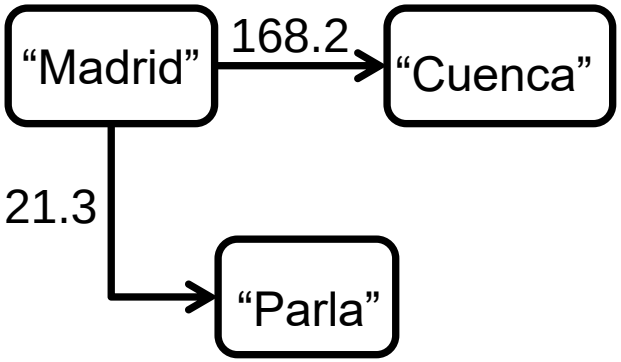
Práctica 5:
Genericidad, colecciones, lambdas y patrones

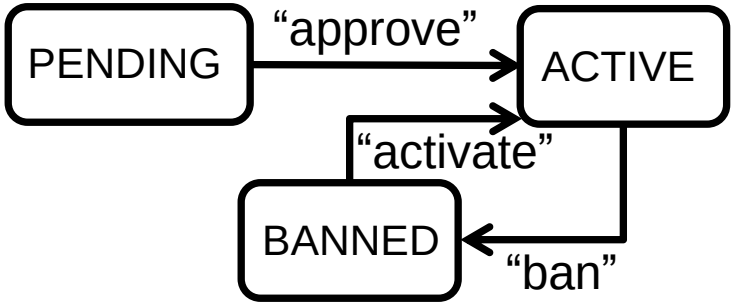
Una guía rápida para empezar

Apartado 1: Grafos (clase Graph)

- Los grafos son estructuras de datos con nodos y enlaces (dirigidos en nuestro caso)
- Crearemos un grafo *genérico*, que pueda almacenar elementos de cualquier tipo en los nodos y en los enlaces.



Grafo con 3 nodos y 2 enlaces:
los nodos almacenan un string,
y los enlaces un double



enum UsrStat { PENDING, ACTIVE, BANNED }

Grafo con 3 nodos y 3 enlaces:
los nodos almacenan un objeto enum,
y los enlaces un string

Apartado 1: Grafos (clase Graph)

- Nuestros grafos deben ser compatibles con las colecciones Java:
  - Deben poder “verse” como una colección de nodos
  - Debemos poder pasar un grafo en el constructor del resto de colecciones

```
Graph<String, Integer> g = new Graph<String, Integer>();
Node<String>  n1 = new Node<String>("s0");
Node<String>  n2 = new Node<String>("s1");
g.addAll(Arrays.asList(n1, n2));
List<Node<String>> nodes = new ArrayList<>(g);
```

- Cada nodo debe tener un identificador único (automáticamente generado)
- Si un nodo se borra, se eliminan los enlaces que salen o entran a él
- No podemos conectar nodos de grafos distintos

Apartado 2a: ConstrainedGraph

- Añadir propiedades sobre los nodos del grafo:
  - exists (...) → devuelve true si existe un nodo que cumple la propiedad
  - forAll (...) → devuelve true si todos los nodos cumplen la propiedad
  - one (...) → devuelve true si exactamente un nodo cumple la propiedad
- Para especificar la propiedad usaremos expresiones lambda(*)

```
g.exists( n -> n.isConnectedTo(n2)))
```

← expresión lambda

```
g.exists(new Predicate<Node<Integer>>() {
            @Override
            public boolean test(Node<Integer> n) {
                return n.isConnectedTo(n2);
            }
        }
       );
```

← Clase anónima equivalente a la expresión lambda

(*) Tema 3.9 “Expresiones Lambda”

Apartado 2b: Comparators

- Un `Comparator` es un objeto conforme la interfaz

```
@FunctionalInterface
public interface Comparator<T> {
   int compare(T o1, T o2);
}
```

- Sirve por ejemplo para ordenar objetos
- Crearemos un comparador de `ConstrainedGraphs` llamado `BlackBoxComparator`
  - Podemos añadir propiedades universales, existenciales o unitaras al comparador
  - Un grafo es mayor que otro si cumple más propiedades

```
BlackBoxComparator<Integer, Integer> bbc = new BlackBoxComparator<>();
 bbc.addCriteria( Criteria.EXISTENTIAL, n -> n.isConnectedTo(2)).
    addCriteria( Criteria.UNITARY, n -> n.neighbours().isEmpty()).
    addCriteria( Criteria.UNIVERSAL, n -> n.getValue().equals(4));

List<ConstrainedGraph<Integer, Integer>> cgs = Arrays.asList(g, g1);
Collections.sort(cgs, bbc);
```

Apartado 3: Reglas (Rule y RuleSet)

- Reglas, con una condición y una acción:
  - Si la condición se cumple, se ejecuta la acción
- Genéricas: aplicables sobre objetos de cualquier tipo
- Se crean con un método estático factoría de la clase `Rule`:

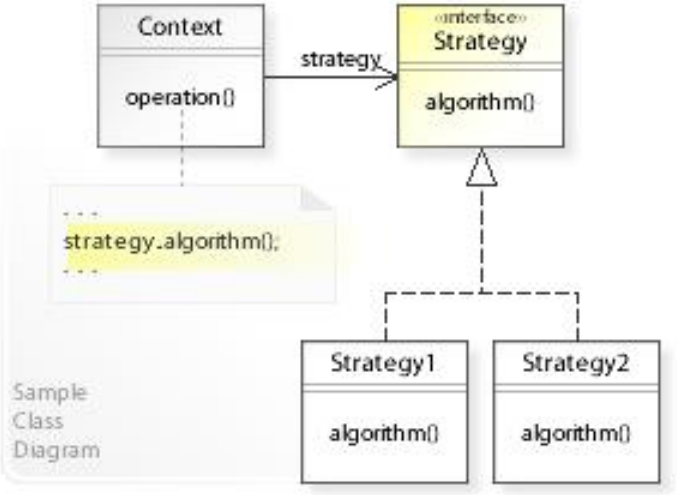
```
Rule.<Producto>rule("r1", "Rebaja un 10% los productos con fecha de caducidad cercana o pasada").
     when(pro -> Producto.getDateDiff(Calendar.getInstance().getTime(),
                                      pro.getCaducidad(),TimeUnit.DAYS) < 2 ).
     exec(pro -> pro.setPrecio(pro.getPrecio()-pro.getPrecio()*0.1))
```

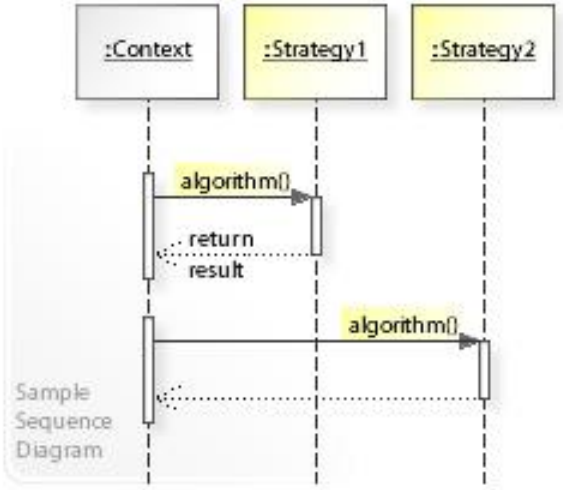
- Conjuntos de reglas (`RuleSet`)
  - Se evalúan sobre una colección de objetos del tipo de las reglas
  - Cada regla se intenta aplicar en orden sobre cada objeto de la colección

```
RuleSet<Producto> rs = new RuleSet<Producto>();
rs.add(...); // se añaden reglas
List<Producto> str = Arrays.asList(...); // una lista
rs.setExecContext(str);  // Las reglas se ejecutarán sobre str
rs.process();
```

Apartado 4: Estrategias de ejecución

- `RuleSetWithStrategy`
  - Sobre las base de `RuleSet`
  - Parametrizable con una estrategia de ejecución
  - Uso del patrón de diseño *Strategy*





- Implementar el algoritmo de Dijskstra usando reglas y los grafos del apartado 1

Apartado 5: (Opcional)

- Reglas con disparador (`TriggeredRule`)
  - Reglas asociadas a la modificación del atributo de un objeto

```
SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");

Producto p1 =     new Producto(10, sdf.parse("15/04/2020"));

TriggeredRule.<Producto>trigRule("r1").
               trigger(p1, "precio"). // se ejecuta cuando se modifica p1.precio
               when(pro ->
                           Producto.getDateDiff(Calendar.getInstance().getTime(),
                           pro.getCaducidad(), TimeUnit.DAYS) < 2 ).
               exec(pro -> { System.out.println("..."); });

p1.setPrecio(17); // dispara la regla
```