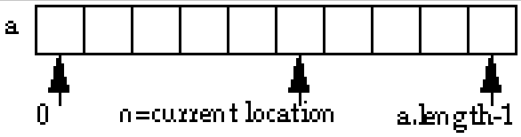
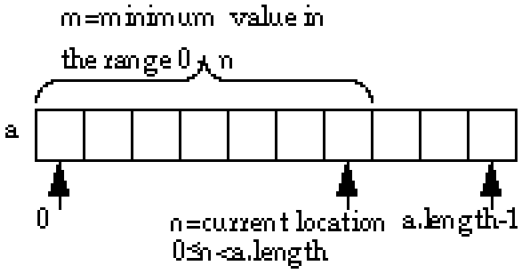
Tutorial on reasoning about Java programs with loops

Q	I find reasoning about programs confusing. I don't know where to start. I don't know what to write down. Anyway I know how to program and I don't see the point of it.	
A	The reasoning is to help you get it right — first time. And to understand your program fully. It also can be used to construct a program, and it often gives simpler programs than you might get if you hacked it. How about going through an example?	
Q	Yes, that might help. How about a simple program — say, to **find the minimum element of an array**?	
A	OK. The first step is to *give a program header and a precise precondition and postcondition*, preferably using logic. If you find logic impossible, English is better than nothing. But you did a whole course on logic, so you shouldn't find it too hard.	
Q	I think I can do the program header:	`int minval(int [] a)`
A	Good, that'll do. You could use reals instead of integers but the idea will be the same, so fair enough.	
Q	Now how do I do the precondition here?	
A	*Ask yourself if there are any restrictions on the input data.*	
Q	Maybe that the array is an array of integers?	
A	No need for that — you put it in the header already.	
Q	How about 0<a.length?	
A	Good idea – otherwise there would be no values to find the min of!	
Q	Now the postcondition. It should be: — I think.	`//pre: 0<a.length` `//post: r = min(a)`
A	OK, but it's a bit informal.	
Q	Why?	
A	Well, "min" normally applies to two numbers, not a whole array's worth. And if you don't define what min means, and min on arrays is not a basic Java method, how are you going to verify that the program calculates it?	
Q	Good point. How about	`//post: r=min{a[0],…,a[a.length-1]}`
A	Better. Could you do it formally in logic if you had to?	
Q	How about (I write **r** for the result returned by the function.)	`∀i:int(0≤i<a.length⇒ r≤a[i])`
A	That says **r** is <= any element in a. But then **r** might be < min(a). You have to say **r** occurs as a value in a as well.	
Q	Ah: so try	`∀i:int(0≤i<a.length⇒ r≤a[i])` `∧∃i:int(0≤i<a.length∧a[i]=r).`
A	Very good.	
Q	So do we use the min version or the logical version?	

A	If you're doing it properly, the logical one. But it can take ages to do. So stick with the simple one. We've got:	`int minval(int [] a)`
		`//pre: 0<a.length`
		`//post: r=min{a[0],…,a[a.length-1]}`

Q	Now what?	
A	*How is the program going to work? What's the idea?*	
Q	Do a loop. Scan through a, keeping track of the least element found.	
A	Good. So *draw a diagram showing the situation at the beginning of an arbitrary iteration of the loop.*	
Q	Here you are:	
A	Not enough detail. That picture would work for just about any problem. So it doesn't help.	
Q	So? I mean, what's the point of a diagram?	
A	What is the point ever? It helps you to think about exactly what's happening. I think it should mention and give the meaning of every variable in the program.	
Q	OK, we'll need a variable, say n, to mark the current place as we scan through (that's in the diagram already), and another, say m, to keep the minimum so far.	
A	Good. Now draw it.	
Q	Here it is. I marked the range that n can take: `0 <=n<a.length`. I think you said you need that to check (i) that array accesses in the loop are legal, and (ii) that n has a particular value (probably a.length-1 here) when the loop terminates — you need this to get the postcondition right.	
A	Exactly. Great. Now you can *do the invariant and variant.*	
Q	How?	
A	The variant is easiest — it *measures how much work there is left to do at that point.*	
Q	So it's how much of the array we haven't seen yet: that is [looks at diagram, to count] a.length-1-n.	`//variant: a.length-1-n`
A	Good. The invariant, on the other hand, is a logical condition expressing that *the program is doing OK so far. It should say what the diagram says, but in logic.*	
Q	But how do I do that?	
A	What are the distinct points the diagram makes?	
Q	Well, it says `0 <=n<a.length`..	
A	Good — that will be in the invariant. What other points does it make?	
Q	Well, it says m is the minimum of the values in a up to n.	
A	Good. How d'you say that in logic?	
Q	How about	`m = min{a[0] .. n}`

A	Well, OK, but you didn't define the notation min{a[0]□ n}. Maybe add :	`// where min{a[0]□ n} means min{a[0],a[1],…,a[n]}`
Q	Right. Should I have used the long way instead?	
A	Not necessarily: we're not trying to make things harder than they have to be, and it's a good idea to use abbreviations, *if you define what they mean.* Now are there any more points the diagram makes?	
Q	— I can't see any.	
A	Are you sure? Have you covered everything you need for the program to be doing OK so far, at the beginning of an iteration of the loop?	
Q	I mentioned all the program variables...and said how they are related. I think I got the lot.	
A	Fine. So can you write the invariant now?	
Q	Alright then, try:	`//invariant: 0≤n<a.length` `//  & m = min{a[0]□ n}`
A	Excellent! You did it!	
Q	Fine. Now what?	
A	*Write the code. Include the precondition, postcondition, invariant and variant as comments.*	

	I like this bit. Here we go:
	
	That should work.
		`int minval(int [] a) {`
		`//pre: 0<a.length`
		`//post: r=min{a[0],…,a[a.length-1]}`
		`  int n = 0;  int m = a[n];`
		`  while (n<a.length-1)`
		`  //invariant: 0≤n<a.length`
		`  //  & m = min{a[0]□ n}`
		`  //variant: a.length-1-n`
		`  n++; if (a[n]<m) m = a[n];`
		`  }`

A	Now you have to verify it.	
Q	I don't like this bit.	
A	That's because you aren't used to it. In fact the verification can go hand in hand with writing the code. Try the first step: *show the initialisation code establishes the invariant.*	

Q	That's this bit:
	
	So what do I do? I don't know how to start.
		`//pre: 0<a.length`
		`//post: r=min{a[0],…,a[a.length-1]}`
		`  int n = 0;  int m = a[n];`
		`  while (n<a.length-1)`
		`  //invariant: 0≤n<a.length`
		`  //  & m = min{a[0]□ n}`

A	*To establish the invariant, find out what values the variables have when the invariant is reached, and see if the invariant is true for those values.*	
Q	Well, n will be 0 and m will be a[n]. So the invariant will be:	`0≤0<a.length` `& a[n] = min{a[0]□ n}`
A	But it's still got n's in. Get rid of them.	
Q	OK: we get: The first line is obviously true. But how can I prove it?	`0≤0<a.length` `& a[0] = min{a[0]□ 0}`

A	It's just true. You don't have to prove it. Formally, 0<=0 is true because the ordering <= on integers is reflexive, and 0<a.length is true because the pre-condition says so. So no, you don't need to prove it.	
Q	How do I know when I have to prove something?	
A	If it's true just because of properties of integers or of Java, you don't have to prove it. But if it relies on things your particular program did, then you do, because you're trying to verify the program works.	
Q	Seems reasonable, I suppose. Now the second line?	`a[0] = min{a[0], 0}`
A	This again is obvious. If we expand the abbreviation min{a[0], 0}, we get: and this equation is obviously true. Unless you want to define min formally ...	`a[0] = min{a[0]}`
Q	No, no, I don't mind, really. So that establishes the invariant. Quite easy really.	
A	Note that you could have *developed* the code by looking to see what you needed to do to establish the invariant. The invariant is true when n=0 and m = a[0], so if the code sets up these values the invariant will be established.	
Q	So we can actually construct programs this way ... [mildly impressed] Now how d'you re-establish the invariant?	
A	*Assume the invariant is true and the loop exit test is false at the beginning of some (arbitrary) iteration of the loop.*	

Q	Right, so we have	`invariant true: 0≤n<a.length`
		`& m = min{a[0], n}`
		`while condition true: n<a.length-1`

A	Now here's the loop code again:
	
	See what it does to the variables, substitute their values at the end of this iteration into the invariant, and see if it's true.
		`while (n<a.length-1)`
		`//invariant: 0≤n<a.length`
		`//  & m = min{a[0], n}`
		`//variant: a.length-1-n`
		`n++; if (a[n]<m) m = a[n];`

Q	OK, well, n is incremented by 1. So if I substitute n+1 for n in the invariant, the first half of it becomes:	`0≤n+1<a.length`
A	Good. Is it true?	
Q	[thinks...] Yes, because we already had 0<=n, and the while condition was true, so n<a.length-1. So how do I write that down?	
A	You just did.	
Q	So this will do—?	`By the old invariant and the true while condition, 0≤n<a.length-1.` `So 0≤n+1<a.length.`
A	Sure, I love it. We're not trying to nit-pick in this course. We just want to know the reasons why things are true.	

Q	I get the idea now. What about the second half of the invariant? I can substitute n+1 for n:
	But I don't know what 'm' is after the iteration. It depends on the "if" test:
	So what do I doooo?
		`m = min{a[0], n+1}`
		
		`if (a[n]<m) m = a[n];`

A	You could take it case by case: *first assume the test in the "if" is true, and check the invariant is reestablished; then do it for when the if-test is false.* That's like -elimination. *It will always work.* There's another way which often works. Look at the *difference between the old and the new invariant. The new invariant (after the iteration) is usually the old one (before the iteration), plus an extra bit,* because the loop has done a bit more work after one more iteration. So try to show the code covers that extra bit.	`[exercise: do it this way!]`
Q	So how d'you do that here?	
A	We're trying to prove that after the iteration we have: What's this in unabbreviated form?	`m = min{a[0], n+1}`
Q	In full, it is	`m = min{a[0], …, a[n], a[n+1]}.`
A	And what did this bit of the invariant say before the iteration? Call the old value of m "m1" if you like.[1]	
Q	It said:	`m1 = min{a[0], …,a[n]}.`
A	And what's the difference between the two?	
Q	Well, m is the min. of all the items that m1 is the min of, plus the extra one, a[n+1]. So I think we have	`m = min{m1,a[n+1]}.`
A	Good! I think you can say that without proof. Now does the code set m to min{m1,a[n+1]}?	

Q	Yes! It does!! The code increments n first, so by the time we get to the loop test, we're dealing with n+1. So in effect, the code sets m to a[n+1] if a[n+1]<m1. This means m ends up the minimum of m1 and a[n+1]. That's what I was thinking of when I wrote the code.
	(I even thought of writing m=min(m,a[n]) but I thought you'd object.) Is that it?
		`while loop code:`
		
		`n++;`
		`if (a[n]<m) m = a[n];`

A	Yes.	
Q	How d'you prove it?	
A	You just did. If you know why it's true, you're 80% towards a proof. Just write what you know and why you know it.	
Q	OK, I'll try, but this is hard.	Re-establishing m = min{a[0], n+1} Before the iteration, the invariant gives m1 = min{a[0], …, a[n]}. After the iteration the invariant is m = min{a[0], …, a[n], a[n+1]}. That is, m = min{m1, a[n+1]}. But the code sets m to the minimum of its old value (m1) and of a[n+1]. QED.
A	Excellent. You should say somewhere that m1 is the value of m before the iteration.	
Q		m1 is the value of m before the iteration.

[1]This is as in lectures. As there are 2 cases for m, we're in danger of confusing the old and new values of m, so having separate symbols for the old and new values of m may help here. We don't really need to bother with this for "n", as there's only 1 possible value of n after the iteration.

A	Thank you. Now *show the loop terminates.*	
Q	How d'you do that?	
A	*Show the variant starts out >=0 on the first iteration,* *drops by at least 1 in each iteration,* *but never goes negative without the loop terminating.*	
Q	Right, here goes:	The variant is a.length-1-n. At the beginning of the first iteration, n is 0. So the variant is a.length-1, and this is ≥0 by the pre-condition.
A	Good. Next bit?	
Q		Each iteration increases n by 1. So the variant drops by ≥1 each iteration.
A	Doing well. How about the last bit?	
Q		The while condition is false if n≥a.length-1, which says a.length-1-n≤0. So if the variant gets to be ≤0, the loop will not execute.
A	Wonderful. Now *show the polishing-off code sets up the postcondition.*	
Q	Right. What do I have to do?	
A	*Assume the invariant and loop exit test are both true, and show that the code after the loop makes the postcondition true. First, write them out:*	invariant: 0≤n<a.length & m = min{a[0]… n)} Loop exit test: n>=a.length-1 post: **r**=min{a[0],…,a[a.length-1]}
Q	Now what?	
A	*Show that the code after the loop makes the postcondition true.* What result does the code return?	
Q	— it doesn't! I forgot! The code should be	minval(int [] a) { … … … **return m;**
A	Naughty. At least the verification caught the mistake. That is what it's for. *Now* can you verify it?	
Q	We know **r** = m. (**r** is the result returned.)	**r**=m
A	So substitute m for r in the postcondition.	
Q	OK: — ah, we know that's true. m is min(a), because the invariant gives and n should be a.length-1 now as it starts at 0 and goes up by 1 each iteration so it'll eventually hit a.length-1 and then the loop will exit.	m = min{a[0],…,a[a.length-1]}. m = min{a[0]… n}
A	Well OK, but this is a bit vague. Can you show n=a.length-1 *just from the invariant and loop test?*	
Q	Errm,	
A	Write them again.	
Q	Here you go... Yes — n>=a.length-1 and n<=a.length-1, so n=a.length-1.	invariant: 0≤n<a.length & m = min{a[0]… n)} Loop exit test: n>=a.length-1
A	GOOD. So write the proof out in order.	

Q		After the loop, the invariant and exit test are both true. So: 0≤n<a.length & m = min{a[0]… n)} and n>=a.length-1. Therefore n=a.length-1. So m = min{a[0]… a.length-1}. As the code returns m, we get **r** = min{a[0]… a.length-1}. This is the postcondition.

A	Very good. Now you only have to *check array accesses are legal in the program.* Here it is again.
	
	Where are the array accesses?
		int minval(int [] a) {
		//pre: 0<a.length
		//post: **r**=min{a[0],…,a[a.length-1]}
		int n = 0; int m = a[n];
		while (n<a.length-1)
		//invariant: 0≤n<a.length
		// & m = min{a[0]… n}
		//variant: a.length-1-n
		n++; if (a[n]<m) m = a[n];
		return m;
		}

Q	Here: and here:	int m = a[n]; if (a[n]<m) m = a[n];
A	Good. Are they legal? Do they always have indexes between 0 and a.length-1?	
Q	The first one does, because n was set to 0 initially.	
A	Fine. What about the ones in the loop? Remember these accesses occur as many times as the loop is done, so you'd better only use the invariant and failure of the loop exit test to verify them.	

Q	Well, we have:
	So after incrementing n, it's <=a.length-1 and >=0 (we did this before). So the a[n] accesses are legal.
		//invariant: 0≤n<a.length
		& m = min{a[0]… n)}
		//while condition true:n<a.length-1
		n++; if (a[n]<m) m = a[n];

A	That's it. Easy, eh?	
Q	Will we have to do all this in the exam?	
A	I sincerely hope so. But honestly, it can be quite long, so often you get asked to do only a part of it, like re-establishing the invariant.	
Q	Phew.	
A	But not always—	
Q	[Transfers to a cheaper university]	

IMH, Feb 00 (with a few minor changes by KB 02)