SCJP – Capitulo 8

Objetos y Colecciones

Agenda

- Sobrescribiendo hashCode() y equals()
- Colecciones
- Garbage Collection

Métodos de la clase Object

Método	Descripción
boolean **equals**(Object o)	Decide si dos objetos son equivalentes
void **finalize**()	Es llamado por el garbage collector cuando el objeto ya no puede ser referenciado
int **hashCode**() Hace un calculo	Retorna el hashcode de un objeto para ser utilizado con colecciones que usan hashing, como Hashtable, HashMap y HashSet
final void **notify**()	Despierta a una "hebra" que esta esperando por el bloqueo de un objeto
final void **notifyAll**()	Despierta a todas las "hebras" que están esperando por el bloqueo de un objeto
final void **wait**()	Causa que la "hebra" actual espere hasta que otra "hebra" llame a *notify()* o *notifyAll()*
String **toString**()	Retorna una "representación en Texto" del objeto

El método toString()

```
public class DificilDeLeer {
    public static void main(String[ ] args){
        DificilDeLeer d = new DificilDeLeer();
        System.out.println(d);
    }
}

//Resultado : DificilDeLeer@a47e0
```

Hash code

El método toString()

```
class Persona {
    private String nombre;
    public Persona(String nombre){
        this.nombre = nombre;
    }
    public String toString(){
        return “mi nombre es : ” + nombre;
    }
}
public class PersonaTest {
    public static void main(String[ ] args){
        Persona b = new Persona(“Alexis”);
        System.out.println(b); // mi nombre es : Alexis
    }
}
```

Sobreescribe cuando se usa == parametros, devolver distinto es sobrecarga

El método equals()

```
public class Nota {
    private int valor;
    public Nota(int valor){ this.valor = valor; } //constructor
    public int getValor(){ return valor; } //accesador
    public boolean equals(Object obj){
        boolean sonIguales = false;
        if (obj instanceof Nota) {
            Nota laOtraNota = (Nota) obj ; // Casting
            sonIguales = (laOtraNota.getValor() == valor);
        }
        return sonIguales;
    }
}
```

Siempre hacer este if para saber si es instancia de

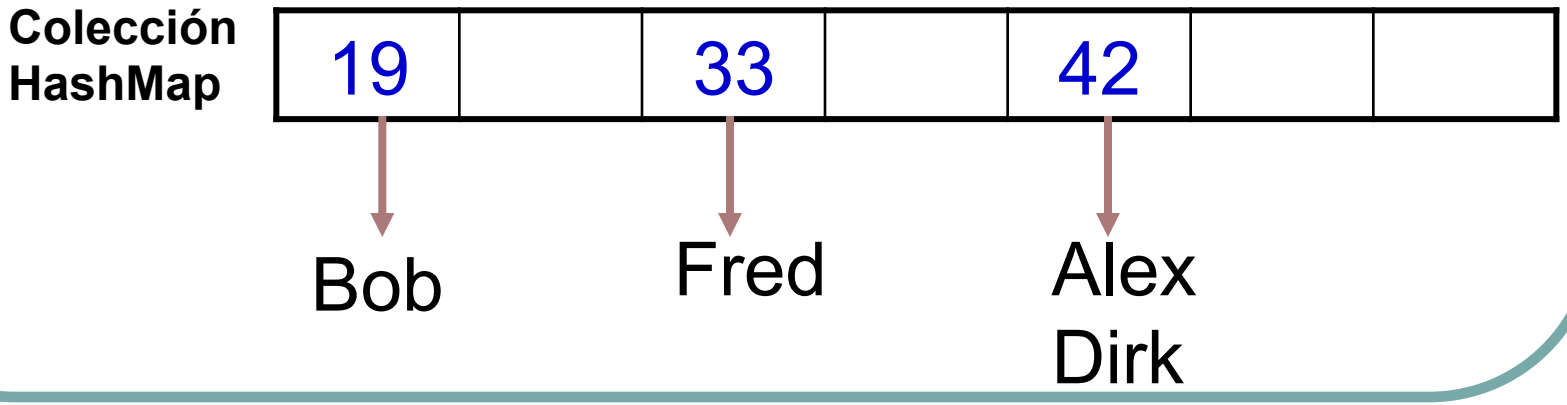
```
Nota nota1 = new Nota (7);
Nota nota2 = new Nota (7);

if (nota1.equals(nota2)) {
  // las notas son iguales
  // (o equivalentes)
}
```

El contrato de equals()

- Reflexiva: x.equals (x) es cierto.
- Simétrica: si x.equals (y) es cierto, entonces y.equals (x) debe ser cierto.
- Transitiva: si x.equals (y) es cierto, y y.equals(z) es cierto, entonces z.equals (x) es cierto.
- Consistente: Múltiples llamadas a x.equals (y) devolverá el mismo resultado.
- null: si x no es nulo, entonces x.equals (null) es falso.

El método hashCode()

Clave	Algoritmo Hashcode	Hascode
Alex	A(1) + L(12) + E(5) + X(24)	42
Bob	B(2) + O(15) + B(2)	19
Dirk	D(4) + I(9) + R(18) + K(11)	42
Fred	F(6) + R(18) + E(5) + D(4)	33



El contrato de hashCode()

- Consistente: Múltiples llamadas a x.hashCode () devuelven el mismo entero.
- Si x.equals(y) es cierto, entonces x.hashCode () = = y.hashCode () debe ser cierto.
- Si x.equals(y) es falso, entonces x.hashCode () = = y.hashCode () puede ser verdadero o falso //en el caso de Alex y Dirk

Implementando hashCode()

```
public class HacerHash{
    public int x;
    public boolean equals(Object otro){
  // se debiò hacer un instanceof
            HacerHash h = (HacerHash) otro;
            return (h.x == this.x);
    }
    public int hashCode(){
            return (x * 17);
    }
}
```

Implementando hashCode()

```
public class HacerHash{
    public int x;
    public boolean equals(Object otro){
        HacerHash h = (HacerHash) otro;
        return (h.x == this.x);
    }
    public int hashCode(){
        return 1977; // legal, pero ineficiente
    }
}
```

Implementando hashCode()

- Una sobreescritura “legal” de *hashCode()* compila y ejecuta.
- Una “adecuada” sobreescritura de *hashCode()* se apega al contrato.
- Una “eficiente” sobreescritura de *hashCode()* distribuye los resultados en una amplia gama de claves.

Implementando hashCode()

```
public class HacerHash implements Serializable {
    public int x;
    public trasient int y;
    public boolean equals(Object otro){
        HacerHash h = (HacerHash) otro;
        return (h.x == this.x && h.y == this.y);
    }
    public int hashCode(){
        return (x ^ y);
    }
}
```

Colecciones

- ¿Que se puede hacer con una colección?
  - Agregar objetos
  - Remover objetos
  - Buscar objetos
  - Iterar (recorrer)

Tipos de Colecciones

- List (lista de cosas): Orden, permite duplicados, con un índice
- Set (conjuntos de cosas): puede o no puede tener orden y / o ordenables, no se permite duplicados
- Map (mapas de las cosas con claves): puede o no puede tener orden y / o ser ordenables, claves duplicadas no son permitidas

Ejemplo Lista Ordenada

```
import java.util.*;
public class Ordered {
    public static void main (String[ ] args){
        List lst = new ArrayList();
        lst.add("Chicago");
        lst.add("Detroit");
        lst.add("Atlanta");
        lst.add("Denver");

        Iterator it = lst.iterator();
        while (it.hasNext()){
           System.out.println("city : " + it.next());
        }
    }
}
```

Salida:

```
city : Chicago
city : Detroit
city : Atlanta
city : Denver
```

Ejemplo Lista Ordenable

```
import java.util.*;
public class Ordered {
    public static void main (String[ ] args){
        Set lst = new TreeSet(); /TReeSet es un arbol ordenado
        lst.add("Chicago");
        lst.add("Detroit");
        lst.add("Atlanta");
        lst.add("Denver");

        Iterator it = lst.iterator();
        while (it.hasNext()){
           System.out.println("city : " + it.next());
        }
    }
}
```

Salida:

```
city : Atlanta
city : Chicago
city : Denver
city : Detroit
```

Tabla Resumen de Colecciones

Class	Map	Set	List	Ordered	Sorted
HasMap (clave valor)	X			No	No
Hastable (es sincronizable)	X			No	No
TreeMap	X			Ordenable	Por orden natural o comparación personalizada
LinkedHashMap (lista enlazada)	X			Por orden de inserción o último acceso	No
HashSet		X		No	No
TreeSet		X		Ordenable	Por orden natural o comparación personalizada
LinkedHashSet		X		Por orden de inserción o último acceso	No
ArrayList			X	Por índice	No
Vector			X	Por índice	No
LinkedList			X	Por índice	No

Garbage Collector

- Cuando el Garbage Collector “corre”, su propósito es buscar y borrar objetos que no pueden ser “alcanzados”

- ¿Cuándo corre el GC?
  - R: Sólo la JVM decide cuando (se puede sugerir mediante código, pero no hay garantías de que la JVM lo haga)

Objetos elegibles por el GC

```
public class ReferenciaNula{
    public static void main (String [ ] args){
        StringBuffer sb = new StringBuffer(“Hola”);
        System.out.println(sb);
        // aquí sb no es elegible por el GC
        sb = null;
        // ahora es elegible por el GC
    }
}
```

Objetos elegibles por el GC

```
public class ReasignandoReferencia {
    public static void main (String [ ] args){
        StringBuffer s1 = new StringBuffer("Hola");
        StringBuffer s2 = new StringBuffer("Chao");
        System.out.println(s1);
        // aquí s1 no es elegible por el GC
        s1 = s2;

        // ahora el puntero de s1 es elegible por el GC, es decir StringBuffer("Hola")
    }
}
```

Objetos elegibles por el GC

```
import java.util.Date;
public class FabricaDeBasura {
    public static void main(String[ ] args){
        Date d = getDate();
        hacerMasCosas();
        System.out.println("d = " + d);
    }
    public static Date getDate(){
        Date d2 = new Date();
        String now = d2.toString();
        System.out.println(now);
        return d2;
    }
}
```

Las variables locales sólo "viven" mientras el método se ejecuta:

Al finalizar el método getDate(), el objeto String "now" es elegible por el GC. Al contrario del objeto "d2", cuya referencia fue retornada por el método.

Aislamientos

```
public class Island {
    Island i;
    public static void main(String[ ] args){
        Island i2 = new Island();
        Island i3 = new Island();
        Island i4 = new Island();

        i2.i = i3; // i2 referencia a i3
        i3.i = i4; // i3 referencia a i4
        i4.i = i2; // i4 referencia a i2

        i2 = null;
        i3 = null;
        i4 = null;
        hacerMasCosas();
    }
}
```

Hay tres objetos “Islas” con referencias entre ellos…

Pero con referencia nula desde la “hebra” principal…

Estos tres objetos (i2, i3, i4) son elegibles por el GC

Forzando al GC

- En verdad, es sólo una petición de ejecución…
  - System.gc();
  - Runtime.getRuntime().gc(); //alternativa

El método finalize()

- El método *finalize( )* está garantizado a ejecutar *una vez y sólo una vez* antes de que el recolector de basura elimina un objeto.
- Dado que el recolector de basura **no** da ninguna garantía, el método *finalize( )* podría nunca ejecutarse.
- No puede dejar inelegible un objeto dentro de *finalize( )*.