SI-pass 4

Johan Brook och Jesper Persson

25 september 2012

1. Sant eller falskt?

Diskutera och svara på om påståendena nedan är äkta sanningar eller listiga lögner.

- Interfaces i Java kan innehålla privata instansvariabler
- Man kan instansiera abstrakta klasser
- Abstrakta metoder i en klass måste implementeras av subklasser
- En klass kan ärva från två andra klasser
- Ett interface kan ärva från ett annat interface
- Ett interface kan ärva från flera interface

2. Modellera mera!

Att diskutera: När är det läge att använda:

- en vanlig klass?
- en abstrakt klass?
- arv?
- ett interface?
- en inre klass?

Ge gärna exempel för varandra.

Att göra:

- Utgå från specifikationen nedan och modellera ett **vapensystem** (kanske kan användas i ett spel). Använd UML för att visualisera sambanden samt ev. arv. Ni får gärna skriva ner kod om ni vill, kanske för att förklara för varandra.
- Implementera korrekta `equals`-metoder där det behövs.
- Vad blir utskriften i testprogrammet längst ner i uppgiften? (enligt er implementation)

Följande modeller ska finnas med, och ha instansvariabler och metoder (nästlade listorna):

- `Sword`
  - `isTwoHanded:boolean`
  - `+ use():void`
- `RocketLauncher`
  - `- ammo:int`
  - `- range:int`
  - `- caliber:int`
  - `- splashArea:int`
  - `+ getAmmo:int`
  - `+ setAmmo(ammo:int):void`
  - `+ use():void`
  - `+ fire():void`
- `ShotGun`
  - `- ammo:int`
  - `- range:int`
  - `- caliber:int`
  - `- bulletSpread:int`
  - `+ getAmmo:int`
  - `+ setAmmo(ammo:int):void`
  - `+ use():void`

- + fire():void

Ni har tillgång till följande interface:

```
/**
* An equippable object.
*/
interface Equippable {

	/**
	* Characters can call this method to invoke an action
	* on the equippable object.
	*/
	public void use();
}
```

Testprogrammet

```
Equippable sword = new Sword();
Equippable rocket = new RocketLauncher(4, 20, 200, 10);
Equippable rocket2 = new RocketLauncher(4, 20, 200, 11);

// Test different weapons
System.out.println("Comparing a sword with a rocket. Should be false");
System.out.println(sword.equals(rocket));

// Test the same weapon
System.out.println("Comparing two rockets. Should be false");
System.out.println(rocket.equals(rocket2));

Player p = new Player();
p.setThing(rocket);
p.thing.use();

p.setThing(sword);
p.thing.use();
```

3. Interface, arv, equals

I alla system och programmeringsspråk vill man kunna jämföra om värden och objekt är likadana. För primitiva datatyper har man operatorn (“==”) som jämför på likhet, men den fungerar inte alltid som man tänkt för klasstyper.

Se gärna närmre på koden hemma om ni inte hinner klart här. Det är ett schysst exempel på hur man kan bygga ett system av lyssnare, och användning av abstrakta klasser och interfaces.

Att göra: Betrakta systemet nedan (bli inte avskräckta av mängden kod :)).

1. Fyll i de tomma metoderna (märkt med **“Implement this!”**).
2. Vad skrivs ut när man kör klassen `ButtonSystem`?
3. Beskriv flödet av koden med egna ord och förklara för varandra om det behövs. *Se till att ni har grepp om vad som händer*. Varför används abstrakta klasser och interface som de gör?

```
public class ButtonSystem {
	public static void main(String[] args) {
		new ButtonSystem();
	}

	// Main program

	public ButtonSystem() {

		Button addButton = new Button("Add", 20, 20);
		Button deleteButton = new Button("Delete", 40, 40);

		deleteButton.setListener(new DeleteListener());
		addButton.setListener(new AddListener());

		// Simulate clicks

		deleteButton.click();
		System.out.println("********");
		addButton.click();
		System.out.println("********");

		System.out.println("Are the buttons equal?");
		System.out.println(addButton.equals(deleteButton));
	}

	private class AddListener implements Listener {
		public void fire(Event evt) {
			System.out.println("Added thing to system with event " + evt);
			System.out.println("The event came from " + evt.getSource());
		}
	}

	private class DeleteListener implements Listener {
		public void fire(Event evt) {
			System.out.println("Delete thing from system with event " +
evt);
			System.out.println("The event came from " + evt.getSource());
		}
	}
}


 abstract class AbstractButton {
```

```
	private Listener listener = null;
	private String label;
	private int width;
	private int height;

	private static int CLICKS = 0;

	public AbstractButton(String label) {
		this(label, 20, 10);
	}

	public AbstractButton(String label, int width, int height) {
		this.label = label;
		this.width = width;
		this.height = height;
	}

	public void setListener(Listener l) {
		this.listener = l;
	}

	public String toString() {
		// ****************
		// Implement this!

		return "<RETURN NICE STRING HERE YES>";
	}

	public void click() {
		if(this.listener != null) {
			int newID = ++CLICKS;
			Event event = new Event(newID, this);

			this.listener.fire(event);
		}
	}

	public boolean equals(Object obj) {
		// ***************
		// Implement this!

		return true;
	}
}

class Button extends AbstractButton {

	public Button(String label) {
		super(label);
	}

	public Button(String label, int width, int height) {
		super(label, width, height);
	}
}

class Event {
	private Object source;
	private int id;

	public Event(int id, Object source) {
```

```
            this.id = id;
            this.source = source;
      }

      public Object getSource() {
            return this.source;
      }

      public String toString() {
            // ***************
            // Implement this!

            return "<RETURN NICE STRING HERE YES>";
      }
}


interface Listener {
      public void fire(Event evt);
}
```