Programming II
Introduction to Imperative Programming

Susan Eisenbach
susan@imperial.ac.uk

With thanks to Tristan Allwood and Nicolai Stawinoga

120.2

Autumn Term - 2017

Textbooks - none required

For beginner programmers:

- **Java Software Solutions: Foundations of Program Design**, John Lewis and William Loftus, Pearson Education, 2012.

For experienced programmers:

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

We use Google's programming style. You can learn about it at https://google.github.io/styleguide/javaguide.html.

Assessment - two possibilities

For experienced imperative language programmers

- next Friday, 17 November, 14.00 – 16.00
- sign up at: https://doodle.com/poll/7qb65tf3hhu3ihmq before Thursday
- get over 80% and you get full marks for the assessment for imperative Java
- get less and there are important things that you still need to learn
- aimed at you
  - advanced programming lectures
  - extending the optional parts on the ppt exercises (get the most out of your UTA)

Main test

- Thursday, 14 December, 14.00 – 17.00
- for all students who have not got over 80% on the 17 November test

Details about the assessment process will be on Piazza.

Declarative vs Imperative Languages

Haskell

- declarative language
- basic unit - the expression
- Say 'what you want' and the computer works out how to do it.
- similar to mathematical functions and "high level" descriptions of algorithms
- *horses for courses*

Java

- imperative language
- basic unit - the statement
- Say 'what the computer should do'.
- similar to a cooking recipe / step by step instructions
- An imperative program executes a *sequence* of instructions that change the program's *state* to reach a desired result.

From Haskell functions to Java methods

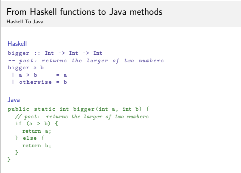
Haskell To Java

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;
  }
}
```



From Haskell functions to Java 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 are returned using the keyword `return`
7. Statements (e.g. `return`), must end in a ;
8. Single line comments start with //

From Functions To Methods

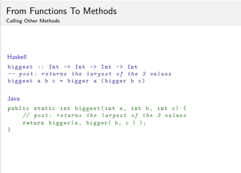
Calling Other Methods

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 ) );
}
```



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));
  }
}
```

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));
}
}

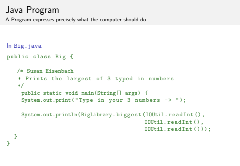
1. Class name matches file name. Java source files end in .java
2. Class is public so it can be used by other Libraries and Programs
3. Methods are public so they can be called by other Libraries and Programs

Java Program

A Program expresses precisely what the computer should do

In Big.java

```
public class Big {

   /* Susan Eisenbach
   * Prints the largest of 3 typed in numbers
   */
    public static void main(String[] args) {
    System.out.print("Type in your 3 numbers -> ");

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



Java Program
A Program expresses precisely what the computer should do
In Big.java
public class Big {
/* Susan Eisenbach
* Prints the largest of 3 typed in numbers
*/
public static void main(String[] args) {
System.out.print("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. Acknowledge it is your code
5. You can print out using System.out.print(...) and System.out.println(...)
6. To use static methods from other classes you need to prefix the method with the name of the class where they were defined.

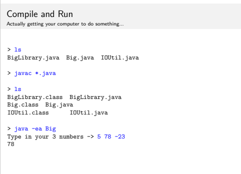
From your code to running code

- Integrated development environments (IDE) make developing code easier.
- They help with all sorts of thing such as helping you to remember what the parameters are for a method you are calling to debugging your code.
- We have chosen Intellij IDEA amongst the several available IDEs because it provides the best support.
- However, it does hide much of the process and computer scientists should know what is actually going on.
- You should be able to write and execute Java code without having Intellij around.

Compile and Run

Actually getting your computer to do something...

```
> ls
BigLibrary.java  Big.java  IOUtil.java

> javac *.java

> ls
BigLibrary.class  BigLibrary.java
Big.class  Big.java
IOUtil.class      IOUtil.java

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



Compile and Run
Actually getting your computer to do something...

```
> ls
BigLibrary.java  Big.java  IOUtil.java

> javac *.java

> ls
BigLibrary.class  BigLibrary.java
Big.class  Big.java
IOUtil.class      IOUtil.java

> java -ea Big
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.

Exercise 1

- Create a library `Util.java` with a method `absolute` which takes an `int` and returns the absolute value of the `int`.
- Create a program, `Absolute.java` which reads in an integer and prints out the absolute value of that number.

Please make sure you use Google style for your Java programs.

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

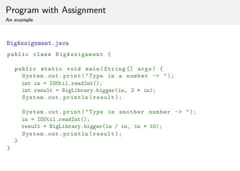
An example

BigAssignment.java

```
public class BigAssignment {

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

    System.out.print("Type in another number -> ");
    in = IOUtil.readInt();
    result = BigLibrary.bigger(in / in, in * 10);
    System.out.println(result);
  }
}
```

2017-12-07

Programming II Introduction to Imperative Programming
└Introduction
└Program with Assignment



Program with Assignment
An example

BigAssignment.java

```
public class BigAssignment {

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

    System.out.print("Type in another number -> ");
    in = IOUtil.readInt();
    result = BigLibrary.bigger(in / in, in * 10);
    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

Exercise 2

- In `Util.java` write a method `sumOrProduct` that takes two `int` arguments and creates two variables containing the sum and the product of the arguments. The method should return the largest number of the two arguments, their sum and their product. (Make use of `BigLibrary` if it helps).
- Write a program `SOP.java` that asks the user for one number and prints out the result of `sumOrProduct` of that number as both arguments.

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.
- The `main` method is special as it is the code that Java executes.
- The signature of main is `public static void main(String[] args)`.
- 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

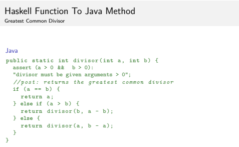
```
divisor :: Int -> Int -> Int
-- pre: the arguments are both > 0
-- post: returns the greatest common divisor
divisor a b | a == b = a
            | a > b  = divisor b (a - b)
            | a < b  = divisor a (b - a)
```

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);
  }
}
```



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.

Exercise 3

Write the following as assert statements

- */* pre: n is positive */*
- */* pre: a is not 0 */*
- */* pre: x and y are different */*
- */* pre: calling foo(n) returns false */*

-
  ```
  /* pre: n is false and m is true,
          or n is true and m is false,
          or a > b */
  ```

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

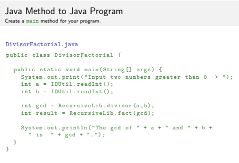
```
public class RecursiveLib {

    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.print("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 + ".");
  }
}
```



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.print("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 + ".");
  }
}
```

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

Exercise 4

Simple Haskell Fibonacci

```
simpleFibonacci :: Int -> Int
simpleFibonacci 0 = 0
simpleFibonacci 1 = 1
simpleFibonacci 2 = 1
simpleFibonacci n = simpleFibonacci (n-1)
                  + simpleFibonacci (n-2)
```

- Translate the above Haskell fibonacci function into a Java method.
- Write a Java program that asks the user to input a number and prints out `The nth fibonacci number is ...`

Helper Functions to Helper Methods

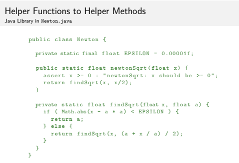
Haskell

```
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);
    }
  }
}
```



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 the parameter 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

Exercise 5

Assume the Util.java library below. What would the Main.java programs do on the following slides? For each, do they compile and why? If they compile and are run, what do they print out?

Util.java

```
public class Util {

  public static double twice(double x) {
    return add(x,x);
  }

  private static double add(double x, double y) {
    return x + y;
  }

}
```

Main1.java

```
public class Main1 {
  public static void main(String[] args) {
    System.out.println(Util.twice(3));
  }
}
```

Main2.java

```
public class Main2 {
  public static void main(String[] args) {
    System.out.println(Util.add(4,3));
  }
}
```

Main3.java

```
public class Main3 {
 private static final double MAGIC = 0.2;

 public static void main(String[] args) {
  System.out.println(Util.twice(MAGIC));
 }
}
```

Main4.java

```
public class Main4 {
 private static void main(String[] args) {
  System.out.println(Math.abs(Util.twice(0.2)));
 }
}
```

Main5.java

```
public class Main5 {
 private static final double MAGIC = -0.2;
 public static void main(String[] args) {
   double addResult = add(MAGIC, Math.abs(MAGIC));
   System.out.println(Util.twice(addResult));
  }

 private static double add(double x, double y) {
   return x + y;
   }
 }
```

Methods Summary

- Haskell has *functions* that return values.
- 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 starting method of a program must have the signature: `public static void main(String[] args)`.
- Java methods can be recursive. It is not wise to make `main` 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 get 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

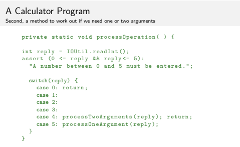
First, a method to present a menu to the user and to get their response

```
private static void presentMenu() {
  // post: Menu appears on the screen.
  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");
 }
```

A Calculator Program

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

```
private static void processOperation( ) {

int reply = IOUtil.readInt();
assert (0 <= reply && reply<= 5):
  "A number between 0 and 5 must be entered.";

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



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

```
private static void processOperation( ) {

int reply = IOUtil.readInt();
assert (0 <= reply && reply<= 5):
  "A number between 0 and 5 must be entered.";

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

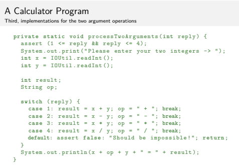
1. Introducing the `switch` statement
2. An expression of `int`, `byte`, `short`, `char` or `String` type*
3. `case value:` which case to jump to

A Calculator Program

Third, implementations for the two argument operations

```
private static void processTwoArguments(int reply) {
  assert (1 <= reply && reply <= 4);
  System.out.print("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);
}
```

└─A Calculator Program



A Calculator Program

Third, implementations for the two argument operations

```
private static void processTwoArguments(int reply) {
  assert (1 <= reply && reply <= 4);
  System.out.print("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 and is optional.

A Calculator Program

Fourth and Fifth, One argument functions and a main method

```
public class Calculator {

  public static void main(String[] args) {
    presentMenu();
    processOperation();
  }

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

  private static void processOperation( ) {
    ... 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

Exercise 6

1. What does `switchy` return when passed the arguments 0, 1, 2, 3, 4 and 5?

```
public static String switchy(int x) {
  String result = "???";

  switch (x) {
    case 0: return "A";
    case 2: result "B";
    case 1:
    case 3: result = "C"; break;
    case 4: result = "D";
    default: return "DEF" + result;
  }
  return result;

}
```

2. Complete the `Calculator` program.

Back to Recursion

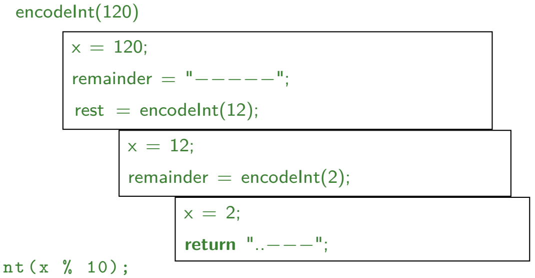
Important things to remember:

- 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 Code Encoder – Another Example

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.

Loops

The `while` loop

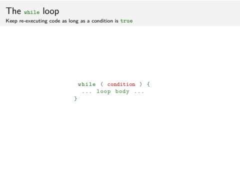
- A loop iterates or 'loops' over a block of code, executing it repeatedly.
- When you need repetition, but you don't know how many times the repetition will occur you can use recursion, or a *while loop*.
- Another type of loop, the `for` loop, is usually used when you know up front how many iterations are wanted. For example, to traverse all the elements of a list whose length you know.
- 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 your loops will terminate.
- Unlike recursion, a non-terminating generalised loop does not cause *stack overflow*, as this is caused by having too many unfinished method calls.

The `while` loop

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

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

Programming II Introduction to Imperative Programming
└─Loops
  └─The `while` loop



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

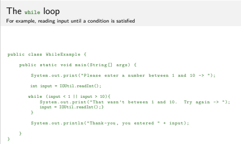
```
 while ( condition ) {
  ... loop body ...
}
```

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.print("Please enter a number between 1 and 10 -> ");

        int input = IOUtil.readInt();

       while (input < 1 || input > 10){
           System.out.print("That wasn't between 1 and 10.  Try again -> ");
           input = IOUtil.readInt();}
        }

        System.out.println("Thank-you, you entered " + input);

    }
}
```



The while loop
For example, reading input until a condition is satisfied

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 ...
      }
    }
  }
}
```

Exercise 7

What will these while loops print out?

For each while loop below, will it compile, and if it does, what does it print when executed?

Meep

```
while(true) {
  System.out.println("Meep!");
}
```

Strung

```
String s = "";
while (s != s + 0) {
  System.out.println(s);
}
```

Exercise 8

Diagonal

```
int i = 0;
int j = 10;

while (i < j) {
  i = i + 1;
  j = j - 1;
  System.out.println(i + j);
}
```

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.

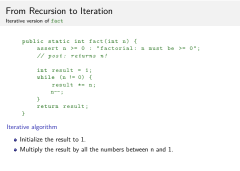
From Recursion to Iteration

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.



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 recursive 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 - Another Example

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 - Another Example

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.

Exercise 9

Remember newtonSqrt? Write it iteratively...

```
public class Newton {

  private static final float EPSILON = 0.00001 f;

  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);
    }
  }
}
```

Other kinds of loops

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

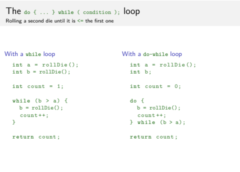
Rolling a second die until it is `<=` the first one

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;
```



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

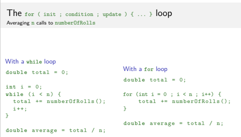
Averaging `n` calls to `numberOfRolls`

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;
```



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
4. `i++` increments `i` by 1. It updates the variable - its counterpart, `++i`, updates first, then returns the updated value.

Exercise 10

- Remember the `fact` function? Re-write the function twice, using a `for` loop and a `do-while` loop instead.
- Write a function `public static void rectangle()` that prompts the user for a width and a height and draws a rectangle of stars. For example:

```
Please enter a width and a height ->  7 3
*******
*******
*******
```

You will need to use two *nested* loops. The outer loop will print out the rows, whereas the inner loop will print out each row.

`break` and `continue`

Jumping around or out of loops

- There might be times when you want to leave a loop early.
  - e.g. you are iterating through a list searching for a value, 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 a list.
- 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. You have seen `break` in switch statements.
  - `continue`: which will jump to the next iteration of the loop.

Consider a predicate `isPrime`

What if you don't want multiple exits from this method?

```
static boolean isPrime(int n) {
  double top = Math.sqrt(n);
  for (int i = 2; i <= top; i++) {
    if ((n % i) == 0) {
      return false;
    }
  }
  return true;
}
```

Another version of the predicate isPrime

You could use break to terminate the loop when you know the number is not prime.

```
static boolean isPrime2(int n) {
  boolean result = true;
  double top = Math.sqrt(n);
  for (int i = 2; i <= top; i++) {
    if ((n % i) == 0) {
      result = false;
      break;  //stops needlessly looping
    }
  }
  return result;
}
```

Yet another version of the predicate isPrime

You could use continue to jump to the next iteration of the loop.

```
 static boolean isPrime3(int n) {
    double top = Math.sqrt(n);
    for (int i = 2; i <= top; i++) {
      if (i % 2 == 0) {
        continue; //only check odd numbers
      }
      if ((n % i) == 0) {
        return false;
      }
    }
    return true;
  }
}
  }
```

break and continue

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

Exercise 11

- First use a while loop to solve this problem.
- Rewrite using a for loop for the attempts to roll n 6's in a row. If we get to the end of the for loop then we are done.
- However, if we don't roll a 6 within the for loop, then we have to try again.
- We may use continue to try the next iteration of a loop and break when we wish to terminate loop.
- We may use return to act like a break, but to also leave the method entirely.
- How would you change your code to just keep running?

Summary

- There are many different ways of performing repeated execution in Java.
- Recursion and while loops are the most general forms of repetition.
- Recursive methods can be written using a loop. 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.

Arrays

Array?

What?

- Space for many items of the same type.
- Each element can be accessed 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 same operations on lots of individual items.

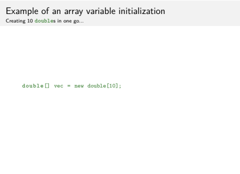
Differences with Haskell Lists

- Every element of an array can be accessed in constant time.
- Arrays are of fixed size (they cannot grow like lists).
- You can't pattern match on arrays.

Example of an array variable initialization

Creating 10 `doubles` in one go…

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



Example of an array variable initialization
Creating 10 doubles in one go…

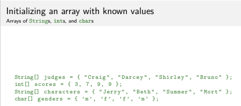
double[] vec = new double[10];

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 take space for 10 double values, by using `double[10]`
6. The 10 new double values will all default to value 0.0
7. The number of elements (10) can be any expression of type `int`

Initializing an array with known values

Arrays of `String`s, `int`s, and `char`s

```
String[] judges = { "Craig", "Darcey", "Shirley", "Bruno" };
int[] scores = { 3, 7, 9, 9 };
String[] characters = { "Jerry", "Beth", "Summer", "Mort" };
char[] genders = { ’m’, ’f’, ’f’, ’m’ };
```



Initializing an array with known values
Arrays of Strings, ints, and chars

```
String[] judges = { "Craig", "Darcey", "Shirley", "Bruno" };
int[] scores = { 3, 7, 9, 9 };
String[] characters = { "Jerry", "Beth", "Summer", "Mort" };
char[] genders = { ’m’, ’f’, ’f’, ’m’ };
```

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

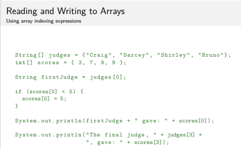
Exercise 12

- Write a statement to create a variable called `flags` that is an array with five `false` values;
- Write a statement to create a variable `empty` which points to an array of length 0 of ints.
- Write a class `AndOr` which has two methods, `and` and `or` which take an array of booleans and returns true if (respectively) all or any of the elements in the array are `true`. You could use a `break` to stop looping if you know the result before all of the elements have been processed.

Reading and Writing to Arrays

Using array indexing expressions

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



Reading and Writing to Arrays
Using array indexing expressions

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

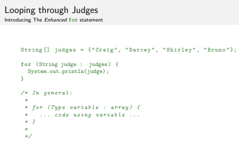
Iteration...

- 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 loop, using the loop variable to access each element of the array at `array[i]`.
- Alternatively, an *enhanced* `for` loop can be used.

Looping through Judges

Introducing The *Enhanced* `for` statement

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

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

/* In general:
 *
 * for (Type variable : array) {
 *    ... code using variable ...
 * }
 *
 */
```



Looping through Judges

Introducing The *Enhanced* for statement

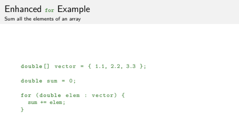
```
String[] judges = {"Craig", "Darcey", "Shirley", "Bruno"};

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

/* In general:
 *
 * for (Type variable : array) {
 *    ... code using variable ...
 * }
 *
 */
```

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



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 %=

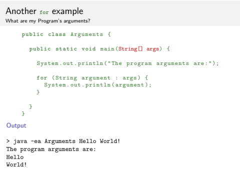
Another for example

What are 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!
```



Another for example

What are 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

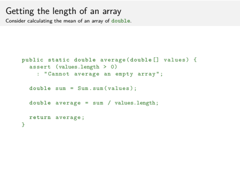
```
> java -ea Arguments Hello World!
The program arguments are:
Hello
World!
```

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

Consider calculating the mean of an array of `double`.

```
public static double average(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

Consider calculating the mean of an array of double.

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

  double sum = Sum.sum(values);

  double average = sum / values.length;

  return average;
}

1. Every array knows its own size.
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

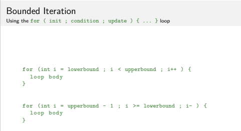
Using the `for ( init ; condition ; update ) { ... }` loop

- 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

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

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


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

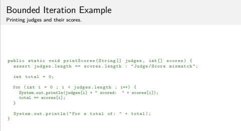

Bounded Iteration
Using the for (init ; condition ; update) { ... } loop
for (int i = lowerbound ; i < upperbound ; i++) {
loop body
}
for (int i = upperbound - 1 ; i >= lowerbound ; i-) {
loop body
}

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. Remember: 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);
}
```


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, so you cannot refer to it.

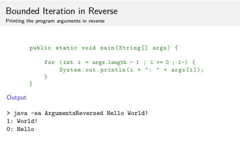
Bounded Iteration in Reverse

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
```



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.

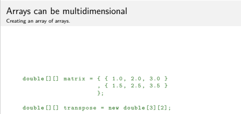
Exercise 13

Write a method `fibArray`, which, given an `int n` produces an array of length `n` filled with the first `n` fibonacci numbers.

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];
```



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];
```

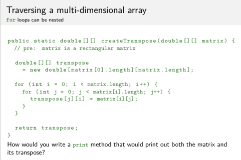
1. matrix is an array of length 2, where each element is an array of doubles of length 3
2. The inner arrays of length 3 are written within { }s, and the two inner arrays are themselves nested within { }s
3. transpose is an array of length 3, where each element is an array of doubles of length 2

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;
}
```

How would you write a print method that would print out both the matrix and its transpose?



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;
}
```

How would you write a print method that would print out both the matrix and its 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]

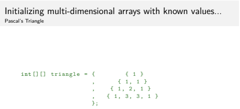
Exercise 14

Write a method `sumAll` that takes a three-dimensional array of `int` as an argument and returns the sum of all the numbers in the array.

Initializing multi-dimensional arrays with known values...

Pascal's Triangle

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



Initializing multi-dimensional arrays with known values...
Pascal's Triangle

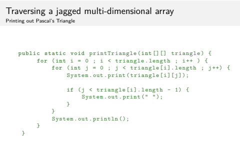
```
int[][] triangle = {         { 1 }
                   ,       { 1, 1 }
                   ,     { 1, 2, 1 }
                   ,   { 1, 3, 3, 1 }
                   };
```

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 maths 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();
     }
 }
```



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 put spaces between the elements, but not at the end, we use an `if` check to see if `j` is before its last index
4. Challenge: how would you print out the triangle centered and not left aligned?

Exercise 15

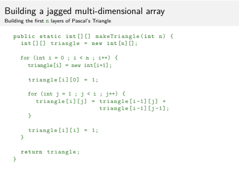
Rewrite `printTriangle` so that it prints out as an isosceles rather than a right triangle.

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;
}
```

└Building a jagged multi-dimensional array



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", "Mario", "Candy Crush"};
```

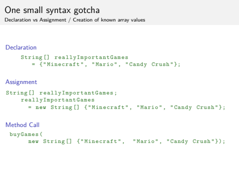
Assignment

```
String[] reallyImportantGames;
    reallyImportantGames
      = new String[] {"Minecraft", "Mario", "Candy Crush"};
```

Method Call

```
 buyGames(
      new String[] {"Minecraft",  "Mario", "Candy Crush"});
```

└One small syntax gotcha



One small syntax gotcha
Declaration vs Assignment / Creation of known array values
Declaration
String[] reallyImportantGames
= {"Minecraft", "Mario", "Candy Crush"};
Assignment
String[] reallyImportantGames;
reallyImportantGames
= new String[] {"Minecraft", "Mario", "Candy Crush"};
Method Call
buyGames(
new String[] {"Minecraft", "Mario", "Candy Crush"});

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 to be allocated for them. This is either done implicitly with values given for all their elements, or explicitly using `new` to take space in the heap.
- 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.

Exercise 16

What do the stack and heap look like when execution reaches each line of the following?

```
String[] dancers = { "Susan", "Konstantinos", "Tony" };
int[] scores = new int[3];
// <here>
scores[0] = 1;
scores[2] = 2;
// <here>
int[] scores2 = { 2,3,4 };
scores = scores2;
// <here>
scores2[2] = 10000;
// <here>
```

Exercise 17

Hand execute the following code in the presence of method `m`, below. Draw the state of the stack and the heap before and after the call to `m(a)`.

```
int[] a = {1, 2, 3};
m(a);
```

Method m

```
public static void m(int[] xs) {
  int[] ys = xs;
  ys[0] = xs[1];
  xs = null;
  ys = null;
}
```

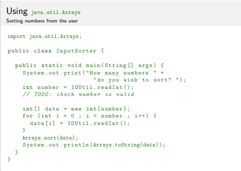
Array Utility Methods

- 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.
- For the complete API see https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html.
- 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.print("How many numbers " +
                     "do you wish to sort? ");
    int number = IOUtil.readInt();
    // TODO: check number is valid

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



Using java.util.Arrays
Sorting numbers from the user

```
import java.util.Arrays;

public class InputSorter {

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

    int[] data = new int[number];
    for (int i = 0 ; i < number ; 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;
}
```

Exercise 18

Draw the stack and the heap after each assignment.

Reverse

Algorithm

- Iterate through the first half of 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);
  }
}
```

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

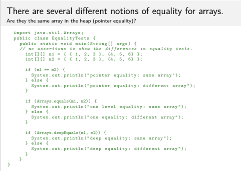
Don’t forget to test!

```
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);

  }
}
```

There are several different notions of equality for arrays.

Are they the same array in the heap (pointer equality)?

```
  import java.util.Arrays;
  public class EqualityTests {
    public static void main(String[] args) {
    // no assertions to show the differences in equality tests.
      int[][] m1 = { { 1, 2, 3 }, {4, 5, 6} };
      int[][] m2 = { { 1, 2, 3 }, {4, 5, 6} };

      if (m1 == m2) {
        System.out.println("pointer equality: same array");
      } else {
        System.out.println("pointer equality: different array");
      }

      if (Arrays.equals(m1, m2)) {
        System.out.println("one level equality: same array");
      } else {
        System.out.println("one equality: different array");
      }

      if (Arrays.deepEquals(m1, m2)) {
        System.out.println("deep equality: same array");
      } else {
        System.out.println("deep equality: different array");
      }
    }
}
```



There are several different notions of equality for arrays.
Are they the same array in the heap (pointer equality)?

1. Do they have the same elements (shallow structural equality)?
2. For nested arrays, are the deeply nested values the same (deep structural equality)?
3. Pointer equality can be tested with `array1 == array2`.
4. Shallow structural equality can be tested with `Arrays.equals(array1, array2)`
5. Deep structural equality can be tested (on 2-or-higher dimensional arrays) with `Arrays.deepEquals(array1, array2)`

Fisher-Yates Shuffle - Another Example

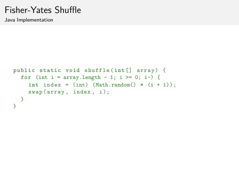
Algorithm

- 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);
  }
}
```



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*. To round rather than round down add 0.5 before rounding.

Exercise 19

Write a method `rotate` that is given an `int[]` and an `int n`, and that rotates the elements of the array `n` steps to the right. For example:

```
int[] xs = { 10, 20, 30, 40 };
rotate(xs, 3);
assert Arrays.equals(new int[] { 20, 30, 40, 10 }, xs);
```

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, there is an API `java.util.Arrays` with a very large number of utility methods. The 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.

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
- 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*.

Classes and Objects

Things that have State, Behaviour and Identity

- A class is a type (for example, `class String`).
- An object is an *instance* of a class (for example, the actual String `"Hello World"`).
- There can be many objects of the same type
- Objects can have fields and methods, which capture and define their *state*, *behaviour*, and *identity*.

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

```
public class Counter {

    private int count = 0;

    public void tick() {
        count++;
    }

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

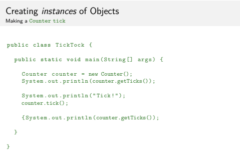
1. `public class Counter` must live in a file called `Counter.java`
2. `private int count` is an *instance field* of the class. It is declared within the class but not inside any method.
3. Each `Counter` instance that is created will get its own `count` value that will store its value as long as the 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



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.

Exercise 20

Create a variation of the `Counter` class that has a method `hasBeenTicked` which returns `true` if `tick` has been called.

- One way is to use an extra `boolean` field.
- Another way it to look at the value of the existing `count` field.

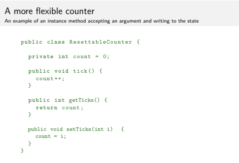
A more flexible counter

- 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;
  }
}
```



A more flexible counter

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

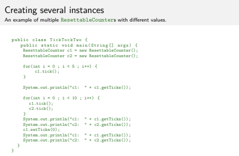
```
public class ResettableCounter {

  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());
  }
}
```



Creating several instances

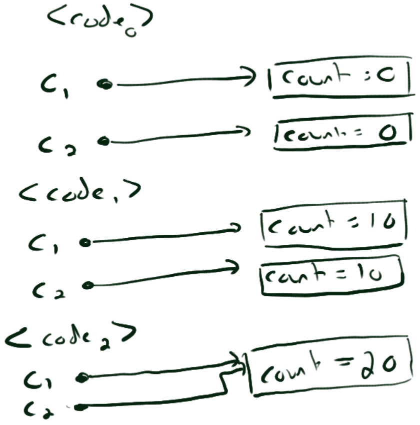
An example of multiple ResettableCounters with different values.

```
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;?

Exercise 21

Given the following snapshots of the stack and heap, what could code0, code1 and code2 be?



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`.

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 `concat`.

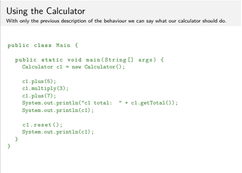
Behaviour

- Methods `plus` and `multiply` that accept an int and update the `total` and `concat` fields accordingly.
- A method `getTotal` to return the current `total`.
- A method `reset` to reset the current `total` and `concat` 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);
  }
}
```



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);
  }
}
```

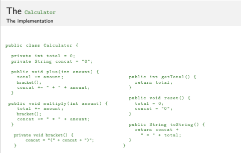
1. The class that describes our calculator will be called `Calculator`.
2. We should be able to call `plus` and `multiply` methods upon it, and it should build up the correct total.
3. In Java, if a class describes a method with the signature `public String toString()` then `println` will use that instead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

The Calculator

The implementation

```
public class Calculator {

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

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

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

   private void bracket() {
        concat = "(" + concat + ")";
   }
```

```
  public int getTotal() {
    return total;
  }

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

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



1. Both plus and multiply need to put brackets around concat, 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

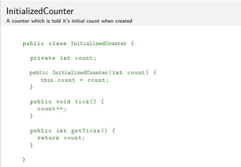
Executing code when creating an instance

- 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;
  }

}
```



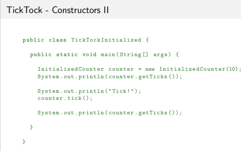
InitializedCounter
A counter which is told it's initial count when created

```
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. Constructors use the same name as the class itself.
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());

  }

}
```



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.

Exercise 22

Write a class `IntHolder` that would make the following `assert` statements pass.

```
public class Main {
  public static void main(String[] args) {
    IntHolder ih = new IntHolder(10);
    assert ih.size() == 10;
    ih.put(0, 3);
    assert ih.get(0) == 3;
    ih.fill(6);
    assert ih.get(4) == 6;
  }
}
```

Exercise 23

What does the following program print?

```
public class Main {
  public static void main(String[] args) {
    InitializedCounter a = new InitializedCounter(10);
    InitializedCounter b = new InitializedCounter(20);

    a = b;
    a.tick();
    System.out.println("The counters have ticked " +
      a.getTicks() + " and " + b.getTicks() + " times");
  }
}
```

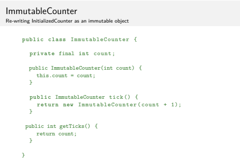
Mutable and immutable objects

- So far, we have seen *mutable* objects – their state can change, and when it does, every variable using the object will see the change.
- Instead of changing the state of an object, a method can return a new object with the desired new state, while the state of the current object remains unchanged.
- In this approach, we can declare all the object's fields as `final`, to guarantee that its state will never change. This is called an *immutable* object.
- The choice between the two is often a matter of taste, and depends on the situation. However, objects that primarily carry 'data' are often immutable (e.g. `String`s).
- Haskell data structures are immutable.

ImmutableCounter

Re-writing InitializedCounter as an immutable object

```
public class ImmutableCounter {

  private final int count;

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

  public ImmutableCounter tick() {
    return new ImmutableCounter(count + 1);
  }

 public int getTicks() {
    return count;
  }

}
```



ImmutableCounter
Re-writing InitializedCounter as an immutable object

```
public class ImmutableCounter {

  private final int count;

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

  public ImmutableCounter tick() {
    return new ImmutableCounter(count + 1);
  }

 public int getTicks() {
    return count;
  }

}
```

1. This example shows how `InitializedCounter` can be implemented as an immutable object
2. The constructor and `getTicks()` method are as before.
3. The `private int count` field has been made `final`, making this class an immutable object.
4. `public void tick()` has been changed to `public ImmutableCounter tick()`, and instead of updating the state returns a new object with the desired state.

Exercise 24

What does the following program print?

```
public class Main {
  public static void main(String[] args) {
    ImmutableCounter a = new ImmutableCounter(10);
    ImmutableCounter b = new ImmutableCounter(20);

    a = b;
    a = a.tick(); // assign back to a
    System.out.println("The counters have ticked " +
      a.getTicks() + " and " + b.getTicks() + " times");
  }
}
```

More on `final`

Declaring something as `final` means its value cannot be changed after the initial assignment. The following declarations can be made final:

- Variables inside a function.
- Fields of an object, such that their value cannot change throughout the lifetime of the object. They must be initialised with an assignment *or* inside a constructor method.
- Method arguments, meaning their value cannot be changed/re-assigned inside the method body. Note, that Java by default allows re-assigning to a method argument, though it is commonly considered bad practice.

The DragonsBreath Dungeons – Another Example

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 to much 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.

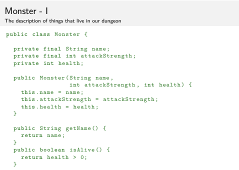
State

- 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;
  }
```



Monster - I
The description of things that live in our dungeon

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 increased 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.

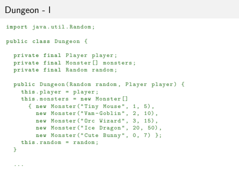
Dungeon

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

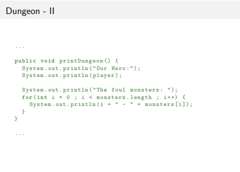
```
import java.util.Random;

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]);
  }
}

...
```

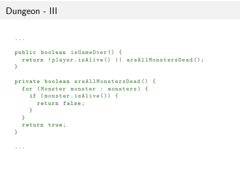


Dungeon - II

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;
}

...
```



Dungeon - III

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;
}

...
```

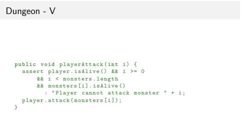
Dungeon - IV

```
/* 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

```
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]);
}
```

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* (e.g. hidden).
- 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`.
- That is, 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 {

 /* Monster behaviour from lecture slides
  * 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
  */
  public static void main(String[] args) {
    System.out.println("Running tests...");

    canRememberName();
    canBeAliveOrNot();
    canBeDamaged();
    attacksPlayers();
    hasReadableStringRepresentation();

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

...
}
```

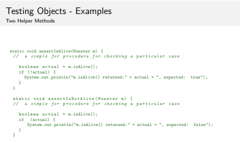
Testing Objects - Examples
Testing Monster

1. We begin with a small program that contains some tests, at least one for each behaviour.
2. The different tests have names that describe behaviour monsters can exhibit. So you would say 'A monster attacks players.' hence the `attack` method could be tested by a method called `attackPlayers`.
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

```
static void assertIsAlive(Monster m) {
 //  a simple for procedure for checking a particular case

   boolean actual = m.isAlive();
   if (!actual) {
     System.out.println("m.isAlive() returned:" + actual + ", expected:  true");
   }
 }

 static void assertIsNotAlive(Monster m) {
 //  a simple for procedure for checking a particular case

   boolean actual = m.isAlive();
   if  (actual) {
      System.out.println("m.isAlive() returned:" + actual + ", expected:  false");
   }
 }
```



1. These methods only print out if results are unexpected.
2. 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.
3. Note that there is no modifier. This means that this method is available in the entire `package` or *package visible*. Anyone can see `public` methods and only within a class are `private` methods visible. For now, a package can be thought of as all the files you can see at once in the IJ ide.

Testing Objects - Examples

Testing `Monster`'s `takeDamage` method

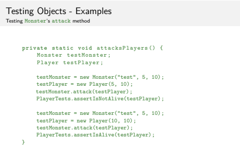
```
// takeDamage tests

  static void canBeDamaged() {
    Monster testMonster;

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

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

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

1. Here 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 attacksPlayers() {
    Monster testMonster;
    Player testPlayer;

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

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



Testing Objects - Examples
Testing Monster's attack method

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

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

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

1. The attack method should change the health status of the testPlayer. To see its effect, we check whether the player is alive or not after the attack.
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 should 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.

Summary

- To test static methods one enumerates simple test cases, (so mapping inputs to outputs).
- Objects have identity, state and behaviour. We need to test the behaviour - that it does what it is supposed to do.
- For each object produce a set of tests that see whether the object behaves properly or not. You need at least one test for each different behaviour.
- To make it easier to see what has gone wrong only print out when a test shows that an object is *not* behaving properly.
- Accumulate your tests for an object. Do not write a test, see that the behaviour is correct and then throw it away. Always run the tests you have written every time you test your code.

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

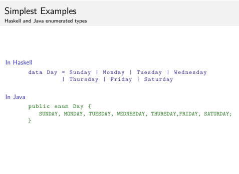
Haskell and Java enumerated types

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 (or Java 5).

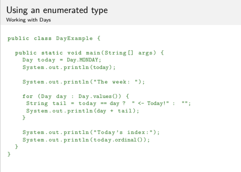
Other Examples

- 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());
  }
}
```



Using an enumerated type
Working with Days

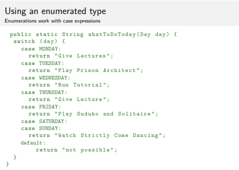
```
public class DayExample {

  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()
6. Setting a single variable to one of two states based on a single condition is such a common use of if-else that a shortcut has been devised for it, the conditional operator, ?:.

Using an enumerated type

Enumerations work with case expressions

```
 public static String whatToDoToday(Day day) {
  switch (day) {
    case MONDAY:
      return "Give Lectures";
    case TUESDAY:
      return "Play Prison Architect";
    case WEDNESDAY:
      return "Run Tutorial";
    case THURSDAY:
      return "Give Lecture";
    case FRIDAY:
      return "Play Suduko and Solitaire";
    case SATURDAY:
    case SUNDAY:
      return "Watch Strictly Come Dancing";
    default:
        return "not possible";
  }
}
```


Using an enumerated type
Enumerations work with case expressions

```
public static String whatToDoToday(Day day) {
 switch (day) {
   case MONDAY:
     return "Give Lectures";
   case TUESDAY:
     return "Play Prison Architect";
   case WEDNESDAY:
     return "Run Tutorial";
   case THURSDAY:
     return "Give Lecture";
   case FRIDAY:
     return "Play Suduko and Solitaire";
   case SATURDAY:
   case SUNDAY:
     return "Watch Strictly Come Dancing";
   default:
       return "not possible";
 }
}
```

1. The cases aren't 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.

Exercise 25

Write a static method `isWeekDay` that takes a `Day` arguments and returns `true` if the given day is a weekday (`Monday` – `Friday`).

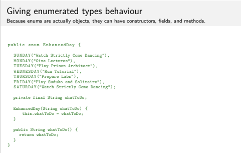
- One way is to use a `switch` statement.
- Another is to use the `.ordinal()` method.

Giving enumerated types behaviour

Because enums are actually objects, they can have constructors, fields, and methods.

```
public enum EnhancedDay {

  SUNDAY("Watch Strictly Come Dancing"),
  MONDAY("Give Lectures"),
  TUESDAY("Play Prison Architect"),
  WEDNESDAY("Run Tutorial"),
  THURSDAY("Prepare Labs"),
  FRIDAY("Play Suduko and Solitaire"),
  SATURDAY("Watch Strictly Come Dancing");

  private final String whatToDo;

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

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

└─Giving enumerated types behaviour



Giving enumerated types behaviour

Because enums are actually objects, they can have constructors, fields, and methods.

```
public enum EnhancedDay {

  SUNDAY("Watch Strictly Come Dancing"),
  MONDAY("Give Lectures"),
  TUESDAY("Play Prison Architect"),
  WEDNESDAY("Run Tutorial"),
  THURSDAY("Prepare Labs"),
  FRIDAY("Play Suduko and Solitaire"),
  SATURDAY("Watch Strictly Come Dancing");

  private final String whatToDo;

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

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

1. The constructor arguments are written between `( )`s after the enum constant's name, to be passed to the constructor.
2. If you declare a constructor, it is `private`, and you cannot write program code to call it. It doesn't need to be explicitly declared as private. It is executed automatically.
3. Within the definition of an enum you can also create fields and methods.

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);

    }

}
```

Exercise 26

Extend `EnhancedDay` with an `isWeekDay` instance method.

Summary

- Enums are lists of constants (or `static final`). Use an enum when you need a small predefined list of values.
- Using enums appropriately both makes your program more readable (hence less error prone) and it may run faster.
- Enums can contain constructors, methods, fields, and constant class bodies.
- `MyEnum.values()` returns an array containing the `MyEnum` values.
- `anEnum.ordinal()` returns the index of `anEnum` in `MyEnum.values()`.
- Enums can be compared with `==`, `.equals()`, and case statements. Even if there is a case for every value in an enumerated type you must either have a `default` or after the swtich statement a separate return in case there is no match (which would be impossible).

Bits and Pieces – Rounding off your Java, ready for next term

Method and constructor overloading

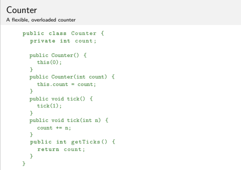
Multiple definitions of the same function

- In a single program a function can be redefined with the same name and same return type, but with strictly different arguments. This is called *overloading*.
- For example, the `System.out.println()` method exists multiple times taking different arguments (or none): `println()`, `println(3)`, `println(false)`.
- Overloading is useful to enable methods to deal with different kinds of arguments, and also to allow the specification of default values.
- Constructors can also be overloaded. This enables the provider of a class to have a default initialisation without parameters and another initialisation with parameters.

Counter

A flexible, overloaded counter

```
public class Counter {
  private int count;

  public Counter() {
    this(0);
  }
  public Counter(int count) {
    this.count = count;
  }
  public void tick() {
    tick(1);
  }
  public void tick(int n) {
    count += n;
  }
  public int getTicks() {
    return count;
  }
}
```



Counter
A flexible, overloaded counter

```
public class Counter {
  private int count;

  public Counter() {
    this(0);
  }
  public Counter(int count) {
    this.count = count;
  }
  public void tick() {
    tick(1);
  }
  public void tick(int n) {
    count += n;
  }
  public int getTicks() {
    return count;
  }
}
```

1. Note the two constructor methods.
2. Constructor methods that can be called with no arguments are called *default constructors*.
3. The default constructor uses `this` to call another *overloaded* constructor method, passing a default value of zero.
4. The `tick` method can tick a single time, or `n` times.
5. We could implement the no-argument version of `tick` by writing `count++`, but instead we chose to call the overloaded version with a default argument of `1`

this

or self-referencing

- `this(...)` references another constructor method.
- `this` followed by a `.` allows us to reference fields and functions of the current object
- `this` can also be used to pass a self-reference to another object. For instance, in a tree-like structure:

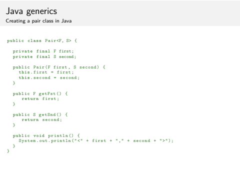
```
Node n = new Node();
n.setParent(this);
```

Generics

Java generics

Creating a pair class in Java

```
public class Pair<F, S> {

  private final F first;
  private final S second;

  public Pair(F first, S second) {
    this.first = first;
    this.second = second;
  }

  public F getFst() {
     return first;
  }

  public S getSnd() {
     return second;
  }

  public void println() {
    System.out.println("<" + first + "," + second + ">");
  }
}
```



Java generics

Creating a pair class in Java

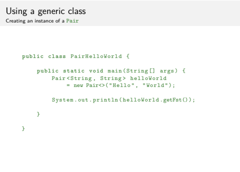
```
public class Pair<F, S> {

  private final F first;
  private final S second;

  public Pair(F first, S second) {
    this.first = first;
    this.second = second;
  }

  public F getFst() {
     return first;
  }

  public S getSnd() {
     return second;
  }

  public void println() {
    System.out.println("<" + first + "," + second + ">");
  }
}
```

1. The type parameters to a class are put between `< >`'s
2. Within the definition of the class Pair, you can use `F` and `S` as types.
3. So, for example, they are used as the types of the `first` and `second` fields.
4. They are also used as the types of the `first` and `second` parameters to the constructor.
5. They are also the return types of the `getFst` and `getSnd` methods.

Using a generic class

Creating an instance of a Pair

```
public class PairHelloWorld {

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

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

    }

}
```



Using a generic class
Creating an instance of a Pair

```
public class PairHelloWorld {

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

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

    }

}
```

1. When calling `new`, you must show that it is generic by creating a `Pair<>`
2. `helloWorld.getFst()` will have a return type of `String` in this example.

Exercise 27

1. Create a static method `equalAllThree` which takes three arguments of the same type, and returns true if they are all `.equals(...)` to each other. The syntax for the signature is:
   `static <T> boolean equalAllThree(T first, T second, T third)`
2. Now create a similar method but this time as an instance method rather than as a static method. In this case it should only take two parameters, because it should compare them with itself.
3. Consider the function `makeDuplicate` in Haskell:
   ```
   makeDuplicate :: a -> (a, a)
   makeDuplicate x = (x, x)
   ```
   Write a similar method in Java.

Reminder: Primitive and Reference Types

Java's Primitive Types

- `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, `char`
- Values of primitive types live on the *stack*, and are *copied* when assigning to variables/fields or when passed into / out of methods.
- By convention they start with a lowercase letter.

Java's Reference Types

- `String`, arrays of anything and instances of classes.
- Their contents live in the *heap*, and variables / fields get a pointer to their contents. This pointer is copied, but the contents themselves are not. So if their contents are changed every use of them will see the change.
- 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).

Type Variables can only represent reference types

- This means you cannot use `Pair<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. they always point to the same place on the heap).

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 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;
```

Exercise 28

What do the stack and heap look like during the execution of the following code:

```
int i = 0;
Integer ii = i;

// << here >>

int j = ii.intValue() + 1;
Integer jj = new Integer(j);

// << here >>

Integer kk = null;

// << here >>

int k = kk.intValue();

// << here? >>
```

Using Generics with Boxed Types

```
public class PairExample {

  public static void main(String[] args) {

    Pair<Integer, Integer> twoInts = new Pair<>(1, 1);
    twoInts.println();

    Pair<Character, Integer> charInt = new Pair<>('x', 1);
    charInt.println();

    Pair<String, Integer> keyValue = new Pair<>("Susan", 569);
    keyValue.println();
  }
}
```

Parametric Polymorphism

- Haskell has polymorphism where you use type variables to write data structures that are can be used to hold elements of any type.
- Java has generics where you use type variables to write data structures that are can be used to hold elements of any type.
- This functionality is called *parametric polymorphism* because it takes type variables as parameters and lets you create data structures (or classes) of the same shape, independent of the types of the elements to be held in the data structure. It makes a language much more expressive. That is it takes less code, to say more and the code is more understandable to read.
- If we did not have parametric polymorphism in Java, instead of a single class `Pair<F,S>`, we would have needed separate classes `StrStrPair`, `IntIntPair`, `ChrIntPair`, `ChrChrPair`, and `StrIntPair` *etc., etc.* each with their own constructor and other methods.

Collections and Interfaces

Collections

- A *collection* is an object that holds a group of objects. Methods are provided to manage the stored objects (such as storing and retrieving elements).
- Modern programming languages provide large libraries (or api's - application program interface) of collections.
- Examples of these are lists, sets, and maps - and there are many, many more and many variations of each of these.
- The api `java.util.Collections` contains the api's for the most commonly used data structures. See https://docs.oracle.com/javase/9/docs/api/java/util/package-summary.html.

Collections

- Parametric polymorphism (using generics) make the collections libraries very expressive. For example, if one had a class `Student` then the `List<E>` class could be used if you wanted a list data structure for your `Student`s, whereas before Java had generics, programmers had to write all the methods for accessing a `StudentList` themselves.
- As is very good programming practice, Java separates each data structure into two - what it does and how it is implemented.
- To use a data structure in your code, you need to know what it does. This is the data structure's specification and you can see what it does by looking at its `interface`.

The List<E> interface

Lists in Java

- Use the List interface to store a list of elements
- Elements can be added to the end of the list (default), or at a specific position.
- Lists do not have a fixed size, and support behaviour for removing elements.
- https://docs.oracle.com/javase/9/docs/api/java/util/List.html shows that there are over 30 methods in the List interface. Each method is described.

The List<E> interface

Important methods in the interface

```
public interface List<E> {
  boolean add(E e);
  void add(int index, E element);
  void clear();
  boolean contains(Object o);
  boolean equals(Object o);
  E get(int index);
  int indexOf(Object o);
  boolean isEmpty();
  E remove(int index);
  int size();
  ...
}
```

The Set<E> interface

- A list has ordered, possibly duplicated elements. Sometimes this level of structure is not needed and a set models the problem better.
- Use the Set interface to hold a unique set of values.
- Sets do not have a way of retrieving an individual element, as they do not commit to storing items in the order they are added.
- https://docs.oracle.com/javase/9/docs/api/java/util/Set.html shows the methods in the Set interface.

The Set<E> interface

Important methods in the interface

```
public interface Set<E> {
  boolean add(E e);
  void clear();
  boolean contains(Object o);
  boolean equals(Object o)
  boolean isEmpty();
  boolean remove(Object o);
  int size();
  ...
}
```

The Map<K,V> interface - lookup tables

Values do not need to be unique in a map.

- Another very useful data structure is a map. Maps contain key-value pairs.
- The key is used to access the key-value pair (or entry), so each key has to be unique. A key is an object that you use to retrieve a value at a later date. Here is an example where a map would be an appropriate data structure:

Key	Value
Susan	569
Tony	354
Alastair	422
Alessandra	560
Marc	304
Konstantinos	228
Mark	228

- Adding entries into a Map<K,V> requires the user to call void put(K key, V value), which will add the key-value pair to the map if the key does not exist, or replace the value of the given key if it is already present. This ensures the set of keys will always be unique.
- https://docs.oracle.com/javase/9/docs/api/java/util/Map.html shows the methods in the Map interface.

The Map<K,V> interface - lookup tables

Important methods in the interface

```
public interface Map<K,V> {
  void clear();
  boolean containsKey(Object key);
  boolean containsValue(Object value);
  Set<Map.Entry<K,V>> entrySet();
  boolean equals(Object o)
  V get(Object key);
  boolean isEmpty();
  Set<K> keySet();
  V put(K key, V value);
  V remove(Object key);
  int size();
  ...
}
```

Interfaces and Implementations

- Interfaces say *what* is implemented but not *how* it should be implemented. They contain *no* method bodies, just fields and method headers.
- Java provides over 2500 interfaces, see http://docs.oracle.com/javase/9/docs/api/
- Some you might find useful are in:
  - java.lang.Math - util methods for mathematics.
  - java.util.Arrays - util methods for handling arrays.
  - java.util.Collections - util methods for handling Collections
- A class can implement an interface (the *how*). It must provide methods for each of the method headers in the interface.
- For example, the interfaces for the data structures list, set, and map have a size() method and so any class that implements one of these data structures must have a size() method.

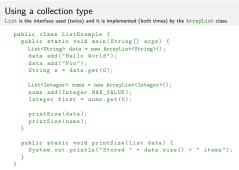
Interfaces and Implementations

- For the interfaces that Java provides, it also provides a variety of classes that can be chosen to implement it and they are listed in the documentation for the interfaces themselves.
- It is usual that there are several ways one can implement a given interface, the *how*.
- For example, look up List and see 'All Known Implementing Classes:'
- There are ten implementing classes including: ArrayList, LinkedList, Stack, Vector
- What the programmer has to do after they decide which interface they wish to use, is to choose which class they want to use to implement it.
- Choose an implementing class that has the features you want for your application. For lists, if you cannot decide I suggest you use an ArrayList.
- Next term you will learn how to write your own interfaces in Java, but you can go a long way with the interfaces and implementing classes that Java provides already.

Using a collection type

List is the interface used (twice) and it is implemented (both times) by the ArrayList class.

```
public class ListExample {
  public static void main(String[] args) {
    List<String> data = new ArrayList<String>();
    data.add("Hello World");
    data.add("Foo");
    String s = data.get(0);

    List<Integer> nums = new ArrayList<Integer>();
    nums.add(Integer.MAX_VALUE);
    Integer first = nums.get(0);

    printSize(data);
    printSize(nums);
  }

  public static void printSize(List data) {
    System.out.println("Stored " + data.size() + " items");
  }
}
```



Using a collection type
List is the interface used (twice) and it is implemented (both times) by the ArrayList class.

```
public class ListExample {
  public static void main(String[] args) {
    List<String> data = new ArrayList<String>();
    data.add("Hello World");
    data.add("Foo");
    String s = data.get(0);

    List<Integer> nums = new ArrayList<Integer>();
    nums.add(Integer.MAX_VALUE);
    Integer first = nums.get(0);

    printSize(data);
    printSize(nums);
  }

  public static void printSize(List data) {
    System.out.println("Stored " + data.size() + " items");
  }
}
```

1. The type parameters for both interface and implementation appear between < >'s
2. Only reference types are allowed as type parameters, no primitive types.

Using a collection type

Using a Map<K,V> inteface implemented by a Hashmap class.

```
import java.util.*;

public class MapExample {

  public static void main(String[] args) {
    Map<String, Integer> officeDB = new HashMap<String, Integer>();
    officeDB.put("Susan", 569);
    officeDB.put("Tony", 354);
    officeDB.put("Alastair", 422);
    officeDB.put("Alessandra", 560);
    officeDB.put("Marc", 304);
    officeDB.put("Konstantinos", 228);
    officeDB.put("Mark", 228);

    System.out.println("Susan is in " + officeDB.get("Susan"));
  }
}
```

Exercise 29

What will the following print? What would the output be if we used a list instead of a set?

```
public class SetExample {

  public static void main(String[] args) {
    Set<Integer> nums = new TreeSet<Integer>();
    nums.add(5);
    nums.add(10);
    nums.add(3);
    nums.add(5);

    for (Integer i : nums) {
      System.out.println(i);
    }
  }
}
```

Summary

- Methods and constructors can be overloaded. For example, `print` can print any type because it is overloaded. Two overloaded methods or constructors take parameters of different types.
- `this` is used to reference an individual object.
- Java has a generics capability, but that we cannot use primitives as the types of the elements. Fortunately, primitives can be boxed so they can be used in generic data structures.
- Java has a large library of interfaces, and these provide a very rich library of data structures or collections.
- We have looked into three of these in a little detail. They are lists, sets, and maps.
- The interfaces are generic so there are `List<Integer>`s and `List<String>`s for example.
- Interfaces need to be implemented by classes and Java also provides a large collection of classes that implement the interfaces that are provided.
- It is good to use interfaces when declaring your data structures and the declarations are of the form:

```
List<String> myList = new ArrayList<String>();
```