
   public Cat(String name) { super(name); }
   public Cat() { super();}
}
public class EgyptianMau extends Cat {
   public EgyptianMau(String name) { super(name); }
   public EgyptianMau() { super(); }
}
```

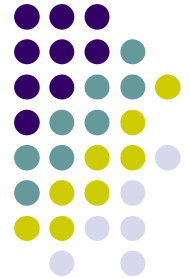
……..

Pet: PetCreator



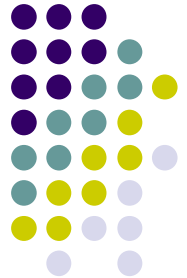
```
public abstract List<Class<? extends Pet>> getTypes();
public Pet randomPet(){
    int n = rand.nextInt(getTypes().size());
    return getTypes().get(n).newInstance();
}
public Pet[] createArray(int size) {
    Pet[] result = new Pet[size];
    for (int i=0; i < size; i++)
        result[i] = randomPet();
    return result;
}
public ArrayList<Pet> arrayList(int size) {
    ArrayList<Pet> result = new ArrayList<Pet>();
    Collections.addAll(result, createArray(size));
    return result;
}
```

Pet: LiteralPetCreator



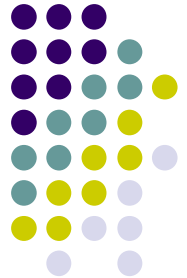
```
public class LiteralPetCreator extends PetCreator {
    private static final List<Class<? extends Pet>> types =
            Arrays.asList(
    Pet.class, Dog.class, Cat.class, Rodent.class, Mutt.class,
     Pug.class, EgyptianMau.class, Manx.class, Cymric.class,
     Rat.class, Mouse.class, Hamster.class
             );
    public List<Class<? extends Pet>> getTypes() {
            return types;
    }
}
```

Pet: TypeCounter



```
public class TypeCounter extends HashMap<Class<?>,Integer>{
  private Class<?> baseType;
  public TypeCounter(Class<?> baseType) {
    this.baseType = baseType;
  }
  public void count(Object obj) {
    Class<?> type = obj.getClass();
    if(!baseType.isAssignableFrom(type)) throw new RuntimeException();
    countClass(type);
  }
  private void countClass(Class<?> type) {
    Integer quantity = get(type);
    put(type, quantity == null ? 1 : quantity + 1);
    Class<?> superClass = type.getSuperclass();
    if(superClass != null && baseType.isAssignableFrom(superClass))
      countClass(superClass);
  }
}
```

Pet: Main



```
public class Main {
public static void main(String[] args) {
    TypeCounter counter = new TypeCounter(Pet.class);
    PetCreator creator = new LiteralPetCreator();
    for(Pet pet : creator.createArray(20)) {
      System.out.print(pet.getClass().getSimpleName() + " ");
      counter.count(pet);
    }
    System.out.println();
    System.out.println(counter);
}
}
```