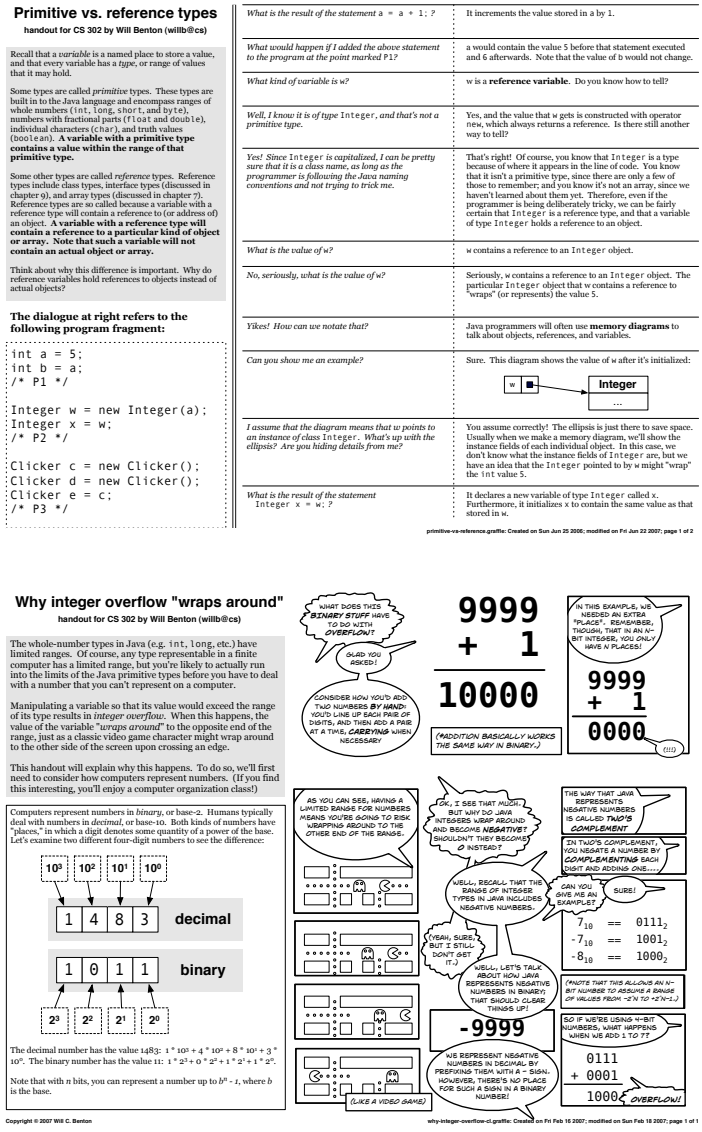
Why handouts?

handout for L&S TA training by Will Benton (willb@acm.org)

Why make handouts, you might ask, *when I could just distribute my slides?* Given the number of students who seem to demand hardcopy slides each term, this is an excellent question. I hope this handout will answer it to some extent.

I have included miniatures of three of my handouts from computer science courses on this page. (Notice that not every page follows my *Anatomy of a simple handout*.) I don't expect that you'll be interested in reading the actual content of these handouts; rather, I want you to glance at these to get an idea about how much *information* each can contain. Each also presents a prototype for a different kind of instructional handout: a dialogue, a comic strip, and an annotated figure.

Of course, you *could* simply make bad slides: put paragraphs of text on each, read every word, and use extremely small type so that you can fit more "stuff" on each slide. (Please don't do this!) Instead, you should use the strengths of slides and handouts to complement your presentations in different ways.



Primitive vs. reference types

handout for CS 302 by Will Benton (willb@cs)

Recall that a *variable* is a named place to store a value, and that every variable has a *type*, or range of values that it may hold.

Some types are called *primitive* types. These types are built in to the Java language and encompass ranges of whole numbers (`int`, `long`, `short`, and `byte`), numbers with fractional parts (`float` and `double`), individual characters (`char`), and truth values (`boolean`). **A variable with a primitive type contains a value within the range of that primitive type.**

Some other types are called *reference* types. Reference types include class types, interface types (discussed in chapter 9), and array types (discussed in chapter 7). Reference types are so called because a variable with a reference type will contain a reference to (or address of) an object. **A variable with a reference type will contain a reference to a particular kind of object or array. Note that such a variable will not contain an actual object or array.**

Think about why this difference is important. Why do reference variables hold references to objects instead of actual objects?

The dialogue at right refers to the following program fragment:

```
int a = 5;
int b = a;
/* P1 */

Integer w = new Integer(a);
Integer x = w;
/* P2 */

Clicker c = new Clicker();
Clicker d = new Clicker();
Clicker e = c;
/* P3 */
```

What is the result of the statement `a = a + 1;` *?*	It increments the value stored in a by 1.
What would happen if I added the above statement to the program at the point marked `P1`*?*	a would contain the value 5 before that statement executed and 6 afterwards. Note that the value of b would not change.
What kind of variable is `w`*?*	`w` is a **reference variable**. Do you know how to tell?
Well, I know it is of type `Integer`*, and that's not a primitive type.*	Yes, and the value that `w` gets is constructed with operator `new`, which always returns a reference. Is there still another way to tell?
Yes! Since `Integer` *is capitalized, I can be pretty sure that it is a class name, as long as the programmer is following the Java naming conventions and not trying to trick me.*	That's right! Of course, you know that `Integer` is a type because of where it appears in the line of code. You know that it isn't a primitive type, since there are only a few of those to remember; and you know it's not an array, since we haven't learned about them yet. Therefore, even if the programmer is being deliberately tricky, we can be fairly certain that `Integer` is a reference type, and that a variable of type `Integer` holds a reference to an object.
What is the value of `w`*?*	`w` contains a reference to an `Integer` object.
No, seriously, what is the value of `w`*?*	Seriously, `w` contains a reference to an `Integer` object. The particular `Integer` object that `w` contains a reference to "wraps" (or represents) the value 5.
Yikes! How can we notate that?	Java programmers will often use **memory diagrams** to talk about objects, references, and variables.
Can you show me an example?	Sure. This diagram shows the value of `w` after it's initialized:
I assume that the diagram means that `w` *points to an instance of class* `Integer`*. What's up with the ellipsis? Are you hiding details from me?*	You assume correctly! The ellipsis is just there to save space. Usually when we make a memory diagram, we'll show the instance fields of each individual object. In this case, we don't know what the instance fields of `Integer` are, but we have an idea that the `Integer` pointed to by `w` might "wrap" the `int` value 5.
What is the result of the statement `Integer x = w;` *?*	It declares a new variable of type `Integer` called `x`. Furthermore, it initializes `x` to contain the same value as that stored in `w`.

primitive-vs-reference.graffle: Created on Sun Jun 25 2006; modified on Fri Jun 22 2007; page 1 of 2

Why integer overflow "wraps around"

handout for CS 302 by Will Benton (willb@cs)

The whole-number types in Java (e.g. `int`, `long`, etc.) have limited ranges. Of course, any type representable in a finite computer has a limited range, but you're likely to actually run into the limits of the Java primitive types before you have to deal with a number that you can't represent on a computer.

Manipulating a variable so that its value would exceed the range of its type results in *integer overflow*. When this happens, the value of the variable "*wraps around*" to the opposite end of the range, just as a classic video game character might wrap around to the other side of the screen upon crossing an edge.

This handout will explain why this happens. To do so, we'll first need to consider how computers represent numbers. (If you find this interesting, you'll enjoy a computer organization class!)

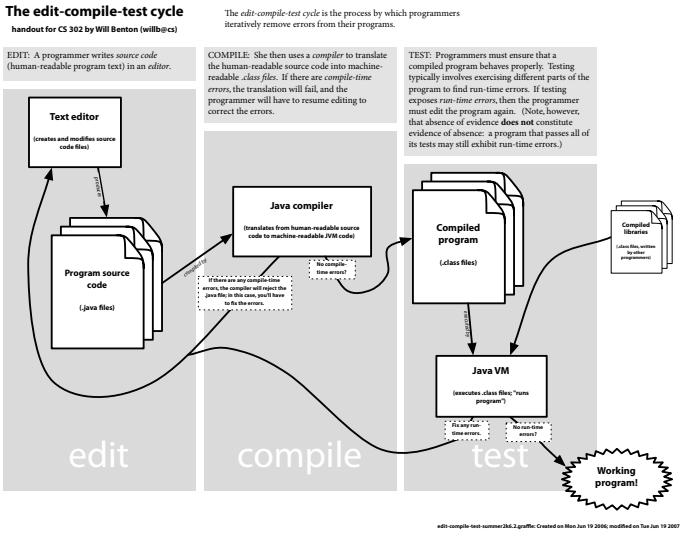
Computers represent numbers in *binary*, or base-2. Humans typically deal with numbers in *decimal*, or base-10. Both kinds of numbers have "places," in which a digit denotes some quantity of a power of the base. Let's examine two different four-digit numbers to see the difference:

The decimal number has the value 1483: $1 * 10^3 + 4 * 10^2 + 8 * 10^1 + 3 * 10^0$. The binary number has the value 11: $1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0$.

Note that with n bits, you can represent a number up to $b^n - 1$, where b is the base.

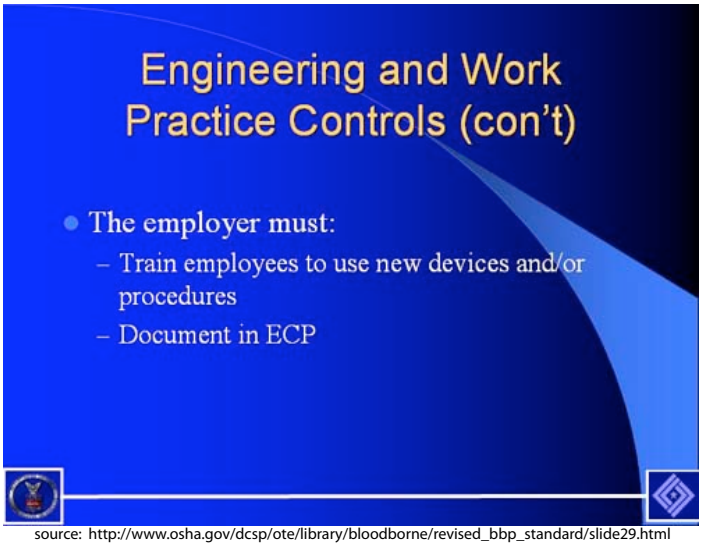
Copyright © 2007 Will C. Benton

why-integer-overflow-cl.graffle: Created on Fri Feb 16 2007; modified on Sun Feb 18 2007; page 1 of 1



edit-compile-test-summer2k6.2.graffle: Created on Mon Jun 19 2006; modified on Tue Jun 19 2007

Now that we've seen some handouts, let's consider slides. Beneath this text I have reproduced a typical slide from a US OSHA presentation. How much useful information can a slide convey? Is a slide really a helpful artifact for further study?



source: http://www.osha.gov/dcsp/ote/library/bloodborne/revised_bbp_standard/slide29.html

In addition to being only sparsely populated with information, the slide above has some other troublesome symptoms. Superficially, it is unattractive. (Sadly, most slide software does not provide adequate support for layout and design.) More functionally, its text appears to be intended for the benefit of the speaker rather than for the benefit of the audience. The appropriate place for an outline is in the speaker's private notes (or, ideally, in his or her memory); putting notes on the screen merely invites the audience to read instead of listening.

Slides should not merely be an outline or a standalone list of bullet points: they should serve and complement an existing presentation. Given the constraints of the slide medium, it is simply not possible for a slide to include as much information as a printed page. Therefore, you should make handouts to serve as durable reinforcements of the concepts you cover in a lecture or presentation, and design slides to support and illustrate your talk as it occurs.