Rethinking OOP in Snap!

Prototypes, Polymorphism & Pedagogy

Jens Mönig
Snap! Team
SAP SE
Walldorf, Germany
jens.moenig@sap.com

Brian Harvey
EECS
University of California, Berkeley
Berkeley, USA
bh@berkeley.edu

Jadga Hügle
Snap! Team
SAP SE
Walldorf, Germamy
jadga.huegle@sap.com

ABSTRACT

Many current novice programming environments offer a "sprite"-based microworld, in which cartoon-like actor objects interact with each other and the user by way of events triggering the execution of stacks of "blocks" representing programming statements. While such "sprites" can be seen as something akin to "objects" in professional programming languages they are for the most part lacking features that are widely considered essential for learning about "real" OOP, in particular the concepts of classes, inheritance and polymorphism. We have tried to address this void by extending our Snap! programming language with prototypical inheritance for sprites. In this talk we will demonstrate how learners can explore traditional OOP concepts for abstraction beginning from concrete sprites, clones and prototypes. We will also share some preliminary thoughts and experiments on a revised curriculum pathway for introducing OOP in schools.

KEYWORDS

OOP, inheritance, polymorphism, prototypes, delegation, Scratch, Snap, BJC

1 Object-Based vs. Object-Oriented

Sprites in Scratch and the many programming environments imitating it have striking similarities with "objects" in professional programming languages: They bundle state, such as x- and y-positions, size, heading, internal variables etc. and behavior ("scripts" reacting to events, custom blocks defined "for this sprite only"). However, sprites lack fundamental features that make OOP attractive for professional programmers. In particular Scratch-like sprites cannot be composed into larger units, nor can their properties be abstracted into "blue-prints" for a group or "kind" of similar specimens. Because of this design choice Scratch and its derivatives are sometimes referred to as "object-based" environments, rather than "object-oriented" ones, which also feature classes, inheritance, message-passing and polymorphism, as well as means for encapsulating internal information [1].

The lack of OOP features in Scratch-like novice programming environments is a problem, because many curriculum frameworks, e.g. in German states, require students to learn professional OOP concepts as early as 7th grade in school. As a consequence, blocks-based programming environments are used for only a very short first introduction to what is often called "coding", before even young students are turned towards studying professional text-based programming languages such as Java, that offer a classical take on OOP but also lack all the supportive scaffolding of Scratch and its dialects.

2 Closures and Dispatch-Procedures

Since Snap! features lexically scoped first-class procedures we have been able to demonstrate and teach dispatch-procedure-style OOP for some years. This is made possible by an implementation that retains a function's original environment (called "context" in Snap!) even after the function has terminated in such cases where it returns another function, thus creating a "closure" side effect.

Figure 1: Creating an anonymous dispatch procedure that serves as a circular buffer object

Calling the returned "reified" function gives it access to the otherwise unreachable environment of its originating function. This way closurized state can be associated with and shared by functions to the same effect as objects bundle internal state with methods operating on it [2]. We use this method to demystify objects by showing the classic "counter" example, but also to create more complex objects such as a circular buffer for sound-synthesis. However, dispatch-procedure-style OOP is admittedly an advanced concept of functional programming and often beyond the abilities of beginners. Also, sprites in Snap! are built-in affordances and therefore cannot be created using this method. Therefore, we felt the need to make "objects" and "inheritance" more accessible to novices by extending Snap's sprite-microworld with OOP concepts.

3 Prototypical Inheritance

Over the past two years we have begun to extend our Snap! programming language with a kind of prototypical inheritance among sprites that is inspired by Henry Lieberman's delegation model [3]. Rather than abstracting traits common to a group of sprites into a "class" like blue-print we introduced arbitrarily deep parent-child relationships among sprites, in which children can inherit certain attributes from their parent. Within such a prototype-clone relationship, children not only assume a parent-sprite's structure, i.e. the slot-names for field-variables, but also dynamically inherit their current value. Dynamic inheritance of values can be compared to class-variables in traditional OO languages such as Smalltalk. The difference is that children may override inherited values with their own ones, thus severing the inheritance-chain on a per-slot basis rather than as a whole. Children can also restore dynamic inheritance per slot, and even do so programmatically.

The same rules apply not only to sprite-only "field"-variables but also to sprite-local custom block definitions and even to built-in visual attributes, such as x- and y-position, size, costume etc. This way, the child inheriting the y-position of its parent turtle-sprite moving in circles can draw a sine-curve simply by repeatedly changing its x-position at constant speed.

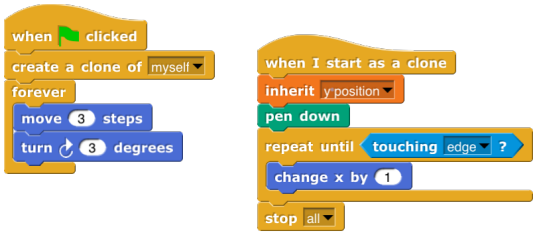

Figure 2: Using a clone and inheritance of the parent's y-position to draw a sine-curve

Also, since Snap! is always "live" even when no script is running, that child-sprite will even follow its parent's y-position when the user drags the parent-sprite around with the mouse. With our design we aim to support exploring powerful concepts such as kind-of relationships through playful tinkering with concrete sprites in a reactive environment.

4 Nesting Sprites

While "kind-of" relationships facilitate code reuse by grouping sprites according to their similarities into classes or prototypical "tribes", "part-of" relationships enable assembling complex structures out of simple components. Nesting sprites has long been supported by Snap! In our recent development we have added ways to nest sprites programmatically using blocks, and to also programmatically create and modify parent-child (inheritance) relationships. By supporting both modes - letting the learner assemble composite sprites and create inheriting descendants interactively using the IDE, as well writing a blocks-script to do it programmatically - we aim to provide a gentle segue for beginners to progress towards more advanced meta-programming techniques as they feel comfortable to explore more.

5 Shifting the Perspective on OOP

As we are extending Snap! with Lieberman-style prototypical inheritance and meta-programming capabilities we are also designing learning activities that we hope will make high level OOP concepts more explorable and also more fun for kids and casual coders. This year we have conducted a number of workshops both for teachers and children in which we have tried various aspects. Our evidence so far is only anecdotal, including a second-hand observation by Eckart Modrow [4] (Uni Göttingen), that Snap's prototypical inheritance model seems to lend itself quite well to learn the full OOP syllabus required by the curriculum framework of the German state of Lower-Saxony. We've also begun to shift our "story line" from the classic triad of "inheritance, polymorphism, encapsulation" to a string of more concrete examples experiencing "first-class-ness, state + behavior, relationships".

6 Outlook

At the time of writing this not all visual attributes of Snap's sprites participate in inheritance, e.g. rotation-style, pen attributes (color, size, shade, down-state). Efforts are ongoing to create and asses learning materials and evaluate the language design.

ACKNOWLEDGMENTS

Thanks to Henry Lieberman for his enthusiastic intellectual and emotional support

REFERENCES

[1] Ingalls, Design Principles Behind Smalltalk, BYTE Magazine, August 1981
[2] Abelson, Sussman, Structure and Interpretation of Computer Programs, MIT Press, Cambridge, MA
[3] Lieberman, Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems, MIT AI LAB, 1985
[4] Modrow, Computer Science with Snap!, Uni Göttingen, 2018