Lecture 1 : Introduction to Programming in Java

Lecturer : Susan Eisenbach

This is the 1st lecture on Java programming. This course is primarily about writing imperative programs using the Kenya system.

Next term you will learn to write object oriented Java programs.

Textbooks

- No textbook is required.
- For programming beginners:
  Java Software Solutions: Foundations of Program Design, John Lewis and William Loftus, Publisher: Addison Wesley, 2002.
- For experienced programmers:
  - **Learning the Java™ Language** at http://java.sun.com/docs/books/tutorial/
  - **Thinking in Java**, Bruce Eckel, Prentice Hall

Software is required

- http://www.doc.ic.ac.uk/kenya/
- download Java onto your home machine
- follow the instructions to install it
- then follow the instructions to install either Kenya or KenyaEclipse

Functional versus imperative languages

- Functional languages are ideal for expressing the functional (the problem to be solved) component of any problem however...
- at least 50% of all programs deal with input/output rather than a problem and functional languages aren't very good at input/output.
- Think of the programs you use now:
  editor
  mail
  language translator (Haskell or Java)
  web browser
- Functional programming should have taught you to appreciate concise elegant programs.

A statement written in Java

```
println("Write this in Haskell!");
```

Commented version

```
/* Susan Eisenbach
 * 12 November 2007
 * a bit of bravado
 */
String exclaim = "Write this in Haskell!";
println(exclaim);
```

every statement is terminated with a ;

print() and println()

- Text can be printed on the screen using `print()` or `println()`.
- Using `println(" ")` puts a carriage return on the end of the line.

```
print( "7*3" );
println( "=" ); println( 7 * 3 );
```

This code prints:

```
7*3=
21
```


Concatenating output with +

- `String drink = "slammers";`
  `print("I like "); println(drink);`
  This code prints: `I like slammers`
- `println("I like Tequila " + drink);`
  This code prints: `I like Tequila slammers`
- `println`
  `("6/9 = " + 6/9 + " or " + 6.0/9.0);`
  This code prints:
  `6/9 = 0 or 0.6666666666666666`

Comments

- There are two ways of commenting code.
- `//` comments are terminated by the end of line

```
// Susan Eisenbach
// 12 November 2007
// a bit of bravado
```

- `/*` comments in Java are also terminated by `*/`

```
/* Susan Eisenbach
 * 12 November 2007
 * a bit of bravado
 */
```

good to make several lines of comments stand out in your program

A function written in Haskell

```
bigger :: Int -> Int -> Int
-- post: returns the larger of two numbers
bigger a b|a>b = a
          |otherwise = b
```

argument types

arguments

Same method written in Java

```
int bigger(int a, int b){
//post: returns the larger of the 2 values
  if (a > b) {return a;}
        else {return b;}
  }
```


A function written in Haskell

```
bigger :: Int -> Int -> Int
-- post: returns the larger of two numbers
bigger a b|a>b = a
          |otherwise = b
```

result type

Same method written in Java

```
int bigger(int a, int b){
//post: returns the larger of the 2 values
  if (a > b) {return a;}
        else {return b;}
  }
```


A function written in Haskell

```
bigger :: Int -> Int -> Int
-- post: returns the larger of two numbers
bigger a b|a>b = a
          |otherwise = b
```

then and else branches are surrounded by `{ }`

Same method written in Java

```
int bigger(int a, int b){
//post: returns the larger of the 2 values
  if (a > b) {return a;}
        else {return b;}
  }
```

method bodies are surrounded by `{ }`

Returning from a method and conditionals

```
int bigger(int a, int b){
//post: returns the larger of the 2 values
  if (a > b) {return a;}
  else {return b;}
  }
```

predicate (test) must be in brackets `()`

results have to be returned using the keyword `return`

conditionals - using the keywords `if` and optionally `else`

A function written in Haskell

```
biggest :: Int -> Int -> Int -> Int
-- post: returns the largest of 3 numbers
biggest a b c = bigger a (bigger b c)
```

Same function written in Java

```
int biggest(int a, int b, int c){
//post: returns the largest of the 3 values
     return bigger(a, bigger(b,c));
  }
```


A Java program must contain a main method

- It is the main method that starts the execution of a program off.
- It doesn't return anything. The return type of a method that does not return anything is void.
- The first statement can be made into a program as follows:

```
void main(){
     println("Write this in Haskell!");
}
```

- By custom the main method is the first method in the program.


```
/*Susan Eisenbach
 *12 November 2007
 *chooses the largest of 3 numbers
 */
void main(){
  print("Type in your 3 numbers -> ");
  println(biggest(readInt(),readInt(),readInt()));
}
int bigger(int a, int b){
//post: returns the larger of the 2 values
  if (a > b) {return a;}
      else {return b;}
}
int biggest(int a, int b, int c){
//post: returns the largest of the 3 values
   return bigger(a, bigger(b,c));
}
```


Variable declarations

- Variables are names of storage locations. Variables can be declared of the following types:
  `int double boolean char String`
- They must be declared before they are used.

```
int j;
double cost;
String firstname; String surname;
```

- Variables can be initialised in declarations

```
int total = 0;
char answer = 'y';
double start = 0;
double sum = 0.0;
boolean finish = false;
```

The assignment statement

- Initialisation is a form of *assignment*.
- Assignment gives a variable (named storage location) a value.
- variables can have their values changed throughout a program.

```
total = total + 1;
total = total / 2;
answer = 'n';
```

- Haskell does not have such a low level feature.

Assignment - don't use too many variables

superfluous to requirements

Poor style	Better style
`int i = 6;`	`int i =6;`
`int j = 5;`	`int j = 5;`
`int k;`	`println(i+j);`
`k = i+j;`	
`println(k);`	

Summary

- The syntax of the Java programming language is introduced in this course for coding solutions to the problems set.
- We have seen
  - methods (Haskell functions) with { }
  - statement terminators - ;
  - variables
  - conditionals – if (predicate) {…} else {…}
  - assignments
  - input/output
  - main method
  - complete Java program

Lecture 2: Recursion

Lecturer : Susan Eisenbach

For extra material read parts of chapters 1,3 and 11

of

Java Software Solutions

Revision from Haskell

- Define the base case(s)
- Define the recursive case(s)
  - Split the problem into subproblems
  - Solve the subproblems
  - Combine the results to give required answer

Haskell program -> Java method

```
divisor :: Int -> Int -> Int
--pre:  the arguments are both
--      integers > 0
--post: returns the greatest common divisor
divisor a b | a==b  = a
            | a>b = divisor b (a-b)
            | a<b = divisor a (b-a)
```


becomes:

```
int divisor (int a, int b){
assert (a > 0 && b > 0):
"divisor must be given arguments > 0";
//post: returns the gcd of a and b
     if (a == b) {return a;}
     else {if (a > b) {return divisor (b, a - b);}
           else {return divisor (a, b - a);}}
}
```


What does assert do?

```
assert (a > 0 && b > 0):
"divisor must be given arguments > 0";
```

evaluates the predicate

true? – continue to execute the code

false? – print the string on the screen and stop the program

Do not execute code which you know may crash or loop forever.

When should you have an assertion?

- If you write a method that expects something special of its inputs then you need to put as a precondition whatever needs to be true before the code can be run.
- The precondition should be coded (if possible) as an assertion.
- Assertions can also be written without the String message. In this case, if the assertion fails then your program stops with an AssertionError.
- If the user has given a method arguments that meet the precondition and the code is correct then the postcondition to the method will hold. Postconditions are written as comments at the top of the method after the word post.

Haskell program -> Java method

```
fact :: Int -> Int
--pre: n>= 0
--post: returns n!
fact 0      =  1
fact (n+1)  =  (n+1)* fact n
```

- *becomes*:

```
int fact( int n ){
assert (n>= 0 && n < 17):
"factorial must be given an argument >= 0";
//post: returns n!
  if (n==0) {return 1;}
  else {return n*fact(n-1);}
}
```


Java method -> Java program

```
void main(){
  print("Factorial number that you want? ");
  println("Answer = " + fact(readInt()));
}
int fact( int n ){
assert (n>= 0):
"factorial must be given an argument >= 0";
//post: returns n!
  if (n==0) {return 1;}
  else {return n*fact(n-1);}
}
```

- Rewrite this program with a more efficient `fact` method.

Methods

- Haskell has functions that return results.
- Java has methods that return results (just like Haskell)
- Java has methods that don't return any values, they just execute some code.
  - their return type is `void`.
  - they frequently consume input and/or produce output
  - The special `main` method must be `void`.
- Both types of methods can be recursive.
- Java programs can *never* be recursive.

Menu method

```
void menu(){
//post: 5 lines of text appear on the screen
  println( "Enter 0 to quit" );
  println( "Enter 1 to add" );
  println( "Enter 2 to subtract" );
  println( "Enter 3 to multiply" );
  println( "Enter 4 to divide" );
}
```


processMenu method

```
void processMenu(int reply){
assert (0<= reply && reply <=4);
  switch (reply){
    case 0:  {println("Bye"); break;}
    case 1:  {println(readInt()+readInt()); break;}
    case 2:  {}
    case 3:  {}
    case 4:  {println(" not yet implemented"); break;}
    default: {println("not possible!");}
  }
}
```

curly brackets are used for each case

switch

int used to choose the case to execute, chars can also be used

```
switch (reply){
  case 0: {println("Bye"); break;}
  case 1: {a = readInt(); b = readInt();
           println(a+b); break;}
  case 2:{}
  case 3:{}
  case 4: {println(" not yet implemented"); break;}
  default:{println("not possible!");}
 }
}
```

break prevents cases falling through

if integer does not match any case -- not required

Question

- Rewrite processMenu changing it in two ways.
  - remove the precondition
  - produce the correct answer on the screen for each of the operations
- *Notes*
  - only read in the numbers once
  - you can put `if`s and `switch`es inside each other or themselves
  - tell the user there is an error if `reply` < 0 or `reply` > 4

Input

- There are a huge number of ways of reading input into Java programs.
- Whitespace means what you get when you hit the space bar or the enter keys.
- We are using the Kenya system which contains:
  - `readInt()` -ignores whitespaces, stops after the last digit
  - `readDouble()` -ignores whitespaces, stops after the last digit
  - `readString()` -ignores whitespaces, stops on the first whitespace after the string
  - `readChar()`-ignores whitespace, then reads one character
  - `read()` - reads the next character (even if it is whitespace)
- readSomething() consumes the carriage return character.

Developing a Java program to reverse a string

- *Specification*:
  - The program should accept a line of text and print it out in reverse order.
- *Remember*:
  - A program cannot be recursive only a method can.
- The main program just calls the method reverse

```
reverse:
  Read a character  //progress- one char closer to CR
  If CR not yet reached //guard the recursive call
     then
          reverse
          print Character.
```


IMPORTANT

- Guard your recursive calls.
- Not guarding your recursive calls can lead to infinite recursion.
- Make sure there is progress towards the terminating condition between invocations of the recursive routine.
- Comment both the guard and the progression.

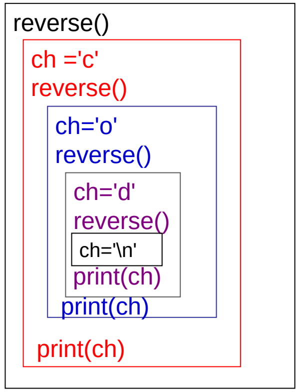
The program

```
void main(){
  print("type in your word to reverse ->");
  reverse();
}
void reverse(){
//post: reads in a string (terminated by ENTER '\n')
//      and prints it out in reverse order
char ch;
  ch = read(); //progress- one char closer to CR
  if (ch != '\n') {
    reverse();
    print(ch);
  }
}
```

Program Input cod Output d o c

```
reverse();

void reverse(){
char ch;
  ch = read();
  if (ch != '\n'){
    reverse();
    print(ch);
  }
}
```



Summary

- A routine that calls itself is called recursive.
- Methods can be recursive, programs cannot.
- Recursive methods that produce a single result are just like Haskell functions.
- Void methods are used when the same operation is to be performed on different data and the result wanted is output on the screen.
- In order that the repetition may be finite, within every recursive method there must appear a terminating condition to guard the recursive call and a progression to distinguish one call from another.
- Switch statements are used rather than conditionals when there are several choices based on an integer or character.

Lecture 3 : Arrays and For Loops

Lecturer : Susan Eisenbach

For extra material read parts of chapters 3 and 6 of Java Software Solutions.

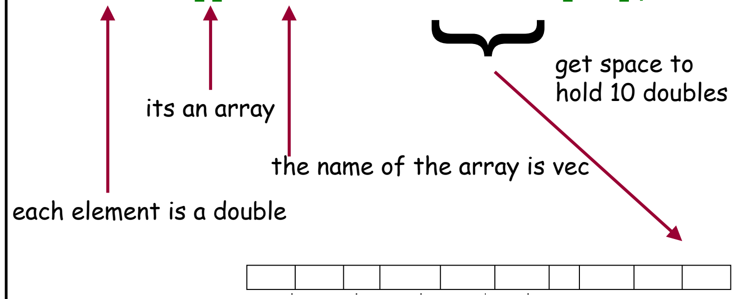
This is the 3rd lecture on Java in which arrays and for loops are examined.

What is an array?

- for problems which deal with large quantities of data
- perform the same, operations on the individual items
- elements of an array are all of the same type and referred to by an index
- arrays can be one or more dimensional
- arrays are called vectors and matrices by non-computing people
- comparison with Haskell lists
  - every element can be accessed with equal ease
  - multi-dimensional arrays are easy to access

Example of an array variable declarations

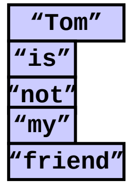
```
double[] vec = new double[10];
```



Another example of an array variable declaration

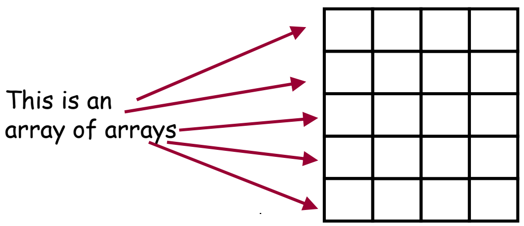
If we want to store the sentence "Tom is not my friend" we would use:

```
String[] words = new String[5];
```



Examples of array variable declarations

```
int[][]mat = new int[5][4];
```



Arrays can be initialised at declaration

- ```
  String[] names = {"Bradley","Eisenbach",
                    "Gillies","Field",
                    "Hodkinson"};
  ```
- ```
  double[] vector = {0.1, 1.2, 0.0, 34.6,
                     -3.0, 34.1, 0.0,
                     0.4, 0.8, 0.1};
  ```

Getting the size of an array

- To get the size(no. of elements) of an array, you write
  `arrayname.length`
- The length of the array is determined by the number of values provided between `{ }`.
- for example if

```
boolean[] answers = {true, false,
                     true, true, false};
```

then `answers.length` is `5`

- Note that `length` is not a method and so does not have `( )`. It is built into Java.
- Once created, the size of the array cannot change.
- The length of the array must be specified when it is created.

Examples of array variable declarations (cont.)

How do you declare and initialise a data structure for the following?

Susan		Eisenbach
Antony	John	Field
Christopher	John	Hogger

```
String[ ][ ] fullNames = {
      {"Susan","","Eisenbach"},
      {"Antony","John","Field"},
      {“Christopher",“John","Hogger"}
};
```


Referencing array elements

- each array element is referenced by means of the array identifier followed by an index expression which uniquely indexes that element
- the first element of an array is at 0, the last at length - 1
- example array references:

```
firstName = fullNames[2][1];

vec[1] = mat[1][0]+mat[1][1]+mat[1][2]+
         mat[1][3]+mat[1][4];

if (i==j) {mat[i][j] = 1;}
else {mat [i][j] = 0;}
```


Using arrays:

- You can pass arrays as arguments to methods:
  `void printNames(String[ ][ ] names)`
- You can return arrays as results from methods:
  `String[ ][ ] copy(String[ ][ ]names)`
- ✖ Do not assign complete arrays:
  `secondYears = firstYears`

  since any change to `firstYears` will happen to `secondYears` as well (more later on this).

Bounded iteration

- arrays exist in order to hold a large number of elements of the same type
- frequently the same operation is performed on each array value
- traversing all the elements of an array can be achieved by means of a control construct known as the **for loop**. Using a `for` statement is called *looping* or *iteration* and causes repetitive execution

```
for (int i =  lowerbound; i < upperbound ; i++) {
     loop body
}
for (int i = upperbound - 1; i >= lowerbound; i--) {
     loop body
}
```


Bounded iteration

on each loop iteration
i=i+1

```
for (int i =  lowerbound; i < upperbound; i++) {
   loop body
}
```

usually 0

first value of i

```
for (int i = upperbound-1; i >= lowerbound; i--) {
   loop body
}
```

stops executing the loop when
i hits this value

on each loop iteration
i=i-1

A for loop example

- for loops are ideal for traversing arrays - each iteration of the loop accesses an element of the array
- a program to calculate the mean of an **array of doubles**:

```
void main(){
  double[] vec = {1,0,3,0,5,0,7,-2,9,10};
  println( mean(vec) );
}
double mean(double[ ] v){
//post:returns the average of the elements in v
double total = 0;
  for (int i =0; i < v.length; i++) {
    total = total + v[i];
  }
  return total/v.length;
}
```


Tracing the execution of some code

- When trying to understand what some piece of Haskell code does, you use rewrites:
- *fact 4 = 4*fact 3 = 4*3*fact 2 = 4*3*2*fact 1 = 4*3*2*1= 24*

When trying to understand what some piece of Java code does you hand execute all the code working out what the values of the variables are:

```
void main(){
  double[] vec = {1,0,3,1,5};
  println( mean(vec) );
}
double mean(double[] v){
int i;
double total = 0.0;
  for (int i =0;i<v.length;i++)
  {
    total = total + v[i];
  };
  return total/v.length;
}
```

mean	total	i	v[0]	v[1]	v[2]	v[3]	v[4]
			1	0	3	1	5

When trying to understand what some piece of Java code does you hand execute all the code working out what the values of the variables are:

```
void main(){
  double[] vec = {1,0,3,1,5};
  println( mean(vec) );
}
double mean(double[] v){
int i;
double total = 0.0;
  for (int i =0;i<v.length;i++)
  {
    total = total + v[i];
  };
  return total/v.length;
}
```

mean	total	i	v[0]	v[1]	v[2]	v[3]	v[4]
			1	0	3	1	5
	0						
	1	0					
	1	1					
	4	2					
	5	3					
	10	4					
		5					
2							

Nested for loops

- a 2-dimensional array requires 2 **for** loops to traverse it:

```
int sum(int[ ][ ] m){
//post: returns the sum of the elements of m
int theSum = 0;
  for (int i = 0; i<m.length; i++){
    for (int j = 0; j<m[i].length; j++){
      theSum = theSum + m[i][j];
    }
  }
  return theSum;
}
```

- an n-dimensional array requires n for loops to traverse
- If **int mat[50][100]** is passed to sum, what is the value of **m.length**? For each **i**, what is the value of **m[i].length**?

What is the value of **names.length**?
For each **i**, what is the value of **names[i].length**?

```
void main(){
  String[][] students ={
    {"BSc", "Homer", "Marge", "Bart", "Lisa", "Maggie"},
     {"MSci", "Moss", "Jen", "Roy"},
     {"BEng", "Peter", "Lois", "Meg", "Brian", "Stewie",},
     {"MEng", “Harry", "Adam", "Ros", "Malcolm","Zafar","Connie"}
  };
   printNames( students );
}
void printNames(String[ ][ ] names){
    for (int i = 0; i < names.length; i++) {
      print(names[i][0] + ": ");
        for (int j = 1; j < names[i].length; j++) {
          print(names[i][j] + " ");
        }
       println();
     }
  }
```

Summary

- Arrays are data structures suitable for problems dealing with large quantities of identically typed data where similar operations need to be performed on every element.
- Elements of an array are accessible through their index values. Arrays using a single index are called vectors, those using *n* indices are *n-dimensional arrays*. A two dimensional array is really an array of arrays, a 3-dim., an array of arrays of arrays, etc.

Summary

- Arrays have a type associated with them: the type of the elements. The index is always a non-negative integer.
- Space has to be allocated explicitly for arrays. Either they are initialised with values and then the right amount of space is allocated or the keyword **new** is used to specify the allocation of space.
- Repetition of the same operation is called *iteration* or *looping*. A **for** loop can be used to do the same operation to every element of an array.

Lecture 4 : Using Arrays

Lecturer : Susan Eisenbach

For extra material read parts of chapters 3 and 6 in Java Software Solutions.

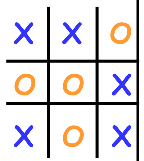
This is the 4th lecture in which code is developed that uses arrays and parameter passing mechanism is examined in some detail.

Consider a game to play noughts and crosses

- Assuming that each space on the board can have a 'X', an 'O' or a ' ', write an array declaration to hold a board, initialising it all to empty.
- `char[ ][ ] board = {{' ',' ',' '}, {' ',' ',' '}, {' ',' ',' '}};`
- Write a statement that puts a 'X' into the middle square
- `board[1][1] = 'X';`

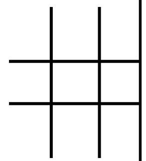
Write a predicate isFull which returns true iff there are no empty spaces on the board.

```
boolean isFull(char[ ][ ] b){
  for (int r = 0; r < 3; r++){
        for (int c = 0; c < 3; c++){
            if (b[r][c] == ' ')
               {return false;}
        }
    }
    return true;
}
```



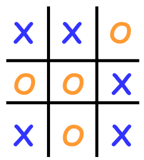
Write a method initBoard which returns a new board filled with spaces.

```
char[ ][ ] initBoard(){
    char[ ][ ] b = {{' ',' ',' '},
                    {' ',' ',' '},
                    {' ',' ',' '}
                   };
    return b;
}
```



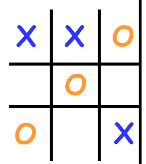
Checking for a winner

- After each move a check has to be made whether the current player has won.
- A player has won if their piece fills one of the diagonals, one of the rows or one of the columns.



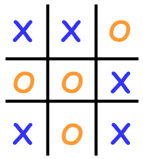
Write a predicate isDiagonal that takes as arguments an X or O and a board and returns true iff one of the diagonals is filled with the piece.

```
boolean isDiagonal(char ch, char[ ][ ] b){
assert (ch='X' || ch='O');
     return b[0][0]== ch &&
            b[1][1]== ch &&
            b[2][2]== ch ||
            b[0][2]== ch &&
            b[1][1]== ch &&
            b[2][0]== ch;
}
```



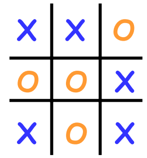
Write a predicate `hasFullRow` that takes as arguments an X or O and a board and returns true iff one of the rows is filled with the piece.

```
boolean hasFullRow(char ch, char[ ][ ] b){
assert (ch='X' || ch='O');
boolean found;
     for (int r = 0; r < 3; r++){
          found = true;
          for (int c = 0; c < 3; c++){
               found = found && b[r][c] == ch;
          }
          if (found) {return true;}
     }
     return false;
}
```



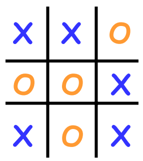
Write a predicate `hasFullCol` that takes as arguments an X or O and a board and returns true iff one of the columns is filled with the piece.

```
boolean hasFullCol(char ch, char[][] b){
assert: ch='X' || ch='O';
boolean found;
     for (int c = 0; c < 3; c++){
          found = true;
          for (int r = 0; r < 3; r++){
               found = found && b[r][c] == ch;
          }
          if (found) {return true;}
     }
     return false;
}
```



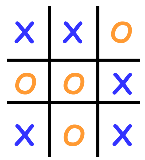
Exercises to do yourself:

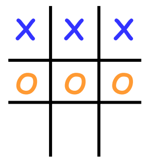
- Write the predicate hasFullCol that takes as arguments an X or O and a board and returns true iff one of the columns is filled with the piece using only one loop.
- Rewrite the code in the slides with the board as a one dimensional array. How much harder is it to write the predicates?



Put them together to produce a predicate `isWinner`

```
boolean isWinner(char ch, char[][] b){
assert: ch='X' || ch='O';
   return isDiagonal(ch,b) ||
          hasFullRow(ch,b) ||
          hasFullCol(ch,b);
}
```





How do you get which square the next player wants?

- You could (mouse) click on the square on the screen and the coordinates could be converted into the appropriate noughts and crosses index.
- This requires very sophisticated input routines.
- Simpler would be to read in from the keyboard chess notation for the square and then convert it to the appropriate array indices.
- So if a user wants the middle square, it is b2 or 2b and the bottom lefthand corner is c1 or 1c

a	X	X	O
b	O	O	X
c	X	O	X
	1	2	3

You need to know if the character the user typed in is for a row or a column

- Write a predicate `IsRow` which takes as an argument a character and returns true iff the argument is an `'a'`, a `'b'` or a `'c'`.

```
boolean isRow(char c){
    return 'a' <= c && c <= 'c';
}
```

- Write a predicate `IsCol` which takes as an argument a character and returns true iff the argument is an `'1'`, a `'2'` or a `'3'`.

```
boolean isCol(char c){
    return '1' <= c && c <= '3';
}
```


Convert the input characters into numbers that can be used for array indices

- Write a method `convert` that takes a character that is a valid row or column and returns the appropriate number to use for the row index or column index. So if `'2'` is passed as an argument to `convert` it returns `1`.

```
int convert(char c){
assert (isRow(c) || isCol(c));
   if (c=='1' || c=='a') {return 0;}
   if (c=='2' || c=='b') {return 1;}
   return 2;
}
```


Arguments to methods

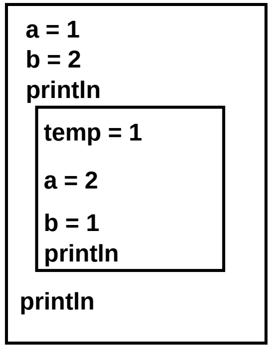
- We have been passing arguments to methods.
- Java's argument passing is slightly more restrictive than Haskell's – you can pass anything to a Java method, except another method.
- In Java methods, arguments are *passed by value*. When invoked, the method receives the value of the variable passed in, creates a local copy, works on it and then discards it when the method is left.
- This means that a method cannot change the value of its arguments.

What happens when you pass a variable to a method and change its value within the method?

```
void main(){
int a = 1;
int b = 2;
  println("a & b = " +  a + b);
  swap(a,b);
  println("after swap " + a + b);
}
void swap(int a, int b){
//post: this method does very little
int temp = a;
   a = b;
   b = temp;
   println("inside swap " + a + b);
}
```

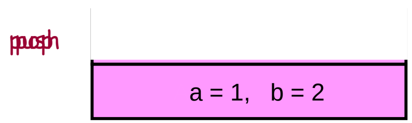

What happens when you pass a variable to a method and change its value within the method?

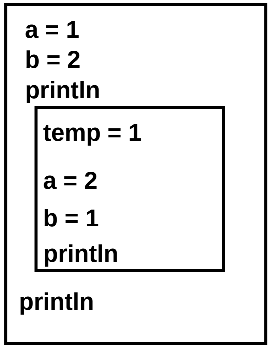
```
void main(){
int a = 1;
int b = 2;
  println("a & b = " +  a + b);
  swap(a,b);
  println("after swap " + a + b);
}
void swap(int a, int b){
//post: this method does very little
int temp = a;
   a = b;
   b = temp;
   println("inside swap " + a + b);
}
```




```
a & b = 1 2
inside swap 2 1
after swap 1 2
```


When a method is called the runtime system of any language holds method data in a stack





Summary

- A predicate is a method that returns a boolean result.
- It is sensible to name predicates starting with **is**, **are**, **can** or some other similar word.
- We have developed a variety of methods that are necessary if one is writing a noughts and crosses game.
- In Java methods, arguments are *passed by value*. When invoked, the method receives the value of the variable passed in, creates a local copy, works on it and then discards it when the method is left.
- This means that a method cannot change the value of its arguments.

Tutorial questions

1. Translate the following Haskell functions into Java functions.

```
power :: Int -> Int -> Int
-- pre:  the second argument is a non-negative
    integer
-- post: computes first arg to power of second arg
power x n | n==0 = 1
          | otherwise = x * power x (n-1)
```



```
int power(int x, int n){
assert (n >= 0);
//post: x^n
    if (n==0) {return 1;}
    else {return x * power(x, n-1);}
 }
```



```
power1 :: Int -> Int -> Int
-- pre:  the second arg is a non-negative integer
-- post: computes first arg to power of second arg
power1 x n |n==0 = 1
       |n==1 = x
       |n `mod` 2 == 0 = z*z
       |otherwise
           = z*z*x
              where z = power1 x (n `div` 2)
```

- In Java `div` is / and `mod` is %


```
int power1(int x, int n){
assert (n > 0);
//post: x^n
    if (n==0) {return 1;}
    else{
         if (n==1) {return x;}
         else{
              int z = power1 (x, n / 2);
              if ((n%2) == 0) {return z*z;}
              else {return z*z*x;}
         }
    }
 }
```

2. Write a Java function that calculates factorials using a **for** loop. In Java you can have for loops that go backwards. They are of the form:

```
for (i = 10; i > 0; i--) {
  loop body
}
```



```
int fact(int n){
assert (n>= 0&& n <17);
//post: computes n!
    int f = 1;
        for (int i=n; i>0; i--){
            f = f*i;
        }
        return f;
    }
```


Lecture 5 : Classes

Lecturer : Susan Eisenbach

For extra material look at chapters 2 and 4 of Java Software Solutions.

This is the 5th lecture in which classes and objects are introduced.

Tuples

- When you want to group many of the same type of element together in Haskell you use a list, in Java you use an array.
- You access elements in a list through the head and an array by index (position of element).
- Sometimes you want to group a few items of (possibly) different types together.
- In Haskell you would use a tuple. The *position* of the piece of data would tell you what it was.
- In Haskell you wanted to hold an applicant's name followed by the A level points of the top 3 A levels you might say:
  - `type Applicant = ([Char], Int, Int, Int)`
  - and you would know that the name was the *first* element of an `Applicant`.

In Java there are classes

- Classes can be used like tuples, (although they are much more powerful as you will see later in the course)
- Classes contain fields and fields are accessed by name (not position like tuples)

```
class Applicant{
  String name;
  int grade1;
  int grade2;
  int grade3;
}
```

- Classes are types, (like Haskell **types**). You create the type with the class declaration and then you need to declare variables of the type you have created.

Using variables of type Applicant

```
Applicant me;
Applicant you;
     me.name = "Susan";
     me.grade1 = 60;
     me.grade2 = 40;
     me.grade3 = 0;
```

```
class Applicant{
     String name;
     int grade1;
     int grade2;
     int grade3;
}
```

- Classes can be passed as arguments and returned from methods

```
boolean areSame(Applicant a1, Applicant a2)
```

- What is the difference between the two statements?
  - `println(me.name + (me.grade1 + me.grade2 + me.grade3));` Susan100
  - `println(me.name + me.grade1 + me.grade2 + me.grade3);` Susan60400

Draw a diagram to understand

- Draw a diagram of an `Applicant`.

Susan	60	40	0
name	grade1	grade2	grade3

- Rewrite the class declaration for `Applicant` so the three grades are held in an array. Draw a diagram of your new class.

Shakil	120	120	120
	0	1	2
name	grades		

```
class Applicant{
     String name;
     int[] grades =
          new int[3];
}
```


Example of an array of classes

```
class Applicant{
  String name;
   int[] grades = new int[3];
}
void main(){
  Applicant[] firstYears = new Applicant[138]; …
```

good style to put class declarations at top of program

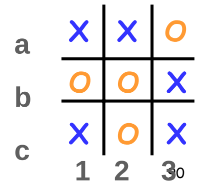
Anita	120	120	120	0
Bintal	120	120	120	1
Carl	120	120	100	2
:				. .
Wing	120	120	120	137

You need to be able to access elements

- How do you access the fifth person's second A level result?
- `firstYears[4].grades[1]`
- How do you write a method that takes `firstYears` as an argument and returns the index to the first student with name `"Dimitri"` and `-1` if there is no Dimitri in `firstYears`.
-
```
int find(String name, Applicant[] aa){
    for(int i=0; i<aa.length; i++){
         if (name == aa[i].name) {return i;}
    }
    return -1;
}
```


Back to noughts and crosses – getting the user's input

- Read in from the keyboard chess notation for the square and then convert it to the appropriate array indices.
- So if a user wants the middle square, it is b2 or 2b and the bottom righthand corner is c3 or 3c



You need to know if the character the user typed in is for a row or a column

- the predicate `IsRow` which takes as an argument a character and returns true iff the argument is an `'a'`, a `'b'` or a `'c'`

```
boolean isRow(char c){
    return 'a' <= c && c <= 'c';
  }
```

- the predicate `IsCol` which takes as an argument a character and returns true iff the argument is an `'1'`, a `'2'` or a `'3'`

```
boolean isCol(char c){
    return '1' <= c && c <= '3';
  }
```


Convert the input characters into numbers that can be used for array indices

- the method `convert` that takes a character that is a valid row or column and returns the appropriate number to use for the row index or column index. So if `'2'` is passed as an argument to `convert` it returns `1`.

```
int convert(char c){
assert (isRow(c) || isCol(c));
   if (c=='1' || c=='a') {return 0;}
   if (c=='2' || c=='b') {return 1;}
   return 2;
}
```

Declare a class to hold the coordinates of a move

- The coordinates need to be integers so they can index the board array.

```
class Coord{
    int row;
    int col;
}
```

- Next write a method **getMove** which reads from the keyboard the user's input. If it isn't a legal input (forget whether the square is occupied) then prompt again and read in the user's input. Continue this until correct input is typed in. The method should return a **Coord**.

Declare getMove

- **getMove** takes no arguments (it reads its inputs from the keyboard) and returns a **Coord**. `Coord getMove()`
- What local variables are needed by **getMove**?
  -two variables to hold the input characters

```
char c1, c2;
```

  -a variable to hold the coordinates of the move to be returned

```
Coord move;
```


What is the algorithm for getMove?

```
c1 = readChar()
c2 = readChar()
  if isRow(c1) && isCol(c2)
    move.row = convert(c1)
    move.col = convert(c2)
    return move
  else if isCol(c1) && isRow(c2)
       move.row = convert(c2)
       move.col = convert(c1)
       return move
     else println("bad coordinates, re-enter-->")
       return getMove()
```


getMove in Java

```
Coord getMove(){
Coord move;
char c1; char c2;
  c1 = readChar();
  c2 = readChar();
  if (isRow(c1) && isCol(c2)){
      move.row = convert(c1);
      move.col = convert(c2);
      return move;
  }
  else{
      if (isCol(c1) && isRow(c2)){
          move.row = convert(c2);
          move.col = convert(c1);
          return move;
      }
      else{
          println("bad coordinates, re-enter-->");
          return getMove();
      }
  }
}
```

While loops

- for loops are ideal to use with arrays, where you know exactly the number of iterations.
- When you want repetition and you don't know in advance how many times the repetition will occur you can use recursion or a *while loop* construct.
- It is a matter of taste whether you use while loops or recursion when you don't know beforehand how many times you need to repeat.
- Like recursion generalised loops can go infinite. When writing code you must ensure that your code will terminate.

Loops

```
while (condition)
{
  body of loop
}
```

the *body of loop* includes something that will make the condition go false eventually

- A loop where the condition cannot become false is an infinite loop. The loop below will not stop.

```
while (true)
{
  body of loop
}
```


Example of a generalised loop

```
void main(){
  char answer = 'y'; char buf = ' ';
  showInstructions();
  while ( answer == 'y' || answer == 'Y' ){
     playGame();
     print("Do you want to play again(y/n)? ");
     answer = readChar();
     buf = readChar() ;
     }
     println("thanks for playing");
}
```

buf is needed to hold the Return character which is never used, just discarded

Example of a generalised loop

- The rest of the code:

```
void showInstructions(){
  println("instructions go here");
}
void playGame(){
  println("the whole game goes here");
}
```

- ¿How would you implement this recursively?
- Trace the code with the input y y k. In addition to the column for the variable answer have a column for the methods being executed, a column for input and a column for output.

meth in ans output method

meth	in	ans	output
main		y	
ShowInstructions			instructions go here
playGame			the whole game goes here
main	y	y	Do you want to play again(y/n)? y
playGame			the whole game goes here
main	y	y	Do you want to play again(y/n)? y
playGame			the whole game goes here
main	k	k	Do you want to play again(y/n)? k

Summary

- To group a few items of (possibly) different types together a *class* is used.
- Access is by *field name* (not position).
- To access the field `f` in class `C` we write `C.f`
- It is good style to place all the class declarations at the very top of the program before the `main` method.

Lecture 6 : Objects

Lecturer : Susan Eisenbach

For extra material read chapter 4 of Java Software Solutions.

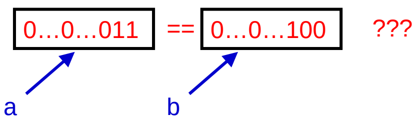
This is the 6th lecture in which primitive types and objects are introduced. The last tutorial question is also gone over.

Primitive values

- `int`s, `double`s, `boolean`s, `String`s, and `char`s are primitive.
- Java has many other number types that are also primitive.
- Primitive variables are names for storage locations that contain the values of the variables.
- What happens when during the execution of your program checking the expression `a==b` where `a` and `b` are both `int`s is reached?

Primitive values

- What happens is the bit pattern that is the value at the storage location called **a** is compared with the bit pattern at the storage location called **b**.
- If they are identical the value of the expression is true otherwise it is false.
- `int a = 3;`
- `int b = 4;`
- `a==b?`



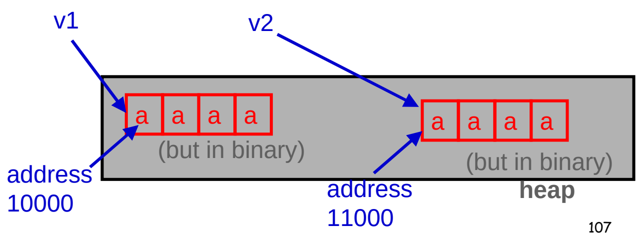
Objects

- Arrays and classes are not primitive. They are data structures and stored as objects.
- Objects (of an array or class type) need to have their space explicitly allocated before they can be used.
- For arrays you do this explicitly by using **new**.
- If you look at the Java code for class declarations generated by the Kenya system you will see the word **new**. This word means create the object on the heap.
- Object variables are names for storage locations that contain a *reference* or pointer to the data structure.
- The actual data structure is stored in a part of memory called the *heap*.

How objects are stored

- Consider the following declarations:

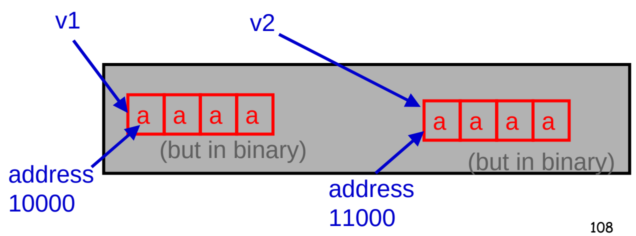
```
char[ ] v1 = {'a','a','a','a'};
char[ ] v2 = {'a','a','a','a'};
```



How objects are stored

- What are the values of **v1** and **v2**?

The two references (or addresses 10000 and 11000) to locations in the heap.



What gets printed?

```
void main(){
char[ ] v1 = {'a','a','a','a'};
char[ ] v2 = {'a','a','a','a'};
     if (v1 == v2) {println("same");}
     else {println("different");}
     v2 = v1;
     v1[0] = 'b';
     if (v1 == v2) {println("same v2[0]="+v2[0]);}
     else {println("different");}
}
```

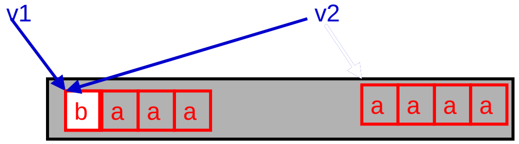
```
different
same v2[0]= b
```


Why?

```
void main(){
char[ ] v1 = {'a','a','a','a'};
char[ ] v2 = {'a','a','a','a'};
      if (v1 == v2) {println("same");}
      else {println("different");}
      v2 = v1;
      v1[0] = 'b';
      if (v1 == v2) {println("same v2[0]="+v2[0]);}
      else {println("different");}
}
```

```
different

same v2[0]= b
```



Take care with = and ==

- If you wish to assign one object to another then you must do it component at a time. Otherwise you will just have 2 names (known as aliases) for the same object.
- Instead of writing

  `me = you;`

  you should write

  `me.name = you.name;`

  `me.grade1 = you.grade1` etc. Then you will have two different objects with the same values.

Java provides `arraycopy` for copying arrays.

- `arraycopy` takes source and copies it to destination
- What gets printed?

```
int[ ] v1 = {1,1,1,1};
int[ ] v2 = {2,2,2,2};
arraycopy(v1,v2);
v1[0] = 33;
for (int i=0; i < v2.length; i++){
   print(v2[i]);
}
```

```
1111
```

Arguments to methods - repeat of earlier slides (reminder)

- We have been passing arguments to methods.
- Java's argument passing is slightly more restrictive than Haskell's - you can pass anything to a Java method, except another method.
- In Java methods, arguments are *passed by value*. When invoked, the method receives the value of the variable passed in, creates a local copy, works on it and then discards it when the method is left.
- This means that a method cannot change the value of its arguments.

What happens when you pass a variable to a method and change its value within the method?

```
void main(){
int a = 1;
int b = 2;
  println("a & b = " +  a + b);
  swap(a,b);
  println("after swap " + a + b);
}
void swap(int a, int b){
//post: this method does very little
int temp = a;
   a = b;
   b = temp;
   println("inside swap " + a + b);
}
```


What happens when you pass a variable to a method and change its value within the method?

```
void main(){
int a = 1;
int b = 2;
  println("a & b = " +  a + b);
  swap(a,b);
  println("after swap " + a + b);
}
void swap(int a, int b){
//post: this method does very little
int temp = a;
   a = b;
   b = temp;
   println("inside swap " + a + b);
}
```

```
a & b = 1 2
inside swap 2 1
after swap 1 2
```

a = 1
b = 2
println

temp = 1
a = 2
b = 1
println

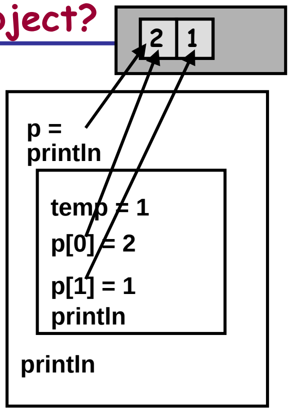
println

What happens when you pass an object to a method and alter the object?

```
void main(){
int[ ] p = {1,2};
  println("p[0] & p[1] = "  + p[0] + p[1]);
  swap(p);
  println("after swap "  + p[0] + p[1]);
}
void swap(int[ ] p){
int temp = p[0];
  p[0] = p[1];
  p[1] = temp;
  println("inside swap " + p[0] + p[1]);
}
```

What happens when you pass an object to a method and alter the object?

```
void main(){
int[ ] p = {1,2};
  println("p[0] & p[1] = " + p[0] + p[1]);
  swap(p);
  println("after swap "  + p[0] + p[1]);
}
void swap(int[ ] p){
int temp = p[0];
  p[0] = p[1];
  p[1] = temp;
  println("inside swap " + p[0] + p[1]);
}
```




```
p[0] & p[1] = 1 2
inside swap 2 1
after swap 2 1
```


What happens when you pass an object to a method and alter the object?

- What is passed to a method is the address of the object.
- Like arguments, this is copied and the local copy is worked on and then discarded, at the end.
- However the object lives in the heap and there is no such thing as a local heap.
- Any alterations to the heap that happen during the execution of a method are permanent.

Details about the heap

- Both arrays and classes are objects that when created live in the heap, which is just a special part of computer memory.
- Anything that lives in the heap must get allocated some space in the heap before it can be accessed.
- The way an array or class is accessed is via its address in the heap also called a pointer.
- If an object has not been allocated space then the address will be a special one called a Null Pointer.
- If you try to access an object that has not yet been allocated some space then you will get a NullPointerException.
- A NullPointerException means you tried to access an object, which did not exist.

What does this program print out?

```
void main(){
int[ ] p = {10,20};
    println("p[0] & p[1] = "  + p[0] + " " + p[1]);
    change1(p);
    println("p[0] & p[1] = "  + p[0] + " " + p[1]);
    change2(p);
    println("p[0] & p[1] = "  + p[0] + " " + p[1]);
}
void change1(int[ ] p){
    int[ ] q = {99,999};
    p = q;
    println("inside change1: p[0] & p[1] = "+p[0]+" "+p[1]);
}
void change2(int[ ] p){
    p[0] = 1000;
    println("inside change2: p[0] & p[1] = "+p[0]+" "+ p[1]);
}
```

Answer:

```
p[0] & p[1] = 10 20
inside change1: p[0] & p[1] = 99 999
p[0] & p[1] = 10 20
inside change1: p[0] & p[1] = 1000 20
p[0] & p[1] = 1000 20
```

- Why do you get this output?

Consider an array of classes

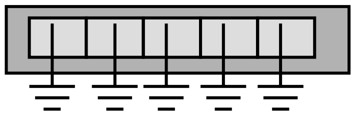
```
class Thing{
    int value = 0;
    char answer = 'y';
}

void main(){
  printThings(initThings());
}

void printThings( Thing[] tt){
    for (int i = 0; i < tt.length; i++){
        println("value = "+tt[i].value+" answer = "+tt[i].answer);
    }
}
```


How do we write **Thing[]initThings**?

- Within **initThings** we need first to create an array.
- **Thing[] tt = new Thing[5];**
- If **tt** just held integers then each array element would contain a 0.
- But since each array element holds an object it needs to contain the address of the object.
- As none of the objects have yet been given addresses then each array element contains the special null address which is normally shown with the symbol you can see below:



To initialise every cell in **tt**

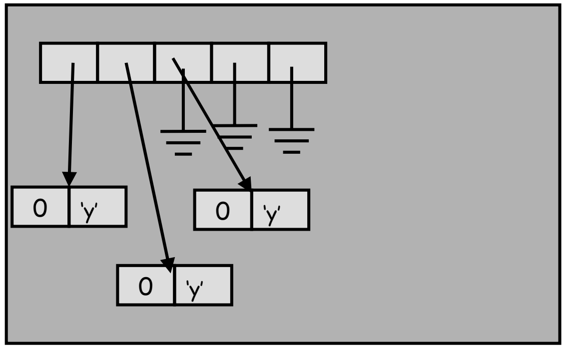
```
for (int i = 0; i < tt.length; i++){
        Thing t;
        tt[i] = t;
    }
```

- If you check each array element before the statement **tt[i] = t;** has been executed you will see that it contains the value null.
- If you check each element after **tt[i] = t;** has been executed you will see that it has been initialised.
- Execute the program in step mode and check the array values at each for loop iteration.

What happens when you execute one iteration of the for loop (say i = 2)

```
Thing t;

tt[2]=t;
```



The complete program

```
class Thing{
    int value = 0;
    char answer = 'y';
}

void main(){
    printThings(initThings());
}


Thing[] initThings(){
    Thing[] tt = new Thing[5];
    for (int i = 0; i < tt.length; i++){
        Thing t; //grabs appropriate space on the heap
        tt[i] = t;
    }
    return tt;
}

void printThings( Thing[] tt){
    for (int i = 0; i < tt.length; i++){
        println("value = " + tt[i].value + " answer = " + tt[i].answer);
    }
}
```


Summary

- Variables declared as a class or array type are objects and not primitive. This means they are actually references to memory addresses in the heap.
- Tests for equality and assignment have to be undertaken subcomponent by subcomponent.
- Arrays can be assigned using `arraycopy`.
- Objects are held on the heap and when changed in a method are permanently changed.

Lecture 7 : Enumerated Types and Simulation

Lecturer : Susan Eisenbach

For extra material read chapter 3 of Java Software Solutions.

This is the 7th lecture in which enumeration types are explained and a simulation program for a vending machine is developed.

Kenya has enumerated types like Haskell

- An *enumerated type* is a type whose legal values consist of a fixed set of constants.
- When the data your program uses is not numeric then using an enumerated type makes your program more readable and hence more maintainable
- Haskell:

```
data Day =
      Sun|Mon|Tues|Wed|Thurs|Fri|Sat
```

- Kenya:

```
enum Day{
  SUN,MON,TUES,WED,THURS,FRI,SAT;
}
```

- By convention the constants are all written in upper case.

Enumerated types - examples

- compass directions, which take the values North, South, East and West
- days of the week, which take the values Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, and Saturday
- suits in a deck of cards
- values in a deck of cards
- planets in our solar system
- operators for a calculator

Using an enumerated type

- You must use the name of the enumerated type before the value so with declaration

```
enum Day{
  SUN,MON,TUES,WED,THURS,FRI,SAT;
  }
```

you write code such as:

```
Day today;
today = Day.MON;
println(today);
```

Day must prefix every value of Day

The next element

```
void main() {
    Days d = Days.SUN;
    while (d != Days.SAT){
        println(d);
        d = enumSucc(d);
}
println(d);
}
```

This gives the next day
In the list of Days

Can use enumerated types with for loops

```
void main() {
    for(Days d = Days.SUN;
        d != null;
        d = enumSucc(d))
    {
        println(d);
    }
}
```

Cannot use < or > on an enumerated type, just == and !=.

Enumerated types and switch statements

- One of the most useful things you can do with an enumerated type is use it for a switch variable.
- In switch cases, you must remember *not* to put the type-name as a prefix for the constants.

Day does not prefix the choices

```
switch ( today ){
        case MON   :{}
        case TUES  :{}
        case WED   :{}
        case THURS :{work(); break;}
        case FRI   :{play(); break;}
        case SAT   :{}
        case SUN   :{doNothing();}
    }
```


Without and with the prefix

```
Day closestWeekDay( Day d){
    switch ( d ){
        case SAT :{return Day.FRI;}
        case SUN :{return Day.MON;}
        default  : {return d;}
    }
}
```

Simulation

- Computer programs are regularly written to simulate something real.
- You have probably all played simulation games (e.g. a computer game of solitaire or a flight simulator) but simulation is also used to help understand some real process that is too difficult to understand any other way
- There is an entire course in the third year for understanding how to write simulation programs called "Simulation and Modelling" – Tony Field and Jeremy Bradley.

Vending Machine

- We will now develop a program to simulate a vending machine that sells sweets.
- Here is the interaction between the machine and the user
  - Machine: lists sweets with numbers and prices
  - User: inputs number of choice
  - Machine: lists price of chosen item
  - User: inputs money
  - Machine: gives change (in fewest possible number of coins)
- Two simplifications on reality:
  - our vending machine always has all the coins it needs to give the appropriate change
  - our users always put in at least enough money

Sample Vending Machine Interaction

```
********** Vending Machine **************
*** Choose From the Following ***********
 1: Mars Bars     ********       50 p    *
 2: Polos         ********       36 p    *
 3: Mini Skittles *********      12 p    *
 4: Crisps        *********      44 p    *
 *****************************************
 Please make your choice  4
  Please pay in 44 pence, paid in 60
 Your change in the fewest possible coins:
 1  one pence
 1  five pence
 1  ten pence
```


First step in implementing: declarations for the data

- Declare variables to hold the choice, the payment and the cost.

```
int choice;
int payment;
int cost;
```

- In the program the prices of the sweets must be known. Declare and initialise a variable to hold the prices of the sweets.

```
int[ ] prices = {50,36,12,44};
```

First step in implementing: declarations for the data

- In order to give the appropriate change in coins, the values of each of the coins must be known. Declare and initialise a variable to hold the values of all coins.

```
int[] values = {1,2,5,10,20,50,100,200};
```

- In order to print out the change, the coin names must be known. Declare and initialise a variable to hold the names of all the coins.

```
String[] coinNames = {"one pence", "two pence",
                      "five pence", "ten pence",
                      "twenty pence",
                      "fifty pence", "one pound",
                      "two pounds"};
```


The hardest problem

- Given an amount it will be necessary to convert it into the (fewest) coins needed to make up that amount. So we need a method that does the following:
  - 3 → {1,1,0,0,0,0,0,0}
  - 65 → {0,0,1,1,0,1,0,0}
  - 48 → {1,1,1,0,2,0,0,0} , etc.
- To do this the array of `values` is also required, since we need to know the value of each of the coins.

Declaring coins

- Write the declaration (header) for a method called coins which takes as arguments the amount (assume it is > 0) and the values of the coins. Include your pre and post conditions.

```
int[] coins(int n, int[] values)
assert (n > 0);
//post: the fewest coins whose values sum to
//      equal n is returned
```


How do you solve the problem?

- You need to create a local (just in the method) array money to return from the method containing the different numbers of coins.
- You need a local variable whatsLeft that contains the amount you haven't yet put into money.

```
walk over the array values from right to left
  money[i] = whatsLeft / values[i]
  whatsLeft = whatsLeft % values[i]
return money
```

In Java

```
{
int money = new int [8];
int whatsLeft = n;
     for (int i = money.length-1; i>=0; i--){
          money[i] = whatsLeft / values[i];
          whatsLeft = whatsLeft % values[i];
     }
     return money;
}
```


What if you want to be able to do the converse of coins?

- Declare a method **sum** which takes as an argument an array money which contains the number of each of the coins and which returns the sum of the coins. You will also need to pass **values** as an argument in order to calculate the sum. Include any pre or post conditions.

```
int sum(int[] money, int[] values)
//post: the monetary value of m is returned
```

- What is the algorithm for the body of the method?

total = 0
walk over the array money (from left to right)
    total = total + money[i]*values[i]
return total

Write the method sum in Java

```
int sum(int[] money, int[] values){
//post: the monetary value of m is returned
int total = 0;
    for (int i = 0; i<money.length; i++){
         total = total + money[i]*values[i];
    }
    return total;
 }
```


You need to be able to print out the change in words

- Declare a method **printMoney**, which takes an array with the money to be printed and an array of the names of the coins and prints on the screen the number of each of the coins.

```
void printMoney(int[] m, String[] names)
//post: the names and numbers of the coins
//      in M are printed on the screen
```

- What is the algorithm for the body of the method?

walk over the array money (from left to right)
    if money[i]>0 println money[i] : names[i]

Write the method `printMoney` in Java

```
void printMoney(int[] m, String[] names){
//post: the names and numbers of the coins
//      in M are printed on the screen
  for (int i = 0; i < m.length; i++){
           if (m[i] > 0) {println(m[i] +
                     "  " + names[i]);}
       }
  }
```

149

Finally, the main program

```
void main(){//all the declarations go here
  println("        ********* Vending Machine *************");
  println("        ******* Choose From the Following ******");
  println("        1: Mars Bars      ******        50 p   *");
  println("        2: Polos          ******        36 p   *");
  println("        3: Mini Skittles  ******        12 p   *");
  println("        4: Crisps         ******        44 p   *");
  println("        ****************************************");
  print("Please make your choice");
  choice = readInt();
  cost = prices[choice-1];
  print("Please pay in " + cost + " pence, paid in ");
  payment = readInt();
  println("Your change in the fewest possible coins: ");
  printMoney (coins(payment - cost, values), coinNames);
}
```

150

Summary

- Kenya has enumerated types like Haskell.
- An *enumerated type* is a type whose legal values consist of a fixed set of constants.
- When the data your program uses is not numeric then using an enumerated type makes your program more readable and hence more maintainable.
- You must use the name of the enumerated type before the value.
- Two values of the same enumerated type can be compared with == and != .

151

Summary

- `enumSucc` is used to get to the next value, so they can be used in for loops as counters
- One of the most useful things you can do with an enumerated type is use it for a switch variable (in this case without the prefix.
- A simulation program for a vending machine was developed.
- It was developed by first deciding on the data needed and then writing the methods that worked on the data.

152

Lecture 8 : The Design of Algorithms

Lecturer : Susan Eisenbach

This is the 8th lecture on Java in which we look at how to design programs.

Sorting an unknown number of numbers

- In the tutorial question last week you were asked to sort 10 numbers.
- This is quite a restrictive problem.
- How would you sort any number of numbers (say up to 100)?
- Firstly you need to know how many numbers you are going to sort.
- There are three ways of doing this. You can type in the number of numbers, followed by the numbers. These can be processed with a `for` loop.

Sentinel values

- If you don't want to count the numbers first and there is at least one value that could not be in the list (say if you were sorting non-negative numbers any negative value would do, otherwise a very large or very small number that wouldn't be in your data for example -9999) put it at the end.
- For example, if you are sorting the numbers
  1, 6, 4, 0, 7, 8
- The list 1, 6, 4, 0, 7, 8, -1 is entered and the first 6 numbers are sorted.
- The value -1 (which mustn't be sorted) is called a sentinel value.

How do you read in the list?

- You need to declare the array at its maximum size.
  `double[] vector = new double [100];`
- You need to declare an integer variable to hold the actual length of the list, which you get by counting. `int len = 0;`
- This will have to be passed to the sort method.
- You need a boolean variable that is true iff you should continue reading numbers.
  `boolean goOn = true;`
- You need a double variable to hold the number that is read in. `double buf;`

Now you need a while loop to read in the numbers

```
while (goOn){
    buf = readDouble();
    goOn = len < 100 && buf > -1;
    if (goOn) {
        vector[len] = buf;
        len = len + 1;
    }
}
```

- It is important not to store the sentinel value in the array

The special End of File character

- When input comes from a file rather than the keyboard after all the input has been consumed a system predicate isEOF is set to true.
- When input comes from the keyboard you can set this predicate to true in 2 ways
  - by typing in ^Z (pronounced Control Z)(windows) or ^D (linux)
  - by pressing the EOF button on the rhs of the input panel
- This predicate can be used to stop reading in values.

Now you need a while loop to read in the numbers

```
double[] vector = new double[100];
int len = 0;
while (!isEOF() && len < 100){
    vector[len] = readDouble();
    len = len + 1;
}
```


The programming process

- programming without much thought only works for small problems
- many different schemes exist to systematise programming
- they encourage programmers of diverse abilities and experience to produce programs of uniform quality
- we'll use a three-stage process which provides a framework to:
  - generate consistent, understandable programs
  - allow scope for individual programmers to apply their own problem-solving skills

The programming process

- *requirements specification*: What should the program do?
- results in a document known as the *requirements specification* or *specification*
- This is written in formal logic and/or concise English.

I know you believe you understand what you think I said,
but I am not sure you realise
that what you heard
is not what I meant.
Anon

The programming process

- *design*: How will the program satisfy these requirements?
  - data - information the program manipulates
  - algorithms - methods to manipulate the data results in the *design documentation* or *design*
- *implementation*: design is transformed into *code*
  - coding - should be routine, results in the "finished product" - tangible *code*
  - testing - does the program perform according to spec?

Design documentation

- design is an iterative process, progressively more detailed decisions are made:
  - data
  - algorithms
- the process of refining the algorithms with detail is known as

  stepwise refinement or *top down design*
- documents required:
  - data declarations - for the data
  - pseudo-code or Haskell - for the algorithms

Specification of a calculator

- Concise verbal statement to start:

  The program should accept two numbers, separated by an arithmetic operator, and should produce the correct arithmetic result, if this is calculable.
- Forms the basis of a discussion between programmer and client to fill in details.
  - What is a number?
  - What is an arithmetic operator?
  - What sums are calculable?
  - What form should the sum be in?
  - What should the program do if the result is not calculable?
  - How many calculations should the program do?

Refining the requirements specification

The program should accept two numbers, separated by an arithmetic operator, and should produce the correct arithmetic result, if this is calculable.

Numbers are non-negative whole numbers.

Arithmetic operators are +, -, * and /.

Calculable means that the result must be finite, whole and non-negative.

Input consists of number operator number return.

Input may be separated by spaces and is separated from the result by a new line.

It is assumed that the user types in correct data.

A potentially incalculable result will produce the error message: "cannot calculate".

The data

- What does a program do? It consumes input and produces output. The first stage of design is to figure out what inputs to the program are and what the outputs from the program are.
- All inputs and outputs identifiers (names) need to be declared and defined.
- Data types are those that are recognised by Java and written in Java.
- Comments are written after // .

Inputs and outputs

- inputs are from the keyboard
  - first
  - op
  - sec
- outputs are to the screen
  - result
  - errorMessage
- program called calculator

Data declarations for the calculator

```
String errorMessage = "cannot calculate";
int first;
char op;  //one of: + - * /
int sec;
int result;
```

Alternatively op could be an enumerated type:
Operator = {PLUS, MINUS, TIMES, DIVIDE}
Operator op;

- Now all we need to do is define a calculator //performs simple arithmetic on //non-negative integers

Pseudo-code

- We need a language to write our algorithms in.
- This could be Java but then you need to worry about syntax details.
- Instead we will use a language called *pseudo-code*. It has no predefined syntax.
- It is close enough to Java to translate obviously. It is close enough to English that you don't have to worry about fussy details of syntax.
- Everyone's pseudo-code is slightly different.
- There has already been some pseudo-code used in this course.

Stepwise refinement

- When writing the algorithm whenever things get complicated make up a name and hive the complication off to another process to be dealt with later. (Use indents instead of brackets and semicolons)

calculator:

```
    read first, op, sec
    if the expression is calculable
    then evaluate the expression  put the result
    else put errorMessage
```


Extend data declarations to include Expression

```
class Expression{
   int first;
   char op; //'+','-','*','/'
   int sec;
}
```


Now define evaluate

evaluate:

```
pre IsCalculable
 switch on op
    '+' : result first + sec
    '-' : result first - sec
    '*' : result first * sec
    '/' : result first / sec
```

How would you write evaluate in Haskell instead of pseudo-code?

Now isCalculable

- Turn isCalculable into something that is obviously a predicate - a boolean method

isCalculable :

```
switch on op
    '-'         : result first >= sec
    '/'         : result sec != 0
    '+', '*'    : result true
                : result false
```

or **isCalculable:**

```
return op == '+' || op == '*' ||
       op == '-' && first >= sec ||
       op == '/' && sec != 0
```


Design complete

- For first year programs, the data declarations (extended with anything new that comes out of the pseudo-code) and pseudo-code form the design.
- Before proceeding to write the code, reread the specification. Check that the design meets the specification and change the design if it does not.
- It should be straightforward to turn the design into code.
- The class, variable and method declarations should come from the data declarations.
- The code should come from the pseudo-code. Amend the pseudo-code and data declarations if you decide on any changes. Programming is an iterative process and there will be changes.

Declarations

```
class Expression{
  int first;
  char op;
  int sec;
}

String errorMessage = "cannot calculate";
Expression expr;
```


isCalculable (both versions)

```
boolean isCalculable (Expression e){
  return e.op == '+' or e.op == '*' ||
         e.op == '-' && e.first >= e.sec ||
         e.op = '/' && e.sec != 0;
}
boolean isCalculable( Expression e ) {
  switch ( e.op ) {
   case '-' : {return e.first >= e.sec;}
   case '/' : {return e.sec != 0;}
   case '+' : {return true;}
   case '*' : {return true;}
   default  : {return false;}
  }
}
```

Evaluate:

```
int evaluate (Expression e){()
  assert (isCalculable(e));
  switch (e.op) {
     case '+' : {return e.first + e.sec;}
     case '-' : {return e.first - e.sec;}
     case '*' : {return e.first * e.sec;}
     case '/' : {return e.first / e.sec;}
     default  : {return -1;}
  }
}
```


Finally, the program:

```
void main(){
  String errorMessage = "cannot calculate";
  Expression expr ;
  expr.first = readInt();
  expr.op = readChar();
  expr.sec = readInt();
  if (isCalculable(expr))
     {println(evaluate (expr));}
  else {println(errorMessage);}
}
```


Summary

- To be able to solve a problem by computer you must be able to decide *what* the problem is and *how* it should be solved.
- Java is less abstract than Haskell so programs written in it must be *designed* before they are committed to code.
- The first step in solving a problem is to understand *what* the problem is; this is called the *specification stage*.

Summary(cont.)

- *How* a problem should be solved should be tackled *after* completely determining what the problem is.
- How to solve the problem comes next - the *design*.
- The method of *stepwise refinement* consists of decomposing a problem into simpler sub-problems.
- This data needs to be decided on as well.
- An algorithm describes how the inputs to a process produce the outputs.
- Algorithms are described either in Haskell or pseudo-code.

Lecture 9 : Designing Input
Lecturer : Susan Eisenbach

This is the 9th lecture on Java in which we look at the kinds of difficulties reasonable input causes a programmer.

Back to the calculator

- The specification of a calculator was too rigid. Any reasonable calculator program would be more flexible about its input. The original specification said:

  Input consists of number operator number return. Input may be separated by spaces and is separated from the result by a new line. It is assumed that the user types in correct data.
- More reasonably would have been a specification that included:

  If the user fails to type in correct data then an error message "not an expression" will be output.

Altering the previous declarations

The data declarations:

```
class Expression{
     int first;
     char op;
     int sec;
}
String notCalculable = "cannot calculate"
String syntaxError = "not an expression"
```

What happened to result?

- The program declaration:

```
calculator //program performs arithmetic on non-negative ints
```

- The method declarations:

```
evaluate = int evaluate(expression)
       pre isCalculable
isCalculable = boolean isCalculable(expression)
```


Pseudo-code

- The only change needed in the main program is that instead of using the Java **`readInt,`** **`readChar`** and **`readInt`** to read in an expression it should be hived off to its own method to be sorted out. Pseudo-code isn't wonderful for such fiddly details so the code is also provided.

Calculator:

readExpression
if the expression *isCalculable*
evaluate the expression
print the result
else print notCalculable

```
expr=readExpression();
if isCalculable(expr)
 {println
   (evaluate(expr));}
else
 {println
   (notCalculable);}
```

- All the real work needs to be done in **readExpression**

How do we do input?

- Previously our input was done with the statements:

```
expr.first = readInt();
expr.op = readChar();
expr.sec = readInt();
```

If you type in:

```
x + 2
```

```
Could not read integer, incorrectly formatted number (x)
```

- You never want your users to see wrong answers or messages you didn't write.

Why?

- Our input routine must be able to deal with receiving syntactically incorrect data and outputting the error message

  `"not an expression"`

- `readInt` terminates on any non-numeric character.
- Given the input `x + 2` `readInt` reads in the `x`, the first non-numeric character, rejects it as a number, and terminates the entire program.

Converting numeric character strings to numbers

- `readInt()` reads in numbers as strings and then converts what it has to a numeric value.
- How does it convert "345" into 345?
- You process the characters from left to right.
- **string left** / **value of number**

string left	value of number
"345"	0
"45"	10*0+3 = 3
"5"	10*3 + 4 = 34
""	10*34 + 5 = 345

3 4 5

345

How do we convert the character '5' to the number 5?

- All characters have ascii values.
  - '0' is 48
  - '1' is 49
  - '2' is 50, etc
- So the ascii value of ('5') minus the ascii value of ('0') is equal to 5
- You can get the ascii value of a character by assigning it to an integer variable.
- So if c is a char, c – 48 will be the value you want.
- Alternatively you can use the Kenya builtin method `charToInt(c)`.
- There is also `intToChar(i)`.

What other builtin methods are in Kenya?

- Check out http://www.doc.ic.ac.uk/kenya/kenya4/ReferenceGuide/
- `String charsToString(char[] source);`
- `void main(){`

```
    char[] myChars = {
        'h','e','l','l','o',' ','w','o','r','l','d' };
     String helloWorld = charsToString(myChars);
     // prints "hello world"
     println(helloWorld);
}
```

- `char[] stringToChars(String argument)`

Switches can be used to convert a character to a numeric value.

- Write a method that takes a character in the range '0' <= character <= '9' and returns the numeric value, eg '0' → 0, etc. Use a switch statement.

```
int convert(char c){
assert (isDigit(c));
    switch (c) {
        case '0' : {return 0;}
        case '1' : {return 1;}
        case '2' : {return 2;}
        case '3' : {return 3;}
        case '4' : {return 4;}
        case '5' : {return 5;}
        case '6' : {return 6;}
        case '7' : {return 7;}
        case '8' : {return 8;}
        case '9' : {return 9;}
        default  : {return -1;}
    }
}
```

this only works if the character is a digit

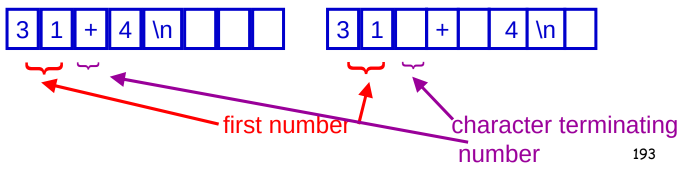
isDigit for the assertion

- Write a predicate (boolean method) `isDigit` that returns true iff a character is in the range '0' to '9'.

```
boolean isDigit(char c){
    return '0' <= c && c <= '9'
}
```

Buffering input

- If you wish your program to print out all error messages then you have to read in the characters and convert them into numbers yourself.
- To write a calculator which will accept both **31+4** and **31 + 4** we need to process a character at a time and convert the characters into numbers where appropriate.



Two ways we can write readExpression:

- ☑ read one character at a time and process it as it arrives
- ☑ read the complete input line into an array of char and then process it
- ☒ do not use `String`, because you cannot process characters in it and because `readString()` will only read up to the first space
- you will only know that you have finished a number when you are looking at the character that follows:
  - 31 + 4
  - 35+4
- in the first example you need to read in the ' ' to know that the number is 31. In the second example you need to read in the '+' before you know that the number is 35.

Processing a line at a time

- If we wish to process a line at a time then readExpression will need to be rewritten.

readExpression:

```
  readLine
  if isValid  //number operator number
      convert  line to expression
  else
      println syntaxError
      readExpression
```

- **readLine**, **isValid** and **convert** still need to be written.


```
if isValid  //number operator number
    convert  line to expression
```

- This says divide the problem in two. First walk over the array of characters checking that you have characters that will convert to what you want, then convert the entire array of characters.
- Dealing with error filled input is difficult and this simplifies the task since you only do the conversion on syntactically correct input.
- You can only use this technique if your input data can be looked at more than once without consuming it.

In Java

```
Expression readExpression(){
char[] line = new char[20];
Expression e;
  line = readLine();
  if (isValid(line))
    {e = convert(line);
     return e}
  else{
   println(syntaxError);
   return readExpression();
  }
}
```


Alternatively you could have a readExpression which is done a character at a time:

readNum
if okay
    readOp
    if okay
    then *readNum*
if not okay
 println syntaxError
 throw away rest of line
 readExpression

- where **readNum** and **readOp** still need to be written.

Summary

- Design is an iterative process so the designer may need to return to an earlier stage for amendments and additions.
- There are frequently several ways to solve a problem. To find the best way solve the problem in the ways you think are good and then compare the solutions for clarity.
- Errors identified and corrected at the design stage are less expensive than those that survive to the implementation stage and beyond.
- Some program somewhere has to convert characters typed in into numbers used in programs.
- Anything but the simplest input is difficult to do.

Lecture 10: Testing and Debugging
Lecturer : Susan Eisenbach

For extra material read part of Chapter 10 of
Java Software Solutions
This is the 10th lecture in which how to test whether a program fulfils its specification and how to debug it, if it doesn't are covered.

Testing

- test that your program does what it was required to do go back to the specification
- test at the limits of data ranges
- alter program if it fails with any legal input data
- document what the program does in situations where the specification does not describe the intended behaviour
- the program should never be allowed to crash or rubbish (an incorrect answer or inappropriate error message) be allowed to appear on the screen
- draw up a test table

The specification for Calculator

- The program should accept two numbers, separated by an arithmetic operator, and should produce the correct arithmetic result, if this is calculable.
- Numbers are non-negative whole numbers.
- Arithmetic operators are +, -, * and /.
- Calculable means that the result must be finite, whole and positive.
- Input consists of number operator number return.
- Input may be separated by spaces and is separated from the result by a new line.
- If the user fails to type in correct data then an error message "not an expression" will be output.
- A potentially incalculable result will produce the error message: "cannot calculate".

Test data

Input	*Expected Outcome*	*Comment*
3+3	6	correct addition
10-2	8	correct subtraction
3* 4	12	correct multiplication
21 /3	7	correct exact division
22/3	7	correct whole num. division
4- 11	cannot calculate	appropriate error message
22/0	cannot calculate	appropriate error message
0/0	cannot calculate	appropriate error message
2&3	not an expression	appropriate error message
3^4	not an expression	appropriate error message

Not so straightforward test data

Input	*Expected Outcome*	*Comment*
-3	???	outside of specification
6+	???	outside of specification
+	???	outside of specification
=	???	outside of specification
a-b	???	outside of specification
0*0	0	at limits of specification
biggest number - biggest number	0	at limits of specification
biggest number *1	biggest number	at limits of number range
biggest number +1	overflow error	outside of specification
biggest number *10	overflow error	outside of specification

Test as you code

- bugs (technical term) are errors in programs
- bugs are difficult to find in complete programs
- test each method as soon as you have written it
- two ways of testing methods:
- test harnesses - write a small program to test each method
- incremental testing - use the whole program to test each new method
- either way, testing as you go takes up less time then post testing

Example test harness to test evaluate

```
class Expression{
   int first;
   char op;
   int sec;
}
void main(){
Expression e;
   e.first = readInt(); e.op = readChar(); e.sec = readInt();
   println(evaluate(e));
}

int evaluate( Expression e ) {
  switch ( e.op ) {
       case '+' : {return e.first + e.sec; }
       case '-' : {return e.first - e.sec; }
       case '*' : {return e.first * e.sec; }
       case '/' : {return e.first / e.sec; }
  }
   return 0;
}
```


Alternatively, use incremental testing

- Write the declarations and main program first.
- All declarations for methods have to be written as well. These should include comments.
- The bodies of each method should be trivial. These are called stubs.
- Execute the program. If there is a bug then fix it.
- Then replace one stub at a time. Each time testing the code.
- Bugs can always be isolated to the latest added code.

Example stubs for testing the main program

```
Expression readExpression(){
Expression e;
  e.first = 2;
  e.op = '*';
  e.sec = 21;
  return e;
}

boolean isCalculable( Expression e ) {
  return true;
}

int evaluate( Expression e ) {
assert (isCalculable(e));
  return 42;
}
```

Calling the stubs

```
class Expression {
   int first = 0;
   char op;
   int sec = 0;
}

void main(){
String notCalculable = "cannot calculate";

Expression expr;
      expr = readExpression();
      if ( isCalculable(expr) ) {println("= " + evaluate( expr ) );}
      else {println( notCalculable );}
}
Expression readExpression(){stub code goes here}
boolean isCalculable( Expression e ) {stub code goes here}
int evaluate( Expression e ) {stub code goes here}
```

code being tested

Debugging complete code

- when a program goes wrong you need:
  - what code was being executed
  - what data was being used
- insert *debugging code*
- need to produce a *trace*

```
main program entered
isCalculable entered
evaluate entered
```

<crash>

Permanent tracing code

- use a boolean constant at the top of the code
  `boolean tracing = true;`
- at the start of each method foo include:
  `if (tracing) {println( "foo entered" );}`
- at the end of each void method include:
  `if (tracing) {println( "foo exited" );}`
- ¿ Why don't non-void methods get this code as well?
- When you don't want to see the trace you change the value of `tracing` to `false`.

Debugging data

- Need to print out values of possible offending variables
- Use another boolean constant for this:
  `boolean debug = true ;`
- Insert code where it might be needed:
  `if debug {println("ch = " + ch);}`
- Write methods to print out classes:
  `void printExpression(Expression e)`

Summary

- Test throughout program development to ease finding bugs.
- Use test harnesses and stubs to find bugs in methods.
- Test a program against its requirements.
- Test with typical data, then at limits then outside the specification.
- If a program does not work properly it needs to be debugged. Insert debugging code to find the source of the error. Do this systematically.
- Trace your program by hand. Time spent this way will be less than the time spent sitting at the machine looking for bugs.

Lecture 11 : Abstract Data Types
Lecturer : Susan Eisenbach

For extra material read Chapter 12 of

Java Software Solutions

This is the 11th lecture on Java in which we define abstract data types and describe an actual use.

Another Calculator

- We will consider a program which can deal with a "long" expression defined as follows:

```
Expression=Operand,{Operator,Operand,} "="
Operand = int
Operator ='+' | '-'| '*' | '/'| '^'
```

- The expression now corresponds to any arithmetic expression with several operators but without brackets.
- In the simplest case do 'left-to-right' evaluation. Thus

```
3 + 4 - 5 + 6 = ( 3 + 4 ) - 5 + 6
              = ( 7 - 5 ) + 6
              = 2 + 6
              = 8
```

- The ideas embodied in the first Calculator could be adapted to give pseudo-code along the following lines

Start Off Calculation:

```
    read first operand
    read operator
    calculate.
    write out the result which is held in first operand.
calculate:
    if operator isn't "="
        read second operand
        evaluate the expression
        assign the result to first operand
        read operator
        calculate.
```

What is wrong with this?

Precedence

Left-to-right evaluation only applies to operations of the same precedence. Consider the expression

a + b * c ^ d / e =

Precedence rules

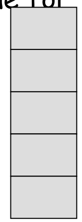
^	highest
* /	high
+ -	low
=	lowest

- The program will need to scan the input expression and can safely evaluate subexpressions from left to right until a higher-precedence operator is encountered.
- The current evaluation will have to be suspended until the higher-precedence operation has yielded a result.

Operations Required

1. Insert a subexpression
2. Remove the most recently inserted subexpression
3. Examine the most recently inserted operator.

- Better to have two data structures one for numbers one for operators.
- This data structure is called a stack.
- Have you seen another data structure that looks like a stack?



Stack Operations

- **isEmpty** returns true iff the stack has no elements in it.
- **empty** returns an empty stack

  stack = empty | push(item, stack)
- **top** returns the top element of the stack.
- **push** takes a **stack** and an **item** and returns the stack with **item** added at the top.
- **pop** takes a **stack** and returns it with the top element removed.

User defined types

- Java cannot provide every data structure that is needed by every programmer.
- Java lets you create new data structures using its classes.
- When accessing elements of these user defined data structures methods are used.
- So instead of getting elements with x[i], like arrays or x.i like fields in classes, the programmer has to write methods to get items from the user defined data structures.

User defined types are not enough

- Although user defined types are useful something like Haskell's polymorphism is important so that the user defined types do not have to contain the type of the elements.
- The latest Java now has generic types which are similar to polymorphic types.
- So now in Java it is possible to define lists, trees, etc which can be used for holding values of any type such as ints, chars or whatever is required by the program.

Many Haskell functions are polymorphic

- `fst :: ( a, b ) -> a` Pair index
- `fst(3,"Hello")` of `3`

Note that the type of fst involves two type variables

- since pair elements can be of any type

Java Generics

- To declare a pair of two elements of the same type in Java:

```
class Pair<T>{
T a;
T b;
}
```

type variable declaration

- To declare a pair of two elements of (possibly) different types in Java:

```
class Pair<S,T>{
S a;
T b;
}
```

type variable declarations

Program using a `Pair<T>`

```
void main() {
  Pair<String> twoStrings;
  twoStrings.a = "hello";
  twoStrings.b = "world";
  println(twoStrings.a);

  Pair<int> twoInts;
  twoInts.a = 3;
  twoInts.b = 4;
  println(twoInts.a);
}

class Pair<T>{
  T a;
  T b;
}
```

Program using a Pair<S,T>

```
void main() {
   Pair<String,String> twoStrings;
   twoStrings.a = "hello";
   twoStrings.b = "world";

   Pair<int,char> intChar;
   intChar.a = 3;
   intChar.b = 'x';
   println(intChar.b);
}

class Pair<S,T>{
   S a;
   T b;
}
```


Generic methods

- Methods can have generic types.
- The generic types must come before the return type.
- Both the arguments and return type may be generic

```
<S,T> S first(Pair<S,T> p){
    return p.a;
  }
```

returns something of the first generic type

Access methods for a stack of items

```
<<T> boolean isEmpty(Stack<T> s) {//code goes here
}
 <T> Stack empty() {//code goes here
//post: isEmpty(empty())
}
<T> T top (Stack<T> s) {//code goes here
assert (! isEmpty(s)) :"no top of an empty stack";
}
<T> Stack push (Stack<T> s, T item) {//code goes here
}
//post top(result)=item
<T> Stack pop(Stack<T> s) {//code goes here
assert (isEmpty(stack)) :"cannot pop an empty stack";
}
```


Using a stack

- In your program you would need the following declarations:

```
enum Operator{
   PLUS, MINUS, TIMES, DIVIDE;
}
Stack<int> numS;
Stack<Operator> opS;
```

- Write `push(numS,3)` to push 3 onto `numS` and `top(opS)` is the top operator on the operator stack.

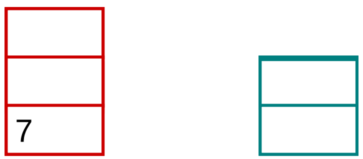
Using a stack

- We have not said how the actual stack is implemented as we have not shown the data declarations. Perhaps our stacks will be implemented as arrays – but they don't have to be.
- When using a stack you don't use the actual data declarations, because they don't model the data structure (stack here) and may be changed.
- You only use the access methods that need to be written: `isEmpty`, `empty`, `pop`, `push` and `top`.
- Use is independent of the implementation of the method.

calculate:

```
if there is another item (operand or operator)
  if it is an operand
    push it onto the numberStack, skip over the item,
    calculate the rest of the expression
  else if the operatorStack isEmpty or its top is of
      lower precedence than the item's precedence
          push the item onto the operatorStack,
          skip over the item
          calculate the rest of the expression
      else pop the top two operands and the top operator,
          evaluate the expression formed,
          push the result onto the numberStack,
          calculate the rest of the expression
```

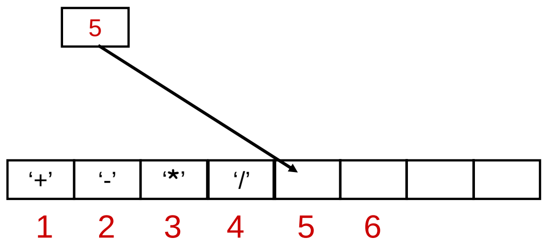

Example: calculate 1+3*4 /2 =




```
if there is another item (operand or operator)
  if it is an operand
    push it onto the numberStack, Skip over the item,
    calculate the rest of the expression
  else if the operatorStack isEmpty or its top is of
      lower precedence than the item's precedence
          push the item onto the operatorStack,
          skip over the item
          calculate the rest of the expression
      else pop the top two operands and the top op,
          evaluate the expression formed,
          push the result onto the numberStack,
          calculate the rest of the expression
```


An array implementation of a stack

- **Declaration of this stack:**



An array implementation of a stack

Data declarations of this stack:

```
class Stacks{
  int items = new int[20];
   int pointer = 0;
//methods go here
}
```

pointer = ___ when the stack is empty

pointer = ___ when the stack is full

Modelling data relationships

- Arrays and records don't model everything.
- In Java you can define your own structures.
- Whether or not Java data structures are suitable follow a three stages process for establishing any data-store:
  - Discern the need for a data store and establish its characteristics and the interrelationships of its components.
  - Make arrangements to create a data store within the program which faithfully reflects the real-world structure.
  - Produce code to manage the structure - i.e. to examine the data store and to insert and remove items.

Important jargon

- In general these operations will not be as simple as for arrays and each operation will be realised as a separate method, called an *access method*.
- In Java you can consider the use and creation of the data structure entirely separately.
- The programmer can consider how the data store will be accessed without needing to bother about the practical details of controlling and manipulating storage - i.e. in the abstract. For this reason, the collection of operations is often known as an *abstract data type*.
- Using abstract data types is a major tool for program decomposition in modern programming practice.

Summary

- When designing a data structure a programmer must:
  - establish those characteristics dictated by the problem
  - create a data structure that has these characteristics
  - produce code to manage the structure.
- Operations designed to manage a data structure are called access methods. A collection of such operations, all applicable to a particular type of data structure, is called an abstract data type.
- A stack is an example of an abstract data type.
- Arithmetic expressions can be evaluated, by using stacks to store both numbers and operators until needed.The use of the stacks ensures that the correct order of operations is observed.
- Next term you will look at many different abstract data types since they are a very powerful programming tool.