Introduction to Java

Java, Principles of OO

Some History

- Developed and maintained by Sun Microsystems
  - Originally called Oak
  - Aimed at producing an operating environment for networked devices and embedded systems
  - …but has been much more successful
- Design objectives for the language
  - Simple, object-oriented,
  - Distributed, multi-threaded, and platform neutral
  - Robust, secure, scaleable

The Virtual Machine

- Java is both compiled and interpreted
  - Source code is compiled into Java *bytecode*
  - Which is then interpreted by the *Java Virtual Machine* (JVM)
  - Therefore bytecode is machine code for the JVM
- Java bytecode can run on any JVM, on any platform
  - …including mobile phones and other hand-held devices
- Networking and distribution are core features
  - In other languages these are additional APIs
  - Makes Java very good for building networked applications, server side components, etc.

Features of the JVM

- The Garbage Collector
  - Java manages memory for you, the developer has no control over the allocation of memory (unlike in C/C++).
  - This is much simpler and more robust (no chance of memory leaks or corruption)
  - Runs in the background and cleans up memory while application is running
- The Just In Time compiler (JIT)
  - Also known as "Hot Spot"
  - Continually optimises running code to improve performance
    - Automatically removes bottlenecks

Object-Oriented Programming

- Understanding OOP is fundamental to writing good Java applications
  - Improves design of your code
  - Improves understanding of the Java APIs
- There are several concepts underlying OOP:
  - Abstract Types (Classes)
  - Encapsulation (or Information Hiding)
  - Aggregation
  - Inheritance
  - Polymorphism

What is OOP?

- Modelling real-world objects in software
- Why design applications in this way?
  - We naturally *class*ify objects into different *types*.
  - By attempting to do this with software aim to make it more maintainable, understandable and easier to reuse
- In a conventional application we typically:
  - decompose it into a series of functions,
  - define data structures that those functions act upon
  - there is no relationship between the two other than the functions act on the data

What is OOP?

- How is OOP different to conventional programming?
  - Decompose the application into *abstract data types* by identifying some useful entities/abstractions
  - An abstract type is made up of a series of behaviours and the data that those behaviours use.
- Similar to database modelling, only the types have both behaviour and state (data)

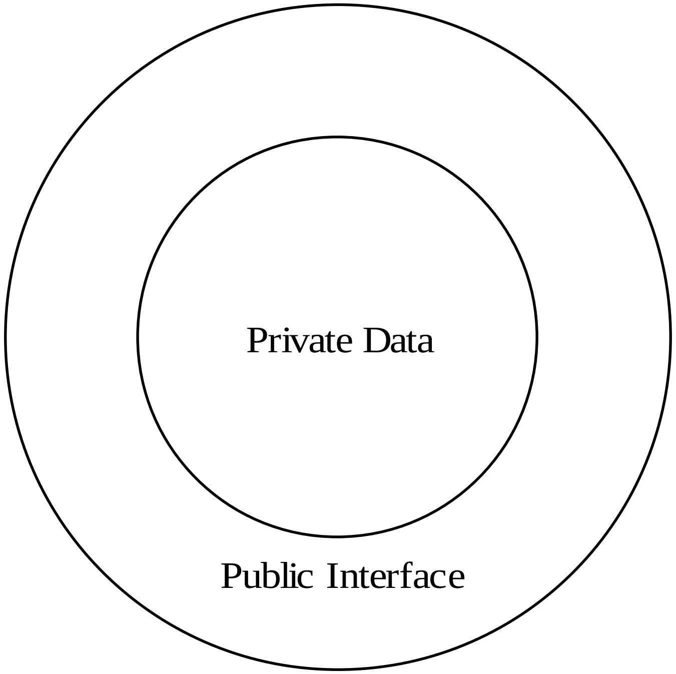
Abstract Data Types - Classes

- Identifying abstract types is part of the modelling/design process
  - The types that are useful to model may vary according to the individual application
  - For example a payroll system might need to know about Departments, Employees, Managers, Salaries, etc
  - An E-Commerce application may need to know about Users, Shopping Carts, Products, etc
- Object-oriented languages provide a way to define abstract data types, and then create *objects* from them
  - It's a template (or 'cookie cutter') from which we can create new objects
  - For example, a Car class might have attributes of speed, colour, and behaviours of accelerate, brake, etc
  - An individual Car *object* will have the same behaviours but its own values assigned to the attributes (e.g. 30mph, Red, etc)

Encapsulation

- The data (state) of an object is private – it cannot be accessed directly.
- The state can only be changed through its behaviour, otherwise known as its public *interface* or *contract*
- This is called *encapsulation*

"The Doughnut Diagram"
Showing that an object has private state and public behaviour. State can only be changed by invoking some behaviour



Encapsulation

- Main benefit of encapsulation
  - Internal state and processes can be changed independently of the public interface
  - Limits the amount of large-scale changes required to a system

What is an OO program?

- What does an OO program consist of?
  - A series of objects that use each others behaviours in order to carry out some desired functionality
  - When one object invokes some behaviour of another it sends it a *message*
  - In Java terms it invokes a *method* of the other object
  - A method is the implementation of a given behaviour.
- OO programs are intrinsically modular
  - Objects are only related by their public behaviour (methods)
  - Therefore objects can be swapped in and out as required (e.g. for a more efficient version)
  - This is another advantage of OO systems

Aggregation

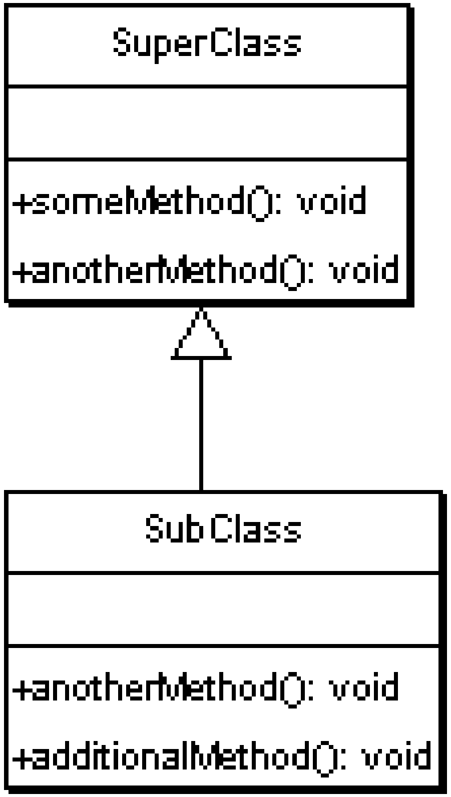
- Aggregation is the ability to create new classes out of existing classes
  - Treating them as building blocks or components
- Aggregation allows reuse of existing code
  - “Holy Grail” of software engineering
- Two forms of aggregation
- Whole-Part relationships
  - Car is made of Engine, Chassis, Wheels
- Containment relationships
  - A Shopping Cart contains several Products
  - A List contains several Items

Inheritance

- Inheritance is the ability to define a new class in terms of an existing class
  - The existing class is the *parent*, *base* or *superclass*
  - The new class is the *child*, *derived* or *subclass*
- The child class inherits all of the attributes and behaviour of its parent class
  - It can then add new attributes or behaviour
  - Or even alter the implementation of existing behaviour
- Inheritance is therefore another form of code reuse

UML -- Inheritance

- Inheritance is shown by a solid arrow from the sub-class to the super-class
- The sub-class doesn’t list its super-class attributes or methods,
- *unless* its providing its own alternate version (I.e. is extending the behaviour of the base class)



Polymorphism

- Means ‘many forms’
- In brief, polymorphism allows two different classes to respond to the same message in different ways
- E.g. both a Plane and a Car could respond to a ‘turnLeft’ message,
  - however the means of responding to that message (turning wheels, or banking wings) is very different for each.
- Allows objects to be treated as if they’re identical

Recap!

- In OO programming we
  - Define classes
  - Create objects from them
  - Combine those objects together to create an application
- Benefits of OO programming
  - Easier to understand (closer to how we view the world)
  - Easier to maintain (localised changes)
  - Modular (classes and objects)
  - Good level of code reuse (aggregation and inheritance)

Java Syntax

Naming

- All Java syntax is case sensitive
- Valid Java names
  - Consist of letters, numbers, underscore, and dollar
  - Names can only start with letter or underscore
  - E.g. `firstAttribute` but not `1stAttribute`
- "Camel case" convention
  - Java encourages long, explanatory names
  - Start with a lower case letter, with words capitalised
  - E.g. `thisIsCamelCase`, `andSoIsThisAsWell`

Keywords

- **keyword**: An identifier that you cannot use because it already has a reserved meaning in Java.

abstract
boolean
break
byte
case
catch
char
class
const
continue

default
do
double
else
extends
final
finally
float
for
goto

if
implements
import
instanceof
int
interface
long
native
new
package

private
protected
public
return
short
static
strictfp
super
switch
synchronized

this
throw
throws
transient
try
void
volatile
while

Java Types

- Java has two basic types
  - Primitive types
  - Reference Types
- Primitive types
  - integers, floating point numbers, characters, etc
  - Refer to actual values
- Reference types
  - Arrays, Classes, Objects, etc
  - Refer to memory locations (by name, not location)

Primitive Types

Type	Description	Size
Boolean (`boolean`)	True/false value	1 bit
Byte (`byte`)	Byte-length integer	1 byte
Short (`short`)	Short integer	2 bytes
Integer (`int`)	Integer	4 bytes
Long (`long`)	Long Integer	8 bytes
Float (`float`)	Single precision floating point number	4 bytes
Double (`double`)	Double precision float	8 bytes
Char (`char`)	Single character	2 bytes

Syntax

- **syntax**: The set of legal structures and commands that can be used in a particular language.
  - Every basic Java statement ends with a semicolon `;`
  - The contents of a class or method occur between `{` and `}`

- **syntax error** (**compiler error**): A problem in the structure of a program that causes the compiler to fail.
  - Missing semicolon
  - Too many or too few `{ }` braces
  - Illegal identifier for class name
  - Class and file names do not match

    ...

Syntax error example

```
1  public class Hello {
2      pooblic static void main(String[] args) {
3          System.owt.println("Hello, world!")_
4      }
5  }
```

- Compiler output:

```
Hello.java:2: <identifier> expected
    pooblic static void main(String[] args) {
         ^
Hello.java:3: ';' expected
}
^
2 errors
```

  - The compiler shows the line number where it found the error.
  - The error messages can be tough to understand!

Escape sequences

- **escape sequence**: A special sequence of characters used to represent certain special characters in a string.

`\t`	tab character
`\n`	new line character
`\"`	quotation mark character
`\\`	backslash character

  - Example:

```
System.out.println("\\hello\nhow\tare \"you\"?\\\\");
```

  - Output:

```
\hello
how     are "you"?\\
```

Comments

- **comment**: A note written in source code by the programmer to describe or clarify the code.
  - Comments are not executed when your program runs.
- Syntax:

  `//` **comment text, on one line**

  or,

  `/*` **comment text; may span multiple lines** `*/`

- Examples:

```
// This is a one-line comment.

/* This is a very long
   multi-line comment. */
```

Using comments

- Where to place comments:
  - at the top of each file (a "comment header")
  - at the start of every method (seen later)
  - to explain complex pieces of code

- Comments are useful for:
  - Understanding larger, more complex programs.
  - Multiple programmers working together, who must understand each other's code.

Strings

- **string**: A sequence of characters to be printed.
  - Starts and ends with a `"` quote `"` character.
    - The quotes do not appear in the output.
  - Examples:

```
"hello"
"This is a string.  It's very long!"
```

- Restrictions:
  - May not span multiple lines.

```
"This is not
a legal String."
```

  - May not contain a `"` character.

```
"This is not a "legal" String either."
```

Syntax Examples (Variables)

How do we declare a variable?

```
int anInteger;
Boolean isSwitchOn;
```

How do we initialize a variable?

```
anInteger = 10;
isSwitchOn = true;
```

Can we combine declaration and initialization?

```
int anInteger = 10;
Boolean isSwitchOn = true;
```

Syntax Examples (if, if else)

```
if (x == y)
{
   //executes if true
}

if (somethingIsTrue())
{
  doSomething();
}
else
{
  doSomethingElse();
}
```

Example (for)

```
int x=0;
for (int i=1; i<=10; i++)
{
  //code to repeat ten times
  x = x + i;
}
```

Example (while)

```
int x=0;
while (x < 10)
{
  doSomething();
  x++;
}

//loop forever
while (true)
{
}
```

Methods

Algorithms

- **algorithm**: A list of steps for solving a problem.
- Example algorithm: "Bake sugar cookies"
  - Mix the dry ingredients.
  - Cream the butter and sugar.
  - Beat in the eggs.
  - Stir in the dry ingredients.
  - Set the oven temperature.
  - Set the timer.
  - Place the cookies into the oven.
  - Allow the cookies to bake.
  - Spread frosting and sprinkles onto the cookies.
  - ...

Problems with algorithms

- *lack of structure*: Many tiny steps; tough to remember.
- *redundancy*: Consider making a double batch...
  - Mix the dry ingredients.
  - Cream the butter and sugar.
  - Beat in the eggs.
  - Stir in the dry ingredients.
  - Set the oven temperature.
  - Set the timer.
  - Place the first batch of cookies into the oven.
  - Allow the cookies to bake.
  - Set the timer.
  - Place the second batch of cookies into the oven.
  - Allow the cookies to bake.
  - Mix ingredients for frosting.
  - ...

Structured algorithms

- **structured algorithm**: Split into coherent tasks.

  1 Make the cookie batter.
  - Mix the dry ingredients.
  - Cream the butter and sugar.
  - Beat in the eggs.
  - Stir in the dry ingredients.

  2 Bake the cookies.
  - Set the oven temperature.
  - Set the timer.
  - Place the cookies into the oven.
  - Allow the cookies to bake.

  3 Add frosting and sprinkles.
  - Mix the ingredients for the frosting.
  - Spread frosting and sprinkles onto the cookies.

  ...

Removing redundancy

- A well-structured algorithm can describe repeated tasks with less redundancy.

1 Make the cookie batter.
- Mix the dry ingredients.
- ...

2a Bake the cookies (first batch).
- Set the oven temperature.
- Set the timer.
- ...

2b Bake the cookies (second batch).

3 Decorate the cookies.
- ...

A program with redundancy

```java
public class BakeCookies {
    public static void main(String[] args) {
        System.out.println("Mix the dry ingredients.");
        System.out.println("Cream the butter and sugar.");
        System.out.println("Beat in the eggs.");
        System.out.println("Stir in the dry ingredients.");
        System.out.println("Set the oven temperature.");
        System.out.println("Set the timer.");
        System.out.println("Place a batch of cookies into the oven.");
        System.out.println("Allow the cookies to bake.");
        System.out.println("Set the oven temperature.");
        System.out.println("Set the timer.");
        System.out.println("Place a batch of cookies into the oven.");
        System.out.println("Allow the cookies to bake.");
        System.out.println("Mix ingredients for frosting.");
        System.out.println("Spread frosting and sprinkles.");
    }
}
```

Methods

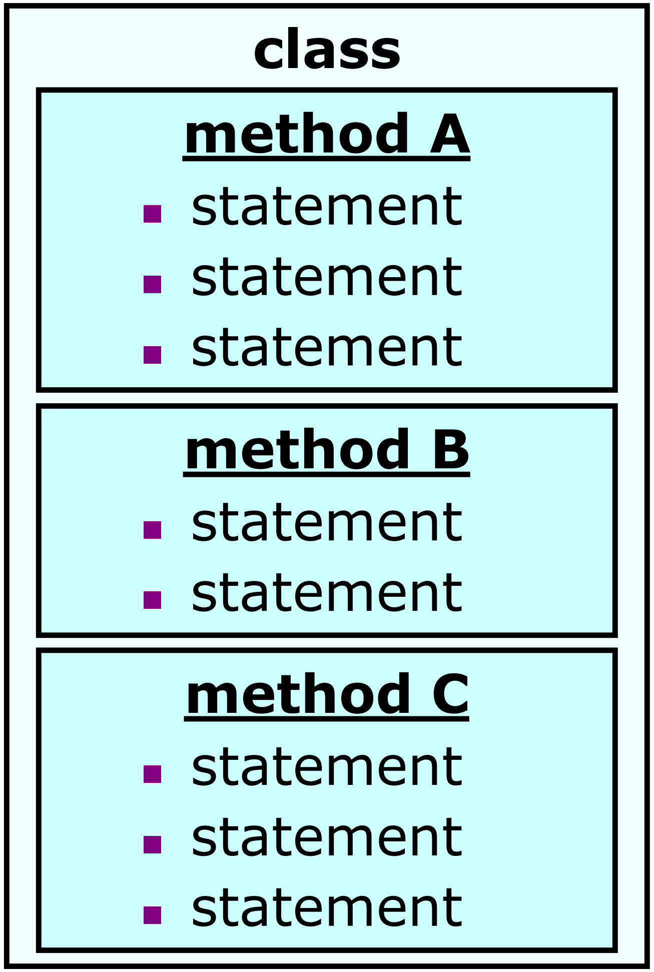
- Define some behaviour of a class
- Method declarations have four basic sections, and a method body:
  - Visibility modifier (who can call the method)
  - Return type (what does it return)
  - Method name
  - Parameter list (what parameters does it accept)

Static methods

- **static method**: A named group of statements.
    - denotes the *structure* of a program
    - eliminates *redundancy* by code reuse

  – **procedural decomposition**: dividing a problem into methods

- Writing a static method is like adding a new command to Java.



class

method A

- statement
- statement
- statement

method B

- statement
- statement

method C

- statement
- statement
- statement

Using static methods

1. Design the algorithm.
   - Look at the structure, and which commands are repeated.
   - Decide what are the important overall tasks.

2. **Declare** (write down) the methods.
   - Arrange statements into groups and give each group a name.

3. **Call** (run) the methods.
   - The program's `main` method executes the other methods to perform the overall task.

Design of an algorithm

```
// This program displays a delicious recipe for baking cookies.
public class BakeCookies2 {
    public static void main(String[] args) {
        // Step 1: Make the cake batter.
        System.out.println("Mix the dry ingredients.");
        System.out.println("Cream the butter and sugar.");
        System.out.println("Beat in the eggs.");
        System.out.println("Stir in the dry ingredients.");

        // Step 2a: Bake cookies (first batch).
        System.out.println("Set the oven temperature.");
        System.out.println("Set the timer.");
        System.out.println("Place a batch of cookies into the oven.");
        System.out.println("Allow the cookies to bake.");

        // Step 2b: Bake cookies (second batch).
        System.out.println("Set the oven temperature.");
        System.out.println("Set the timer.");
        System.out.println("Place a batch of cookies into the oven.");
        System.out.println("Allow the cookies to bake.");

        // Step 3: Decorate the cookies.
        System.out.println("Mix ingredients for frosting.");
        System.out.println("Spread frosting and sprinkles.");
    }
}
```

Declaring a method

Gives your method a name so it can be executed

- Syntax:

```
public static void name() {
    statement;
    statement;
    ...
    statement;
}
```

- Example:

```
public static void printWarning() {
    System.out.println("This product causes cancer");
    System.out.println("in lab rats and humans.");
}
```

Calling a method

Executes the method's code

- Syntax:

  name();

  - You can call the same method many times if you like.

- Example:

```
printWarning();
```

  - Output:

```
This product causes cancer
in lab rats and humans.
```

Program with static method

```java
public class FreshPrince {
    public static void main(String[] args) {
        rap();                 // Calling (running) the rap method
        System.out.println();
        rap();                 // Calling the rap method again
    }

    // This method prints the lyrics to my favorite song.
    public static void rap() {
        System.out.println("Now this is the story all about how");
        System.out.println("My life got flipped turned upside-down");
    }
}
```

Output:

```
Now this is the story all about how
My life got flipped turned upside-down

Now this is the story all about how
My life got flipped turned upside-down
```

Final cookie program

```
// This program displays a delicious recipe for baking cookies.
public class BakeCookies3 {
    public static void main(String[] args) {
        makeBatter();
        bake();       // 1st batch
        bake();       // 2nd batch
        decorate();
    }

    // Step 1: Make the cake batter.
    public static void makeBatter() {
        System.out.println("Mix the dry ingredients.");
        System.out.println("Cream the butter and sugar.");
        System.out.println("Beat in the eggs.");
        System.out.println("Stir in the dry ingredients.");
    }

    // Step 2: Bake a batch of cookies.
    public static void bake() {
        System.out.println("Set the oven temperature.");
        System.out.println("Set the timer.");
        System.out.println("Place a batch of cookies into the oven.");
        System.out.println("Allow the cookies to bake.");
    }

    // Step 3: Decorate the cookies.
    public static void decorate() {
        System.out.println("Mix ingredients for frosting.");
        System.out.println("Spread frosting and sprinkles.");
    }
}
```

Methods calling methods

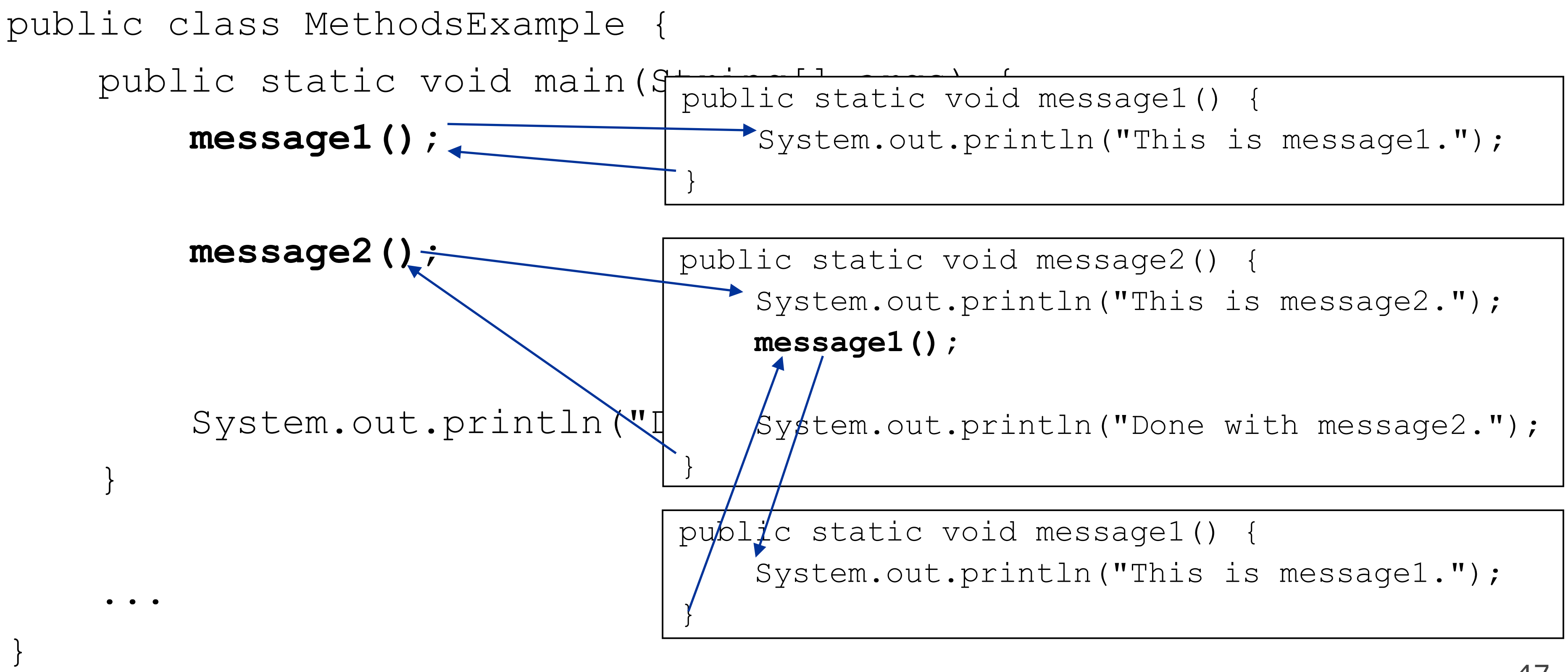
```
public class MethodsExample {
    public static void main(String[] args) {
        message1();
        message2();
        System.out.println("Done with main.");
    }

    public static void message1() {
        System.out.println("This is message1.");
    }

    public static void message2() {
        System.out.println("This is message2.");
        message1();
        System.out.println("Done with message2.");
    }
}
```

- Output:

```
This is message1.
This is message2.
This is message1.
Done with message2.
Done with main.
```

Control flow

- When a method is called, the program's execution...
  - "jumps" into that method, executing its statements, then
  - "jumps" back to the point where the method was called.



```
public class MethodsExample {
    public static void main(S
        message1();

        message2();


        System.out.println("D
    }

    ...
}
```

```
public static void message1() {
    System.out.println("This is message1.");
}
```

```
public static void message2() {
    System.out.println("This is message2.");
    message1();

    System.out.println("Done with message2.");
}
```

```
public static void message1() {
    System.out.println("This is message1.");
}
```

When to use methods

- Place statements into a static method if:
  - The statements are related structurally, and/or
  - The statements are repeated.

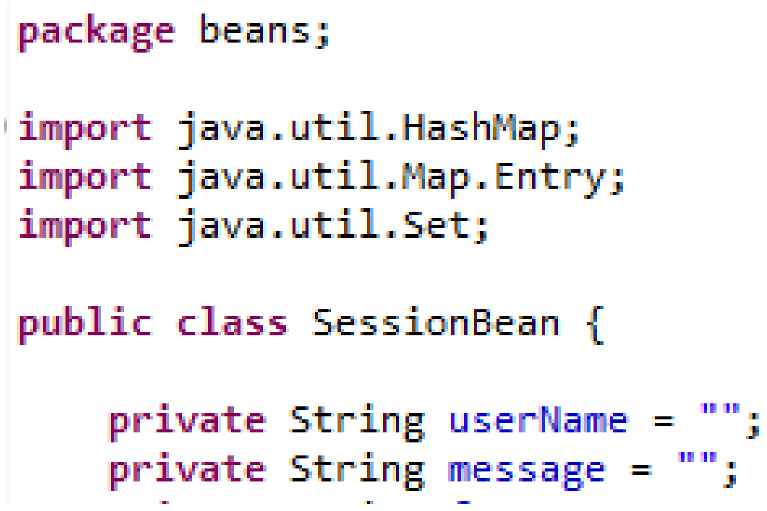
- You should not create static methods for:
  - An individual `println` statement.
  - Only blank lines. (Put blank `println`s in `main`.)
  - Unrelated or weakly related statements.
    (Consider splitting them into two smaller methods.)

Classes

- One Java class defined in each .java file
- File name must match the name of the class
  - Otherwise there will be compilation errors
  - Class names start with an upper case letter
- Compiler will generate a .class file with same name
  - Contains the *bytecode*
- Classes defined using the `class` keyword.

Packages

- Group related classes together
- Each class in a package must have a unique name
- Indicate the package a class belongs to with the `package` keyword
- Recommended each class is put in a package
- Gain access to public classes in other packages using the `import` keyword
  - The JVM needs to know where the classes are defined before you can use them



```
package beans;

import java.util.HashMap;
import java.util.Map.Entry;
import java.util.Set;

public class SessionBean {

    private String userName = "";
    private String message = "";
```

More fundamentals

Expressions

- **expression**: A value or operation that computes a value.
  - Examples:

    ```
    1 + 4 * 5
    (7 + 2) * 6 / 3
    42
    ```

  - The simplest expression is a *literal value*.
  - A complex expression can use operators and parentheses.

Arithmetic operators

- **operator**: Combines multiple values or expressions.

`+`	addition
`-`	subtraction (or negation)
`*`	multiplication
`/`	division
`%`	modulus (a.k.a. remainder)

- As a program runs, its expressions are *evaluated*.
  - `1 + 1` evaluates to `2`
  - `System.out.println(3 * 4);` prints `12`
    - How would we print the text `3 * 4` ?

Integer division with /

- When we divide integers, the quotient is also an integer.
  - `14 / 4` is `3`, not `3.5`

```
       3                  4                     52
4 ) 14          10 ) 45          27 ) 1425
    12                   40                 135
     2                    5                   75
                                              54
                                              21
```

- More examples:
  - `32 / 5` is `6`
  - `84 / 10` is `8`
  - `156 / 100` is `1`

  - Dividing by 0 causes an error when your program runs.

Integer remainder with %

- The % operator computes the remainder from integer division.
  - 14 % 4 is 2
  - 218 % 5 is 3

```
         3
    _______
4 )  14
     12
     __
      2
```

```
        43
    _______
5 )  218
     20
     __
      18
      15
      __
       3
```

What is the result?

```
45 % 6
2 % 2
8 % 20
11 % 0
```

- Applications of % operator:
  - Obtain last digit of a number: 230857 % 10 is 7
  - Obtain last 4 digits: 658236489 % 10000 is 6489
  - See whether a number is odd: 7 % 2 is 1, 42 % 2 is 0

Precedence

- **precedence**: Order in which operators are evaluated.
  - Generally operators evaluate left-to-right.

    `1 - 2 - 3` is `(1 - 2) - 3` which is `-4`

  - But `* / %` have a higher level of precedence than `+ -`

    ```
    1 + 3 * 4                 is 13

    6 + 8 / 2 * 3
    6 +   4   * 3
    6 +     12                is 18
    ```

  - Parentheses can force a certain order of evaluation:

    ```
    (1 + 3) * 4               is 16
    ```

  - Spacing does not affect order of evaluation

    ```
    1+3 * 4-2                 is 11
    ```

Real numbers (type `double`)

- Examples: `6.022` , `-42.0` , `2.143e17`
  - Placing `.0` or `.` after an integer makes it a `double`.

- The operators `+ - * / % ()` all still work with `double`.
  - `/` produces an exact answer: `15.0 / 2.0` is `7.5`
  - Precedence is the same: `()` before `* / %` before `+ -`

String concatenation

- **string concatenation**: Using + between a string and another value to make a longer string.

```
"hello" + 42    is "hello42"
1 + "abc" + 2   is "1abc2"
"abc" + 1 + 2   is "abc12"
1 + 2 + "abc"   is "3abc"
"abc" + 9 * 3   is "abc27"
"1" + 1         is "11"
4 - 1 + "abc"   is "3abc"
```

- Use + to print a string and an expression's value together.

  - `System.out.println("Grade: " + (95.1 + 71.9) / 2);`

  - Output: `Grade: 83.5`

Variables

Declaration

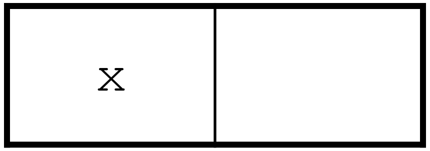
- **variable declaration**: Sets aside memory for storing a value.
  - Variables must be declared before they can be used.

- Syntax:

  type name;

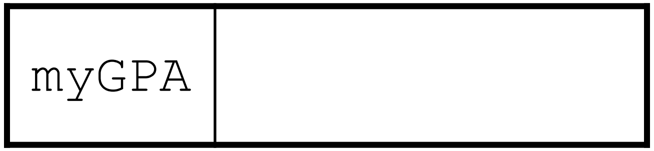
  - The name is an *identifier*.

  - `int x;`



x

  - `double myGPA;`



myGPA

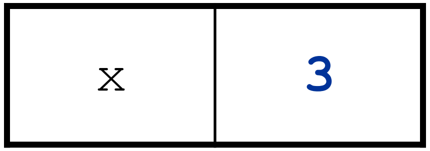
Assignment

- **assignment**: Stores a value into a variable.
  - The value can be an expression; the variable stores its result.

- Syntax:

  name = **expression**;

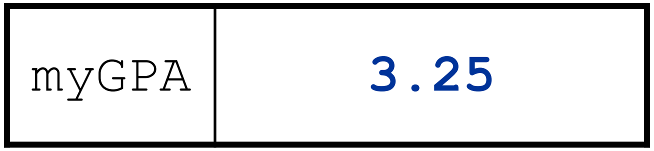
  -
    ```
    int x;
    x = 3;
    ```



x

3

  -
    ```
    double myGPA;
    myGPA = 1.0 + 2.25;
    ```



myGPA

3.25

Using variables

- Once given a value, a variable can be used in expressions:

```
int x;
x = 3;
System.out.println("x is " + x);       // x is 3

System.out.println(5 * x - 1);         // 5 * 3 - 1
```

- You can assign a value more than once:

x	11

```
int x;
x = 3;
System.out.println(x + " here");       // 3 here

x = 4 + 7;
System.out.println("now x is " + x); // now x is 11
```

Declaration/initialization

- A variable can be declared/initialized in one statement.

- Syntax:

  type name = **value**;

  - `double myGPA = 3.95;`

myGPA	3.95

  - `int x = (11 % 3) + 12;`

x	14

Assignment and algebra

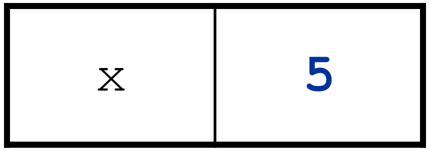
- Assignment uses = , but it is not an algebraic equation.

  = means, *"store the value at right in variable at left"*

  - The right side expression is evaluated first, and then its result is stored in the variable at left.

- What happens here?

```
int x = 3;
x = x + 2;    // ???
```



x 5

Assignment and types

- A variable can only store a value of its own type.
  - `int x = 2.5;      // ERROR: incompatible types`

- An `int` value can be stored in a `double` variable.
  - The value is converted into the equivalent real number.
  - `double myGPA = 4;`

myGPA	4.0

  - `double avg = 11 / 2;`

avg	**5.0**

    - Why does `avg` store `5.0` and not `5.5` ?

Compiler errors

- A variable can't be used until it is assigned a value.

  -
    ```
    int x;
    System.out.println(x);   // ERROR: x has no value
    ```

- You may not declare the same variable twice.

  -
    ```
    int x;
    int x;                   // ERROR: x already exists
    ```

  -
    ```
    int x = 3;
    int x = 5;               // ERROR: x already exists
    ```
    - How can this code be fixed?

Printing a variable's value

- Use + to print a string and a variable's value on one line.

  -
    ```
    double grade = (95.1 + 71.9 + 82.6) / 3.0;
    System.out.println("Your grade was " + grade);

    int students = 11 + 17 + 4 + 19 + 14;
    System.out.println("There are " + students +
                       " students in the course.");
    ```

- Output:

  ```
  Your grade was 83.2
  There are 65 students in the course.
  ```

Type casting

- **type cast**: A conversion from one type to another.
  - To promote an `int` into a `double` to get exact division from `/`
  - To truncate a `double` from a real number to an integer

- Syntax:

  (**type**) **expression**

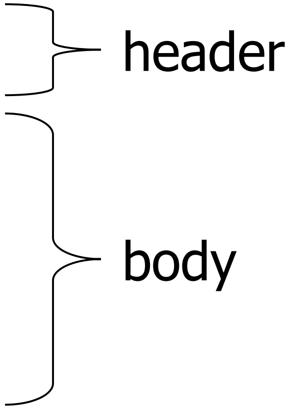
  Examples:

```
double result = (double) 19 / 5;      // 3.8
int result2 = (int) result;           // 3
int x = (int) Math.pow(10, 3);        // 1000
```

The `for` loop

for loop syntax

```
for (initialization; test; update) {
    statement;
    statement;
    ...
    statement;
}
```



header

body

- Perform **initialization** once.
- Repeat the following:
  - Check if the **test** is true. If not, stop.
  - Execute the **statement**s.
  - Perform the **update**.

Initialization

```java
for (int i = 1; i <= 6; i++) {
    System.out.println("I am so smart");
}
```

- Tells Java what variable to use in the loop
  - Performed once as the loop begins
  - The variable is called a *loop counter*
    - can use any name, not just `i`
    - can start at any value, not just `1`

Test

```
for (int i = 1; i <= 6; i++) {
    System.out.println("I am so smart");
}
```

- Tests the loop counter variable against a limit
  - Uses comparison operators:

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

Increment and decrement

shortcuts to increase or decrease a variable's value by 1

Shorthand	Equivalent longer version
variable`++;`	**variable** `=` **variable** `+ 1;`
variable`--;`	**variable** `=` **variable** `- 1;`

```
int x = 2;
x++;                    // x = x + 1;
                        // x now stores 3

double gpa = 2.5;
gpa--;                  // gpa = gpa - 1;
                        // gpa now stores 1.5
```

Modify-and-assign

shortcuts to modify a variable's value

Shorthand	Equivalent longer version
variable += **value**;	**variable** = **variable** + **value**;
variable -= **value**;	**variable** = **variable** - **value**;
variable *= **value**;	**variable** = **variable** * **value**;
variable /= **value**;	**variable** = **variable** / **value**;
variable %= **value**;	**variable** = **variable** % **value**;

```
x += 3;             // x = x + 3;

gpa -= 0.5;         // gpa = gpa - 0.5;

number *= 2;        // number = number * 2;
```

System.out.print

- Prints without moving to a new line
  - allows you to print partial messages on the same line

```
int highestTemp = 5;
for (int i = -3; i <= highestTemp / 2; i++) {
    System.out.print((i * 1.8 + 32) + "  ");
}
```

- Output:

```
26.6  28.4  30.2  32.0  33.8  35.6
```

  - Concatenate " " to separate the numbers

Nested `for` loops

Nested loops

- **nested loop**: A loop placed inside another loop.

```
for (int i = 1; i <= 5; i++) {
    for (int j = 1; j <= 10; j++) {
        System.out.print("*");
    }
    System.out.println();    // to end the line
}
```

- Output:

```
**********
**********
**********
**********
**********
```

- The outer loop repeats 5 times; the inner one 10 times.
  - "sets and reps" exercise analogy

Common errors

- Both of the following sets of code produce *infinite loops*:

```
for (int i = 1; i <= 5; i++) {
    for (int j = 1; i <= 10; j++) {
        System.out.print("*");
    }
    System.out.println();
}

for (int i = 1; i <= 5; i++) {
    for (int j = 1; j <= 10; i++) {
        System.out.print("*");
    }
    System.out.println();
}
```

Class constants and scope

Scope

- **scope**: The part of a program where a variable exists.
  - From its declaration to the end of the `{ }` braces
    - A variable declared in a `for` loop exists only in that loop.
    - A variable declared in a method exists only in that method.

```
public static void example() {
    int x = 3;
    for (int i = 1; i <= 10; i++) {
        System.out.println(x);
    }
    // i no longer exists here
} // x ceases to exist here
```

i's scope

x's scope

Scope implications

- Variables without overlapping scope can have same name.

```
for (int i = 1; i <= 100; i++) {
    System.out.print("/");
}
for (int i = 1; i <= 100; i++) {   // OK
    System.out.print("\\");
}
int i = 5;                       // OK: outside of loop's scope
```

- A variable can't be declared twice or used out of its scope.

```
for (int i = 1; i <= 100 * line; i++) {
    int i = 2;                   // ERROR: overlapping scope
    System.out.print("/");
}
i = 4;                           // ERROR: outside scope
```

Class constants

- **class constant**: A fixed value visible to the whole program.
  - value can be set only at declaration; cannot be reassigned

- Syntax:

  `public static final` **type name** `=` **value**`;`

  - name is usually in ALL_UPPER_CASE

  - Examples:

    ```
    public static final int DAYS_IN_WEEK = 7;
    public static final double INTEREST_RATE = 3.5;
    public static final int SSN = 658234569;
    ```

Using a constant

- Constant allows many methods to refer to same value:

```
public static final int SIZE = 4;

public static void main(String[] args) {
    topHalf();
    printBottom();
}

public static void topHalf() {
    for (int i = 1; i <= SIZE; i++) {     // OK
        ...
    }
}

public static void bottomHalf() {
    for (int i = SIZE; i >= 1; i--) {     // OK
        ...
    }
}
```

Syntax and Objects

Overview

- More new syntax
  - Arrays
  - Parameter passing
- Working with objects
  - Constructors
  - Constants

Java Arrays – The Basics

- Declaring an array

```
int[] myArray;
int[] myArray = new int[5];
String[] stringArray = new String[10];
String[] strings = new String[] {“one”, “two”};
```

- Checking an arrays length

```
int arrayLength = myArray.length;
```

- Looping over an array

```
for(int i=0; i<myArray.length; i++)
{
    String s = myArray[i];
}
```

Java Arrays – Bounds Checking

- Bounds checking
  - Java does this automatically. Impossible to go beyond the end of an array (unlike C/C++)
  - Automatically generates an `ArrayIndexOutOfBoundsException`

Java Arrays – Copying

- Don't copy arrays "by hand" (e.g. by looping over the array)
- The System class has an `arrayCopy` method to do this efficiently

```
int array1[] = new int[10];
int array2[] = new int[10];
//assume we add items to array1

//copy array1 into array2
System.arrayCopy(array1, 0, array2, 0, 10);
//copy last 5 elements in array1 into first 5 of array2
System.arrayCopy(array1, 5, array2, 0, 5);
```

Strings

- Strings are objects
- The compiler automatically replaces any string literal with an equivalent String object
  - E.g. `“my String”` becomes `new String(“my string”);`

Strings

- Strings have methods to manipulate their contents:

```
int length = someString.length();
String firstTwoLetters =
  someString.substring(0,2);
String upper = someString.toUpperCase();
boolean startsWithLetterA =
  someString.startsWith(“A”);
boolean containsOther =
  (someString.indexOf(otherString) != -1)
```

Passing Parameters

- Java has two ways of passing parameters
  - Pass by Reference
  - Pass by Value
- Pass by Value applies to primitive types
  - int, float, etc
- Pass by Reference applies to reference types
  - objects and arrays

Passing Parameters

```
public class PassByValueTest
{
  public void increment(int x)
  {
    x = x + 1;
  }

  public void test()
  {
    int x = 0;
    increment(x);
    //whats the value of x here?
  }
}
```

Passing Parameters

```
public class PassByReferenceTest
{
  public void reverse(StringBuffer buffer)
  {
    buffer.reverse();
  }

  public void test()
  {
    StringBuffer buffer = new StringBuffer(“Hello”);
    reverse(buffer);
    //what does buffer contain now?
  }
}
```

Initialising Objects

- Variables of a reference type have a special value before they are initialised
  - A “nothing” value called `null`
- Attempting to manipulate an object before its initialised will cause an error
  - A `NullPointerException`
- To properly initialise a reference type, we need to assign it a value by creating an object
  - Objects are created with the `new` operator

```
String someString = new String(“my String”);
```

Constructors

- `new` causes a *constructor* to be invoked
  - Constructor is a special method, used to initialise an object
  - Class often specifies several constructors (for flexibility)
  - `new` operator chooses right constructor based on parameters (*overloading*)
- Constructors can only be invoked by the `new` operator

Constructors – Example 1

```
public class MyClass
{
  private int x;
  public MyClass(int a)
  {
    x = a;
  }
}
```

We can then create an instance of MyClass as follows:

```
MyClass object = new MyClass(5);  //constructor is
  called
```

What are constructors for?

- Why do we use them?
  - Give us chance to ensure our objects are properly initialised
  - Can supply default values to member variables
  - Can accept parameters to allow an object to be customised
  - Can validate this data to ensure that the object is created correctly.
- A class *always* has at least one constructor
  - …even if you don’t define it, the compiler will
  - This is the *default constructor*

Destroying Objects

- No way to explicitly destroy an object
- Objects destroyed by the Garbage Collector
  - Once they go out of scope (I.e. no longer referenced by any variable)
- No way to reclaim memory, entirely under control of JVM
  - There is a `finalize` method, but its not guaranteed to be called (so pretty useless!)
  - Can request that the Garbage Collector can run, buts its free to ignore you

Modifiers

- Public/private are visibility modifiers
  - Used to indicate visibility of methods and attributes
- Java has a range of other modifiers
  - Control “ownership” of a method or attribute
  - Control when and how variable can be initialised
  - Control inheritance of methods (and whether they can be overridden by a sub-class)

Static

- `static` – indicates a *class variable* or *class method*. It's not owned by an individual object
  - This means we don't have to create an object to use it
  - `Arrays.sort` and `System.arrayCopy` are static methods

Static -- Example

```
public class MyClass
{
  public static void utilityMethod() { … }
  public void otherMethod() { … }
}

//using the above:
MyClass.utilityMethod();
MyClass objectOfMyClass = new MyClass();
objectOfMyClass.otherMethod();
objectOfMyClass.utilityMethod();

//this is illegal:
MyClass.otherMethod();
```

Final

- `final` – to make a variable that can have a single value
  - Can be assigned to *once and once only*
  - Useful to ensure a variable isn't changed once its assigned.

```
final int count;
count = 10;
//the following will cause an error
count = 20;
```

Defining Constants

- Unlike other languages, Java has no `const` keyword
- Must use a combination of modifiers to make a constant
  - `static` – to indicate its owned by the class
  - `final` – to make sure it can’t be changed (and initialise it when its declared)
- Naming convention for constants is to use all capitals
- Example…

Constants – An Example

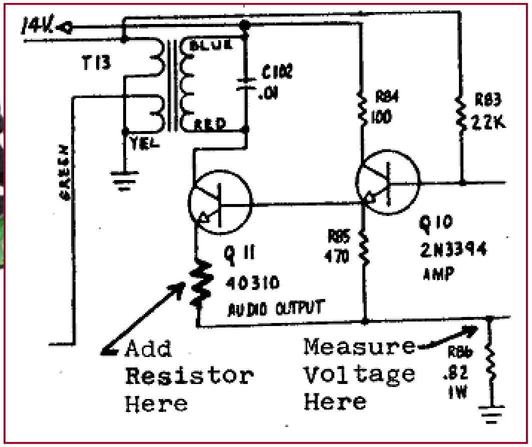
```
public class MyClass
{
  public static final int COUNT = 0;
  public static final boolean SWITCHED_ON = false;
}

//example usage:
if (MyClass.COUNT > 0) { … }

if (MyClass.SWITCHED_ON) {…}
```

Encapsulation

Encapsulation

- **encapsulation**: Hiding implementation details from clients.

  - Encapsulation forces *abstraction*.
    - separates external view (behavior) from internal view (state)
    - protects the integrity of an object's data



Private fields

A field that cannot be accessed from outside the class

```
private type name;
```

- Examples:

```
private int id;
private String name;
```

- Client code won't compile if it accesses private fields:

```
PointMain.java:11: x has private access in Point
System.out.println(p1.x);
                      ^
```

Accessing private state

```
// A "read-only" access to the x field ("accessor")
public int getX() {
    return x;
}

// Allows clients to change the x field ("mutator")
public void setX(int newX) {
    x = newX;
}
```

– Client code will look more like this:

```
System.out.println(p1.getX());
p1.setX(14);
```

Benefits of encapsulation

- Abstraction between object and clients

- Protects object from unwanted access
  - Example: Can't fraudulently increase an `Account`'s balance.

- Can constrain objects' state Example: Only allow `Account`s with non-negative balance.
  - Example: Only allow `Date`s with a month from 1-12.

The `this` keyword

- **`this`** : Refers to the implicit parameter inside your class.

  (a variable that stores the object on which a method is called)

– Refer to a field:	`this.`**field**
– Call a method:	`this.`**method**`(`**parameters**`);`
– One constructor can call another:	`this(`**parameters**`);`