Programming II
Introduction to Imperative Programming

Tristan Allwood
tristan.allwood@imperial.ac.uk
Based on material by Susan Eisenbach

120.2

Autumn Term - 2013

Textbooks

- No textbook is required.

For programming beginners:

- **Java Software Solutions: Foundations of Program Design**, John Lewis and William Loftus, Pearson Education, 2011.

For experienced programmers:

- **Learning the Java™Language**, online at `http://download.oracle.com/javase/tutorial/java/`
- **Thinking in Java™, Bruce Eckel**, Prentice Hall, 2006.
- **Effective Java™Second Edition**, Joshua Bloch, Addison-Wesley, 2008.
- **Java™Puzzlers: Traps, Pitfalls and Corner Cases**, Joshua Bloch, Neal Gafter, Addison-Wesley, 2005
- **Java Language Specification**, online at `http://docs.oracle.com/javase/specs/`

Java Software

Editions

- SE - *Standard Edition* for desktop development.
- ME - *Micro Edition* for embedded devices.
- EE - *Enterprise Edition* for “enterprise”.

JRE vs JDK

- JRE - *Java Runtime Environment*. Includes `java`, which executes compiled Java code.
- JDK - *Java Development Kit*. Includes `javac`, a compiler to turn Java source code into compiled Java code. Also includes the JRE.

Version

- In the labs we will use *Java SE JDK v7*.
- Download: `http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html`

Declarative vs Imperative Languages

Haskell

- Declarative language
- Say ‘what you want’ and the computer works out how to do it.
- Similar to mathematical functions and “high level” descriptions of algorithms.

Java

- Imperative Language
- Say ‘what the computer should do’.
- You have to remember what you want the end result is.
- Similar to a recipe / step by step instructions.

From Functions to Methods

Haskell To Java

Haskell

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

Java

```
public static int bigger(int a, int b) {
  // post:  returns the larger of two numbers
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```


1. Argument Types
2. Arguments
3. Result Type
4. Method body delimited by {}
5. Predicate (test) must be surrounded by ()s
6. Results have to be returned using the keyword `return`
7. Statements (e.g. `return`), must end in a ;
8. Single line comments start with //

From Functions To Methods

Calling Other Methods

Haskell

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

Java

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


1. Called method must be followed by ()s
2. Method arguments are inside the ()s

Java Library

Collecting methods together

In BigLibrary.java

```
public class BigLibrary {

  public static int bigger(int a, int b) {
    // post: returns the larger of two numbers
    if (a > b) {
      return a;
    } else {
      return b;
    }
  }

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


1. Class name matches file name. Java source files end in .java
2. Class is public so it can be used by other Libraries and Programs
3. Methods are public so they can be called by other Libraries and Programs

Java Program

A Program expresses precisely what the computer should do

In BigProgram.java

```
public class BigProgram {

  /* Tristan Allwood
   * Prints the largest of 3 inputted numbers
   */
  public static void main(String[] args) {
    System.out.println("Type in your 3 numbers:");

    System.out.println(BigLibrary.biggest(IOUtil.readInt(),
                                          IOUtil.readInt(),
                                          IOUtil.readInt()));
  }

}
```


1. Java programs always start in a public static void main(String[] args) method
2. The return type void means the method doesn't return anything.
3. Multi line comments start with a /* and finish with a */
4. You can print out using System.out.println(...)
5. To use static methods from other classes you need to prefix the method with the class name

Compile and Run

Actually getting your computer to do something...

```
> ls
BigLibrary.java  BigProgram.java  IOUtil.java

> javac *.java

> ls
BigLibrary.class  BigLibrary.java
BigProgram.class  BigProgram.java
IOUtil.class      IOUtil.java

> java -ea BigProgram
Type in your 3 numbers:
5 78 -23
78
```


Compile and Run
Actually getting your computer to do something...
> ls
BigLibrary.java BigProgram.java IOUtil.java
> javac *.java
> ls
BigLibrary.class BigLibrary.java
BigProgram.class BigProgram.java
IOUtil.class IOUtil.java
> java -ea BigProgram
Type in your 3 numbers:
5 78 -23
78

1. `javac` turns Java source (`.java`) into compiled class files (`.class`)
2. `java` runs a compiled class given its name (*without* the `.class` extension)
3. The `-ea` flag enables *assertions*, which we will shortly see.

Variable Declarations

- Variables are names of storage locations.
- They can be of many different types, e.g.
  - `boolean char int double String`
- They must be *declared* before they are used:

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

- They can be *initialised* in the declaration:

```
int total = 0;
char answer = 'y';
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 (re-assigned) throughout a method.

```
boolean answer = false;
int total = 0;

total = total + 1;
total = total * 2;
answer = total >= 2;
```

- Haskell doesn't let you change a variable's value.
  - (Haskell's variables are really *identifiers*).

Program with Assignment

An example

BigProgramAssignment.java

```
public class BigProgramAssignment {

  public static void main(String[] args) {
    System.out.println("Type in a number:");
    int input = IOUtil.readInt();                    // 1 ***
    int result = BigLibrary.bigger(input, 2 * input); // 2 ***
    System.out.println(result);

    System.out.println("Type in another number:");
    input = IOUtil.readInt();                        // 3 ***
    result = BigLibrary.bigger(input, input / 2);     // 4 ***
    System.out.println(result);
  }
}
```


1. Declaring and assigning a variable for the input
2. Declaring and assigning a variable for the result
3. Assigning a new input value
4. Assigning a new result value
5. Don't need new variables for every subexpression

Summary

We have seen...

- Methods (in Haskell, functions), delimited by `{}`.
- Collecting methods into a library using `class`.
- Statement Terminators - `;`.
- Conditionals - `if (predicate) { ... } else { ... }`.
- Variables, Declarations, Assignments.
- Input and Output.
- Programs starting with a `public static void main(String[] args) { ... }` method.
- Compiling (`javac`) and running (`java -ea`) a program.

Recursive Static Methods

Revision from Haskell

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

Haskell Function To Java Method

Greatest Common Divisor

Haskell

```
divisor :: Int -> Int -> Int
-- pre: the arguments are both > 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)
```


Haskell Function To Java Method

Greatest Common Divisor

Java

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


1. Multiple conditionals:
   `if ( p1 ) { ... } else if ( p2 ) { ... } else { ... }`
2. Preconditions expressed with `assert predicate : "message"`

What does `assert` do?

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

- If the predicate is `true` - continue as normal.
- If the predicate is `false` - stop the program with the an error and the message.
- The `: "message"` part is optional, but *strongly* recommended.

When should you use an assertion?

- If you write a method that expects something special of its arguments then you need a *precondition* to state what should be true of the arguments.
- Where possible, use an `assert` to express the precondition.
- If the user has given method arguments that meet the precondition, and the code is correct, then the *postcondition* of the method will hold. Postconditions are written as comments at the top of the method using *//post: ....*

Haskell Program To Java Method

Haskell

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

Java

```
public static int fact(int n) {
  assert n >= 0 : "factorial: n must be >= 0";
  //post: returns n!
  if (n == 0) {
    return 1;
  } else {
    return n * fact(n-1);
  }
}
```


Java Method To Java Program

First put your algorithmic methods in a suitable library.

RecursiveLib.java

```
public class RecursiveLib {

    public static int divisor(int a, int b) {
      ... as before ...
    }

    public static int fact(int n) {
      ... as before ...
    }
}
```

Java Method to Java Program

Create a `main` method for your program.

DivisorFactorial.java

```
public class DivisorFactorial {

  public static void main(String[] args) {
    System.out.println("Please input two numbers" +
      ", greater than 0:");
    int a = IOUtil.readInt();
    int b = IOUtil.readInt();

    int gcd = RecursiveLib.divisor(a,b);
    int result = RecursiveLib.fact(gcd);

    System.out.println("The gcd of " + a + " and " + b +
      " is: " + gcd + ".");
    System.out.println(gcd + "! is: " + result);
  }
}
```


Java Method to Java Program
Create a main method for your program.

1. You can glue `Strings` (and other values onto `String`s) with `+`

Helper Functions to Helper Methods

Haskell

```
epsilon :: Float
epsilon = 0.00001

newtonSqrt :: Float -> Float
-- pre: x >= 0
newtonSqrt x = findSqrt ( x / 2 )
  where
    findSqrt :: Float -> Float
    findSqrt a | abs (x - a * a) < epsilon = a
               | otherwise = findSqrt ( (a + x / a) / 2 )
```


Helper Functions to Helper Methods

Java Library in `Newton.java`

```
public class Newton {

  private static final float EPSILON = 0.00001f;

  public static float newtonSqrt(float x) {
    assert x >= 0 : "newtonSqrt: x should be >= 0";
    return findSqrt(x, x/2);
  }

  private static float findSqrt(float x, float a) {
    if ( Math.abs(x - a * a) < EPSILON ) {
      return a;
    } else {
      return findSqrt(x, (a + x / a) / 2);
    }
  }
}
```

1. You can't directly nest methods, so the helper method needs `x` as-well as `a`
2. The helper method is `private` so it can only be seen by methods inside class `Newton`
3. `EPSILON` is declared as a `private` constant
4. `float` literals need to end with an `f`, otherwise they default to being `double`
5. The built in `Math` library has lots of helpful methods, e.g. `Math.abs`

Methods Summary

- Haskell has *functions* that return results.
- Java has *methods* that can return values.
- Java also has methods that don't return values.
  - They only execute code.
  - Their return type is `void`.
  - They frequently consume input and/or produce output.
- The start of a program must have the signature: `public static void main(String[] args)`.
- Java methods can be recursive.

A Calculator Program

An excuse to introduce more syntax...

Description

Write a simple calculator that prompts the user for an operation (+, -, *, /, negation), one or two numbers as appropriate, and prints out the result.

Stages

1. Presenting a menu to the user, and getting their response.
2. Some control flow to work out if we need one or two arguments.
3. Implementations for the two argument operations.
4. Implementation for the one argument operation.
5. A `main` method to start the program.
6. A class to contain all the methods.

A Calculator Program

First, a method to present a menu to the user and to get their response

```
private static int presentMenu() {
  // post: Menu appears on screen. 0 <= result <= 5;
  System.out.println("Enter 0 to quit");
  System.out.println("Enter 1 to add");
  System.out.println("Enter 2 to subtract");
  System.out.println("Enter 3 to multiply");
  System.out.println("Enter 4 to divide");
  System.out.println("Enter 5 to negate");

  int result = IOUtil.readInt();
  assert (0 <= result && result <= 5);
  return result;
}
```

A Calculator Program

Second, a method to work out if we need one or two arguments

```
private static void processMenu(int reply) {
  assert (0 <= reply && reply <= 5) ;

  switch(reply) {
    case 0: return;
    case 1:
    case 2:
    case 3:
    case 4: processTwoArguments(reply); return;
    case 5: processOneArgument(reply);
  }
}
```


1. Introducing the `switch` statement
2. An expression of `int`, `byte`, `short` or `char` type*
3. *Or an enum type, which we'll see later. In Java 7 you can also use `String`
4. `case value:` for where to jump to based on value

A Calculator Program

Third, implementations for the two argument operations

```
private static void processTwoArguments(int reply) {
  assert (1 <= reply && reply <= 4);
  System.out.println("Please enter your two integers:");
  int x = IOUtil.readInt();
  int y = IOUtil.readInt();

  int result;
  String op;

  switch (reply) {
    case 1: result = x + y; op = " + "; break;
    case 2: result = x - y; op = " - "; break;
    case 3: result = x * y; op = " * "; break;
    case 4: result = x / y; op = " / "; break;
   default: assert false: "Should be impossible!"; return;
  }
  System.out.println(x + op + y + " = " + result);
}
```


1. `break` leaves the switch (stops *fall-through*)
2. `default` is a place to jump to if no other value matches (usually optional)

A Calculator Program

Fourth and Fifth, One argument functions and a main method

```
public class Calculator {

  public static void main(String[] args) {
    int menuResult = presentMenu();
    processMenu(menuResult);
  }

  private static int presentMenu() {
    ... as before ...
  }

  private static void processMenu(int reply) {
    ... as before ...
  }

  private static void processTwoArguments(int reply) {
    ... as before ...
  }

  private static void processOneArgument(int reply) {
    // TODO
    System.out.println("TODO: not implemented yet");
  }

}
```


An aside, Java's primitive types

Type	Size in bits	Notation	Use in switch
`byte`	8	0	Yes
`short`	16	0	Yes
`int`	32	0	Yes
`long`	64	0L	No
`float`	32	0.0f	No
`double`	64	0.0d	No
`boolean`	1	false / true	No
`char`	16	'\u0000' (or 'A', '\n' etc)	Yes

Back to Recursion

Important things to remember:

- Base Cases
  - Guard your recursive calls.
  - Not guarding your recursive calls leads to infinite recursion.
- Recurse on simpler inputs.
  - Make sure there is progress towards the base cases between invocations of the recursive routine.
- Use comments to make things clearer if possible.

Morse Encoder

A recursive function with 10 base cases!

```
public class Encoder {

  public static String encodeInt(int x) {
    assert x >= 0 : "Can only encode non-negative integers";

    switch (x) {
      case 0: return "-----";
      case 1: return ".----";
      case 2: return "..---";
      case 3: return "...--";
      case 4: return "....-";
      case 5: return ".....";
      case 6: return "-....";
      case 7: return "--...";
      case 8: return "---..";
      case 9: return "----.";
      default:
        String remainder = encodeInt(x % 10);
        String rest      = encodeInt(x / 10);
        return rest + " " + remainder;
    }
  }
}
```

encodeInt(120)

```
x = 120;
remainder = "-----";
rest = encodeInt(12);
```

```
x = 12;
remainder = encodeInt(2);
```

```
x = 2;
return "..---";
```

Summary

- A method that calls itself is called *recursive*.
- Recursive methods that produce a single result are just like Haskell functions.
- `void` methods do not produce a result.
    - They are used when you are interested in their *side effects*.
    - For example input / output.
    - In the next lectures you will see other forms of side effect.
- To ensure recursive calls will eventually terminate, every recursive method must be guarded by terminating conditions (base cases), and progression towards those conditions in the recursive calls.
- `switch` statements can be used rather than conditionals (`if (p1) { ... } else if (p2) { ... } else { ... }`) for choices based on `int`-like values.

Arrays and Loops

Array?

What?

- Space for many items of the same type.
- You can access each element via its index in the array.
- Arrays can be multi-dimensional.

Why?

- Sometimes you'll need to deal with large quantities of data.
- Sometimes you'll want to perform the operations on lots of individual items.
- Sometimes you'll want to process an unknown number of items.

Differences to Haskell Lists

- You can't pattern match on an array.
- Every element of an array can be accessed in constant time.

Example of an array variable initialization

Creating 10 `double`s in one go...

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


Example of an array variable initialization
Creating 10 doubles in one go...
double[] vec = new double[10];

1. To declare an array variable of a given type, we add `[]` after the type
2. This variable is called `vec`
3. `vec` therefore is a variable for an array of doubles
4. To initialize `vec` we use the keyword `new` to ask for space
5. Here we ask for space for 10 double values, by `double[10]`
6. The 10 new double values will all default to being 0.0
7. The number of elements (10) can be any expression of type `int`

Initializing an array with known values

Arrays of `String` and `int`

```
String[] judges = { "Craig", "Darcey", "Len", "Bruno" };
int[] scores = { 3, 7, 10, 9 };
```


Initializing an array with known values
Arrays of String and int
String[] judges = { "Craig", "Darcey", "Len", "Bruno" };
int[] scores = { 3, 7, 10, 9 };

1. The items are listed between `{ }`
2. Java automatically creates a new array of the right size and populates it

Reading and Writing to Arrays

Using array indexing expressions

```
String[] judges = { "Craig", "Darcey", "Len", "Bruno" };
int[] scores = { 3, 7, 10, 9 };

String firstJudge = judges[0];

if (scores[0] < 5) {
  scores[0] = 5;
}

System.out.println(firstJudge + " gave: " + scores[0]);

System.out.println("The final judge, " + judges[3] +
                   ", gave: " + scores[3]);
```


1. You can read the element at index `i` out of array `a` with the syntax `a[i]`
2. The first element of an array is at index `0`
3. You can change the value of the element at index `i` in array `a` with the syntax `a[i] = newValue;`
4. The last element of an array is at an index one smaller than the length of the array

Iteration...

- Arrays exist in order to hold multiple values that should be treated similarly.
- Frequently the same operation needs to be performed on each array value.
- Traversing all the elements of an array can be achieved with a control flow statement called a *for loop*.
- Using a `for` statement is called looping, and causes repetitive execution.

Looping through Judges

Introducing The *Enhanced* `for` statement

```
String[] judges = { "Craig", "Darcey", "Len", "Bruno" };

for (String judge : judges) {
  System.out.println(judge);
}
```


Doing something with each element of an array

The *Enhanced* `for` statement in general

```
for (Type variable : array) {
  ... code using variable ...
}
```


Doing something with each element of an array
The Enhanced for statement in general
for (Type variable : array) {
... code using variable ...
}

1. The block of `code` will be executed once for each element in `array`
2. Each time `code` is executed, `variable` will be bound to a successive element of `array`.

Enhanced `for` Example

Sum all the elements of an array

```
double[] vector = { 1.1, 2.2, 3.3 };

double sum = 0;

for (double elem : vector) {
  sum += elem;
}
```


1. `elem` will be 1.1, then 2.2, then 3.3
2. `sum += elem` is a Java shortcut for `sum = sum + elem`
3. You might also want to use `*=`, `-=`, `/=` and `%=`

Another `for` example

What were my Program's arguments?

```
public class Arguments {

  public static void main(String[] args) {

    System.out.println("The program arguments are:");

    for (String argument : args) {
      System.out.println(argument);
    }

  }
}
```

Output

```
> java -ea Arguments Hello World!
The program arguments are:
Hello
World!
```


1. On the command line you can give your program extra arguments
2. These get turned into a `String` array and passed into your `main` method

Getting the length of an array

For example, to calculate the mean average of an array of `double`

```
public static double meanAverage(double[] values) {
  assert (values.length > 0)
    : "Cannot average an empty array";

  double sum = Sum.sum(values);

  double average = sum / values.length;

  return average;
}
```


Getting the length of an array
For example, to calculate the mean average of an array of double

1. Every array knows what its own size is
2. To get the size of the array `a`, you write `a.length`
3. This is called a *field lookup*, where `length` is a *field* of every array
4. This is not a method call, you don't put `()` after `length`
5. The `length` field is read only, and is of type `int`
6. Once created, an array cannot change its size

Bounded Iteration

Introducing the `for ( init ; condition ; update ) { ... }` loop

- Sometimes we need to traverse the array in a different order than first to last.
- Sometimes we want to talk about the elements at the same index in different arrays.

Bounded Iteration

Introducing the `for ( init ; condition ; update ) { ... }` loop

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

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


1. These are two common patterns for using `for` loops
2. The variable `i` is in scope within the loop
3. `i++` is shorthand for `i = i+1`, similarly `i--` is shorthand for `i = i-1`
4. `for ( init ; condition ; update ) { body }` - `init` is executed, then `condition ; body; update` is repeatedly executed as long as `condition` evaluates to `true`.
5. When the loop is being used to traverse an array `a`, `lowerbound` is typically `0`, and `upperbound` is typically `a.length`
6. The first loop counts up, and is useful if an array needs to be traversed in order
7. The second loop counts down, and is useful if an array needs to be traversed in reverse order

Bounded Iteration Example

Printing judges and their scores

```
public static void printScores(String[] judges, int[] scores) {
  assert judges.length == scores.length : "Judge/Score mismatch";

  int total = 0;

  for (int i = 0 ; i < judges.length ; i++) {
    System.out.println(judges[i] + " scored: " + scores[i]);
    total += scores[i];
  }

  System.out.println("For a total of: " + total);
}
```


1. We use a `for` loop to walk through successive elements of the `judges` and `scores` arrays
2. On each execution of the body, `i` will be incremented due to the `i++`
3. This means we can access a judge's name, and their score at the same time in the loop body
4. After the for loop, `i` is no longer in scope, you cannot refer to it

Bounded Iteration in Reverse

Printing the program arguments in reverse

```
public static void main(String[] args) {

    for (int i = args.length - 1 ; i >= 0 ; i--) {
        System.out.println(i + ": " + args[i]);
    }
}
```

Output

```
> java -ea ArgumentsReversed Hello World!
1: World!
0: Hello
```


1. The loop starts at `args.length - 1`, which is the index of the last element in the array
2. The loop continues as long as `i` is non-negative, decrementing each time round.

Arrays can be multidimensional

Creating an array of arrays..

```
double[][] matrix = { { 1.0, 2.0, 3.0 }
                    , { 1.5, 2.5, 3.5 }
                    };

double[][] transpose = new double[3][2];
```


1. `matrix` is an array of length 2, where each element is an array of doubles of length 3
2. The inner arrays of length 3 are written within `{ }`s, and the two inner arrays are themselves nested within `{ }`s
3. `transpose` is an array of length 3, where each element is an array of doubles of length 2

Traversing a multi-dimensional array

`for` loops can be nested

```
public static double[][] createTranspose(double[][] matrix) {
  // pre: matrix is a rectangular matrix

  double[][] transpose
    = new double[matrix[0].length][matrix.length];

  for (int i = 0; i < matrix.length; i++) {
    for (int j = 0; j < matrix[i].length; j++) {
      transpose[j][i] = matrix[i][j];
    }
  }

  return transpose;
}
```


1. Accessing the `length` of a multimensional array will give the number of sub-arrays within it. i.e. the size of that dimension of the array
2. Each inner array will also have its own `length`
3. Here we require as a precondition that the `matrix` parameter is rectangular
4. What happens if `matrix.length` is 0?
5. In order to build the transpose array, we use nested for loops, one for traversing each dimension of `matrix`
6. For the inner loop, we can't write `int i = 0` again, (we've already got a variable called `i`!) so the convention is to use `j`, then `k`, etc.
7. Since our `i` and `j` loops are traversing over `matrix` and `matrix[i]` respectively, inside the body of the loop the element we are interested in will be at `matrix[i][j]`

Initializing multi-dimensional arrays with known values...

Pascal's Triangle

```
int[][] triangle = {          { 1 }
                   ,       { 1, 1 }
                   ,    { 1, 2, 1 }
                   ,  { 1, 3, 3, 1 }
                   };
```


1. Useful for tabulating binomial expansions and combinations
2. In the triangle, the edges are always 1, and inner numbers are the sum of the two values above them
3. In the array form, the math is a little different - don't *ever* trust indentation - Java doesn't care about it at all!
4. The triangle is represented as an array of arrays, but each of the inner arrays has a different length
5. Such arrays are called *jagged*

Traversing a jagged multi-dimensional array

Printing out Pascal's Triangle

```
public static void printTriangle(int[][] triangle) {
     for (int i = 0 ; i < triangle.length ; i++ ) {
          for (int j = 0 ; j < triangle[i].length ; j++) {
               System.out.print(triangle[i][j]);

               if (j < triangle[i].length - 1) {
                    System.out.print(" ");
               }
          }
          System.out.println();
     }
 }
```


1. We use nested loops to walk through each part of the triangle
2. Each of the inner arrays has its own length, so we can use that to get the right number of elements
3. To print without printing a newline, we can use `System.out.print( ... )`
4. To put spaces between the elements, but not at the end, we use an `if` check to see if `j` is before its last index
5. Challenge: how would you print out the triangle centered and not left aligned?

Building a jagged multi-dimensional array

Building the first `n` layers of Pascal's Triangle

```
public static int[][] makeTriangle(int n) {
  int[][] triangle = new int[n][];

  for (int i = 0 ; i < n ; i++) {
    triangle[i] = new int[i+1];

    triangle[i][0] = 1;

    for (int j = 1 ; j < i ; j++) {
      triangle[i][j] = triangle[i-1][j] +
                       triangle[i-1][j-1];
    }

    triangle[i][i] = 1;
  }

  return triangle;
}
```


1. We can ask for space for `n` arrays of arrays, but not give the size of the inner arrays (yet)
2. The `i` loop traverses the rows of the triangle. Row `i` has `i + 1` columns
3. You can create sub arrays and assign them to their parent array. For example, `triangle[i]` can be assigned `int[]` values.
4. The innermost `j` loop traverses from index 1 to one less than the row length

One small syntax gotcha

Declaration vs Assignment / Creation of known array values

Declaration

```
String[] reallyImportantGames
  = { "Minecraft", "GTAV", "Watch Dogs" };
```

Assignment

```
String[] reallyImportantGames;
reallyImportantGames
  = new String[] { "Minecraft", "GTAV", "Watch Dogs"};
```

Method Call

```
buyGames(
  new String[] { "Minecraft", "GTAV", "Watch Dogs" });
```


1. If you declare and initialize an array in one line, then the compiler knows the type of the array, and you can just use `{ }` as we've been doing so far
2. However if you are creating a new array, and e.g. assigning it, or calling a method, then you need to say that you want a `new` something, and then use `{ }`s to build it

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 accessed through their index values. Arrays using a single index are sometimes called vectors, those using *n* indexes are *n-dimensional*. A two-dimensional array is really an array of arrays.
- The number of items in an array can be found through the length field, `array.length`. For multi-dimensional arrays, `array.length` will contain the number of sub arrays, and `array[i].length` will be the number of elements in sub-array `i`.
- Array indexes are `int` expressions. The first element is always at index `0`, and the last at `array.length - 1`.
- Arrays need space allocating for them. This is either done implicitly with values given for them, or explicitly using `new`.
- Repetition of the same operation is called `iteration` or `looping`. A `for` loop can be used to do the same operation on every element of an array.

In-Place Array Operations

Pass by Value

- We have been passing arguments to methods.
- Java methods can accept primitive types as arguments (`int`, `boolean`, `double`, etc).
- They can also accept more complicated types (called *reference types*, for reasons we'll shortly see) such as arrays and `String`s.
- In Java, all method parameters are *passed by value*. This means a copy of the *value* of a parameter is made before the method receives it.
- If the method makes changes to the parameter values, they are not visible to the method's caller.
- However the *value* could point to some shared memory through which changes could be seen.

Not a swap method

```
public class NotSwap {

  public static void main(String[] args) {
    int a = 1;
    int b = 2;
    System.out.println("Before swap: " + a + ", " + b);
    swap(a,b);
    System.out.println("After swap: " + a + ", " + b);
  }

  public static void swap(int x, int y) {
    // this method doesn't do very much!
    int temp = x;
    x = y;
    y = temp;
    System.out.println("Inside swap: " + x + ", " + y);
  }
}
```

Output

```
Before swap: 1, 2
Inside swap: 2, 1
After swap:
```


An Array Swap

```
public class ArraySwap {

  public static void main(String[] args) {
    int[] a = { 1, 2 };

    System.out.println("Before arraySwap: " + a[0] + ", " + a[1]);
    arraySwap(a);
    System.out.println("After arraySwap: " + a[0] + ", " + a[1]);
  }

  public static void arraySwap(int[] array) {
    assert array.length == 2 : "Can only swap 2 elements";
    int temp = array[0];
    array[0] = array[1];
    array[1] = temp;
    System.out.println("In arraySwap: " + array[0] + ", " + array[1]);
  }
}
```

Output

```
Before arraySwap: 1, 2
In arraySwap: 2, 1
After arraySwap:
```

Update in place

- Even though methods can't alter the caller's parameters directly, they can modify their contents if they are a reference type.
- For arrays, this means a method can alter the contents of the array, without having to allocate space for and then returning a new one.
- It is very important that the documentation (postcondition) of methods makes it clear when they perform such updates.
- Note that even though `String`s are a reference type, they are *immutable*, and their contents can never change.

Array Utility Methods

- Java comes with a utility library of helpful methods that act on arrays, called `Arrays`.
- To use it, you will have to `import java.util.Arrays;` at the top of your source file (before the `public class ... {` line).
- It features methods to perform searches, equality checks and pretty printing on arrays.
- It also has methods to sort and fill arrays. These methods are `void` as they update the argument array in place.
- Next term you'll learn in-place algorithms for binary searching and sorting in your Reasoning course.
- In this rest of this lecture we'll look at two other algorithms:
  - Reverse an array.
  - A Fisher-Yates Shuffle.

Using `java.util.Arrays`

Sorting numbers from the user

```
import java.util.Arrays;

public class InputSorter {

  public static void main(String[] args) {
    System.out.println("How many numbers " +
                       "do you wish to sort?");
    int amount = IOUtil.readInt();
    // TODO: check amount is valid

    int[] data = new int[amount];
    for (int i = 0 ; i < amount ; i++) {
      data[i] = IOUtil.readInt();
    }
    Arrays.sort(data);
    System.out.println(Arrays.toString(data));
  }
}
```


1. We have to explicitly `import` the `Arrays` class at the top of our file
2. The `sort` method sorts our array of `int` for us, modifying it in place
3. The utility method `toString` returns a pretty printed version of the array as a `String` which we can print out

A slightly more general swap

Another example of update in place

```
private static void swap(int[] array, int x, int y) {
  int temp = array[x];
  array[x] = array[y];
  array[y] = temp;
}
```


Reverse

Algorithm

- Iterate through the first half the array.
- For each element in the first half, swap it with its corresponding element in the second half.

Reverse

Java Implementation

```
public static void reverse(int[] array) {
  for (int i = 0 ; i < array.length / 2 ; i++) {
    swap(array, i, array.length - 1 - i);
  }
}
```


1. The for loop only traverses the first half of the array
2. If the array has an odd length we don't visit the middle element, since `int` division rounds down
3. If the first element is at index 0, then the last element (the one we swap it with) is at `array.length - 1`

Don't forget to test!

Actually, this is an excuse to talk about equality

```
import java.util.Arrays;

public class ReverseTests {

  public static void main(String[] args) {

    int[] test = { 5, 4, 3, 2, 1 };
    ReverseShuffle.reverse(test);
    assert Arrays.equals(new int[] { 1, 2, 3, 4, 5 }, test);

    test = new int[] { 4, 3, 2, 1 };
    ReverseShuffle.reverse(test);
    assert Arrays.equals(new int[] { 1, 2, 3, 4 }, test);

  }
}
```


1. There are several different notions of equality for arrays
2. Are they the same array in the heap (pointer equality)?
3. Do they have the same elements (shallow structural equality)?
4. For nested arrays, are the deeply nested values the same (deep structural equality)?
5. Pointer equality can be tested with `array1 == array2`.
6. Shallow structural equality can be tested with `Arrays.equals(array1, array2)`
7. Deep structural equality can be tested (on 2-or-higher dimensional arrays) with `Arrays.deepEquals(array1, array2)`

Fisher-Yates Shuffle

Algorithm

- Loop from the end of the array towards the start.
- At each step, swap the current element for a random array element between the first and the current (inclusive).

Fisher-Yates Shuffle

Java Implementation

```
public static void shuffle(int[] array) {
  for (int i = array.length - 1; i >= 0; i--) {
    int index = (int) (Math.random() * (i + 1));
    swap(array, index, i);
  }
}
```


1. The loop starts at the end of the array and walks backwards toward the front
2. The utility method `Math.random()` returns a `double` value that is uniformly distributed between 0 (inclusive) and 1 (exclusive)
3. To produce a random number between `0` and `i` inclusive we multiply the random value by `i + 1`.
4. To convert a `double` to an `int`, we *cast* it, by writing `(int)`. This will round the `double` towards 0.
5. i.e. for positive `double` values like we have here, it will round *down*.

Summary

- Java has *pass by value* semantics. Methods receive a copy of their arguments and changes made are not passed back to the calling method.
- However, Java also has *reference types*, which a method can make changes to. These changes are seen by the calling method.
- Reference types, like arrays and `String`s live on the *heap*, unlike primitive values, which live on the *stack*.
- For arrays, this means there are many utility methods perform updates in place, for example sorting, without needing to create space for a new array.
- Arrays have several different forms of equality, and you must be careful about using `==`, as it compares if two arrays are the same thing in the heap, not if they have the same values.
- There are utility methods in `java.util.Arrays` for checking the structural equality of two arrays.

General Loops

The `while` loop

- `for` loops are usually used when you know up front how many iterations are wanted. For example, to traverse all the elements of an array.
- When you need repetition, but you don't know how many times the repetition will occur you can use recursion, or a *while loop*.
- The choice between using loops or recursion is usually a matter of taste.
- Like recursion, generalised loops can repeat indefinitely. When writing code you must ensure that the loops will terminate.

The `while` loop

Keep re-executing code as long as a condition is `true`

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


The while loop
Keep re-executing code as long as a condition is true
while (condition) {
... loop body ...
}

1. The loop body should include code that eventually makes the `condition` false
2. Loops where the condition cannot become `false` are infinite loops

The `while` loop

For example, reading input until a condition is satisfied

```
public class WhileExample {

  public static void main(String[] args) {

    System.out.println("Please enter a number between 1 and 10:");

    int input = IOUtil.readInt();

    while (input < 1 || input > 10) {
      System.out.println("That wasn't between 1 and 10.  Please try again!");
      input = IOUtil.readInt();
    }

    System.out.println("Thank-you, you gave me: " + input);

  }
}
```


1. The variable in the condition, `input`, is modified in the loop
2. We don't know in advance how many times the loop will need to be run
3. If the user enters a value between 1 and 10 immediately then the loop body will not be run at all

When is the condition checked?

You can imagine a `while` loop as a potentially infinite stacking of `if` statements

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

```
if ( condition ) {
  ... loop body ...
  if ( condition ) {
    ... loop body ...
    if ( condition ) {
      ... loop body ...
      if ( condition ) {
        ... loop body ...
        ... etc ...
      }
    }
  }
}
```


From Recursion to Iteration

Recursive version of `fact`

```
public static int fact(int n) {
    assert n >= 0 : "factorial: n must be >= 0";
    // post: returns n!
    if (n == 0) {
        return 1;
    } else {
        return n * fact(n - 1);
    }
}
```

Recursive algorithm

- Base case: if n is 0, return 1
- Recursive case: multiply n by the factorial of n - 1.

From Recursion to Iteration

Iterative version of `fact`

```
public static int fact(int n) {
    assert n >= 0 : "factorial: n must be >= 0";
    // post: returns n!

    int result = 1;
    while (n != 0) {
        result *= n;
        n--;
    }
    return result;
}
```

Iterative algorithm

- Initialize the result to 1.
- Multiply the result by all the numbers between n and 1.

1. The loop runs until the base case is `true`
2. This means the loop condition is the negation of the recursive base case condition
3. The argument that changes during the recursive call (`n`) is modified in place (`n--`)

From Recursion to Iteration II

Recursive version of `divisor`

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

Recursive algorithm

- If the values are the same, they are their own divisor - return that.
- Otherwise return the divisor of the smaller value and the difference of the values.

From Recursion to Iteration II

Iterative version of `divisor`

```
public static int divisor(int a, int b) {
    assert (a > 0 && b > 0) :
        "divisor must be given arguments > 0";
    // post: returns the greatest common divisor
    while (a != b) {
        if (a > b) {
            a = a - b;
        } else {
            b = b - a;
        }
    }
    return a;
}
```

Iterative algorithm

- Repeatedly make the larger value equal to the difference of the values.
- When the values are the same, we are done.

Other types of loop...

A method to simulate the roll of a die. The result is a random `int` between 1 and 6 (inclusive)

```
public static int rollDie() {
  return (int) (Math.random() * 6) + 1;
}
```

Thought experiment

- I roll one die. (⚀⚁⚂⚃⚄⚅)
- I then roll a second die until I get a number smaller than or equal to the first die.
- How many times will I have to roll the second die?

The `do { ... } while ( condition );` loop

Rolling a second die until it is `<=` the first one

With a `while` loop

```
int a = rollDie();
int b = rollDie();

int count = 1;

while (b > a) {
  b = rollDie();
  count++;
}

return count;
```

With a `do-while` loop

```
int a = rollDie();
int b;

int count = 0;

do {
  b = rollDie();
  count++;
} while (b > a);

return count;
```


1. In the `while` loop version, we have to roll `b` both outside and inside the loop
2. Frequently this pattern of code is better expressed as a `do-while` loop
3. In a `do { code } while (condition);` loop, `code` is executed first, and then `condition` is checked before possibly looping back.

The `do { ... } while ( condition );` loop

Rolling a second die until it is `<=` the first one

```
public static int numberOfRolls() {
  int a = rollDie();
  int b;

  int count = 0;

  do {
    b = rollDie();
    count++;
  } while (b > a);

  return count;
}
```

- We can use this method to try to answer our thought experiment.
- We can call the method n times, and then average the results.

The `for ( init ; condition ; update ) { ... }` loop

Averaging `n` calls to `numberOfRolls`

With a `while` loop

```
double total = 0;

int i = 0;
while (i < n) {
  total += numberOfRolls();
  i++;
}

double average = total / n;
```

With a `for` loop

```
double total = 0;

for (int i = 0 ; i < n ; i++) {
  total += numberOfRolls();
}

double average = total / n;
```


1. Using a `while` loop we can see when `init`, `condition` and `update` are executed in a `for` statement
2. Be careful though, in the `for` version, `i` is out of scope after the loop, whereas in the `while` version it is in scope
3. Usually the `for` behaviour is what you want - don't keep variables in scope that you don't need

`break`, `continue` and labels

Jumping around or out of loops

- There might be times when you want to leave a loop early.
  - e.g. you are searching for a value in an array, and you can finish the loop early if you find it
- There might be times when you want to skip the current iteration of the loop, and go on to the next one
  - e.g. you only want to process even numbers in an array.
- In order to make writing this kind of code easier, there are two control flow constructs you can use in any of the loops seen so far:
  - `break`: which will exit the loop and carrying on execution from the next statement after the loop.
  - `continue`: which will jump to the next iteration of the loop.
- Sometimes loops will be *nested*, so you can use *labels* to indicate which loop you wish to `break` or `continue`.

`break`, `continue` and labels

Rolling `n` sixes in a row, and reporting how many attempts it took

```
private static int rollNSixesInARow(int n) {
    int diceThrown = 0;

    outer: while (true) {
        for (int i = 0 ; i < n ; i++) {
            int attempt = rollDie();
            diceThrown++;
            if (attempt != 6) {
                continue outer;
            }
        }
        break;
    }

    return diceThrown;
}
```


1. `outer:` is an optional label. Is says this `while` loop is called `outer` (you can replace `outer` with any name you like)
2. The `while (true)` loop says it will loop forever. However we can leave the loop using `break` (or `return`)
3. The `for` loop attempts to roll `n` 6's in a row. If we get to the end of the `for` loop then we are done and can leave the `while` loop, which we do using `break`
4. However, if we don't roll a 6 within the `for` loop, then we have to try again. We use `continue` to try another iteration of the `while` loop.
5. Since we are nested within a `for` loop, we have to say `continue outer` to try the next iteration of the `while` loop

Summary

- There are many different ways of performing repeated execution in Java.
- `while` statements are the most general form of looping.
- Recursive methods can be written using a loop instead. However care must be taken to ensure they have the same behaviour.
- There are some common patterns that occur when using `while` statements, which gives rise to the `do-while` statement and the `for` statement.
- Sometimes you will want to skip an iteration of a loop, or to exit it early, in which case a `continue` or `break` statement is needed.
- If you have nested loops, then you can label your loops if you want to refer to a particular loop in a `continue` or `break` statement.

Objects

Programming II

The story so far...

- So far we have been using Java to develop methods that could be placed into utility libraries.
- These tend to be small and self contained, usually performing a single job. e.g.
  - `biggest` : returning the largest of three numbers.
  - `encodeInt` : converting an `int` into its Morse code representation.
  - `reverse` : reversing the contents of an array
  - `rollDie` : simulating a dice roll
- This is a *procedural* style of program writing.
- However Java is primarily an *Object Oriented* programming language, and has many sophisticated language features for creating and working with *Objects*.

Objects

Things that have State, Behaviour and Identity

State

- Internal information that the object uses to know how to behave.
- Usually hidden, or only accessed / updated through a well defined interface.
- For example, a watch knows the current time, traffic lights know how long until they change to red.
- State is modelled in Java by using *fields*. These are variables that persist across multiple method calls on the object.

Objects

Things that have State, Behaviour and Identity

Behaviour

- This is the external stimuli an object can respond to.
- Usually publicly available, this is the well defined interface that the object lets the rest of the world interact with it by.
- For example, if asked to change, a traffic light can tell you the next colours it will display.
- Behaviour is modelled in Java by *instance methods*. These can:
  - Accept arguments.
  - Read and write to the object's state.
  - Return results.

Objects

Things that have State, Behaviour and Identity

Identity

- There can be many different objects, each with different internal state and possessing different behaviours.
- We may want to create many similar objects that have the same state and behaviour descriptions, but can co-exist in different states at the same time.
- For example, most traffic lights in London look the same, but they don't all show red at the same time.
- In Java, the description of an object is called its *class*, and an object that follows the description given by a class is said to be an *instance* of that class.
- Classes are described by the `class` construct, and instances are created using `new`.

A Clicky Counter

An example of an Object

Imagine a simple device with two buttons labelled `tick` and `getTicks`. The `tick` button increments a count of how many times it has been pressed. The `getTicks` button tells you how many times the `tick` button has been pressed.

State

- The number of times the button has been pressed.
- Can be stored in an `int` called `count`.

Behaviour

- `tick` will accept no arguments, increment the state, and return no results.
- `getTicks` will accept no arguments, read the state and return it.

Identity

- We could create many counters and increment them separately.

Classes describe Objects

The description of a counter

```
public class Counter {

    private int count = 0;

    public void tick() {
        count++;
    }

    public int getTicks() {
        return count;
    }
}
```


Classes describe Objects
The description of a counter

1. `public class Counter` must live in a file called `Counter.java`
2. `private int count` is an *instance field* of the class. It is declared within the class but not inside any method.
3. Each `Counter` instance that is created will get its own `count` value that will store its value as long as the instance exists.
4. The `= 0` is optional (as `int` fields default to `0`), but makes things clearer.
5. We make the `count` variable `private` to keep it hidden. Only methods declared within the class `Counter` can access it.
6. The `public void tick()` is an *instance method* declaration. It can access the field `count` and modify it. Note the *lack* of the `static` keyword.
7. Since we only care about the side effect of incrementing the count, `tick` is a `void` method. It doesn't return anything.
8. The `getTicks` instance method reads the current value of `count` and returns it.

Creating *instances* of Objects

Making a `Counter` `tick`

```
public class TickTock {

  public static void main(String[] args) {

    Counter counter = new Counter();
    System.out.println(counter.getTicks());

    System.out.println("Tick!");
    counter.tick();

    System.out.println(counter.getTicks());

  }

}
```


Creating *instances* of Objects
Making a Counter tick
public class TickTock {
public static void main(String[] args) {
Counter counter = new Counter();
System.out.println(counter.getTicks());
System.out.println("Tick!");
counter.tick();
System.out.println(counter.getTicks());
}
}

1. To create a new `Counter` object, write `new Counter()`
2. This will create space in the heap for the fields of `Counter` and return a pointer to it that we store in the `counter` variable.
3. In order to invoke the instance methods `tick` and `getTicks` we have to say which instance of `Counter` we want to call them on.
4. This specification happens through the use of a `.`, e.g. `counter.getTicks()` or `counter.tick()`
5. You can read `counter.tick()` as, on the instance of `Counter` pointed to by `counter`, invoke the `tick` method with no arguments.

A more flexible counter

- We can write a description of more flexible counter that also allows you to fix the value of `count`.
- To do this, we'll add a new behaviour, `setTicks` that accepts an `int` argument and uses that as the new value of `count`.

A more flexible counter

An example of an instance method accepting an argument and writing to the state

```
public class ResettableCounter {

  private int count = 0;

  public void tick() {
    count++;
  }

  public int getTicks() {
    return count;
  }

  public void setTicks(int i) {
    count = i;
  }
}
```


1. The setTicks method accepts a single argument.
2. Again, we are only interested in the side effect of updating the state of ResettableCounter, so it is also a void method.
3. In Java, if a private field is to be updatable, it is a common pattern to use methods named get* and set* (getters and setters).

Creating several instances

An example of multiple ResettableCounters with different values.

```
public class TickTockTwo {

  public static void main(String[] args) {

    ResettableCounter c1 = new ResettableCounter();
    ResettableCounter c2 = new ResettableCounter();

    for(int i = 0 ; i < 5 ; i++) {
      c1.tick();
    }

    System.out.println("c1: " + c1.getTicks());

    for(int i = 0 ; i < 10 ; i++) {
      c1.tick();
      c2.tick();
    }
    System.out.println("c1: " + c1.getTicks());
    System.out.println("c2: " + c2.getTicks());

    c1.setTicks(0);

    System.out.println("c1: " + c1.getTicks());
    System.out.println("c2: " + c2.getTicks());

  }
}
```


1. We first create two different counters, and store pointers to them in c1 and c2.
2. We then tick c1 five times and print out it's getTicks
3. Next we tick both counters ten times. Printing out their ticks will give different internal counts.
4. Finally we reset c1 back to a count of 0.
5. Again, printing out the ticks of c1 and c2 will have different results.
6. What happens if instead of writing new ResettableCounter() we put ResettableCounter c2 = c1;?

Static vs Instance

Static

- Static methods and fields are not associated with any instance.
- If they are `public` you can call or read/write to them from anywhere - within instances or static methods.
- They are denoted by the keyword `static`.
- Think “there can only be one”.

Instance

- Instance methods and fields are associated with an instance of a class.
- If they are `public` They can be called or read/written to only if you have an instance of that class already.
- They are denoted by the *absence* of the keyword `static`.

Static vs Instance

The elephant in the room

`System.out.println()`

- `System` - the name of a class.
- `out` a *static* field of class `System`. Which points to an instance of a `PrintStream` (things that can be printed to).
- `println()` - an *instance* method of `PrintStream`.

You can find another instance of `PrintStream` through the static field `System.err`.

Objects: Another Example

A Simple Calculator

Imagine a small simple calculator. It should start at zero, and it has methods to add or multiply its current value by an `int`. It should also be able to represent its current calculation as a `String`.

State

- The current value of the calculation, represented by an `int` field called `total`.
- The `String` representing the calculation so far. Call it `calculation`.

Behaviour

- Methods `plus` and `multiply` that accept an int and update the `total` and `calculation` fields accordingly.
- A method `getTotal` to return the current `total`.
- A method `reset` to reset the current `total` and `calculation` back to `0`.
- A method `toString` which will represent the state of the calculator as a `String`.

Using the Calculator

With only the previous description of the behaviour we can say what our calculator should do.

```
public class Main {

  public static void main(String[] args) {
    Calculator c1 = new Calculator();

    c1.plus(5);
    c1.multiply(3);
    c1.plus(7);
    System.out.println("c1 total: " + c1.getTotal());
    System.out.println(c1);

    c1.reset();
    System.out.println(c1);
  }
}
```


1. The class that describes our calculator will be called `Calculator`.
2. We should be able to call `plus` and `multiply` methods upon it, and it should build up the correct total.
3. In Java, if a class describes a method with the signature `public String toString()` then `println` will implicitly use that.

The `Calculator`

The implementation.

```
public class Calculator {

  private int total = 0;
  private String calculation = "0";

  public void plus(int amount) {
    total += amount;
    bracket();
    calculation += " + " + amount;
  }

  public void multiply(int amount) {
    total *= amount;
    bracket();
    calculation += " * " + amount;
  }

  private void bracket() {
    calculation
      = "(" + calculation + ")";
  }

  public int getTotal() {
    return total;
  }

  public void reset() {
    total = 0;
    calculation = "0";
  }

  public String toString() {
    return calculation +
      " = " + total;
  }
}
```


1. Both `plus` and `multiply` need to put brackets around `calculation`, so we create a `private` helper method.
2. `bracket()` is this private instance method. It can only be called by other methods defined withing `Calculator`.
3. Within an instance method, you can call other instance methods on the same instance *implicitly*, i.e. without needing the `instance.` syntax.

Constructors

Executing code when creating an instance

- Sometimes when you create an instance you would like some custom code to execute.
- Frequently this is used to initialize the fields of the object to a known state, to make sure some property of the fields holds.
- You may also wish to pass into the object some initial values to use for the fields.
- This code is specified by a special method called a *constructor*. Constructors can accept arguments and modify the fields of an instance, but they *can't* return results.

InitializedCounter

A counter which is told it's initial count when created

```
public class InitializedCounter {

  private int count;

  public InitializedCounter(int count) {
    this.count = count;
  }

  public void tick() {
    count++;
  }

  public int getTicks() {
    return count;
  }

}
```


1. Note the `InitializedCounter` constructor method.
2. The constructor doesn't have a return type, as it doesn't return results.
3. Frequently the parameter names of the constructor will *shadow* the names of fields. To get around this, fields can be referred to by prefixing with `this.`
4. `this` is a variable that refers to the current instance.

TickTock - Constructors II

```
public class TickTockInitialized {

  public static void main(String[] args) {

    InitializedCounter counter = new InitializedCounter(10);
    System.out.println(counter.getTicks());

    System.out.println("Tick!");
    counter.tick();

    System.out.println(counter.getTicks());

  }

}
```


1. If a class constructor expects arguments, you can pass them to it during the `new` call.
2. The arguments are passed between `( )`s after the class name.

The DragonsBreath Dungeons

Working with Objects

The DragonsBreath Dungeons

We are going to build up a slightly larger example, with several classes and lots of different instances working together in a single program.
Our program is going to be a very simple dungeon game, which features four classes:

- The `Player` class. This describes our hero, who braves the fearsome dungeon, fighting monsters and gaining experience, while trying not too lose all their health.
- The `Monster`s. This describes the template of a monster, which attacks our hero and dies when they run out of health.
- The `Dungeon`. This holds a player and the monsters within. It also co-ordinates the attack phases between monsters and players, and signals when the game is over.
- `DragonsBreath`. Contains the static `main` method, and manages the main game loop and input routines.

Monsters

We wish to create several different variations of `Monster`, for example Orcs, Dragons and Bunnies.

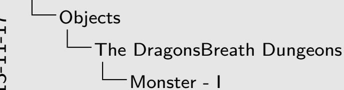
State

- Their `name`, a `String` that will not change after the instance is created.
- Their `attackStrength`, an `int` that will not change after the instance has been created.
- Their `health`, an `int` that will be initialized to a set value.

Monster - I

The description of things that live in our dungeon

```
public class Monster {

  private final String name;
  private final int attackStrength;
  private int health;

  public Monster(String name,
                 int attackStrength, int health) {
    this.name = name;
    this.attackStrength = attackStrength;
    this.health = health;
  }

  public String getName() {
    return name;
  }
  public boolean isAlive() {
    return health > 0;
  }
```

1. The `public Monster( ... )` declaration is the class constructor
2. Constructors are methods without a return type, and with the same name as the class
3. If you don't write a constructor, a default is created for you which is roughly equivalent to `public ClassName() { }`
4. The `name` and `attackStrength` fields are marked `final`. Instance methods cannot change the value of a final field.
5. `final` fields *must* be initialized in place or in a constructor.
6. Frequently the parameter names of the constructor will *shadow* the names of fields. To get around this, fields can be referred to by prefixing with `this.`
7. `this` is a variable that refers to the current instance.

Calling the Monster Constructor

Bunny!

```
public class MainMonsters {

  public static void main(String[] args) {

    Monster monster = new Monster("Cute Bunny", 0, 7);

    System.out.println(monster.getName()
      + " " + monster.isAlive());
  }

}
```


Monster

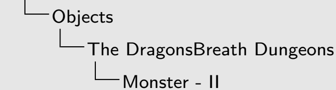
Behaviour

- `getName` - returns the name of the Monster
- `isAlive` - returns whether the health of the monster is > 0.
- `takeDamage` - receives an amount of damage to take and reduces health by that amount.
- `toString` - represents the monster as a `String`
- `attack` - accepts a `Player` as an argument, and attacks them.

Monster - II

Implementations of Behaviour

```
  ...

  public void takeDamage(int damage) {
    health = Math.max(health - damage, 0);
  }

  public String toString() {
    String aliveOrDead = isAlive() ? ":)" : "x";
    return name + "    H: " + health
      + "    A:" + attackStrength + "    " + aliveOrDead;
  }

  public void attack(Player player) {
    player.takeDamage(attackStrength);
  }

}
```

1. The `condition ? expression : expression` syntax is an expression-level conditional.
2. (compare to `if (condition) { ... } else { ... }` which is a statement-level conditional.)
3. The `attack` method receives a `Player` as an argument, and then instructs them to `takeDamage` according to the `attackStrength` of the monster.
4. Note that the `player.takeDamage(...)` method will be a method declared in the `Player` class, *not* the `takeDamage` method declared here.

Player

State

- `int health` - the remaining health of the Player.
- `int experience` - This is increase by killing monsters, and will make the player tougher and stronger.
- `final int attackStrength` - The base attack damage the player does. It will be multiplied by their experience.

Behaviour

- `attack` - attacks a monster. If they succeed in killing the monster, the player gains experience.
- `takeDamage` - reduces the players health by an amount of damage, modified by `experience`.
- `isAlive` - returns whether the player's health is `>0`.
- `toString` - returns a `String` representation of the player.

Player

```
public class Player {

  private int health;
  private final int attackStrength;
  private int experience;

  public Player(int health,
                int attackStrength) {
    this.health = health;
    this.attackStrength
      = attackStrength;
    this.experience = 1;
  }

  public void attack(Monster monster) {
    if (!monster.isAlive()) {
      return;
    }

    monster.takeDamage(
      attackStrength * experience);

    if (!monster.isAlive()) {
      experience++;
    }
  }

  public void takeDamage(
          int monsterAttackStrength) {
    health = Math.max(0, health -
     monsterAttackStrength / experience);
  }

  public boolean isAlive() {
    return health > 0;
  }

  public String toString() {
    String aliveOrDead
      = isAlive() ? ":)" : "x";
    return "Player: H: " + health
     + "    A:" + attackStrength
     + "    E: " + experience
     + "    " + aliveOrDead;
  }
}
```


Dungeon

Holds a player and some monsters. Coordinates the attacking of creatures held within, and knows when the game is over.

State

- `final Player player` - The `Player` that has braved the dungeon.
- `final Monster[] monsters` - An array of dead and alive `Monster`s that live in the dungeon.
- `final Random random` - An instance of a Java utility class that provides more flexible random numbers than just using `Math.random()`.

Note that although all the state is `final`, the states of the individual `Player` and `Monsters` can change, just that the `Dungeon` cannot change which `Player` instance it knows about.

Dungeon

Behaviour

- `printDungeon` - Print out a representation of the dungeon to the console.
- `isGameOver` - The game is over if the player dies, or all the monsters have died.
- `randomMonsterAttack` - Causes a random monster to attack the player.
- `playerAttack` - Causes the player to attack a particular monster.

Dungeon - I

```
import java.util.Random;

public class Dungeon {

  private final Player player;
  private final Monster[] monsters;
  private final Random random;

  public Dungeon(Random random, Player player) {
    this.player = player;
    this.monsters = new Monster[]
      { new Monster("Tiny Mouse", 1, 5),
        new Monster("Vam-Goblin", 2, 10),
        new Monster("Orc Wizard", 3, 15),
        new Monster("Ice Dragon", 20, 50),
        new Monster("Cute Bunny", 0, 7) };
    this.random = random;
  }

  ...
```


1. We need to explicitly `import java.util.Random;` to refer to the `Random` class.
2. We initialize `monsters` with an array of new Monster instances. Every time the constructor is called, new, fresh monsters are created.

Dungeon - II

```
  ...

  public void printDungeon() {
    System.out.println("Our Hero:");
    System.out.println(player);

    System.out.println("The foul monsters: ");
    for(int i = 0 ; i < monsters.length ; i++) {
      System.out.println(i + " - " + monsters[i]);
    }
  }

  ...
```


1. `println( )` will use the `toString` method defined on `Player` to represent them.
2. The `toString` method is also implicitly used if you try and concatenate (`+`) a instance object onto a `String`.
3. Here, for example, `monsters[i]` is a `Monster`, and its `toString` is used in the `println( ... )` call.
4. Note the use of the `for( ... ; ... ; ... )` loop to let us print out both the index of the monster, and the monster itself.

Dungeon - III

```
...

public boolean isGameOver() {
  return !player.isAlive() || areAllMonstersDead();
}

private boolean areAllMonstersDead() {
  for (Monster monster : monsters) {
    if (monster.isAlive()) {
      return false;
    }
  }
  return true;
}

...
```


1. areAllMonstersDead is a private helper method in Dungeon.
2. If it detects any monster is alive, then it can immediately return false.
3. Note the use of the enhanced for loop to check all the monsters when we don't care about their positions in the array.

Dungeon - IV

```
/* Causes a random living monster to attack the Player
 * return The monster which attacked the player
 */
public Monster randomMonsterAttack() {
  assert !isGameOver() : "Monster cannot attack if game is over";

  Monster attackingMonster;
  do {
    attackingMonster = monsters[random.nextInt(monsters.length)];
  } while (!attackingMonster.isAlive());

  attackingMonster.attack(player);

  return attackingMonster;
}

...
```


1. This method does two things, it causes a random monster to attack the player, *and* it returns the monster that attacked.
2. The precondition of this method is that the game isn't over.
3. In order to keep choosing random monsters until we get one that is alive, we use a do-while loop. do-while is appropriate as we definitely want to run the body of the loop at least once.
4. In order to choose a random monster, we use the instance method nextInt on our random field.
5. nextInt(value) returns a random number between 0 (inclusive) and value (exclusive) - perfect for choosing a random value from an array!
6. The method causes the chosen attackingMonster to attack the player, and also returns it for printing out later.

Dungeon - V

```
public void playerAttack(int i) {
  assert player.isAlive() && i >= 0
      && i < monsters.length
      && monsters[i].isAlive()
        : "Player cannot attack monster " + i;
  player.attack(monsters[i]);
}
```


Dungeon - V

1. This method tells the `player` to attack the monster at index `i` in the `monsters` array.
2. It has a big precondition: the player must be alive, the index must be a valid index into the array, and the monster at that index must be alive.
3. What would happen if the `monsters[i].isAlive()` check was at the beginning of the precondition instead of at the end?

DragonsBreath

Tying it all together.

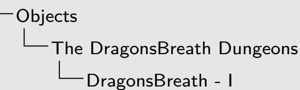
This class has two static methods:

- The `main` method that runs the game loop
- and a helper method, `checkDifficultyAndGetPlayer` which prints out a menu to choose the difficulty of the game.

The difficulty setting changes the initial strength and health of the `Player` which is put into the dungeon.

DragonsBreath - I

```
import java.util.Random;

public class DragonsBreath {

  private static Player checkDifficultyAndGetPlayer() {
    System.out.println("Please choose your difficulty:");
    System.out.println("1: Easy");
    System.out.println("2: Normal");
    System.out.println("3: Hard");

    while (true) {
      int response = IOUtil.readInt();
      switch (response) {
      case 1:
        return new Player(1000, 100);
      case 2:
        return new Player(100, 5);
      case 3:
        return new Player(15, 1);
      default:
        System.out.println("Invalid response, please try again!");
      }
    }
  }
```

1. Since DragonsBreath will also need to use Random it needs to import java.util.Random too.
2. The while (true) loop is used to keep asking the user for a difficulty until they choose a correct one.
3. We vary the constructor parameters to the Player that is returned to alter the difficulty.

DragonsBreath - II

The main loop of the game

```
  public static void main(String[] args) {
    Random random = new Random();
    System.out.println("Hello and welcome to DragonsBreath!");
    Player player = checkDifficultyAndGetPlayer();
    Dungeon dungeon = new Dungeon(random, player);

    while (!dungeon.isGameOver()) {
      dungeon.printDungeon();

      System.out.println("Which monster do you wish to attack?");
      int monsterId = IOUtil.readInt();
      //TODO: check this is a valid monster;
      dungeon.playerAttack(monsterId);

      if (dungeon.isGameOver()) {
        break;
      }

      Monster monsterThatAttacked = dungeon.randomMonsterAttack();
      System.out.println("You were attacked by the: "
        + monsterThatAttacked.getName() + "!");
    }

    System.out.println("Game over!");
    dungeon.printDungeon();
  }
}
```


Testing

Testing Static Methods

- When testing functions in Haskell and simple static methods in Java it was enough to enumerate simple test cases matching inputs to expected outputs.
- For example:

```
public static void sumSquareDigitsTests() {
        checkSumSquareDigits(10, 1);
        checkSumSquareDigits(103, 10);
        ...
}
```

- These test cases represented the fact that `sumSquareDigits(10)` should equal `1`, and that `sumSquareDigits(103)` should equal `10`.

Testing Objects?

- Testing objects is different. You can't think of an object as being a mapping from inputs to outputs.
- Recall that an object consists of three parts: State, Behaviour and Identity.
  - Identity - this is managed for us by Java. New, unique things are created via `new`.
  - State - this is internal and hidden and used only by the object.
  - Behaviour - this is external and visible to others using the object.

Testing Objects?

State

- From outside an object you can't see its internal state.
- Furthermore, we don't really want to - we want the state to be *encapsulated*.
- We don't care how the object does what it does, only that it does it correctly.
- This means it should be safe to change how an object works internally.
  - e.g. `Monster`s could store a `boolean` field saying if they are dead or alive and update and use that instead of checking if `health > 0` in `isAlive`.
- i.e. we don't want to test the state directly.

Testing Objects?

Behaviour

- The behaviour of an object is specified by its public instance methods.
- We can observe the return values of these methods and whether they are what we expect.
  - e.g. if we have just created a `Monster` with `10` health, we expect `isAlive` to return `true`.
- Some methods are `void`. However we can also observe their side effects on the current object.
  - e.g. After calling `takeDamage(20)` on a `Monster` that has been created with `10` health, we'd expect a subsequent call of `isAlive` to return `false`.
- We can also observe the side effects of `void` methods on other objects.
  - e.g. After calling `attack(player)` on a `Monster` that has been created with an attack damage of 5, we'd expect a newly created `Player` with health `10` to still be alive after the call.

Testing Objects - Examples

Testing Monster

```
public class MonsterTests {

  public static void main(String[] args) {
    System.out.println("Running tests...");

    constructorTests();
    takeDamageTests();
    attackTests();

    System.out.println("...tests complete");
  }
```


1. We begin with a small program that contains some tests.
2. The different tests are grouped together by category for the method they are primarily testing, tests for the constructor, for the `takeDamage` method and the `attack` method
3. For more complicated objects it may be important to test the interaction of multiple methods, and so new categories could be created for them.

Testing Objects - Examples

Two Helper Methods

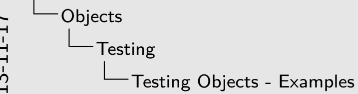
```
public static void testGetName(Monster monster,
                               String expectedName) {
  String actualName = monster.getName();
  if (!expectedName.equals(actualName)) {
    System.out.println("monster.getName() returned:"
      + actualName + ", expected: " + expectedName);
  }
}

public static void testIsAlive(Monster monster,
                               boolean expected) {
  boolean actual = monster.isAlive();
  if (expected != actual) {
    System.out.println("monster.isAlive() returned:"
      + actual + ", expected: " + expected);
  }
}
```


1. `getName` and `isAlive` are two methods that return results that we can check.
2. We will use these helper methods to check that the actual results from calling those methods matches what we expect.
3. These methods only print out if results are unexpected.
4. If we run large numbers of tests, it is clearer to only see those that fail than going through lots of output trying to work out which pass and which fail.
5. Note that to test whether two `String`s are equal, we use the instance method `equals( otherString )`.

Testing Objects - Examples

Testing Monster's Constructor

```
private static void constructorTests() {
  testGetName(new Monster("test", 5, 0), "test");
  testIsAlive(new Monster("test", 5, 0), false);
  testIsAlive(new Monster("test", 5, 1), true);
  testIsAlive(new Monster("test", 5, 10), true);
}
```

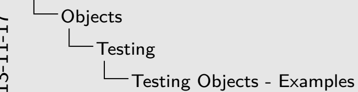



1. When creating tests, try to make each test as self-contained as possible.
2. Here, we create a new Monster for each test, to that we know we are only testing the constructor call and one public method call.
3. We could create a new category for more complicated calls later, but only after we've tested the simplest cases first.

Testing Objects - Examples

Testing Monster's takeDamage method

```
private static void takeDamageTests() {
    Monster testMonster;

    testMonster = new Monster("test", 5, 10);
    testMonster.takeDamage(5);
    testIsAlive(testMonster, true);

    testMonster = new Monster("test", 5, 10);
    testMonster.takeDamage(10);
    testIsAlive(testMonster, false);

    testMonster = new Monster("test", 5, 10);
    testMonster.takeDamage(5);
    testMonster.takeDamage(5);
    testIsAlive(testMonster, false);
}
```

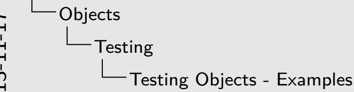



1. There are three of many possible examples of testing the takeDamage method here
2. The simplest cases are tested first (just calling takeDamage once), and then a more complicated example calling takeDamage twice.

Testing Objects - Examples

Testing `Monster`'s `attack` method

```
private static void attackTests() {
    Monster testMonster;
    Player testPlayer;

    testMonster = new Monster("test", 5, 10);
    testPlayer = new Player(5, 10);
    testMonster.attack(testPlayer);
    PlayerTests.testIsAlive(testPlayer, false);

    testMonster = new Monster("test", 5, 10);
    testPlayer = new Player(10, 10);
    testMonster.attack(testPlayer);
    PlayerTests.testIsAlive(testPlayer, true);
}
```

1. The `attack` method should change the health status of the `testPlayer`. So to see its effect, we look at the `isAlive` status of the `testPlayer` after the `attack` method call.
2. Again notice that before each test, we recreate the `Monster` and `Player`, so that the tests are as minimal as possible.
3. We could also add some tests that `attack` doesn't change our expectations of whether the `testMonster` is alive.

When can you write tests

- Before writing the code that implements it.
  - You'll know when you've implemented the feature because the tests all pass.
  - Writing the tests can sometimes guide the design of your object.
- Before fixing a bug found in a program.
  - If you have a test that isolates the bug, then debugging gets easier.
  - If you already have a test suite, then adding new test cases should make this easy.
  - If when you introduce a bug you find it, fix it and add a test for it (not necessarily in that order!), you'll never have to worry that the bug might come back. (This does happen!)
- Before changing/restructuring the internal workings of an object.
  - Arrange to have passing tests before making the change.
  - Once you've changed the code, you can rerun the tests.
  - If any fail then you've changed the behaviour of the object, as-well as its state.

Note

- This is just scratching the surface of testing Java code.
- Next term you'll see more features of Java that will make it possible to create modular, flexible test suites in a disciplined way.
- You will also get to see (and create!) much larger codebases and be exposed to different forms of testing, for example:
  - *Integration Testing* - testing a whole program from end to end.
  - *Unit Testing* - testing the individual components (in this case objects).
  - *Regression Testing* - using existing tests to check changed or new code still works.
  - *Automated Testing* - using tools to help you create tests.

Generics

Polymorphism

Parametric Polymorphism

- Meaning methods and classes can be parameterized by one or more type variables.
- This allows for general purpose data structures and algorithms.
- Both Haskell and Java (>= 1.5) have this (thanks to generics).
- When people talk about polymorphism in the context of Haskell, this is usually what they mean.

Ad-hoc polymorphism

- This is often called *overloading*.
- Methods or functions can have the same name but different argument types.
- Haskell has this through its type class mechanism (e.g. `show` or `==`).
- Java supports this. For example, `println()`, `println(3)`, `println(false)`.

Polymorphism

Subtype polymorphism

- Also known as *inheritance*.
- When people say polymorphism in the context of Java, this is usually what they mean.
- Haskell doesn't have this.
- Next term you will learn how to use Java's inheritance capabilities.

Parametric Polymorphism in Haskell

Extracting the first element out of a tuple

```
fst :: ( a, b ) -> a
```


Parametric Polymorphism in Haskell
Extracting the first element out of a tuple
fst :: (a, b) -> a

1. In Haskell, type variables are declared implicitly by using types with a lower case letter
2. Data structures can also use type variables
3. Type variables can be instantiated to concrete types or other type variables when used
4. For example, the `fst` function extracts the first element out of a tuple of any type

Parametric Polymorphism in Haskell

Creating your own tuple data structure

```
data Tuple x y = Tuple x y

getFst :: Tuple x y -> x
getFst (Tuple a b) = a

getSnd :: Tuple x y -> y
getSnd (Tuple a b) = b
```

```
Main> getFst (Tuple "hello" "world")
"hello"
```


Parametric Polymorphism in Haskell
Creating your own tuple data structure
data Tuple x y = Tuple x y
getFst :: Tuple x y -> x
getFst (Tuple a b) = a
getSnd :: Tuple x y -> y
getSnd (Tuple a b) = b
Main> getFst (Tuple "hello" "world")
"hello"

1. The `Tuple` type takes two types
2. Its data constructor (also called `Tuple` stores a value of of each inside itself
3. `getFst` and `getSnd` access those elements and return them

Generics in Java

Creating a tuple class in Java

```
class Tuple<X, Y> {

  private final X x;
  private final Y y;

  public Tuple(X x, Y y) {
    this.x = x;
    this.y = y
  }

  public X getFst() {
    return x;
  }

  public Y getSnd() {
    return y;
  }
}
```


1. The type parameters to a class are put between `< >`'s
2. Within the definition of the class Tuple, you can use `X` and `Y` as though they were a type.
3. So, for example, they are used as the types of the `x` and `y` fields.
4. They are also used as the types of the `x` and `y` parameters to the constructor.
5. They are also the return types of the `getFst` and `getSnd` methods.

Generics in Java

Creating an instance of a `Tuple`

```
public class TupleHelloWorld {

    public static void main(String[] args) {
        Tuple<String, String> helloWorld
            = new Tuple<>("Hello", "World");

        System.out.println(helloWorld.getFst());

    }

}
```


1. Variables of Tuple type must be given their type arguments, again using `< >`'s.
2. When calling `new`, you also have to give the type parameters
3. Note, in Java 6 you would have to write
   `new Tuple<String, String>("Hello", "World");`
4. `helloWorld.getFst()` will have a return type of `String` in this example.

A duplicate function Haskell

```
makeDuplicate :: a -> (a, a)
makeDuplicate x = (x, x)
```

```
Main> (snd . makeDuplicate) "hello"
"hello"
```


A duplicate method in Java

Type variable declarations for methods

```
public class UsingTuples {

  public static <T> Tuple<T, T> makeDuplicate(T value) {
    return new Tuple<>(value, value);
  }

  public static void main(String[] args) {
    String result = makeDuplicate("hello").getSnd();
    System.out.println(result);
  }

}
```


1. Sometimes you will want a method to be parameterized by a type variable.
2. New type variables need to be explicitly declared between `< >`'s before the return type of the method is given.
3. For example, within the `makeDuplicate` method:
4. The type parameters are `T` (from `<T>`)
5. The return type is `Tuple<T, T>`
6. The argument type is `T` (from `T value`)
7. You can only refer to `T` within the `makeDuplicate` method.

A lookup function in Haskell

```
stringLookup :: String -> a -> [(String, a)] -> a
stringLookup key defalt [] = defalt
stringLookup key defalt ((k,v) :xs)
  | key == k     = v
  | otherwise    = stringLookup key defalt xs
```

```
*Main> stringLookup "hello" 3 [ ("world", 1), ("hello", 2) ]
2
```

A lookup method in Java

```
public static <Y> Y lookup(String key,
                           Tuple<String,Y>[] elems,
                           Y defalt) {
    for (Tuple<String,Y> elem : elems) {
        if (key.equals(elem.getFst())) {
            return elem.getSnd();
        }
    }
    return defalt;
}
```


Reminder: Primitive and Reference Types

Java's Primitive Types

- `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, `char`
- Their value lives on the *stack*, and is *copied* when assigning to variables/fields or when passed into / out of methods.
- Always start with a lowercase letter

Java's Reference Types

- `String`, arrays of anything and instances of classes.
- Their contents lives in the *heap*, and variables / fields get a pointer to their contents. This pointer is copied, but the contents themselves are not.
- By convention they start with a capital letter.
- Variables and fields of reference type can have the value `null` which means they don't point to a value (yet).

Generic Restrictions

- Java's generics were introduced in Java 1.5, and as such had to maintain backwards compatibility with exisiting Java code and virtual machines.
- Java's generics have several (seemingly odd) restrictions:
  - A type variable can only be used to represent reference types.
  - You cannot create arrays (with `new` or `{ elem1, elem2, ... }`) for classes that have type parameters.
  - You cannot create a new instance of a type variable using `new`.
- These restrictions all have workarounds, often supported by the standard library.

Type Variables can only represent reference types

- This means you cannot use `Tuple<String, int>` as a type for a variable, for example, as `int` is a primitive type.
- However, Java has a set of reference types that *box* the primitive types.
- These boxes live on the heap like other reference types, but are immutable (i.e. you cannot change the value they wrap).

Primitive Type	Reference Type
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`boolean`	`Boolean`
`char`	`Character`

Boxes for Primitive Types

- You can create instances of the box classes using their constructors, as per normal classes. You can then use the box's instance methods to unwrap the primitive they contain. For example:

```
Integer i = new Integer(2);
int j = i.intValue();
```

- In many cases, Java can work out when you need to do the wrapping / unwrapping and can do it for you. This is a feature called *autoboxing*. The above example could equally be written as:

```
Integer i = 2;
int j = i;
```

- The box classes have lots of useful `static` and instance methods.
- Be aware, that autoboxing will likely crash your program if you try and convert a `null` box into a primitive, e.g.

```
Integer i = null;

//this line will crash
int j = i;
```


Back to Lookup

- So the workaround to convert the Haskell example is to create a `Tuple<String, Integer>`.
- However, we have the second restriction (you can't create an array with a parameterized type). Which means at the moment even if we could call the method, we can't create an array of values to lookup in!
- i.e. you cannot write:

```
Tuple<String, Integer>[] values = {
        new Tuple<>("one", 1),
        new Tuple<>("two", 2),
        new Tuple<>("three", 3) };
```

- Or, being more explicit (also Java rejects this):

```
Tuple<String, Integer>[] values =
  new Tuple<String, Integer>[] {
        new Tuple<>("one", 1),
        new Tuple<>("two", 2),
        new Tuple<>("three", 3) };
```


The workaround?

- Since Java has to be backwards compatible with versions of itself before generics were introduced, it is possible to "forget" that a class needs type parameters.
- In general, you should *always* put the type parameters in.
- You will get a warning when compiling when you use this workaround.

```
Tuple<String, Integer>[] values = new Tuple[] {
   new Tuple<>("one", 1),
   new Tuple<>("two", 2),
   new Tuple<>("three", 3) };

int result = lookup("two", values, -1);
```

- Next term you'll learn about Java's Collections which have type safe lists, maps and sets which means you'll very rarely need to use arrays and this workaround!

Enumerations

Enumerations

- An *enumerated type* is a type whose legal values consist of a fixed set of constants.
- If your program needs a fixed set of constants then using an enumerated type makes your program more readable and more maintainable.
- In Java, the values in the enumerated type are also objects, which means they can have constructors and instance methods which makes it easy to have per-constant behaviour.

Simplest Examples

Haskell and Java enumerated types

In Haskell

```
data Day = Sunday | Monday | Tuesday | Wednesday
         | Thursday | Friday | Saturday
```

In Java

```
public enum Day {
  SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
  FRIDAY, SATURDAY;
}
```


Simplest Examples
Haskell and Java enumerated types
In Haskell
data Day = Sunday | Monday | Tuesday | Wednesday
| Thursday | Friday | Saturday
In Java
public enum Day {
SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
FRIDAY, SATURDAY;
}

1. In Java, `enum` is bit like `class`.
2. The `Day` enum must live in a file called `Day.java`
3. By convention, Java constants (and enumeration constants) are written in all capital letters
4. Note: enums were added in Java 1.5.

Other Examples

- Compass Directions (North, East, South and West)
- Days of the week
- Months of the year
- Ranks and Suits in a deck of cards
- Planets in our solar system

Using an enumerated type

Working with Days

```
public class DayExample {

    public static void main(String[] args) {
        Day today = Day.MONDAY;
        System.out.println(today);

        System.out.println("The week: ");

        for (Day day : Day.values()) {
            String tail = today == day ? " <-- Today!" : "";
            System.out.println(day + tail);
        }

        System.out.println("Today's index:");
        System.out.println(today.ordinal());
    }
}
```


1. To reference an enum constant you have to prefix it with the name of the enum. i.e. `Day.MONDAY`
2. By default, when you convert an enum constant to a `String` it will return its name (`"MONDAY"`)
3. Enum classes (e.g. `Day`) have some `static` methods automatically declared for them, for example `values()` which returns all of that enum's constants, in an array, in order.
4. You are guaranteed by Java that there will only ever be one instance of the enum for each enum constant. This means that `==` will work on them.
5. If you wish to know what the index of an enum value is in the array, you can call its instance method `.ordinal()`

Using an enumerated type

Enumerations work with case expressions

```
public static String whatToDoToday(Day day) {
    switch (day) {
    case MONDAY:     return "Lectures";
    case TUESDAY:    return "Minecraft";
    case WEDNESDAY:  return "Lectures";
    case THURSDAY:   return "Prepare Labs";
    case FRIDAY:     return "GTAV";
    case SATURDAY:
    case SUNDAY:     return "Strictly Come Dancing";
    }

    return "Impossible!";

}
```


1. Each case doesn't need to be prefixed with `Day`.
2. Even if the `switch` is exhaustive, Java will still require you to put a default case in or an extra return statement.

Giving enumerated types behaviour

Because enums are objects too...

```
public enum EnhancedDay {
    SUNDAY("Strictly Come Dancing"),
    MONDAY("Lectures"),
    TUESDAY("Minecraft"),
    WEDNESDAY("Lectures"),
    THURSDAY("Prepare Labs"),
    FRIDAY("GTAV"),
    SATURDAY("Strictly Come Dancing");

    private final String whatToDo;

    private EnhancedDay(String whatToDo) {
        this.whatToDo = whatToDo;
    }

    public String whatToDo() {
        return whatToDo;
    }
}
```


1. Within the definition of an enum you can create fields (`private final String whatToDo;`), constructors and methods.
2. If you declare a constructor, it *must* be `private`, i.e. only the enum constants can call it, not anyone else.
3. The constructor arguments are passed between `( )`s after the enum constant's name.

Using an enumerated type's behaviour

```
public class EnhancedDayExample {

    public static void main(String[] args) {

        EnhancedDay today = EnhancedDay.MONDAY;
        String activity = today.whatToDo();

        System.out.println(activity);

    }

}
```