LEARNABLE PROGRAMMING

배우기 쉬운 프로그래밍

번역중

10/08 스피드 번역 **-** 오늘밤 **10**시 **-** 수고하셨습니다**!!**

수다는 요기로.. (구글문서 채팅창이 오류나는 관계로..)
http://www.facebook.com/events/437230242999660/

전체 작업일지를 보시려면 이곳으로~
찜하기 표시 참여방법
번역 참여방법
참여하신 분
<1차 번역작업한 예시>
찜하세요~ 001 : Nassol 찜 & 작업완료 & 리뷰 완료
찜하세요~ 002 : sahoon 찜 & 작업끝 리뷰요망
찜하세요~ 003 : 끝 리뷰요망!
찜하세요~ 004 : Wony 찜 & 작업 끝 & 리뷰요망
찜하세요~ 005 : JunYoung 찜 & 작업끝 리뷰요망
찜하세요~ 006 : 작업완료!
찜하세요~ 007 : 작업완료!
찜하세요~ 008 : nassol 찜 & 작업완료
찜하세요~ 009 : Wony 찜 & 작업 끝 & 리뷰 끝
찜하세요~ 010 : nassol 찜 & 작업완료 & 확인필요
찜하세요~ 011 : sahoon 찜 & 작업끝 리뷰요망 & nassol 리뷰
찜하세요~ 012 : sahoon 찜 & 작업끝 리뷰요망 & nassol 리뷰
찜하세요~ 013 : sahoon 찜 & 작업완료 리뷰요망 & nassol 리뷰완료
찜하세요~ 014 : Wony 찜 & 작업 끝
찜하세요~ 015 : Wony 찜 & 작업 끝
찜하세요~ 016 : Wony 찜 & 작업 끝 & 리뷰요망 & 나솔 리뷰

찜하세요~ 017 : Wony 찜 & 작업 끝 & 리뷰요망
찜하세요~ 018 : 유영호찜 & 작업완료 & 리뷰 요망 & 리뷰 완료
찜하세요~ 019 : 김소혜찜 & 작업완료 & 리뷰요망
찜하세요~ 020 : 김소혜찜 & 작업완료 & 리뷰요망
찜하세요~ 021 : nassol 찜 & 작업중
찜하세요~ 022 : 김소혜찜 & 작업완료 & 리뷰요망
찜하세요~ 023 : 김소혜찜 & 작업완료 & 리뷰요망
찜하세요~ 024 : 유영호찜 & 작업완료 & 리뷰요망
찜하세요~ 025 : whdnfl21 찜 & 작업완료 & 리뷰요망
찜하세요~ 026 : 김훈민 찜 & 작업완료 & 리뷰요망
찜하세요~ 027 : 김훈민 찜 & 작업완료 & 리뷰요망
찜하세요~ 028 : Justin 찜 & 작업 완료 & 리뷰 요망
찜하세요~ 029 : Justin 찜 & 작업 완료 & 리뷰 요망
찜하세요~ 030 : 사훈 찜 & 작업완료 & 리뷰요청
찜하세요~ 031 : sahoon 찜 & 작업중 & 리뷰요청
찜하세요~ 032 : sahoon 찜 & 작업완료 & 리뷰요청
찜하세요~ 033 : 유영호찜 & 작업완료 & 리뷰요망.
찜하세요~ 034 : Justin 찜 & 작업 완료 & 리뷰 요망
찜하세요~ 035 : Justin 찜 & 작업 완료 & 리뷰 요망
찜하세요~ 036 : 김훈민 찜 & 작업완료& 리뷰요청
찜하세요~ 037 : 김훈민 찜 & & 작업완료& 리뷰요청
찜하세요~ 038 : 김병진 발번역
찜하세요~ 039 : 김병진 발번역
찜하세요~ 040 : 김병진 발번역
찜하세요~ 041 : nassol 찜 & 작업완료
찜하세요~ 042 : nassol 찜 & 작업완료 & 리뷰 요청
찜하세요~ 043 : nassol 찜 & 작업완료
찜하세요~ 044 : nassol 찜 & 작업완료
찜하세요~ 045 : nassol 찜& 작업완료
찜하세요~ 046 : nassol 찜 & 작업중
찜하세요~ 048 : 김소혜찜 & 작업중
찜하세요~ 049 : 김소혜찜 & 작업중
찜하세요~ 050 :김훈민 찜 & 작업완료 & 리뷰요망
찜하세요~ 051 : 김훈민 찜 & 작업완료 & 리뷰요망
찜하세요~ 052 : 누렁이 찜 & 작업완료 & 리뷰완료
찜하세요~ 053 : 누렁이 찜 & 작업끝 &리뷰완료
찜하세요~ 055 : 누렁이 찜 & 진행중
찜하세요~ 056 : 누렁이 찜
찜하세요~ 057 :
찜하세요~ 058 :
찜하세요~ 059 :
찜하세요~ 060 :
찜하세요~ 061 :

찜하세요~ 062 :
찜하세요~ 063 :
찜하세요~ 064 :
찜하세요~ 065 :
찜하세요~ 066 :
찜하세요~ 067 :
찜하세요~ 068 :
찜하세요~ 069 :
찜하세요~ 070 :
찜하세요~ 071 :
찜하세요~ 072 :
찜하세요~ 073 :
찜하세요~ 074 :
찜하세요~ 075 :
찜하세요~ 076 :
찜하세요~ 077 :
찜하세요~ 078 :
찜하세요~ 079 :
찜하세요~ 080 :
찜하세요~ 081 :
찜하세요~ 082 :
찜하세요~ 083 :
찜하세요~ 084 :
찜하세요~ 085 :
찜하세요~ 086 :
찜하세요~ 087 :
찜하세요~ 088 :
찜하세요~ 089 :
찜하세요~ 090 :
찜하세요~ 091 :
찜하세요~ 092 :
찜하세요~ 093 :
찜하세요~ 094 :
찜하세요~ 096 :
찜하세요~ 097 :
찜하세요~ 098 :
찜하세요~ 099 :
찜하세요~ 100 :
찜하세요~ 101 :
찜하세요~ 102 :
찜하세요~ 103 :
찜하세요~ 104 :
찜하세요~ 105 :

찜하세요~ 106 :
찜하세요~ 107 :
찜하세요~ 108 :
찜하세요~ 109 :
찜하세요~ 110 :
찜하세요~ 111 :
찜하세요~ 112 :
찜하세요~ 113 : 장재원찜 & 작업완료 & 리뷰요망
찜하세요~ 114 :
찜하세요~ 115 :
찜하세요~ 116 :
찜하세요~ 117 :
찜하세요~ 118 :
찜하세요~ 119 :
찜하세요~ 120 :
찜하세요~ 121 :

전체 작업일지를 보시려면 이곳으로~

http://opentutorials.org/course/245/2903

번역할 원문 : http://worrydream.com/LearnableProgramming/

찜하기 표시 참여방법

1. 찜하기 표시 안된 부분을 찾습니다.
2. 한 번 번역하는 분량이 부담되지 않도록, 약, 5-10줄 이내로 분량을 선택해서, 시작하는 쪽에다가 찜하세요~ 를 표시합니다. **(예: 찜하세요~ 001 :)**
3. 줄을 클릭한 상태에서, 서식을 일반텍스트=>제목**1**로 바꿉니다.
4. 문서 젤 위인 목차로 가서 동그라미 모양의 화살표를 클릭해서 목차를 업데이트합니다.

번역 참여방법

1. 찜하세요 옆에 이름을 씁니다.
2. 번역합니다. : 1차 번역작업한 예시 참고..

(참 번역하시기 전에, 원문http://worrydream.com/LearnableProgramming/ 페이지에서 내가 번역하려는 부분을 함 살펴보면서 그림도 같이 보고 하시면 좋겠죠? ^^
3. 개영에 가서 했다고 자랑합니다(또는 표현 제안해달라고, 함 봐달라고 부탁합니다.) 아참, 번역하신 부분의 번호를 알려주시거나 고 부분 링크를 달아주시면 좋겠죠 :) http://www.facebook.com/groups/engfordev/

참여하신 분

Gist에 옮기는 작업

- Tw Shim

찜하기 표시하는 작업
- 나솔

1차 번역

- 나솔
- 사훈
- Wony
- 유영호
- 김소혜
- Justin
- Junyoung
- 누렁이
- 김훈민

10월 8일 스피드 번역 참여
- Justin 유영호 김사훈 지영배 Dongwoo Kim 김소혜 Nasol

리뷰, 확인 및 표현 제안

- 숟가락(장재원)

지켜보는 역할
- 지영배

<1차 번역작업한 예시>

찜하세요~ 001 : Nassol 찜 & 작업완료 & 리뷰 완료

LEARNABLE PROGRAMMING
배우기 쉬운 프로그래밍

(좋은 표현이 바로 떠오르지 않으시면 너무 고민하지 마시고, 선택 + 마우스 우클릭해서 메모로 남겨주세요.)

Designing a programming system for understanding programs
프로그램을 이해하는데 도움을 주는 프로그래밍 시스템을 설계하기.

(한국어 부분이 잘 보이도록, 색깔을 파랗게 바꾸어주세요~0

- [Bret Victor](http://worrydream.com/) / September 2012
- [원문](http://worrydream.com/LearnableProgramming/)

(#, ##, [ ] , ( ) 등은 markdown 형식이니, 번역문에도 고스란히 넣어 주세요~)

Here's a trick question: How do we get people to understand programming?
"어떻게 하면 사람들이 프로그램을 이해할 수 있을까요? 참 알쏭달쏭한 문제입니다.

Khan Academy recently launched an [online environment](http://www.khanacademy.org/cs) for learning to program. It offers a set of tutorials based on the JavaScript and Processing languages, and features a "live coding" environment, where the program's output updates as the programmer types.

최근 칸 아카데미에서 프로그램 학습을 위한 [online environment](http://www.khanacademy.org/cs) 를 선보였습니다. 이곳에서는 자바스크립트와 프로세싱 언어로 진행되는 강좌를 제공합니다. 그리고 "실제로 코딩"해볼 수 있는 환경도 제공합니다. 프로그래머가 타이핑을 하면 프로그램의 출력결과가 바로바로 업데이트 됩니다.

찜하세요~ **002 : sahoon** 찜 **&** 작업끝 리뷰요망

-- 리뷰하는 방법 **:** 전번호부터 한국어로 쭉 읽으면서 대략 흐름을 파악합니다.
리뷰하려는 글을 한국어로 읽으면서 흐름이 자연스럽게 이어지는지 확인합니다.
어색하면, 표현을 제안합니다. **--**

Because my work was cited(link: http://ejohn.org/blog/introducing-khan-cs) as an inspiration for the Khan system, I felt I should respond with two thoughts about learning:
나의 개발작업은 [사이트](http://ejohn.org/blog/introducing-khan-cs) 칸 시스템에 영향을 받았기 때문에 나는 학습에 대해 두가지 생각으로 답해야할 것 같습니다.

- **Programming is a way of thinking, not a rote skill.** Learning about "for" loops is not learning to program, any more than learning about pencils is learning to draw.
- **People understand what they can see.** If a programmer cannot see what a program is doing, she can't understand it.
- **프로그래밍은 기계적인 기술이 아니라 생각하는 방법입니다.** "for" 반복문을 배우는것은 프로그래밍을 배우는 것이아닙니다. 마치 드로잉을 배우는것이 연필을 배우는것이 아닌것 처럼요.

(리뷰) - 확인했습니다. 자연스럽고 좋네요!

- **사람들은 직접 보았을 때 이해를 잘 합니다. ** 자기가 만든 프로그램이 무엇을 해내는지 보지 못하면, 프로그래밍하는 것을 이해하지 못합니다.

(리뷰-표현제안) - ** 사람들은 직접 보았을 때 이해를 잘 합니다. ** 자기가 만든 프로그램이 무엇을 해내는지 보지 못하면, 프로그래밍하는 것을 이해하지 못하죠.

찜하세요~ **003 :** 끝 리뷰요망**!**

Thus, the goals of a programming system should be:

- to support and encourage powerful ways of thinking
- to enable programmers to see and understand the execution of their programs

따라서, 프로그래밍 설계의 목표는 이래야 합니다:

- 생각이 풍성해지도록 지원하고 격려하는 것

- 프로그래머가 실행 결과를 보고 이해할 수 있을 것

A live-coding Processing environment addresses neither of these goals. JavaScript and Processing are poorly-designed languages that support weak ways of thinking, and ignore decades of learning about learning. And live coding, as a standalone feature, is worthless.

프로세싱의 라이브 코딩 환경은 이들 목표중에 어느것도 배려하고 있지 않습니다. 자바스크립트와 프로세싱은 생각의 지원에 빈약하게 설계된 언어였고 수년간 쌓인 배움을 대한 지식을 무시해 왔었습니다. 그리고 독립적인 기능으로써의 라이브코딩은 가치가 없었죠.

(숟가락_제안) JavaScript와 Processing은 둘 다 고유명사니까 원어로 표기하면 어떨까요?

찜하세요~ **004 : Wony** 찜 **&** 작업 끝 **&** 리뷰요망

Alan Perlis wrote, *"To understand a program, you must become both the machine and the program."* This view is a mistake, and it is this widespread and virulent mistake that keeps programming a difficult and obscure art. A person is not a machine, and should not be forced to think like one.

앨런 펄리스는 *"프로그램을 이해하기 위해서는 당신 스스로 기계와 프로그램, 그 자체가 되어야 한다"*라고 했습니다. 이러한 관점은 프로그래밍을 어렵고 이해하기 힘들도록 만드는 실수, 그것도 치명적이고 널리 퍼져버린 실수입니다. 사람은 기계가 아니므로 기계처럼 생각하도록 강요받아서는 안됩니다.

How do we get people to understand programming?

그렇다면, 어떻게 사람들이 프로그램을 이해할 수 있도록 할까요?

We change programming. We turn it into something that's understandable by people.

우리가 프로그래밍을 바꿀 것입니다. 우리는 그것을 사람들로 하여금 이해할 수 있는 것으로 만들 것입니다.

찜하세요~ **005 : JunYoung** 찜 **&** 작업끝 리뷰요망

Contents

A programming system has two parts. The programming "environment" is the part that's installed on the computer. The programming "language" is the part that's installed in the programmer's head.

목차

프로그래밍은 두 부분으로 이루어집니다. 프로그래밍 '환경'은 컴퓨터 안에 있는 부분입니다. 프로그래밍 '언어'는 사람의 머릿속에 있습니다.

This essay presents a set of design principles for an environment and language suitable for learning.

이 문서에서는 프로그래밍을 배우기에 적합한 환경과 언어를 설계하는 원칙을 제시합니다.

찜하세요~ 006 : 작업완료!

The **environment** should allow the learner to:

- **read the vocabulary** -- *what do these words mean?*
- **follow the flow** -- *what happens when?*
- **see the state** -- *what is the computer thinking?*
- **create by reacting** -- *start somewhere, then sculpt*
- **create by abstracting** -- *start concrete, then generalize*

환경은 학습자가 다음을할 수 있도록 해야합니다:

- **어휘를 읽기** -- *저 단어들은 어떤 뜻이지?*
- **흐름을 파악하기** -- *언제 어떤 일이 일어났지?*
- **상태를 파악하기** -- *컴퓨터는 어떤 생각을 하지?*
- **반응을 중심으로 만기** -- *일단 시작하고 조각하기*
- **추상화를 중심으로 만들기** -- *기반을 다지고 일반화하기*

찜하세요~ 007 : 작업완료!

The **language** should provide:

- **identity and metaphor** -- *how can I relate the computer's world to my own?*
- **decomposition** -- *how do I break down my thoughts into mind-sized pieces?*

- **recomposition** -- *how do I glue pieces together?*
- **readability** -- *what do these words mean?*

언어는 다음을 제공 해야합니다:

- **아이덴티티, 메타포** -- *어떻게 컴퓨터 세상을 나의 것 으로 할 수 있을까?*
- **분해** -- *어떻게 나의 생각을 분해해 의미 단위 조각으로 만들수 있을까?*
- **재구성** -- *어떻게 조각들을 붙일 수 있을까?*
- **가독성** -- *저 언어들은 어떤 의미가 있을까?*

찜하세요~ **008 : nassol** 찜 **&** 작업완료

The features are not the point

We often think of a programming environment or language in terms of its *features* -- this one "has code folding", that one "has type inference". This is like thinking about a book in terms of its *words* -- this book has a "fortuitous", that one has a "munificent". What matters is not individual words, but how the words together convey a *message*.

중요한 건 기능이 아닙니다.

프로그래밍 환경이나 언어에 대해 생각할 때, *기능*을 위주로 생각하기 쉬운데요, 예를 들면 이 프로그래밍 환경에서는 “코드 접는 기능”이 있고, 저 프로그래밍 환경에서는 “데이터형을 추측해주는 기능”이 있다고 얘기하죠. 그런데 이것은 마치, 책에 대해서 이야기할 때, *단어*를 위주로 얘기하는 것이나 마찬가지입니다. 예를 들어, 책에 대해 이렇게 얘기하는 거죠. 이 책에는 “우연한"이라는 단어가 들어 있고, 저 책에는 “후한"이라는 단어가 들어있다고 하는 거에요. 하지만 책에 대해 중요한 것은 개별적인 단어가 아니라, 단어들이 모여서 어떤 *메세지*를 전달하느냐 입니다.

찜하세요~ **009 : Wony** 찜 **&** 작업 끝 **&** 리뷰 끝

Likewise, a well-designed system is not simply a bag of features. A good system is designed to encourage particular **ways of thinking**, with all features carefully and cohesively designed around that purpose.

마찬가지로 잘 설계된 시스템은 단순히 여러 기능들을 모아 놓은 것이 아닙니다. 모든 기능들이 특정한 사고방식을 유도하도록 정교하고 일관성 있게 설계된 것이야말로 좋은 시스템입니다.

This essay will present many features! The trick is to see *through* them -- to see the underlying design principles that they represent, and understand how these principles enable the programmer to think.

이 에세이 역시 많은 기능들을 제공합니다! 중요한 점은 그 기능들을 속속들이 들여다보는 것, 즉, 그 기능들이 표방하는 저변에 깔려있는 설계 원칙들을 파악하고, 그 설계 원칙들이 어떻게 프로그래머들로 하여금 생각하도록 하는 지를 이해하는 것입니다.

찜하세요~ **010 : nassol** 찜 **&** 작업완료 **&** 확인필요

READ THE VOCABULARY

Here is a simple tutorial program that a learner might face:

Before a reader can make any sense of this code, before she can even *begin* to understand how it works, here are some questions she will have:

READ THE VOCABULARY (??)

코드에 나오는 표현들을 읽게 하기

다음은 프로그래밍 학습자가 접할 만한 간단한 튜토리얼 프로그램입니다 :

이 코드에 대해서 이해하기 전에, 그리고 프로그램의 작동방식에 대해 이해하기 *시작*하기도 전에, 학습자는 다음에 나오는 것들이 궁금할 것입니다.

찜하세요~ **011 : sahoon** 찜 **&** 작업끝 리뷰요망 **& nassol** 리뷰

Khan Academy's tutorials encourage the learner to address these questions by *randomly adjusting numbers* and trying to figure out what they do.

칸 아카데미 튜토리얼에서는 학습자들이 위에서 제기한 질문에 접근하도록 다음의 방식을 써보게 합니다. *숫자를 임의로 바꿔보고 결과 나오는 것을 보면서* 이 부분이 어떤 기능을 하는지 이해하게 하기 위함입니다.

(표현제안) 칸 아카데미 튜토리얼에서는 학습자들이 위에서 제기한 질문에 접근하도록 다음의 방식을 써보게 합니다. *숫자를 임의로 바꿔보고 결과 나오는 것을 보면서* 이 부분이 어떤 기능을 하는지 이해하게 하기.

Thought experiment. Imagine if you bought a new microwave, took it out of the box, and found a panel of unlabeled buttons.

생각해보세요. 전자렌지를 새로 샀는데 버튼에 아무것도 적혀 있지 않아요.

찜하세요~ **012 : sahoon** 찜 **&** 작업끝 리뷰요망 **& nassol** 리뷰

Imagine if the microwave encouraged you to randomly hit buttons until you figured out what they did.
무작위로 버튼을 눌러보면서 이 버튼의 기능이 뭔지 알아내도록 전자렌지가 유도하는 거죠

Now, imagine if your cookbook advised you that randomly hitting unlabeled buttons was *how you learn cooking*.
자, 이번에는 이런 경우를 생각해보죠. 요리책을 봤더니 이렇게 적혀 있는 거에요. “라벨 없는 버튼을 이것저것 눌러보는 것, 바로 이것이 *요리를 배우는 방법*”입니다..

찜하세요~ **013 : sahoon** 찜 **&** 작업완료 리뷰요망 **&**

nassol 리뷰완료

Make meaning transparent
알기쉽게 하기
(표현제안) 의미가 잘 드러나게 하기

Learning cooking is not about guessing the functionality of your kitchen appliances. It's about understanding how ingredients can be combined.
요리를 배우는 것은 당신의 주방 가전의 기능을 추측하는 것이 아닙니다. 그것은 어떻게 요리의 재료가 잘 섞여 요리가 될 수 있는지를 이해하는 것입니다.

(표현제안) 요리를 배운다는 것은 무엇을 의미할까요? 주방 가전의 기능을 배운다는 의미일까요? 그렇지 않습니다. 요리를 배운다는 것은 여러가지 재료를 어떻게 조합할 수 있는지를 이해하는 것입니다.

Likewise, guessing the third argument of the "ellipse" function isn't "learning programming". It's simply a *barrier* to learning. In a modern environment, memorizing the minutia of an API should be as relevant as memorizing times tables.
마찬가지로 "타원" 함수의 세 번째 변수를 추측하는 것은 프로그래밍을 배우는 것이라고 할 수 없습니다. 그것은 오히려 프로그래밍을 배우는데 있어서 *걸림돌*일 뿐입니다. 현대적인 환경에서는 API의 세부사항을 기억하는 것은 시간표를 기억하는 것 만큼 관련성이 있어야 합니다.

(표현제안) 프로그래밍도 마찬가지입니다. “타원" 함수의 세번째 변수를 추측한다는 것이 프로그래밍을 배우는 것을 의미하지는 않습니다. 오히려 프로그래밍을 배우는 데 “걸림돌"이 될 수 있죠. 오늘날 프로그래밍을 할 때에는, 나에게 꼭 필요할 때에만 API의 세부 사항을 기억해야 합니다. 마치 나에게 의미있는 시간표를 기억하는 것처럼 말입니다.

찜하세요~ **014 : Wony** 찜 **&** 작업 끝

The environment is responsible for making meaning transparent. The environment must enable the reader to *effortlessly read the program*, to decode the code, so she can concentrate on genuine programming concepts -- how the algorithmic "ingredients" combine.

바람직한 프로그래밍 환경에서는 각 구성 요소들의 의미가 알기 쉬워야 합니다. 환경은 프로그래머가 큰 노력을 들이지 않고도 프로그램을 읽고 해석하도록 함으로써 보다 근본적인 프로그래밍 개념, 즉 어떻게 알고리즘 요소들이 결합하는 지에 집중할 수 있도록 해야 합니다.

(표현제안) 프로그래밍 환경에서는 각 구성 요소들의 의미가 알기 쉽도록 해야 합니다. 그리고 프로그래머가 큰 노력을 들이지 않고도 프로그램을 읽고 해석하도록 지원해야 합니다. 그래야

근본적인 프로그래밍 개념, 즉 알고리즘 요소들이 어떻게 결합하는지에 집중할 수 있습니다.

Here is one example of how a programming environment can make meaning transparent, by providing labels on mouse-over:

구성 요소에 마우스를 갖다 대었을 때 그에 대한 설명을 제공함으로써 의미를 명료하게 하는 프로그래밍 환경의 예시를 참고하세요.

(표현제안) 구성 요소의 의미를 알기 쉽게 하는 방법으로 다음의 예를 참고하세요. 구성 요소에 마우스를 갖다 대었을 때 그에 대한 설명을 제공해서 의미를 명료하게 한 예입니다.

찜하세요~ **015 : Wony** 찜 **&** 작업 끝

```
<div class="example" data-verbose-button="1" style="left: 0px; ">
  <video width="640" height="110" preload="">
    <source
src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/Vocab6.mp4"
type="video/mp4">
    <source
src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/Vocab6.webm
" type="video/webm">
  </video>
<div class="videoChrome" style="display: block; width: 640px; height: 110px; top: 0px; left: 0px;
">
    <div class="marker">
        <canvas class="markerProgressCanvas" width="24" height="24"></canvas>
        <div class="markerProgressOverlay"></div>
        <div class="markerPlayAgain">play<br>again</div>
    </div>
    <div class="videoOverlay" style="width: 638px; height: 108px; left: 0px; ">
        <div class="videoPlayButton verbose" style="left: 281px; top: 11px; "></div>
    </div>
    <div class="videoDarken" style="left: 0px; background-color: transparent; "></div>
</div></div>
```

찜하세요~ **016 : Wony** 찜 **&** 작업 끝 **&** 리뷰요망 **&** 나솔

리뷰

Control structures can be labeled as well.

분기나 반복과 같이 프로그램 제어에 필요한 구성 요소들에도 역시 레이블을 붙일 수 있습니다.

(표현제안) 제어하는 구성요소에도 레이블을 붙일 수 있습니다.



It's tempting to think of this as "inline help", but it's not help -- it's simply labeling. The problem with the following UI isn't that it lacks a "help feature". The problem is that nothing is labeled.

이 것이 각 라인 상의 도움말 기능이라고 생각하기 쉽지만 엄밀히 말해 이는 도움말 기능이라기 보다는 단순한 레이블입니다. 반면, 아래 제시된 UI의 문제점은 도움말 기능이 없는 것이 아니라, 그 어떤 구성 요소에도 레이블이 붙어있지 않은 것입니다.

(표현제안) 레이블을 붙이는 것이, 도움말 기능이라고 생각하기 쉽지만, 이는 도움말 기능이 아닙니다. 그냥 레이블을 붙이는 거에요. 아래 제시한 UI의 진짜 문제는, 도움말 기능이 없는 것이 아니라, 레이블이 붙어있지 않은 것입니다.



찜하세요~ **017 : Wony** 찜 **&** 작업 끝 **&** 리뷰요망

That UI is exactly as informative as this line of code:

이 UI는 정확히 다음 한 라인의 코드가 제공하는 만큼의 정보 만을 제공하지요.

Why do we consider the code acceptable and the UI not? Why do we expect programmers to "look up" functions in "documentation", while modern user interfaces are designed so that documentation is typically unnecessary?

왜 우리는 아래 제시된 예시의 코드는 괜찮다고 여기면서, 위에 제시된 예시의 UI는 형편없다고 생각할까요? 왜 우리는 오늘날의 사용자 인터페이스들이 기본적으로 설명서가 필요 없을 정도로 쉽게 설계되기를 기대하면서, 정작 프로그래머들에게는 여러 기능에 대한 설명들을 그들이 알아서 찾아가며 일을 하기를 요구하나요?

찜하세요~ **018 :** 유영호찜 **&** 작업완료 **&** 리뷰 요망 **&** 리뷰 완료

Explain in context
A programming environment is a *user interface for understanding a program*. Especially in an environment for learning, the environment must be *designed to explain*.

One attribute of great explanations is that they are often embedded in the context of what they are explaining. That is, they *show* as well as *tell*.

문맥 안에서의 설명하기
(표현제안) ## 문맥 안에서 설명하기

프로그래밍 환경은 프로그램을 이해할 수 있는 접점 역할을 합니다.(user interface를 그냥 풀어서 썼어요.) 특히나 배우기위한 환경이라면, 그 환경은 설명적이어야만 합니다.

(저는 이렇게 해봤어요 :) 정답이 있는 것은 아닐테니 함 비교해보세요 ^^) 프로그래밍 환경은 학습자에게 *프로그램을 이해하는 데 있어 인터페이스 역할*을 합니다. 특히 교육을 염두에 두고 만든 환경이라면, *설명하는 역할을 하도록* 설계해야 합니다.

좋은 설명은 보통 설명하려는 대상을 문맥 안에 둡니다. 즉, *말해줄*뿐 아니라 *보여줌*으로서 설명을 도와줍니다.

좋은 설명은 보통 설명하려는 대상을 어떤 문맥 안에다 둡니다. 즉 이것이 무엇이다..라고 *말해줄* 뿐 아니라, 이것은 이런 것이다.. 라고 *보여*줌으로서 학습자가 감을 잡을 수 있게 도와줍니다.

찜하세요~ **019 :** 김소혜찜 **&** 작업완료 **&** 리뷰요망

Instead of just *describing* what vocabulary means, we can often *show* it in the context of the data. In the following example, the labels *connect* the code and its output:

단어가 무엇을 의미하는지 단지 *설명하는 것* 대신에, 우리는 많은 경우에 데이터의 전후 관계에서 *보여*줄 수 있습니다. 다음의 예에서, 라벨은 코드와 그 결과를 *연결합니다*.

```
<div class="example">
        <video width="640" height="110" preload>
```

```
        <source
src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/Vocab12.mp4
" type="video/mp4">
        <source
src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/Vocab12.web
m" type="video/webm">
    </video>
</div>
```

Such a connection can be especially powerful when a line of code does multiple things:

한 줄의 코드가 여러가지 일을 할 때 그런 연결은 특히 효과적일 수 있습니다.

```
<div class="example">
    <video width="640" height="110" preload>
        <source
src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/Vocab13.mp4
" type="video/mp4">
        <source
src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/Vocab13.web
m" type="video/webm">
    </video>
</div>
```

찜하세요~ **020 :** 김소혜찜 **&** 작업완료 **&** 리뷰요망

Summary — Read the vocabulary

##요약 — 코드에 나오는 단어를 읽기

(표현제안) ## 요약 - 코드에 나오는 단어를 읽기

The particular solutions shown here are merely examples. What matters is the underlying purpose: enabling the learner to read the program.

여기에서 보여지는 개별적인 설명은 단지 예일 뿐입니다. 근본적인 목적은 학습자가 프로그램을 읽을 수 있게 해주는 것입니다.

(뒷부분 표현제안) - 근본적인 목적은 학습자가 프로그램을 읽을 수 있게 해주는 것입니다.

* The environment should **make meaning transparent**, so the learner can concentrate on high-level concepts, not vocabulary.
* The environment should **explain in context**. Show and tell. Annotate the data, not just the code.

* 환경은 **의미를 명백하게 만들**어서 학습자가 단어가 아닌 좀 더 고차원적인 개념(이를테면 설계)에 집중할 수 있게 합니다.
* 환경은 **전후 관계에서 설명합니다**. 보여주고 말합니다. 단지 코드가 아니라 데이터에 설명을 추가합니다.

찜하세요~ **021 : nassol** 찜 **&** 작업중

The examples above are just one of many ways of achieving these goals. All that really matters is that somehow the learner's questions get answered:

An environment which allows learners to get hung up on these questions is an environment which discourages learners from even getting started.

The examples above are just one of many ways of achieving these goals. All that really matters is that somehow the learner's questions get answered:

An environment which allows learners to get hung up on these questions is an environment which discourages learners from even getting started.

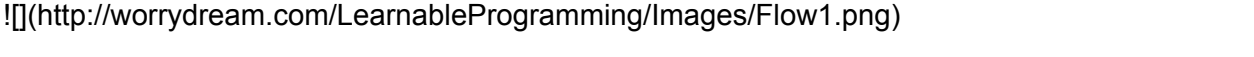
찜하세요~ **022 :** 김소혜찜 **&** 작업완료 **&** 리뷰요망

FOLLOW THE FLOW

흐름을 따라가기

The Khan Academy system presents the learner with code on the left, and the output of the code on the right. When the code is changed, the output updates instantaneously.

칸 아카데미 시스템은 학습자에게 왼쪽에는 코드를, 오른쪽에는 코드의 출력 결과를 보여줍니다. 코드가 변경됨과 동시에, 출력 결과가 변경됩니다.



Another thought experiment. Imagine a cooking show, ruthlessly abbreviated. First, you're shown a counter full of ingredients. Then, you see a delicious soufflé. Then, the show's over.

또 다른 실험이 있습니다. 무자비하게 생략된 요리 쇼를 상상해보세요. 먼저, 재료로 꽉 찬 조리대가 보입니다. 그 다음에는 맛있는 수풀레가 보입니다. 그리고 나면 쇼는 끝이 납니다.



찜하세요~ **023 :** 김소혜찜 **&** 작업완료 **&** 리뷰요망

Would you understand how that soufflé was made? Would you feel prepared to create one yourself?

수플레가 어떻게 만들어졌는지 이해할 수 있습니까? 스스로 무언가를 만들 준비가 되었다고 느낍니까?

Of course not. You need to see how the ingredients are combined. You need to **see the steps**.

물론 아닙니다. 재료가 어떻게 조합되는지 볼 필요가 있습니다. **그 과정을 보**아야 합니다.

The programming environment exhibits the same ruthless abbreviation as this hypothetical cooking show. We see code on the left and a result on the right, but it's the steps *in between* which matter most. The computer traces a path through the code, looping around loops and calling into functions, updating variables and incrementally building up the output. *We see none of this*.

프로그래밍 환경은 예로 든 요리 쇼처럼 무자비한 생략을 나타냅니다. 왼쪽에는 코드가, 오른쪽에는 결과가 있지만, *그 사이에* 발생한 과정이 가장 중요합니다. 컴퓨터는 코드를 통해 실행 경로를 따라가면서, 반복문을 수행하고 함수를 호출하고, 변수의 값을 변경하고 점차적으로 결과물을 축적합니다. *우리는 이 과정을 전혀 보지 않습니다.*

찜하세요~ **024 :** 유영호찜 **&** 작업완료 **&** 리뷰요망

People understand things that they can *see and touch*. In order for a learner to understand what the program is actually doing, the program flow must be made *visible and tangible*.

사람들은 *볼 수 있고 만질 수 있는 것*들을 쉽게 이해합니다. 학습자들에게 프로그램이 실제로 무엇을 하는지 알게 하려면, 프로그램의 흐름이 *직접 제어할* 수 있게 해주고 *눈으로 볼* 수 있게 해주어야 합니다.

(뒷부분 표현제안) 학습자들이 프로그램이 실제로 무엇을 하는지 알도록 도와주려면,
프로그램의 흐름을 *직접 제어*할 수 있게 해주고, *눈으로 볼* 수 있게 해주어야 합니다.

Make flow tangible
프로그램의 흐름 실체화하기
(표현제안) ## 학습자가 프로그램의 흐름을 제어할 수 있게 해주기

That example program again:
다시 예를 들어:

찜하세요~ **025 : whdnfl21** 찜 **&** 작업완료 **&** 리뷰요망

This is a particularly difficult example for a beginner to follow. The "for" construct, with its three statements on a single line, makes the control flow jump around bizarrely, and is an unnecessarily steep introduction to the concept of looping.

이 예는 초보 학습자가 따라가기에는 너무 어렵습니다.
"for" 의 구조는 한 줄에 3개의 구문이며, 특이하게도 통제 흐름을 훌쩍 넘어가게 만듭니다.
그리고 그것은 반복(문) 컨셉에 관해 불필요하게 급격한 입문입니다.

To make the flow more sane for a learner, the loop can be rewritten using "while":

학습자를 위해 더 분별있는 흐름을 만드려면, 반복(문)을 "while"로 쓸 수 있습니다.

찜하세요~ **026 :** 김훈민 찜 **&** 작업완료 **&** 리뷰요망

Now, the control flow must be made *tangible*. We must put the execution of the program into the programmer's hand, let her feel that it is a *real thing*, let her *own* it.

이제 제어 흐름을 명백하게 만들어야 합니다. 우리는 프로그램의 실행과정을 학습자에게 넘겨주어 이게 진짜라고 느끼고, 소유하도록 해야 합니다.

In the following example, the programmer uses a slider to scrub through the execution:

아래의 예제에서, 학습자는 슬라이더를 이용해서 실행과정을 앞뒤로 오가며 보게 됩니다.

```
<div class="example" data-top="16">
	<video width="640" height="126" preload>
		<source
src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/Flow5.mp4"
type="video/mp4">
		<source
src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/Flow5.webm"
type="video/webm">
	</video>
</div>
```

찜하세요~ **027 :** 김훈민 찜 **&** 작업완료 **&** 리뷰요망

This control allows the programmer to move around the loop at her own pace, and understand what is happening at each step. She can go backwards and forwards, dwell in difficult areas, and compare what is happening at different times. She can study how the output is built up over time, instead of seeing it magically appear all at once.

학습자에게 제어를 넘겨주면 학습자는 자신의 속도에 맞게 이동하며 각 단계에서 일어나는 일들을 살필 수 있습니다. 그러면서 앞뒤로 이동하고, 어려운 부분에 머무르거나, 다른 시간대에는 어떤 일이 일어나고 있는지 비교할 수 있습니다. 그러면 학습자는 모든 것들이 마법처럼 한 번에 나타나는 상황을 보는 것 대신에 시간이 지남에 따라 출력이 어떻게

만들어지는지 배울 수 있습니다.

찜하세요~ 028 : Justin 찜 & 작업 완료 & 리뷰 요망

Make flow visible
흐름을 시각화하기

The example above allows the programmer to follow the program's execution over time. But she's peeking through a pinhole, only seeing a single point in time at any instant. She has no *visual context*.
위의 예제는 학습자가 프로그램의 실행을 따라갈 수 있게 합니다. 하지만, 학습자는 조그만 구멍을 통해서 특정 시점에 특정 지점만 살짝 엿볼 수 있을 뿐입니다. 학습자는 *시각적인 콘텍스트(맥락)*를 전혀 갖고 있지 않습니다.

To illustrate what I mean, here are two representations of a trip around my neighborhood, one where the neighborhood itself isn't visible, and one where it is.
제가 말하고자 하는 바를 표현하기 위해, 여기 내 이웃동네 주변을 돌아다니는 두가지 방법을 제시합니다. 하나는 이웃동네가 보이지 않는 것, 다른 하나는 보이는 것.

찜하세요~ 029 : Justin 찜 & 작업 완료 & 리뷰 요망

This "overhead view" lets a person understand the trip at a higher level. She can see the shape of the trip. She can see *patterns*.
이 “오버헤드 뷰”는 누군가 그 여행(돌아다니기)을 좀 더 높은 위치에서 보고 이해할 수 있게 해줍니다. 학습자는 그 여행의 모양을 볼 수 있습니다. 또한 학습자는 그 여행의 *패턴*을 볼 수 있습니다.

In the following example, the program flow is plotted on a timeline. Each line of code that is executed leaves a dot behind. The programmer can take in the entire flow at a glance:
다음 예제에서, 프로그램의 흐름을 타임라인에 그리게 됩니다. 각각의 실행된 코드 라인들은 점들을 남겨둡니다. 학습자는 그렇게 함으로써 전체 흐름을 한 눈에 볼 수 있게 됩니다.

<video width="792" height="124" preload>
<source
src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/Flow8.mp4"

type="video/mp4">
<source src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/Flow8.webm" type="video/webm">
</video>

찜하세요~ **030 :** 사훈 찜 **&** 작업완료 **&** 리뷰요청

The patterns that emerge are especially helpful in the presence of conditionals and other forms of flow control:
등장한 패턴은 조건의 존재와 다른 흐름 컨트롤 형태에 특별히 도움이 됩니다.

<video width="792" height="232" preload>
<source src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/Flow9.mp4" type="video/mp4">
<source src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/Flow9.webm" type="video/webm">
</video>

This visualization allows the programmer to see the "shape" of an algorithm, and understand it at a higher level. The program flow is no longer "one line after another", but a *pattern of lines over time*.
이런 시각화는 프로그래머에게 알고리즘의 “모양”을 볼 수 있게 하고, 높은 수준에서 이해할 수 있게 합니다. 프로그램 흐름은 더 이상 하나가 “실행된 후 한 라인”이 아니라 *시간의 흐름에 따른 라인의 패턴*입니다.

찜하세요~ **031 : sahoon** 찜 **&** 작업중 **&** 리뷰요청

Make time tangible
실체화

Line-by-line execution is a very fine-grained view of time. The programmer also thinks about

time at other granularities.
라인 단위의 실행은 매우 세밀한 보기입니다. 프로그래머는 뭉쳐서 생각해야 합니다.

For instance, animations and games run at a frame rate, say, sixty frames per second. Every 1/60th of a second, the program prepares the next frame to display on the screen. Other programs are event-driven -- they respond to an external event, such as a button click or network request, by performing some computation, and then they wait for the next event.
예를 들어, 애니메이션과 게임이 초당 60프레임으로 보여진다고 해요. 두번째의 모든 1/60초는 다음 프레임을 화면에 표시할 준비를 합니다. 다른 프로그램은 버튼을 클릭하다거나 네트워크 요청을 하거나, 어떤 계산을 수행하거나 다음 이벤트를 기다리는 외부 이벤트에 반응하는 이벤트 중심입니다.

찜하세요~ **032 : sahoon** 찜 **&** 작업완료 **&** 리뷰요청

These frames or event responses form a natural way of "chunking" time. If the execution of a line of code is like a sentence, then a frame is like a chapter. These chapters can also be made tangible, so the programmer can understand the execution at this granularity as well.
이런 프레임 또는 이벤트 응답은 “청킹”(정보를 의미있게 연결시키거나 묶는 인지 과정을 지칭)타임을 형성합니다. 코드 라인의 실행이 문장과 같다면, 프레임은 챕터와 같습니다. 이런 챕터들은 실체화 할 수도 있는데 이를 통해 프로그래머는 이 단위에서 실행을 이해할 수 있습니다.

The following example provides a timeline for exploring line-by-line execution, and a slider for exploring frame-by-frame.
아래 예제는 라인이 차례대로 실행되는 순서와 프레임이 차례대로 실행되는 슬라이더를 보여줍니다.

```
<div class="example" data-top="14" data-bottom="36">
	<video width="640" height="268" preload>
		<source
src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/Flow10.mp4"
type="video/mp4">
		<source
src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/Flow10.webm
" type="video/webm">
	</video>
</div>
```

찜하세요~ **033 :** 유영호찜 **&** 작업완료 **&** 리뷰요망**.**

This control enables the programmer to go backwards and forwards through time, study interesting frames, and compare the execution across different frames.
이러한 제어는 학습자에게 시간적으로 프로그램의 앞과 뒤로 갈 수 있게 해줌으로서, 서로 다른 프레임의 실행을 비교 할 수 있게 해줍니다.

Make time visible
시간 시각화 하기

In the above example, we are once again peeking through a pinhole, seeing just one frame at a time. In the following example, all frames are lightly overlaid, in order to give *context* to the active frame. The entire path of the ball can be seen at once.
위의 예제에서는 한번에 한 프레임만을 보았었습니다. 그러나 아래 예제에서는 실행되는 프레임의 전체 문맥을 보여주기 위해 모든 프레임을 잔상으로 보여줍니다. 공의 모든 경로가 한번에 보여집니다.

```
<div class="example" data-bottom="30">
	<video width="640" height="248" preload>
		<source
src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/Flow11.mp4"
type="video/mp4">
		<source
src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/Flow11.webm
" type="video/webm">
	</video>
</div>
```

The output of the program is no longer a series of fleeting moments, but can be seen as a single, solid thing that extends over time. There is great power in this way of thinking.*
이제 더 이상 결과는 각 시간의 연속이 아닌, 전체 시간을 하나로 보여줍니다. 이러한 방식은 사고에 큰 힘을 줍니다.

찜하세요~ **034 : Justin** 찜 **&** 작업 완료 **&** 리뷰 요망

Summary — Follow the flow
정리 - 흐름을 따라가라

Again, the particular solutions shown here are merely examples. What matters is the underlying purpose: enabling the learner to follow the program flow, by *controlling time* and *seeing patterns across time*. Transforming flow from an invisible, ephemeral notion into a solid thing that can be studied explicitly.
다시 한 번, 여기에 보여준 특정 솔루션들은 단순히 예제에 지나지 않습니다. 중요한 것은 그 아래 놓인 목적이죠. *시간을 통제함으로써* 그리고 *그 시간 동안 패턴을 찾는 것으로써* 학습자가 프로그램의 흐름을 따라갈 수 있게끔 해주는 것이 그 목적입니다. 보이지 않거나 덧없는 개념에서 출발하여 굳건한 무언가로 바뀌는 것을 명시적으로 학습하는 것이 또다른 목적이지요.

찜하세요~ **035 : Justin** 찜 **&** 작업 완료 **&** 리뷰 요망

The environment can **make flow tangible**, by enabling the programmer to explore forward and backward at her own pace.
환경은 학습자가 스스로의 페이스에 맞추어 앞뒤로 왔다갔다 할 수 있게 함으로 인해 **흐름을 실체화 시킬 수 있습니다**.
The environment can **make flow visible**, by visualizing the pattern of execution.
환경은 프로그램의 실행 결과 패턴을 시각화하는 것에 의해 **흐름을 시각화 시킬 수 있습니다**.

The environment can represent time at **multiple granularities**, such as frames or event responses, to enable exploration across these meaningful chunks of execution.
환경은 프레임이나 이벤트 응답과 같이 의미있는 실행 단위로 시간을 **다양하게 쪼개는 것들**에서 표현할 수 있습니다.

(표현제안) 프로그램 환경에서는 타임 프레임이나 이벤트 응답 등을 표시하는 방법으로, 실행 단위로 시간을 **쪼개는 방식**으로 시간을 표시해줄 수 있습니다. 이렇게 하면 학습자는 의미있는 실행단위로 프로그램의 흐름을 탐색할 수 있습니다.
(나도나도) 환경은 시간을 프레임이나 이밴트 응답같은 **여러 단위**로 표시해줄 수 있습니다. 이는 사용자로 하여금 의미있는 실행의 덩어리를 중심으로 탐색할 수 있도록 합니다.

(최종) 이 환경은 시간을 프레임 (frame) 이나 이벤트 응답 (event response) 과 같은 **여러개의 조각들**로 재현해서, 그 의미있는 여러개의 조각들을 넘나들며 탐색할 수 있게 합니다.

찜하세요~ **036 :** 김훈민 찜 **&** 작업완료**&** 리뷰요청

SEE THE STATE

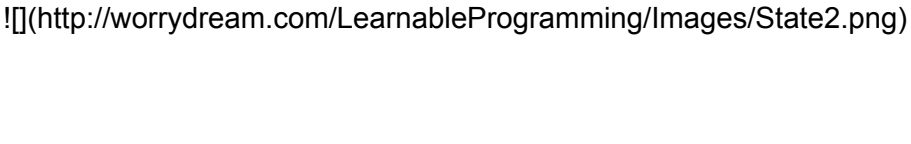
A simple program:

상태 보기
간단한 프로그램 :



The third line declares a variable named "scaleFactor", which varies with each iteration of the loop.

세번째 라인은 “scaleFactor”라는 이름의 변수를 선언하는데, 루프의 각 이터레이션에 따라 값이 달라집니다.



찜하세요~ **037 :** 김훈민 찜 **& &** 작업완료**&** 리뷰요청

Take a moment to look at that line, and think about these questions:

잠시 동안 세번째 라인을 보고 아래의 질문들에 대해 생각해봅시다. :

* What values does scaleFactor take on? 1? 100? -1? pi / 2?
* What is scaleFactor at the beginning of the loop? At the end?
* How does scaleFactor change over the course of the loop? Linearly up? Linearly down? Does the change get faster or slower?

* scaleFactor는 어떤 값을 맡게 되죠? 1? 100? -1? pi / 2?
* 루프가 시작될 때 scaleFactor는 무엇일까요? 루프가 끝났을때는?
* scaleFactor는 루프가 반복되는 과정을 통해 어떻게 변하나요? 선형 증가? 선형 감소? 변화가 더 빨라지거나 느려지나요?

찜하세요~ **038 :** 김병진 발번역

Wait. Wait a minute. Were you trying to answer those questions by doing arithmetic in your

head? The computer somehow drew that picture, so the computer must have calculated all those scaleFactors itself. Are you seriously *recalculating them in your head?*

Now imagine if scaleFactor also depended on some other variables, or some other functions, or external input. There would be *no way* to easily answer those questions.*
잠깐, 잠시만 기다리세요. 이 질문의 해답을 머리속으로 암산해볼려고 하셨나요? 컴퓨터는 이 사진을 모든 scaleFactor들을 스스로 계산해서 그렸습니다. 이것을 다시 머리속으로 암산하기 위해서 심각하세요? 만약 scaleFactor가 어떤 다른 변수들과 다른 함수들 다른 외부 입력에 의존적이라고 상상해봅시다. 이 문제를 쉽게 해결할 방법이 없을 거에요.

찜하세요~ **039 :** 김병진 발번역

Think about this. We expect programmers to write code that manipulates variables, without ever seeing the values of those variables. We expect readers to *understand* code that manipulates variables, without ever seeing the values of the variables. **The entire purpose of code is to manipulate data, and we never see the data.** We write with blindfolds, and we read by playing pretend with data-phantoms in our imaginations.
이것에 대해 생각해봐요. 우리는 프로그래머가 그 변수들의 값을 확인하지 않고 변수들을 다루는 코드를 작성하는 것을 바라죠. 우리는 읽는 사람들이 변수들의 값을 확인하지 않고 변수들을 다루는 코드를 *이해하길* 바라죠. **코드의 궁극의 목적은 데이터를 다루고 우리는 데이터를 보지않는 것입니다.** 우리는 눈을 가린채 작성하고 우리의 상상속의 데이터 유령과 함께 하는 것 같이 읽어요.

찜하세요~ **040 :** 김병진 발번역

Information design pioneer Edward Tufte has one primary rule, and this rule should be the principle underlying any environment for creating or understanding.

Show the data.

If you are serious about creating a programming environment for learning, the **number one thing you can do** -- more important than live coding or adjustable constants, more important than narrated lessons or discussion forums, more important than badges or points or ultra-points or anything else -- is to **show the data**.
정보 설계 선구자인 에드워드 터프트는 기본 원칙을 가지고 있고 이 원칙은 창작 또는 이해를 위한 어떤 환경에서도 근본적인 원칙이 되었죠.

데이터를 보세요

만약 학습을 위한 프로그래밍환경을 만드는 것에 직면해있다면 **당신이 할 수 있는 한가지 방법은** -- 실제코딩 이나 조절가능한 상수보다중요하고, 해설강의나 토론 포럼보다 중요하며, 표시나 요점들 또는 매우 중요한 요점 또는 그 어떤 것 보다 더 중요한 것은 -- **데이터를 보는 것**입니다.

발번역 show the data를 저는 명령문으로 해석했어요..ㅠㅠ 원칙이라고 해서..ㅠㅠ 어제밤 꿈에 개발자영어가 꿈에 나타나서 정모를 하더라고요. 무서워서 아침에 접속해보았어요.

찜하세요~ **041 : nassol** 찜 **&** 작업완료

Show the data

Because the value of a variable varies over time, showing the data is intimately connected with showing time.

The previous section presented a timeline representation that showed the data at each step. In the following example, the programmer mouses over a particular row of the timeline to concentrate on a single line.

데이터를 보여주기

시간이 지남에 따라 변수의 값이 바뀌기 때문에, 변수의 값을 시간정보하고 같이 보여주는 것이 중요합니다.

이전 섹션에서는 타임라인에다가 각 스텝별 변수의 값을 보여주었습니다. 아래의 예에서, 프로그래머가 타임라인의 특정 행에 마우스를 갖다대서 그 행에 대해 집중적으로 볼 수 있습니다.

	<video width="792" height="196" preload>
		<source src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/State3.mp4" type="video/mp4">
		<source src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/State3.webm"

type="video/webm">
</video>

찜하세요~ **042 : nassol** 찜 **&** 작업완료 **&** 리뷰 요청

In this example, it is easy to answer the first two questions. By skimming over the execution of that line of code, we can see all of the values that scaleFactor takes on, and when.

However, it is still difficult to answer the third question: *how* does the variable vary? What is the shape of its change? The question is difficult because we are, once again, peeking through a pinhole, only seeing a single point at a time.

Edward Tufte has a second rule. It is not enough to just show the data. We must **show comparisons**.

이 예에서 처음 두 개의 물음에 답할 수 있습니다. 각 라인의 실행을 훑어보면서 우리는 scaleFactor가 언제, 어떤 값을 취하는지 알 수 있습니다.

하지만 변수가 어떻게 변화하는가? 라는 세 번째 물음은 여전히 대답하기 어렵습니다. 그것이 변화하는 형태는 어떠한가? 이 문제가 어려운 이유는, 다시한번 말하지만, 사람들은 작은 바늘구멍을 통해 들여다 보면서 한번에 한 부분만을 보기 때문입니다.

에드워트 투프트는 두 번째 원칙을 제시합니다. 데이터를 보여주는 것만으로는 충분치 않습니다. 학습자가 데이터의 **차이를 비교**해서 볼 수 있게 해주어야 합니다.

찜하세요~ **043 : nassol** 찜 **&** 작업완료

##Show comparisons

Data needs context. It is rarely enough to see a single data point in isolation. We understand data by comparing it to other data.

The timeline examples so far have used dots to represent executed lines. But instead of dots, we can show data. The following timeline shows each of the scaleFactors:

데이터를 비교해서 볼 수 있게 해주기

데이터는 문맥 안에서 봐야 의미가 있습니다. 개별 데이터를 따로 떼어놓고 보는 것은 의미가 없습니다. 데이터를 다른 데이터와 비교해서 보았을 때 우리는 데이터를 이해할 수 있습니다.

지금까지의 타임라인 방식은 실행되고 있는 라인을 점으로 표시해 왔습니다. 하지만 점 대신에 데이터를 보여주는 방법도 있습니다. 아래 타임라인에서는 각각의 scafeFactor를 보여줍니다.

타임라인 방식으로 보여준 예에서는 실행되는 코드행을 나타내기 위해서 점을 사용하였는데요, 점 대신에 데이터를 표시해서 데이터를 서로 비교해서 볼 수 있게 해줄 수 있습니다.

아래 제시한 타임라인에는 각각의 scaleFactor를 보여줍니다:

찜하세요~ 044 : nassol 찜 & 작업완료

Almost every line of code here calculates something. The environment should provide the best visualization of whatever that something is. For example, the "rotate" line can show the rotations.

The "fill" line sets a fill color. That color can be shown.

The "triangle" line draws a triangle to the canvas, rotated and colored. The timeline can show a thumbnail of each triangle produced.

거의 모든 코드가 무언가를 계산합니다. 계산하는 대상이 무엇이 되었든, 프로그램 환경에서는 계산하는 대상을 시각적으로 최대한 잘 보여주어야 합니다. 예를 들어 “회전시키는" 코드는

회전하는 것을 보여주어야 합니다.



“색깔을 채우는" 코드는 색깔을 채우는 것을 시각적으로 보여주어야 합니다. 그 색깔을 보여주어야 합니다.

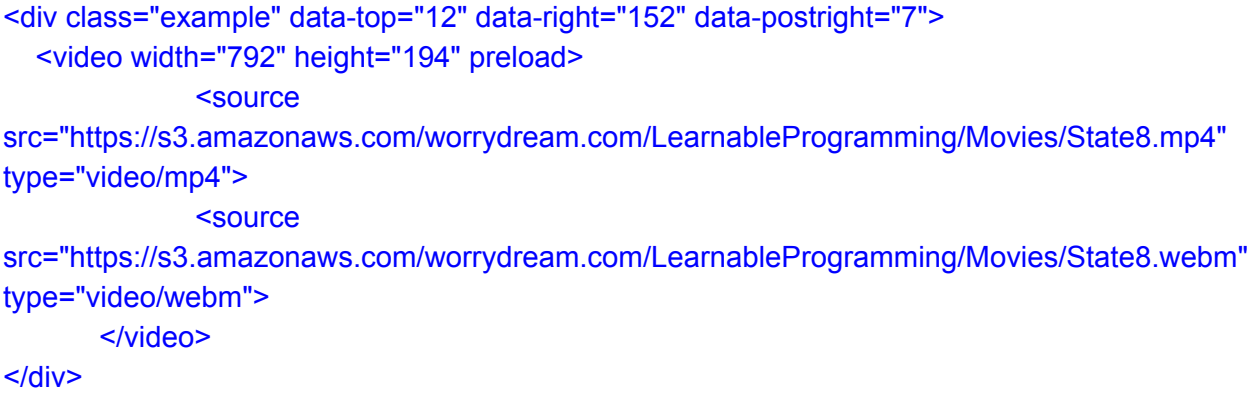


“삼각형" 코드행은 캔버스에다가 삼각형을 그려주고, 회전시키고 색깔을 표시해주어야 합니다. 타임라인 상에서는 각 시점의 삼각형의 모양을 썸네일로 보여주어야 합니다

찜하세요~ 045 : nassol 찜& 작업완료

Taken together, we have a timeline that depicts not just the flow, but *all* of *the data calculated in that flow*. The execution of the program is laid bare for the reader. At a glance, she can see *which* lines were executed, *when* they were executed, and *what* they produced. The flow and the data are both shown *in context*.

다 모아서 보면, 이 타임라인은 프로그램의 흐름을 보여줄 뿐 아니라, 프로그램이 진행되는 과정에서 계산되는 데이터를 *모두* 보여줍니다. 프로그램을 실행하는 것은 학습자가 직접할 수 있고요. 학습자는 이 타임라인을 보면서 *어느* 행이 실행되고 있는지를 알 수 있고, 그 행이 *언제* 실행되는지를 알 수 있으며, 그 행이 *어떤 결과*를 만들어내는 지를 알 수 있습니다. 학습자는 프로그램의 흐름과 데이터 둘다 *문맥* 안에서 볼 수 있습니다.



찜하세요~ 046 : nassol 찜 & 작업중

The example above only loops twenty times. Is it possible to understand a loop with, say, thousands of iterations, without drowning in thousands of numbers?

Yes -- there is an entire field of study devoted to depicting large amounts of numbers. To visualize this data, we can use all of the standard techniques of *data visualization*.

In the following example, as the programmer zooms the timeline out, the visualization automatically switches from a table to a plot.

	<video width="584" height="86" preload>
		<source src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/State9.mp4" type="video/mp4">
		<source src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/State9.webm" type="video/webm">
	</video>

찜하세요~ 047 : twshim작업끝

Eliminate hidden state

In order to understand what a line of code does, the learner must *see its effect*. For example, as the programmer moves over iterations of the "triangle" line, she sees each triangle appear on the canvas:

숨겨진 상태 제거하기

코드 한줄이 무슨 일을 하는지 이해하기 위해서, 학습자는 *효과를 볼 수 있어야* 합니다. 예를들어, 프로그레머가 “삼각형”을 이동하려 한다면, 각 삼각형이 캔버스에 나타나는 걸 볼 것입니다:

	<video width="792" height="194" preload>

<source src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/State10.mp4" type="video/mp4">
<source src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/State10.webm" type="video/webm">
</video>

찜하세요~ **048 :** 김소혜찜 **&** 작업중

The "fill" line, on the other hand, sets the fill color for subsequent drawing operations. When the programmer moves over this line, what effect does she see? She sees *nothing happen*, because the "fill" function *modifies hidden state*.

The Processing graphics library relies heavily on implicit state, in the form of the "current" fill color, stroke color, transform matrix, and so on. Code that modifies this state produces *no visible effect* on the canvas. In an interactive environment, this is unacceptable.

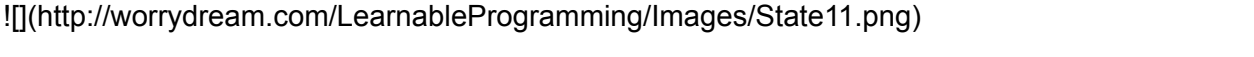
”채우기”행은 이후의 그리기 연산을 위한 채우기 색을 지정합니다. 프로그래머가 이 행 위로 움직였을 때, 어떤 결과를 보게 될까요? “채우기" 연산은 *숨은 상태를 수정하기" 때문에 아무런 변화도 없습니다.

프로세싱 그래픽스 라이브러리는 “현재" 채우기 색, 선긋기 색, 변환 매트릭스 등등의 형태로 암시적인 상태에 심하게 의존합니다. 이런 상태를 수정하는 코드는 캔버스 상에서는 “시각적인 결과가 없습니다.” 양방향의 환경에서는 이것은 받아들일 수 없습니다.

찜하세요~ **049 :** 김소혜찜 **&** 작업중

There are two design options here. One alternative is to **eliminate the state**. For example, color could be passed as a parameter to the "triangle" function.

두 가지 종류의 설계 방법이 있습니다. 한 가지 방법은 *“상태를 없애는 것**입니다.예를 들어, 색은 “삼각형" 함수에 인자로 전달될 수 있습니다.



The other alternative is to **show the state**. In the following example, the current fill and stroke colors are shown above the canvas. Now, when a line of code changes the fill color, the programmer actually *sees something change*. Making something visible *makes it real*.

다른 방법은 **상태를 보여주는 것"입니다. 다음의 예에서, 현재의 채우기와 선긋기 색은 캔버스 상에서 보여집니다. 이제, 한 줄의 코드가 채우기 색을 변경할 때, 프로그래머는 실제로 *어떤 것이 변화하는 것을 알 수 있습니다.". 어떤 것을 시각적으로 만드는 것은 *실제로 만드는 것*입니다.

	<video width="792" height="196" preload>
		<source src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/State12.mp4" type="video/mp4">
		<source src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/State12.webm" type="video/webm">
	</video>

찜하세요~ **050 :**김훈민 찜 **&** 작업완료 **&** 리뷰요망

All state must be *eliminated* or *shown*. Either can be a reasonable design decision. An environment that does neither -- forcing learners to *imagine* the state and make sense of functions that produce no visible effect -- is irresponsible design, and disrespectful to the learner.

모든 상태는 *제거*되거나 *보여져*야 하며, 둘 중 어느 하나가 합리적인 디자인 결정이 될 수 있습니다. 둘 다 가능하지 않은 환경은 사용자에게 무책임하고, 무례한 일입니다. -- 학습자에게 상태를 *상상*하게 하고 어떠한 시각적인 효과도 만들어내지 못하는 기능을 이해하도록 하는 --

찜하세요~ **051 :** 김훈민 찜 **&** 작업완료 **&** 리뷰요망

The current transform matrix is a particularly critical and confusing member of the state. Drawing anything interesting with the Processing graphics library requires matrix transforms, but the *current transform is invisible*. Functions such as "scale" and "rotate" have no visible effect,

and compound transformations (such as translation followed by scale, or should it be the other way around?) often involve groping blindly through trial-and-error.

현재의 변환 매트릭스는 특히 중요하며 혼란을 주는 상태의 멤버입니다. 프로세싱 그래픽 라이브러리로 흥미로운 무언가를 그리는 일은 매트릭스의 변환을 필요로 하지만, *현재 변환 상태는 눈에 보이지 않습니다*. “scale” 과 “rotate” 같은 함수가 만들어내는 효과는 눈에 보이지 않으며, 종종 시행착오들을 통해서 맹목적으로 모색해나가는 과정들을 포함하는 변화를 혼합하는 역할을 합니다. (scale에 따른 해석하거나, 아니면 그 반대여야 하겠죠?)

찜하세요~ **052 :** 누렁이 찜 **&** 작업완료 **&** 리뷰완료

In the following example, the transform is visualized, and the effect of every function can be seen directly.
다음의 예제에서는 변환 상태가 그려지고, 각각의 함수가 내는 효과를 직접 눈으로 볼 수 있습니다.

```
<div class="example" data-top="14" data-bottom="20" data-right="168" data-postright="23">
        <video width="808" height="216" preload>
                <source
src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/State13.mp4"
type="video/mp4">
                <source
src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/State13.webm" type="video/webm">
        </video>
</div>
```

찜하세요~ **053 :** 누렁이 찜 **&** 작업끝 **&**리뷰완료

Summary — See the state
정리 - 상태를 파악하라

Code manipulates data. To understand code, a learner must *see the data*, and see the *effect of code on the data*.

코드는 데이터를 다룹니다. 그러므로 코드를 이해하기 위해서 학습자들은 반드시 *데이터* 와 *코드가 데이터에 미치는 효과* 를 눈으로 보아야 합니다.

* The environment must **show the data**. If a line of code computes a thing, that thing should be immediately visible.
* The environment must **show comparisons**. If a program computes many things, all of those things should be shown in context. This is nothing more than data visualization.
* The system must have **no hidden state**. State should either be eliminated, or represented as explicit objects on the screen. Every action must have a visible effect.

* 개발 환경은 *데이터를 보여주어야* 합니다. 만약 한 줄의 코드가 한 결과를 산출해내면, 결과가 바로 보여야 합니다.
* 개발 환경은 학습자가 데이터 간의 *차이를 볼 수 있게* 해주어야 합니다. 만약 어떤 프로그램이 여러 결과물을 산출해내면 이 모든 것들이 맥락을 따라 보여야 합니다. 이것은 데이터의 가시화와도 같습니다.
* 시스템은 *숨겨진 상태* 를 가져서는 안됩니다. 보이지 않을 거라면, 그런 상태는 아예 없애거나, 모니터 등 눈으로 분명히 볼 수 있는 물체 위에서 표현되어야 합니다. 모든 실행 하나하나는 반드시 보여지는 효과를 지녀야 한다는 것이죠.

찜하세요~ **054 :** 누렁이 찜

CREATE BY REACTING
#반응을 중심으로 만들기

I was recently watching an artist friend begin a painting, and I asked him what a particular shape on the canvas was going to be. He said that he wasn't sure yet; he was just "pushing paint around on the canvas", reacting to and getting inspired by the forms that emerged. Likewise, most musicians don't compose entire melodies in their head and then write them down; instead, they noodle around on a instrument for a while, playing with patterns and reacting to what they hear, adjusting and sculpting.
최근에 미술을 하는 제 친구가 그림 그리는 것을 지켜보다가 저는 그 친구에게 정확히 어떤 그림이 될 것이냐고 물었습니다. 그러자 그는 아직 확실치 않다고 했습니다. 그는 단지 드러나는 형상을 보고 반응하고 영감을 얻어 “물감을 캔버스에 묻히고 있는 것일 뿐” 이라고 했습니다. 비슷하게도 대부분의 음악가들 역시 미리 머릿속에 전체 멜로디를 구상하고 곡을 쓰는 것이 아닙니다. 대신에 그들은 악기로 일정하게 연주도 해보고 그것이 내는 소리를 들으며 반응하면서 악기를 한동안 만지작거리고 형상화합니다 .

찜하세요~ **055 :** 누렁이 찜 **&** 진행중

An essential aspect of a painter's canvas and a musical instrument is the immediacy with which the artist gets *something there* to react to. A canvas or sketchbook serves as an "external imagination", where an artist can grow an idea from birth to maturity by continuously *reacting to what's in front of him*.

Programmers, by contrast, have traditionally worked in their heads, first imagining the details of a program, then laboriously coding them.

찜하세요~ **056 :** 누렁이 찜

Working in the head doesn't scale. The head is a hardware platform that hasn't been updated in millions of years. To enable the programmer to achieve increasingly complex feats of creativity, the environment must get the programmer *out* of her head, by providing an *external* imagination where the programmer can always be reacting to a work-in-progress.

찜하세요~ **057 :**

Some programming systems attempt to address this with a so-called "live coding" environment, where the output updates immediately as the code changes. An example of live coding:*

```
<div class="example">
	<video width="640" height="110" preload>
		<source
src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/React1.mp4"
type="video/mp4">
		<source
src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/React1.webm
" type="video/webm">
	</video>
</div>
```

As you can see, live coding, on its own, is almost worthless. The programmer still must type at least a full line of code before *seeing any effect*. This means that she must already understand what line of code she needs to write. The programmer is still doing the creative work entirely in her head -- imagining the next addition to the program and then translating it into code.

찜하세요~ **058 :**

Get something on the screen as soon as possible

Live coding does, however, provide a foundation for *other* features which can jump-start the create-by-reacting process. In the following example, the environment offers *autocomplete with default arguments*. After typing just a couple of characters, the programmer immediately *sees something on the screen*, and can proceed to adjust it.

찜하세요~ **059 :**

```
<div class="example" data-push-left="18" data-bottom="16" data-right="6">
	<video width="664" height="198" preload>
		<source
src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/React2.mp4"
type="video/mp4">
		<source
src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/React2.webm
" type="video/webm">
	</video>
</div>
```

Autocomplete is a common feature of most programming environments, but there are two critical subtleties here. First, the functions all have default arguments (position, width, height, and so on are already filled in), so each completion is a complete statement that produces a visible effect. Second, a default completion is selected immediately. Here's what this means for the programmer's thought process:

찜하세요~ **060 :**

In the example above, the programmer wants to draw a roof on the house. She doesn't need to mentally plan out *how* to draw the roof beforehand -- she doesn't need to imagine which functions would be appropriate. She just needs the vague notion: "I want to draw *something*." She starts typing "draw", and **immediately sees a shape on the screen**.

찜하세요~ 061 :

At this point, she can stop *imagining* and start *reacting*:

* *"This is the wrong shape. Which shape will work better?"* She goes down the list and turns the shape into a triangle.
* *"This is a right triangle. I want a different triangle."* She adjusts the triangle's points into a more roof-like shape.
* *"The roof isn't lying on the house."* She adjusts the triangle to lower it onto the house.

찜하세요~ 062 :

This example assumed a hypothetical graphics library which was *designed for autocomplete* -- all of the drawing functions begin with "draw", so the completion list would appear as the designer intended.*

A different way to structure the library would be to provide a single "shape" function, which takes the type of shape (triangle, ellipse, etc.) as an argument. For example:

```
<div class="example" data-push-left="18" data-right="6">
	<video width="664" height="182" preload>
		<source
src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/React4.mp4"
type="video/mp4">
		<source
src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/React4.webm
" type="video/webm">
	</video>
</div>
```

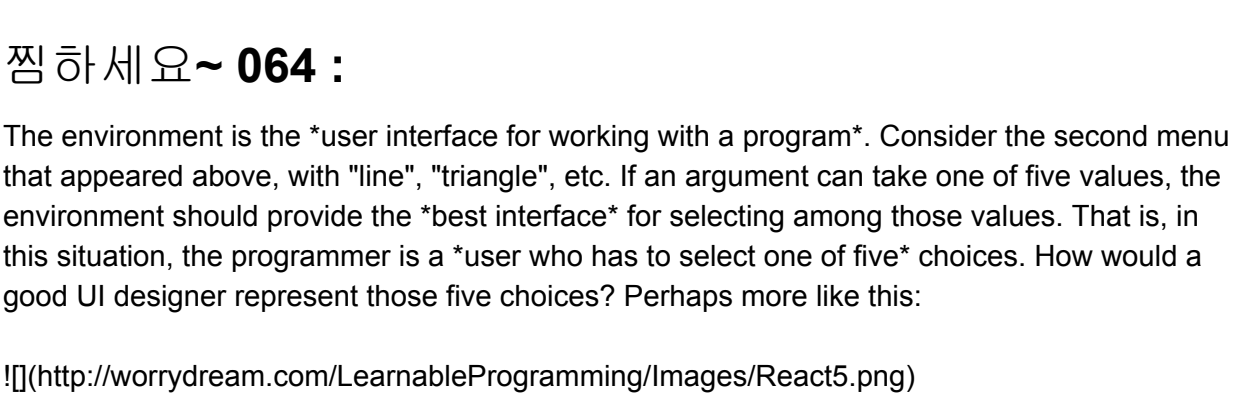
찜하세요~ 063 :

This could help to further encourage the create-by-reacting way of thinking. Because "drawTriangle" and "drawRect" aren't in the vocabulary, the programmer would never find herself thinking about specific shape functions before something is on the screen. Her starting

point is always just "shape".

찜하세요~ 064 :

The environment is the *user interface for working with a program*. Consider the second menu that appeared above, with "line", "triangle", etc. If an argument can take one of five values, the environment should provide the *best interface* for selecting among those values. That is, in this situation, the programmer is a *user who has to select one of five* choices. How would a good UI designer represent those five choices? Perhaps more like this:

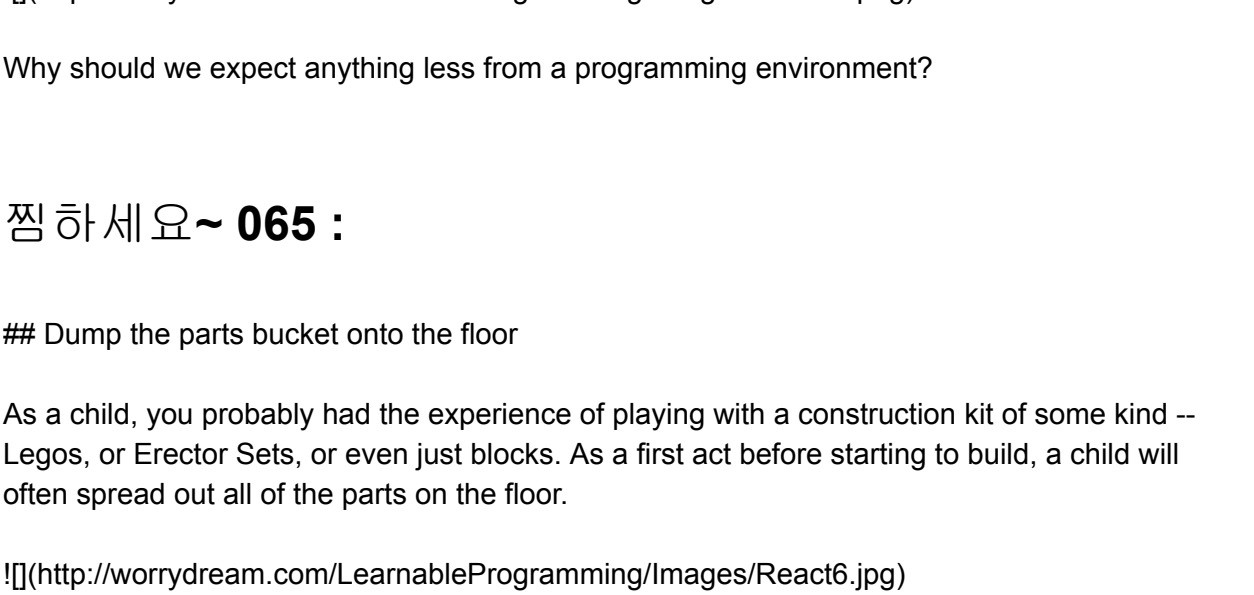


Why should we expect anything less from a programming environment?

찜하세요~ 065 :

Dump the parts bucket onto the floor

As a child, you probably had the experience of playing with a construction kit of some kind -- Legos, or Erector Sets, or even just blocks. As a first act before starting to build, a child will often spread out all of the parts on the floor.



찜하세요~ 066 :

This provides more than simply quick access. It allows the child to scan the available parts and *get new ideas*. A child building a Lego car might spot a wide flat piece, and decide to give the car wings.

This is a second form of create-by-reacting. In addition to reacting to the object under construction, the child is also reacting to the *parts she has available*.

찜하세요~ 067 :

In the following example, the available functions are located adjacent to the coding area, and the programmer can skim over these "parts" and get ideas.

```
<div class="example" data-left="160" data-bottom="28">
	<video width="640" height="210" preload>
		<source
src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/React7.mp4"
type="video/mp4">
		<source
src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/React7.webm
" type="video/webm">
	</video>
</div>
```

찜하세요~ 068 :

The above example *encourages* the programmer to explore the available functions. A learner who would never think to try typing the "bezier" function, with its unfamiliar name and eight arguments, can now easily stumble upon it and discover what it's about.

The example above is one way of representing the "parts bucket" for programmatic drawing. But would an user interface designer consider that to be the *best interface* for drawing a picture on a computer screen? What about the following?

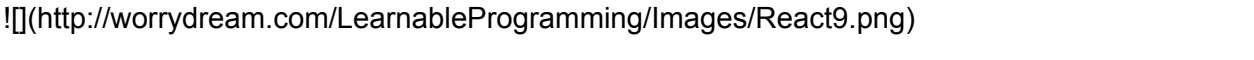
찜하세요~ 069 :

```
<div class="example" data-left="33">
	<video width="640" height="146" preload>
		<source
src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/React8.mp4"
type="video/mp4">
		<source
src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/React8.webm
" type="video/webm">
	</video>
```

An objection might arise at this point. With this interface, is this even "programming"?

No, not really. *But none of the examples in this section are "programming".* Typing in the code to draw a static shape --

찜하세요~ 070 :



-- is not programming! It's merely a very cumbersome form of illustration. It becomes genuine *programming when the code is abstracted* -- when arguments are *variable*, when a block of code can do different things at different times. The next section will discuss how create-by-reacting leads into *create-by-abstracting*.

찜하세요~ 071 :

Summary — Create by reacting

The create-by-reacting way of thinking could be stated as: start with *something*, then adjust until it's right.*

The programmer must be able to do her thinking out in the environment, not trapped in her head. The environment must serve as an *external imagination*, where the programmer can be continuously reacting to a work-in-progress.

찜하세요~ 072 :

To be clear, this does not relieve the programmer from thinking! It simply makes those thoughts *immediately visible*. I am happy to be composing this essay in a text editor, where my words become visible and editable as soon as I think of them, as opposed to working entirely internally like the orators and playwrights of the distant past.

찜하세요~ 073 :

* The environment must be designed to **get something on the screen as soon as possible**,

so the programmer can start reacting. This requires modeling the programmer's thought process, and designing a system that can pick up on the earliest possible seed of thought.
* The environment must **dump the parts bucket onto the floor**, allowing the programmer to continuously react to her raw material and spark new ideas.

찜하세요~ 074 :

CREATE BY ABSTRACTING

Learning programming is learning abstraction.

A computer program that is just a list of fixed instructions -- draw a rectangle here, then a triangle there -- is easy enough to write. Easy to follow, easy to understand.

It also makes *no sense at all*. It would be much *easier* to simply draw that house by hand. What is the point of learning to "code", if it's just a way of getting the computer to do things that are easier to do directly?

찜하세요~ 075 :

Because code can be *generalized* beyond that specific case. We can change the program so it draws the house anywhere we ask. We can change the program to draw many houses, and change it again so that houses can have different heights. Critically, we can draw all these different houses from a *single description*.

찜하세요~ 076 :

The description still says "draw a rectangle here, then a triangle there", but the here and there have been *abstracted*. Different parameters give us different heres and different theres.

How does a programmer learn to write this abstract code? How does she learn to write a single

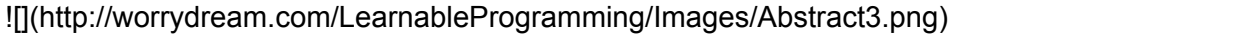
description that is generalized for many cases?

She *doesn't*. The learner should start by writing concrete code, and then *gradually* change it to introduce abstraction. And the environment must provide the tools to perform this process, in such a way that the learner can *understand* the program at each stage.

찜하세요~ **077 :**

Start constant, then vary

In the create-by-abstracting way of thinking, the programmer starts by creating a specific case, typically involving *constants*. She then moves to the general case by turning those constants into *variables*. Here's an example of how the environment can encourage this way of thinking, starting with the house from earlier.



찜하세요~ **078 :**

The programmer wants to move the house to a different location. She can't move it by adjusting a single number in the code, because there are four different points which all need to change at the same time -- the rectangle's origin, and the triangle's three points.

Here, the programmer selects one of the numbers, and *converts it into a variable*.

	<video width="640" height="164" preload>
		<source src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/Abstract4.mp4" type="video/mp4">
		<source src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/Abstract4.webm" type="video/webm">
	</video>

She then *connects* the variable to another number, by dragging from one to the other.

	<video width="640" height="164" preload>

<source
src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/Abstract5.mp4" type="video/mp4">
<source
src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/Abstract5.webm" type="video/webm">
</video>

찜하세요~ 079 :

There are two additional arguments to "triangle" which need to vary as well. When she connects the variable, whose value is 80, to the constant 100, the environment offers a choice of four possible relationships between the numbers.

<video width="640" height="164" preload>
<source
src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/Abstract6.mp4" type="video/mp4">
<source
src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/Abstract6.webm" type="video/webm">
</video>

The four expressions involve addition, subtraction, multiplication, and division respectively. One of them will typically be either the correct relationship or a good starting point.

Here is the entire process of introducing the variable.

<video width="640" height="164" preload>
<source
src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/Abstract7.mp4" type="video/mp4">
<source
src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/Abstract7.webm" type="video/webm">
</video>

찜하세요~ 080 :

Start with one, then make many

In the example above, the house is now *abstracted* -- the code doesn't just draw one fixed house, but can draw a house anywhere. This abstracted code can now be used to draw *many different houses*.

In the following example, the programmer wants to draw a row of houses. She selects the abstracted code, and *converts it into a loop*.

	<video width="640" height="164" preload>
		<source src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/Abstract8.mp4" type="video/mp4">
		<source src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/Abstract8.webm" type="video/webm">
	</video>

찜하세요~ 081 :

The variable in the first line of the selection becomes an *induction variable*, and the programmer can then adjust its bounds.

This is a process of starting with a specific case, and progressively generalizing:

* First, the programmer creates a house at a fixed location. Here, she has interactive control over each *individual shape*.
* She turns the house's location into a variable. Now, she has interactive control over the variable, which affects *many shapes*.
* She introduces a loop to vary that variable. Now, she has interactive control over the bounds of the loop, which affects *many houses*, which affect many shapes.

찜하세요~ 082 :

At each stage, the programmer has interactive control over the relevant parameters, but the parameters are at successively higher levels of abstraction. That is, the programmer can still create by reacting, but she's **creating and reacting at higher levels**.

Instead of drawing an evenly-spaced row of houses, the programmer now wants individual control over each of the houses. Starting from the variable abstraction, she selects the code and *converts it into a function*.

	<video width="640" height="164" preload>
		<source src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/Abstract9.mp4" type="video/mp4">
		<source src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/Abstract9.webm" type="video/webm">
	</video>

찜하세요~ 083 :

By duplicating the function call, she obtains several houses which can be controlled individually.

Now, instead of identical houses, she wants to vary the heights of the houses. She introduces another variable, and then converts it into *an additional argument to the function*.

	<video width="640" height="164" preload>
		<source src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/Abstract10.mp4" type="video/mp4">
		<source src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/Abstract10.webm" type="video/webm">

</video>

The process, again, consists of starting concrete, and progressively introducing abstraction:*

* The programmer creates a drawing of a house.
* She turns x into a variable, so she can control the house's position.
* She turns x into a function argument, so different houses can have different positions.
* She turns y into a variable, so she can control the house's height.
* She turns y into a function argument, so different houses can have different heights.

찜하세요~ 084 :

Summary — Create by abstracting

Start concrete, start grounded. Start with one specific case, entirely understood. Then gradually generalize, level by level, in such a way that the programmer still fully understands the program at each level of abstraction.

Fully concrete code can be micromanaged -- the programmer has explicit control over every step of the execution. Abstraction means giving up some of this control, and this can be scary for a learner.

찜하세요~ 085 :

But the learner can work up to it -- introduce a variable, interactively control it, connect the variable to another value, interactively control it, turn the variable into a function argument, interactively control it. **The learner always gets the experience of interactively controlling the lower-level details, understanding them, developing trust in them, before handing off that control to an abstraction and moving to a higher level of control.**

The environment must support this process. A typical text editor only provides direct support for growing "outward" -- adding new lines of code. The environment must also support growing "upward" -- abstracting over existing code.*

찜하세요~ 086 :

* The environment should encourage the learner to **start constant, then vary**, by providing meaningful ways of gradually and seamlessly transitioning constant expressions into variable expressions.
* The environment should encourage the learner to **start with one, then make many**, by providing ways of using those variable expressions at a higher level, such as function application or looping.

찜하세요~ 087 :

LANGUAGE

A programming system has two parts. The environment is installed in on the computer, and the language is installed in the programmer's head.

The design of the language is just as critical to the programmer's way of thinking as the design of the environment. In the best cases, they are co-designed and inseparable. Many recent learning environments use JavaScript or Processing, and for the sake of comparison, the examples in this essay used them as well. But neither is a well-designed language for learning.

찜하세요~ 088 :

Fortunately, there are giant shoulders to stand on here -- programming systems that were carefully and beautifully designed around *the way people think and learn*. This section will briefly offer some design principles that have been distilled from these great systems of the past.

Great works

The canonical work on designing programming systems for learning, and perhaps the greatest book ever written on learning in general, is Seymour Papert's "Mindstorms".

찜하세요~ 089 :

Designing a learning system without a solid understanding of the principles in this book is like designing a mechanical system without understanding "the lever". Or "gravity". If you are reading this essay (and I'm pretty sure you are!) then you need to read "Mindstorms".

Seriously. I mean it. If you are going to design anything whatsoever related to learning, then you literally need to read "Mindstorms".

For fuck's sake, [read "Mindstorms"](http://books.google.com/books?id=HhIEAgUfGHwC&printsec=frontcover).

찜하세요~ 090 :

This section will make reference to four seminal programming systems that were designed for learning, and I strongly recommend studying each of them.



To be clear, I'm not advocating *using* any of these systems, in either their historical or modern incarnations. I'm advocating *understanding* them, and building on their insights.

찜하세요~ 091 :

Identity and metaphor

In **Logo**, the programmer draws pictures by directing the "turtle", an onscreen character which leaves a trail as it moves:



Watch just two minutes of this video -- the children, and the beardy guy talking:

<iframe width="420" height="315" src="http://www.youtube.com/embed/BTd3N5Oj2jk?rel=0#t=37" frameborder="0" allowfullscreen></iframe>

찜하세요~ 092 :

That's Seymour Papert explaining the Logo turtle. The turtle serves a number of brilliant functions, but the most important is that the programmer can *identify* with it. To figure out how to make the turtle perform an action, the programmer can ask how she would perform that action herself, if *she were the turtle*.

찜하세요~ 093 :

For example, to figure out how to draw a circle, a learner will walk around in circles for a bit, and quickly derive a "circle procedure" of taking a step forward, turning a bit, taking a step forward, turning a bit. After teaching it to herself, the learner can then teach it to the computer.* The turtle is the in-computer embodiment of the programmer herself, a "self", like the player-character in a video game, and thereby allows the learner to transfer her knowledge of her own body into knowledge of programming.

찜하세요~ 094 :

Every programming language is made of metaphors, but some fit the mind better than others. Standard imperative programming uses the metaphor of "assigning to variables", shuffling bits between little boxes. Unlike the Logo turtle, this metaphor was not designed to resonate with how people learn and understand; it simply evolved as a thin layer over the metaphors used in the underlying machine architecture, such as "storing to memory".*

찜하세요~ 095 :

Smalltalk, like Logo, also has a strong resonant metaphor, which is the *message*. All computation in Smalltalk is represented by objects sending and responding to messages from other objects. In order to program the behavior of an object, the programmer casts herself into the role of that object (to the extent of referring to the object as "self"!) and thinks of herself as *carrying on a conversation* with other objects. This is a powerful metaphor, because role-playing and conversing are powerful innate human facilities. As with Logo, tremendous time and thought went into the crafting and honing of Smalltalk's metaphors.

찜하세요~ 096 :

In **HyperCard**, the program is represented as a stack of cards, with the programmer drawing objects onto each card. Unlike a typical programming language, where an "object" is an abstract ethereal entity floating inside the computer, every object in HyperCard has a "physical presence" -- it has a location on a particular card, it can be seen, it can be interacted with. Every object in

HyperCard is a "real thing", and this is a powerful metaphor which allows programmers to apply their intuition and understanding of the physical world.

찜하세요~ 097 :

Rocky's Boots is structured as a video game, with a player-character that can be moved around directly. The player not only can pick up and move objects, but also acts as a power source -- a *literally* powerful metaphor. Everything is visible and tangible -- electricity is not some abstract voltage reading, but can be seen directly as orange fire, flowing through wires. This beautiful metaphor makes it trivial to *follow the flow and see the state*.

In **Processing**, by contrast, the programmer has no identity within the system. There are no strong metaphors that allow the programmer to translate her experiences as a person into programming knowledge. The programmer cannot solve a programming problem by performing it in the real world.

찜하세요~ 098 :

Processing's core metaphor is the "painter's algorithm" -- the computer places a series of shapes on the screen, like drawing on paper. Because this metaphor carries no computational power (you cannot *compute* by filling in pixels), all computation occurs outside the bounds of the metaphor. In this example of a bouncing-ball animation --

```
<div class="example" data-no-marker="1">
	<video width="640" height="218" preload>
		<source
src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/Language5.mp4" type="video/mp4">
		<source
src="https://s3.amazonaws.com/worrydream.com/LearnableProgramming/Movies/Language5.webm" type="video/webm">
	</video>
</div>
```

-- the simulated properties of the ball (position, velocity) are not associated with the picture of the ball onscreen. They are computed and stored abstractly as "numbers" in "variables", and the ball is merely a shadow that is cast off by this ethereal internal representation. The ball cannot be picked up and moved; it cannot be told how to interact with other objects. It is not a "living thing", and the simulation cannot be *understood* or *thought* about in any way other than

"numbers in variables". This is a very weak way of thinking.*

찜하세요~ 099 :

Decomposition

Modularity is the human mind's lever against complexity. Breaking down a complex thing into understandable chunks is essential for understanding, perhaps the *essence* of understanding.

A programming language must encourage the programmer to *decompose* -- to approach a complex problem by breaking it into simpler problems. Papert refers to this as breaking down a program into "mind-size bites".

찜하세요~ 100 :

Logo uses the metaphor of "teaching the turtle a new word". To draw a face consisting of four circles, we can teach the turtle a subprocedure for drawing a circle, and then apply that subprocedure four times. Long and careful thought was given to the process by which a learner *discovers* the need for subprocedures, and then factors a large procedure into subprocedures.

Smalltalk is, in essence, a philosophy of decomposition in the form of a programming language. This is Alan Kay inventing objects:

찜하세요~ 101 :

> Bob Barton [said] "The basic principle of recursive design is to make the parts have the same power as the whole." For the first time I thought of the whole as the entire computer, and wondered why anyone would want to divide it up into weaker things called data structures and procedures. Why not divide it up into little computers... Why not thousands of them, each simulating a useful structure?

Smalltalk's key insight was that a complex computer program could be decomposed into smaller computers, called "objects". Programming in Smalltalk is almost entirely an exercise in decomposition -- breaking down thoughts into classes and messages.

찜하세요~ 102 :

Almost every computer language provides some facility for decomposition, but some are better than others. In his wonderful essay [Why Functional Programming Matters](http://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf), John Hughes argues that decomposition lies at the heart of the power of languages like Haskell:

> When writing a modular program to solve a problem, one first divides the problem into subproblems, then solves the subproblems, and finally combines the solutions. The ways in which one can divide up the original problem depend directly on the ways in which one can glue solutions together. Therefore, to increase one's ability to modularize a problem conceptually, one must provide new kinds of glue in the programming language.

찜하세요~ 103 :

> Functional languages provide two new, very important kinds of glue... This is the key to functional programming’s power -- it allows improved modularization.

Processing allows for Logo-style decomposition into subprocedures, in the form of function definitions. The more powerful Smalltalk-style decomposition, where submodules can be *thought about independently*, is not supported. In Processing, drawing and input events are tied to single entry points -- top-level functions such as "draw" and "mouseDown". The behavior of submodules must be tangled across these global functions. Clean decomposition is not possible.

찜하세요~ 104 :

Consider a programmer who has made a bouncing ball animation. How does she go from one ball to two, to a hundred? How does she make the balls bounce off one another? How does she make balls draggable with the mouse? In a genuine learning environment such as Etoys, this progression is natural and encouraged. In Processing, each of these steps is a nightmare of needless complexity.

A language that discourages decomposition is a language that cripples a programmer's most valuable way of thinking.

찜하세요~ **105 :**

Recomposition

Creating is remixing. To a large extent, new ideas are old ideas in new combinations.

A programming language must encourage *recomposition* -- grabbing parts of other programs, assembling them together, modifying them, building on top of them. This gives creators the initial material they need to create by reacting, instead of facing every new idea with a blank page. It also allows creators to *learn from each other*, instead of deriving techniques and style in a vacuum.

찜하세요~ **106 :**

HyperCard was designed for recomposition, and is perhaps still unsurpassed in that respect. Bill Atkinson fully intended for creators to assemble a program by copying and pasting objects from other programs, and then gradually tweaking and customizing them. Every program thus serves as a parts kit for creating new programs. Because all source code, if any, is embedded in individual objects in the form of scripts, and because scripts use loose, relative references to other objects, groups of related objects can be transplanted much more easily and successfully than in other systems.*

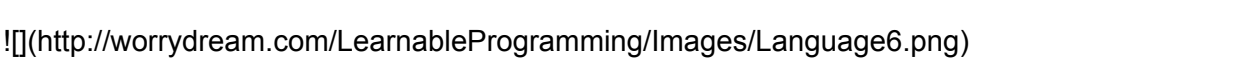
찜하세요~ **107 :**

Many people revere HyperCard for initiating them into programming. Any user can remix their software with copy and paste, thereby subtly transitioning from user to creator, and often eventually from creator to programmer.

Processing's lack of modularity is a major barrier to recomposition. The programmer cannot simply grab a friend's bouncing ball and place it alongside her own bouncing ball -- variables must be renamed or manually encapsulated; the "draw" and mouse functions must be woven together, and so on. One can easily start from an existing Processing program and modify it, but the language does not encourage *combining two programs*.

찜하세요~ **108 :**

Worse, Processing's dependence on global state hinders even the simplest forms of recomposition. As an analogy, imagine you're writing an email. You copy some red text from a website, paste it into your email, and *everything else in your email turns red*:



This is *exactly* what can happen when copying and pasting lines of Processing code, because Processing's way of handling color is inherently leaky:

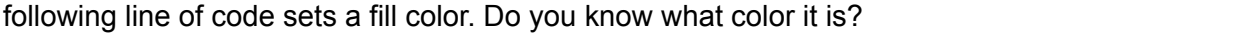


Experienced programmers might look at this example and consider this a programmer's error, because this is "just how code works." But this error is not intrinsic to programming; it's a consequence of specific design decisions -- mutable state, global variables, no encapsulation.

찜하세요~ 109 :

Worse yet, Processing has global modes which *alter the meaning of function arguments*. The following line of code sets a fill color. Do you know what color it is?

Trick question -- it's impossible to know what color it is, because the meaning of "255" depends on the global "color mode". It could be *any* of these colors:



If two Processing programs specify their colors in different color modes, then combining the two programs is almost hopeless.

Designing a system that supports recomposition demands long and careful thought, and design decisions that make programming more convenient for individuals may be detrimental to social creation.

찜하세요~ 110 :

Readability

A learner must be able to look at a line of code and know what it means.

Syntax matters. Here are two statements in HyperCard's scripting language, and their equivalents in a more conventional syntax:

    HyperTalk:
    Write "hello" to file "greetings".
    Drag from "0,0" to "100,100" with optionKey.

    C-like syntax:
    writeFile("hello", "greetings");
    dragMouse(0, 0, 100, 100, OPTION_KEY);

HyperTalk happens to use an English-like syntax, but that's not the point here. What matters is that every argument can be *understood in context*. It's clear that "hello" is a string and "greetings" is a filename, and that "0,0" and "100,100" are start and end points. In the conventional syntax, both are ambiguous.

As another example, here's how a programmer might draw an ellipse in three languages:

    Smalltalk:
    canvas drawEllipseCenteredAtX:50 y:50 width:100 height:100.

    Processing:
    ellipse(50,50,100,100);

    x86 assembly:
    push 100; push 100; push 50; push 50; call _ellipse

찜하세요~ 111 :

In Smalltalk, arguments have context. Processing's "ellipse" is exactly as cryptic as assembly language. The reader must look up or memorize every argument, a significant barrier to reading.

Names matter. Below are four array methods from Apple's Cocoa framework, and the equivalent JavaScript methods:

    mutate array and return nothing
    mutate nothing and return new array

    Cocoa:
    addObject
    addObjectsFromArray
    arrayByAddingObject
    arrayByAddingObjectsFromArray

    JavaScript:
    push
    splice
    concat
    concat

찜하세요~ 112 :

Cocoa follows strong grammatical conventions which immediately convey the meanings of methods. Verb phrases ("addObject") perform an action and return nothing. Noun phrases ("arrayByAddingObject") return the noun so named, and generally do not have stateful effects

unless the name suggests so. Expected arguments are clearly indicated by the name, in Smalltalk style. ("addObject" takes an object; "addObjectsFromArray" takes an array.) Most Cocoa code can thus be read and at least vaguely understood without documentation.

찜하세요~ 113 : 장재원찜 & 작업완료 & 리뷰요망

By contrast, many of Processing's function names are grammatically ambiguous or misleading. Many nouns, such as "ellipse" and "triangle", perform actions. Many verbs, such as "fill" and "stroke", do not.* The programmer constructs a color using a noun ("color"), and constructs an image using a verb ("createImage"). This sort of linguistic sloppiness is inappropriate, especially in a system for learning. A language must be parsed by people, not just compilers.

이와는 대조적으로 다수의 Processing 명령어들은 문법적으로 모호하거나 오류를 유발한다. ellipse나 triangle과 같은 명사(로 된 명령어들은) 동작을 수행한다. fill이나 stroke같은 동사들은 그렇지 않다. 프로그래머는 명사인 'color'를 사용해서 색깔을 넣고, 동사인 'createImage'를 사용해서 이미지를 넣는다. 이런류의 언어의 너저분함은 특히나 학습을 위한 체계로는 부적절한다. 언어는 컴파일러가 아니라 인간에 의해서 분석될 수 있어야 한다.

OKAY THEN!

그래 좋다!

The design principles presented in this essay can be used as a checklist to evaluate a programming system for learning.

이 에세이에서 제시하는 디자인 원칙은 프로그래밍 체계가 배우기에 적절한지 평가하는 점검표로 활용될 수 있다.

Does the *environment* allow the learner to...

*환경*이 학습자에게 다음과 같은 것을 허용하는가

찜하세요~ 114 :

* **read the vocabulary?** -- *Is meaning transparent? Is meaning explained in context, by showing and telling?*
* **follow the flow?** -- *Is time visible and tangible? At all meaningful granularities?*
* **see the state?** -- *Does the environment show the data? Show comparisons? Is hidden state eliminated?*
* **create by reacting?** -- *Is something on screen as soon as possible? Is the parts bucket on

the floor?*
* **create by abstracting?** -- *Can the programmer start concrete, then generalize?*

찜하세요~ 115 :

Does the **language** provide...

* **identity and metaphor?** -- *Is the computer's world connected to the programmer's world?*
* **decomposition?** -- *Can the programmer break down her thoughts into mind-sized pieces?*
* **recomposition?** -- *Can the programmer put diverse pieces together?*
* **readability?** -- *Is meaning transparent?*

This essay suggested some features and references that address these questions, but the questions matter more than my answers.

If you are designing a system and you can't answer these questions, it's time to reopen your sketchbook, because your design's not done yet.

찜하세요~ 116 :

These are not training wheels

These design principles were presented in the context of systems for learning, but they apply universally. An experienced programmer may not need to know what an "if" statement means, but she does need to understand the runtime behavior of her program, and she needs to understand it while she's programming.

A frequent question about the sort of techniques presented here is, **"How does this scale to real-world programming?"** This is a reasonable question, but it's somewhat like asking how the internal combustion engine will benefit horses. The question assumes the wrong kind of change.

찜하세요~ 117 :

Here is a more useful attitude: **Programming has to work like this.** Programmers *must* be able to read the vocabulary, follow the flow, and see the state. Programmers *have to* create by reacting and create by abstracting. Assume that these are *requirements*. Given these requirements, how do we *redesign programming*?

Here's an example. In many styles of programming today, when an application launches, it creates a large set of interconnected stateful objects. To see the effect of a code change, the application must be "relaunched" -- that is, its entire world is destroyed, and rebuilt again from scratch. How can we "create by reacting", continuously changing the code and seeing continuous effects in the flow and data, when there is no continuity between the application's state before and after the change?

찜하세요~ 118 :

We can't. That's the wrong question. A better question is: *How do we design a new programming model that does allow for continuous change?* We already have clear hints.*

Another example. Most programs today manipulate abstract data structures and opaque objects, not pictures. How can we visualize the state of these programs?

Again, wrong question. A better attitude is to assert that we have to be able to understand the state of our programs. We can then ask: *How do we design data structures that can be visualized?* Can we invent data structures that are *intended* to be visualized? How do we move towards a culture where only visually-understandable data is considered sound? Where opaque data is regarded in the same way that "goto" is today?*

찜하세요~ 119 :

In his influental essay [No Silver Bullet](http://faculty.salisbury.edu/~xswang/Research/Papers/SERelated/no-silver-bullet.pdf), Fred Brooks makes the case that software is inherently "invisible and unvisualizable", and points out the universal failure of so-called "visual programming" environments. I don't fault Fred Brooks for his mistake -- visual programming is indeed worthless. But that's because it visualizes the wrong thing.

찜하세요~ 120 :

Traditional visual environments visualize the code. They visualize static structure. But that's not what we need to understand. We need to understand what the code is doing.

Visualize data, not code. Dynamic behavior, not static structure.

Maybe we don't need a silver bullet. We just need to take off our blindfolds to see where we're

firing.

Much thanks to Star Simpson, Dan Amelang, Dave Cerf, Patrick Collison, Christina Cacioppo, and Oliver Steele for their feedback on this essay.

찜하세요~ 121 :

This essay was an immune response, triggered by hearing too many times that Inventing on Principle was "about live coding", and seeing too many attempts to "teach programming" by adorning a JavaScript editor with badges and mascots.

Please read [Mindstorms](http://books.google.com/books?id=HhIEAgUfGHwC&printsec=frontcover). Okay?