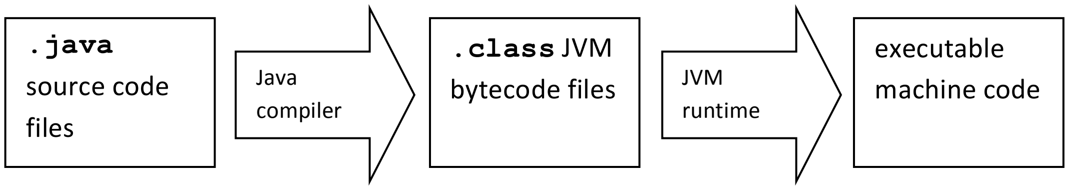
Java & OO SE 2 – Lecture #1 – Jeremy.Singer@glasgow.ac.uk

About the Java Programming Language

Java is an object-oriented, high-level programming language. It is a platform-neutral language, with a 'write once run anywhere' philosophy. This is supported by a virtual machine architecture called the Java Virtual Machine (JVM). Java source programs are compiled to JVM bytecode class files, which are converted to native machine code on platform-specific JVM instances.



Java is currently one of the top programming languages, according to most popularity metrics.[1] Since its introduction in the late 1990s, it has rapidly grown in importance due to its familiar programming syntax (C-like), good support for modularity, relatively safe features (e.g. garbage collection) and comprehensive library support.

Our First Java Program

It is traditional to write a 'hello world' program as a first step in a new language:

```
/**
 * a first example program to print Hello world
 */
public class Hello {
  public static void main(String [] args) {
    System.out.println("Hello world");
  }
}
```

Contrast with Python

Whereas Python programs are concise, Java programs appear verbose in comparison. Python has dynamic typing, but Java uses static typing. Python scripts are generally interpreted from source, whereas Java programs are compiled to bytecode then executed in a high-performance just-in-time native compiler.

[1] E.g. see http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

Supporting User Input in Simple Java Programs

There are two ways to receive text-based user input in simple programs like our 'hello world' example.

1) Users can pass command line arguments to the program, which are stored in consecutive elements of the String array that is the only parameter of the main method. Here is a simple example:

```
public static void main(String [] args) {
  System.out.println("Hello " + args[0]);
}
```

2) Programs can request input from the standard input stream using the Scanner library class[2]. The Scanner class will parse the input and return it (if possible) as a value of the appropriate type. Here is a simple example:

```
public static void main(String [] args) {
  java.util.Scanner scanner =
                  new java.util.Scanner(System.in);
  int i = scanner.nextInt();
  System.out.println("Come in, number " + i);
```

Questions

1) What does object-orientation mean?

2) So far we have come across at least five different Java types. How many can you spot in the source code examples above?

[2] Check out the Scanner library documentation at http://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html

Java & OO SE 2 – Lecture #2 – Jeremy.Singer@glasgow.ac.uk

Java Primitive Types

Java has eight *primitive* types defined and supported in the language and underlying virtual machine. These are shown in the table below.

Type	Description	Default initial value
`byte`	8-bit signed integer value	`0`
`short`	16-bit signed integer value	`0`
`int`	32-bit signed integer value	`0`
`long`	64-bit signed integer value	`0L`
`float`	32-bit single precision IEEE 754 floating point value	`+0.0F`
`double`	64-bit double precision IEEE 754 floating point value	`+0.0`
`boolean`	boolean value	`false`
`char`	16-bit Unicode character value	`\u0000`

Since Java is a statically typed language, types must be declared:

- local variables e.g. `int i;`
- class fields e.g. `class C { boolean b;}`
- method return and parameter values e.g. `public float getValue(long l) {…}`

Matching Values with Types

Match the following values with their corresponding primitive types from the table above.

```
false    1.3e-9   'a'    -256  256   1L  Double.POSITIVE_INFINITY
```

Integer Bases

An integer literal value may be expressed in decimal (default) or octal (leading `0`) or hex (leading `0x`). Integer values may be printed in these bases using the `System.out.printf()` method with the `%d`, `%o` and `%x` format specifiers respectively.

Java Identifiers

An *identifier* is the textual label for a named Java entity, such as a class, field, method, parameter, local variable… There are rules and conventions for identifiers.

The main rule is that an identifier must begin with a letter or an underscore, and must be at least one character long. Non-initial characters may be letters, numbers or underscores. Note that since Java supports the Unicode character set, letters are not restricted to the 26 characters of the Latin alphabet.

Conventions are not enforced by the Java compiler, but should be observed by careful programmers to make source code easier to understand. Coding standards and guidelines will specify different conventions. Near-universal conventions include:

- initial capital letter for class name, initial lower case for other identifiers

- multi-word identifiers are encoded in CamelCase e.g. `ArrayIndexOutOfBoundsException`
- constant values have all caps identifiers e.g. `Math.PI`

Simple For Loops

The Java `for` loop has the same semantics as in C. The three clauses in the parentheses are for initialization, termination condition check and post-loop-body update of the iteration variable(s). Thus:

```
for (int i=0; i<10; i++) { doSomething();}
```

is equivalent to:

```
int i; while (i<10) { doSomething(); i++;}
```

Below is a simple method that counts the number of vowel characters in a `String` object.

```
/**
 * count the number of vowel chars [aeiou] in a String
 * @arg s String to process
 * @return number of vowel chars in s
 */
public static int numVowels(String s) {
  int vowels = 0;
  for (int i=0; i<s.length(); i++) {
    char c = s.charAt(i);
    if (c=='a' ||
        c=='e' || …) {
      vowels++;
    }
  }
  return vowels;
}
```

Notice the use of `String` API (application programmer interface) methods `length()` and `charAt()`. We can investigate the full set of `String` methods via the online Java API[1].

Questions

1) Check out the `switch` control-flow construct for Java. Can you replace the above `if` statement to check for vowels with a switch?

2) Is `void` a primitive type in Java?

[1] Just google for **Java String API Oracle** and the appropriate webpage should be top of the search results.

Java & OO SE 2 – Lecture #3 – Jeremy.Singer@glasgow.ac.uk

Identifier Scope

In C-like languages, a block of statements is enclosed in curly braces { }. Generally all statements in the block have the same indentation level, but this is not mandatory (unlike Python). A local variable declared in a block is in scope (i.e. accessible) from the point of its declaration to the end of the enclosing block. For method parameters, the parameter is in scope until the end of the method. For iteration variables declared in a `for` loop initializer, the variables are in scope until the end of the loop body.

No two local variables or method parameters that are in scope can share the same identifier name. However a variable may have the same name as a package, type, method, field or statement label. Below is a pathological example, from the Java Language Specification (2nd ed, p113).

```
class a {
  a a(a a) {
    a:
    for(;;) {
      if (a.a(a) == a)
        break a;
    }
    return a;
  }
}
```

Switch statements

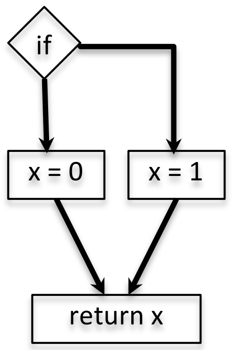
Whereas an `if` statement evaluates a `boolean` expression and conditionally executes one of two statements/blocks, on the other hand a `switch` statement evaluates an integer expression (`byte`, `short`, `int`, `long`, `char`, or `enum`) or a `String` (Java 7+) and conditionally executes one or more of the multiple `case` blocks.

In the example source code below, note the use of a catch-all `default` case as the last case block in the `switch` construct. This is recommended unless all the possible values are covered by explicitly labeled cases. Case labels must be constant expressions. Note the use of `break` statements to prevent fall-through from one case to another (except where the cases share a code block).

```
/**
 * lookup scrabble points for a single letter
 * @arg c the letter to lookup
 * @return the scrabble score for this letter
 */
public static int letterScore(char c) {
  int score;
  switch (c) {
    case 'z':
      score = 10;
      break;
    case 'x':
      score = 8;
      break;
    // …
    case 'g':
    case 'd':
      score = 2;
      break;
    default:
      score = 1;
      break;
  }
  return score;
}
```

Representing Control Flow with UML *Activity Diagrams*

Schematic diagrams like UML activity diagrams are often used to show complex control flow of programs. Diamond nodes indicate conditional tests (if or switch statements, or loop conditions). Square nodes indicate single statements of blocks of consecutive statements. Arrows indicate flow of control from one region to another.



Advanced Loop Control Flow

We have already looked at `for` and `while` loops. The `break` and `continue` statements[1] can be used inside these loop bodies to direct control flow explicitly. A `break` is used to exit a loop entirely. A `continue` is used to skip to the end of the current iteration and commence the next iteration (if any).

```
  for (int i=0; i<args.length; i++) {
    if (args[i].equals("needle") {
      System.out.println("found needle in haystack");
      break;
    }
  }
```

[1] See the helpful official documentation at http://docs.oracle.com/javase/tutorial/java/nutsandbolts/branch.html for more details.

Java & OO SE 2 – Lecture #4 – Jeremy.Singer@glasgow.ac.uk

Type Conversions

Since Java is a statically typed language[1], a declared variable can only hold values of a single, specific type. In order to store a value of type t_1 in a variable of type t_2, the value must be converted to type t_2 before the assignment occurs. Some type conversions are implicit, i.e. the programmer does not need to indicate the conversion in the source code. These are generally *widening* conversions, where little or no information is lost. Example widening conversions include `byte` to `long`, or `int` to `double`.

When a type conversion would result in significant potential loss of information, e.g. `double` to `float` or `int` to `short`, this is known as a *narrowing* conversion. In such cases, the conversion must be made explicit using a type-cast operator which specifies the target type in brackets. For example:

```
int i = 1025;
byte b = (byte)i;   // b has value 1
```

Floating-point to integer type conversions use the *round to zero* convention if the floating-point value is in the representable range of the target integer type. For –ve and +ve numbers that are too large in magnitude to represent, the `MIN_VALUE` or the `MAX_VALUE` of the integer type is selected, respectively. The Java Language Specification gives full details of the type conversion rules[2].

It is worth mentioning one other kind of conversion, from `String` objects to primitive values. Each of the primitive wrapper classes has a static method to convert from a String to a primitive value of that type. For instance, `Integer.parseInt("42")` will return the value 42 of type `int`. Consider the program below, which takes a sequence of integers from the program arguments and sums the positive integers until a 0 value appears, or there are no more arguments.

```
public static void main(String [] args) {
  int i, sum = 0;
  try {
    for (i=0; i<args.length; i++) {
      int value = Integer.parseInt(args[i]);
      if (value==0) break;
      sum += value;
    }
    System.out.printf("Sum of first %d args is %d\n", i, sum);
  }
  catch (NumberFormatException e) {
    // error in parsing
  }
}
```

[1] In contrast to dynamically typed languages, such as Python, Ruby and Javascript.
[2] http://docs.oracle.com/javase/specs/jls/se7/html/jls-5.html

Constant Values

The Java `final` modifier indicates that the relevant entity (for now, we only consider class variables and local variables) is a constant. Constant class variables are useful values for general calculations, for example `Math.E` and `Math.PI`. (Note that constant class variables generally have all-caps identifiers.) Constant local variables are useful to indicate values that should not change after their initial assignment, e.g. the length of an array or a `String` can be stored in a `final` variable. The use of `final` is encouraged[3] because it makes source code easier to read and also to optimize.

Math Library Methods

The Java Math library has some useful static methods for numeric calculations. These include trigonometric functions like `Math.sin()`, simple utility functions like `Math.pow()`, etc. Check out the full documentation at http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html. Note that most of these methods operate on double values.

One particularly useful method is `Math.random()` which returns a pseudo-random `double` value, in the range [0.0, 1.0). Values are uniformly distributed in this range. So, to get an `int` value in the range [1,100], the code would look like:

```
int i = (int)(1 + Math.random()*100);
```

Fun Task

Write a Java program that computes a 'secret' random number in the range 1 to 20. The program prompts the user for guess values. For each guess, the program outputs HIGHER, LOWER or CORRECT. Until the guess is CORRECT, the user is prompted for another guess.

Hints: use the random number generation method above. Use `Scanner.nextInt()` on `System.in` to acquire user input. Use `System.out.println()` to output messages to the user. A `while` loop (or a `do/while`) is appropriate for the control flow.

Question

1) Should method parameters be marked as `final`? If so, why? If not, why not?

[3] http://www.javapractices.com/topic/TopicAction.do?Id=23

Java & OO SE 2 – Lecture #5 – Jeremy.Singer@glasgow.ac.uk

Classes and Objects

Classes are *types* and objects are *instances* of types. Compare with the `boolean` primitive type, which has `true` and `false` instance values. An example class is `Planet`, with objects such as `mercury`, `venus`, or `skaro`. A class contains *members*, which are either data *fields* or *methods*. The data fields store state that describes some attributes of the object. The methods represent behaviour that processes and transforms the object state. Effectively a class is an abstract description of a set of real-world entities. (This is the object-oriented principle of *abstraction*.) Each instance of a class has data and behaviour associated with it. (This is the object-oriented principle of *encapsulation*.)

Example Case Study

Below is a simple class representing a bank account.

```
public class BankAccount {
  int balance;
  String name;
  int id;
  static int nextId = 0;

  void deposit(int value) { this.balance += value; }
  void withdraw(int value) { this.balance -= value; }
  BankAccount(String name, int initialAmount) {
    this.name = name;
    this.balance = initialAmount;
    this.id = BankAccount.nextId++;
  }
}
```

Object Instantiation

The *constructor* for an object looks like a method that has the same name as the class. The constructor sets up initial values for the data fields to initialize the object state. If no constructor is explicitly defined, then a default *no-args* constructor is automatically included. To invoke the constructor, use the `new` keyword in Java. e.g. `BankAccount b = new BankAccount("test", 0);`

Static members

A static member is associated with a class, rather than with any object created from that class. So for static data fields, there is exactly *one* variable, no matter how many objects of the class have been instantiated. A static method performs a general task for the class, rather than for any specific object. A static method can only access static variables and call static methods in the class. A static method is generally invoked via the class name, rather than via an object reference. e.g. `BankAccount.setInterestRate(0.5);`

Inherited Methods

All objects inherit some methods from the class at the root of the inheritance tree, which is `java.lang.Object`[1]. One such method is `toString()`, which generates a String representation of the object. By default, this String displays the name of the class type and the address in memory of the object. However you can *override* this behaviour by supplying a custom `toString()` definition. Another method inherited by all objects is `equals()` which compares two objects for equality and returns a `boolean` value. Note that the == operator implements value equality for primitives, and reference equality for objects (i.e. an object is only equal to itself). A custom `equals()` method allows us to implement some kind of value equality for objects of a specific class.

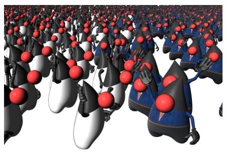
Here is an example equality test for `BankAccount` objects. We assume that if two `BankAccount` objects have equal `int` ids then the corresponding accounts are equal.

```
public boolean equals(Object o) {
  boolean equal = false;
  if (o instanceof BankAccount) {
    BankAccount b = (BankAccount)o;
    equal = (this.id == b.id);
  }
  return equal;
}
```

Questions

For the `BankAccount` class as defined above, how do we stop client code from resetting the `nextId` static field to allow multiple accounts to share the same id? Also, how do we stop client code from directly modifying the id fields of individual `BankAccount` instances?

[1] See http://docs.oracle.com/javase/tutorial/java/IandI/objectclass.html for details.



Java & OO SE 2 – Lecture #6 – Jeremy.Singer@glasgow.ac.uk

Member Visibility Modifiers

In order to limit the visibility of class members, i.e. fields and methods, it is possible to specify an access modifier as part of a member declaration. The table below shows the extent of visibility for members with the various modifiers.

Modifier	Same class	Same package	Any subclass	Any class
`public`	✓	✓	✓	✓
`protected`	✓	✓	✓	
(default)	✓	✓		
`private`	✓			

`private` data fields are used for internal class state that must be accessed in a controlled way, perhaps through getter and setter methods. See example below.

```
public class Person {
  private int age;

  public int getAge() {
    return this.age;
  }

  public void setAge(int age) {
    assert(age>=0);  // age must be non-negative
    this.age = age;
  }
}
```

`private` methods are used for non-API methods that are utility/helper methods internal to a class.

Class Inheritance

Some classes are related to each other via an inheritance hierarchy. More general classes will have characteristics in common with more specialized classes. A class B can be defined as a subclass of class A, in which case B inherits the members of A. Effectively B ***is-a*** specialized version of A, or an extension of A[1]. Subclasses are declared in Java using the `extends` keyword i.e.

```
public class B extends A { … }
```

This notion of class inheritance is one of the most powerful object-oriented features of Java. The Java language supports single inheritance (rather than multiple inheritance like C++). This means that the

[1] See http://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html for more details.

Java inheritance hierarchy is a tree rather than a directed acyclic graph. The root of the Java inheritance hierarchy is the `java.lang.Object` class.

Method Overriding

If a subclass has a method with an identical signature (name, return type and parameter types) as a superclass, then the subclass method is said to ***override*** the superclass method. Effectively, this is the way that the subclass specifies alternative behaviour to the superclass. See the example below.

```
public class Person {
  private Gender g;
  public String getTitle() {
    String title;
    if (g==Gender.MALE) title = "Mr.";
    else title = "Ms.";
    return title;
  }
}

public class TitledPerson extends Person {
  private String title;
  public String getTitle() {
    return this.title;
  }
}
```

Polymorphism

Polymorphism[2] literally means 'many forms'. It means that wherever an instance of class `A` is expected in a program, one may supply an instance of class `B` which is a subclass of `A`. This is an application of the *Liskov substitution principle*[3]*.* Polymorphism is supported by virtual method invocation in Java – method calls are dynamically dispatched based on the runtime type of the receiver object.

Questions

1) Can `static` methods be overridden in the same way as instance methods? If so, why? If not, why not?
2) What is the point of a class with only `private` constructors?

[2] See http://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html for details.
[3] This is starting to get into hard-core CS theory, see http://c2.com/cgi/wiki?LiskovSubstitutionPrinciple if interested.

Java & OO SE 2 – Lecture #7 – Jeremy.Singer@glasgow.ac.uk

Static Methods

Recall that `static` methods are associated with a class rather than any particular instance. These `static` methods are generally utility methods. Examples include:

- `Math.random()` which returns a `double` value in the range [0,1)
- `System.exit(int status)` which terminates all threads and aborts the running JVM
- `Integer.parseInt(String s)` which tries to interpret the parameter `s` as a 32-bit integer value

Exceptions

When errors occur in program execution, `Exception` objects are *thrown*. All `Exception` objects belong to classes that are subclasses of `java.lang.Exception`[1]. Some `Exception` objects are subclasses of `RuntimeException` – these are *unchecked*. All other `Exception`s are *checked*, and if they may be thrown then they must be caught or declared in the enclosing method's `throws` clause.

An example of a checked exception is `FileNotFoundException`. An example of an unchecked exception is `ArrayIndexOutOfBoundsException`.

It is possible to instantiate and `throw` exceptions directly in your own code, i.e.

```
    throw new Exception();
```

Customized exceptions can be created – either (1) by supplying an error message `String` in the `Exception` constructor (the String can be retrieved via the `Exception.getMessage()` instance method) – or (2) by extending the `Exception` class and possibly adding new instance fields.

Handling Exceptions

A `try` block should enclose code that may throw an `Exception` instance. A `try` block may be followed by one or more `catch` blocks, each of which takes a single `Exception` parameter. The `catch` blocks are evaluated in sequential order, and the first `catch` block whose parameter type matches the thrown exception is executed. A `try` block may also be associated with a `finally` block, which is executed either after the non-exceptional exit from the `try` block, or after any matching `catch` block has been executed. Example source code is shown below:

[1] See http://www.oracle.com/technetwork/articles/entarch/effective-exceptions-092345.html for a discussion of Exceptions in Java.

```
try { …
}
catch (Exception e) { …
}
finally { …
}
```

Abstract Classes and Methods

Some superclasses have 'holes' in them, which subclasses can 'fill in' when they extend the superclass. The `holey' superclasses are marked as `abstract` classes, which have `abstract` methods declared in them. The `abstract` class only defines a partial implementation. An `abstract` class cannot be instantiated. An `abstract` method only has a signature and no method body, thus it cannot be called. A subclass of an `abstract` class must supply an implementation for the inherited `abstract` methods, or the subclass itself must be marked as `abstract`.

The `abstract` method mechanism is a way to enforce that subclasses conform to a particular API. An example is shown below. All subclasses of `TwoDimensionalPoint` must implement the `distanceToOrigin()` method.

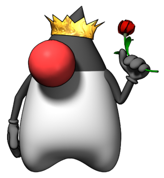
```
public abstract class TwoDimensionalPoint {
  double x;
  double y;
  public abstract double distanceToOrigin();
}

public class CartesianPoint extends TwoDimensionalPoint {
  public double distanceToOrigin() {
    return Math.sqrt(x*x+y*y);
  }
}

public class ManhattanPoint extends TwoDimensionalPoint {
  public double distanceToOrigin() {
    return Math.abs(x) + Math.abs(y);
  }
}
```

Questions

1) Can an `abstract` class have constructors? If so, why? If not, why not?
2) What is the relationship between an `abstract` class and an `interface`?



Java & OO SE 2 – Lecture #8 – Jeremy.Singer@glasgow.ac.uk

Constructor Chaining

In a constructor body, the first action must be to call a superclass constructor. If there is no explicit superclass constructor call, then the compiler inserts a default no-args constructor to the superclass, i.e. `super()`. Every time a constructor is invoked, there is a chain of constructor calls going up the inheritance hierarchy all the way back to `java.lang.Object`. We can see this by inserting `println` statements into a set of constructors:

```
public class A {
  public A() { /*super();*/ System.out.println("A constructor"); }
}
public class B extends A {
  public B() { /*super();*/ System.out.println("B constructor"); }
}
public class C extends B {
  public C() { /*super();*/ System.out.println("C constructor"); }
}
```

If there is not a no-args constructor in the superclass, then the subclass constructor must specify *explicitly* which superclass constructor is to be called.

Calling Superclass Methods

In a similar way to superclass constructor invocation, it is possible to invoke a method from the superclass that is overridden in a subclass, using the `super` pseudo-variable[1]. The `super` variable is a reference to the current instance, with the type of its immediate superclass in the inheritance hierarchy. Invoking a method through the `super` reference is not subject to polymorphic overriding (unlike method invocation via the `this` reference.)

Leaves on the Inheritance Tree

In some cases, a developer may not want a class to be subclassed. If the class is marked as `final`, then it cannot be subclassed. Similarly, if a method is marked as `final`, then it cannot be overridden in a subclass. `final` classes and methods can improve security[2] (or predictability) – a developer can be certain that an instance of a `final` class does what is expected, rather than any overriding behaviour. In the code below, marking the PasswordChecker class as `final` (or the check method) would prevent subclass injection attacks.

[1] See http://docs.oracle.com/javase/tutorial/java/IandI/super.html for more details about super.
[2] See http://www.oracle.com/technetwork/java/seccodeguide-139067.html#4 for attacks and corresponding defence techniques.

```
public class PasswordChecker {
  public boolean check(String username, String password) {
    String passwordHash = hash(password);
    String correctHash = lookupHash(username);
    return (passwordHash.equals(correctHash));
  }
}

public class DodgyChecker {
  public boolean check(String username, String password) {
    return true;
  }
}
```

More on Exceptions

Recall that when an `Exception` is thrown in a `try` block, the associated `catch` blocks are examined in sequential order and only the *first* matching `catch` block (if any) is executed. This means that `catch` blocks should be ordered from least general to most general. The Java compiler will complain about unreachable code if more general `catch` blocks (e.g. `catch (Exception e){}`) are positioned above less general catch blocks.

Three useful methods in Exception objects are:

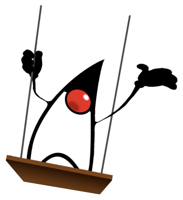
- `e.getMessage()` – returns a String with some information about the exception
- `e.printStackTrace()` – prints out the calling context of the exception at the point it was thrown
- `e.toString()` – generally returns a String indicating the concrete type of the Exception instance

`Exception` messages may be printed to the `System.err PrintStream`, rather than the usual `System.out PrintStream`.

Questions

How would you create a constructor for class Foo that creates an exact copy of another instance of Foo? See the helpful Java Practices website[3] for more details.

[3] http://www.javapractices.com/topic/TopicAction.do?Id=12



Java & OO SE 2 – Lecture #9 – Jeremy.Singer@glasgow.ac.uk

Java Arrays

An array is a fixed length sequence of consecutive memory locations, indexed by an integer subscript. Arrays are supported directly by the underlying Java Virtual Machine, so they are efficient to use.

Declaring Array Types

Each array has a *type*, which specifies the type of the individual elements and the dimensionality of the array. For example, `int[]` is a one-dimensional `int` array and `String[][]` is a two-dimensional `String` array. Element types may be Java primitive types or Object (reference) types.

When an array is declared (perhaps as a method parameter, a local variable or a class member) it is given a *name*. The name either comes after the type (i.e. `String [] args`) or is inserted within the type (i.e. `String args[]`). This latter form is a C-style hangover.

Initializing Array Values

An array declaration does not reserve space for the array elements, or specify the length of the array. Instead it only declares a reference to the (currently uncreated) array. This means that the uninitialized array reference is a `null` pointer value. The array may be created via a call to `new` or with an explicit initializer.

```
int [] a = new int[10];

String [] as = { "each", "peach", "pear", "plum" };
```

Subscripting Array References

Once the array has been created, array elements can be indexed via integer subscripts, e.g. `a[3]`, `as[1]`. Subscripts start from 0 (unlike Fortran, COBOL or Matlab). The maximum allowable subscript is slightly less than `Integer.MAX_VALUE`. However if a subscript is greater than or equal to the length of the array, then an `ArrayIndexOutOfBoundsException` unchecked Java exception is thrown at runtime.

The length of an array is constant, stored in a field of that name, e.g. `a.length`, `as.length`. Note that for `String` objects, `length()` is a method whereas for arrays, `length` is a field.

Iterating over Arrays

The standard idiom for iterating over an array is to use a `for` loop.

```
for (int i=0; i<a.length; i++) {

  a[i] = …;

}
```

An alternative, more concise, notation is to use the for-each loop idiom, in cases where the array indexing does not need to be explicit.

```
for (String s: as) {
  System.out.println(s);
}
```

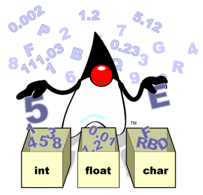
Helper Methods for Arrays

Since an array is effectively an object in Java, it inherits all the methods from `java.lang.Object`. The `java.util.Arrays`[1] class contains a set of static helper methods for array manipulation, including `Arrays.toString()` and `Arrays.fill()`.

Example Array Code

In the lecture, we will compute values for elements in Pascal's triangle and store these in a two-dimensional int array.

[1] See http://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html



Java & OO SE 2 – Lecture #10 – Jeremy.Singer@glasgow.ac.uk

Java Collections

The Collections framework[1] is a Java standard library. Each Collections class supports the processing of a series of related elements in a common data structure. In this course, we are going to consider the `java.util.ArrayList` class, but there are other kinds of Collections including `Set`, `Stack` and `HashMap`. You will explore some of these data structures in the ADS2 course next semester. The advantage of the Collections framework is that it provides a standardized, reusable implementation for these common data structures.

The Collections framework forms an object-oriented class hierarchy. The base class[2] is `java.util.Collection`[3] which defines methods that all Collections must implement including `add()`, `remove()`, `contains()`, `size()` and `toArray()`.

The ArrayList Data Structure

The major limitation of Java arrays is that they have a fixed length. The `java.util.ArrayList`[4] class is a more flexible (although correspondingly less efficient) Collections class that implements variable length arrays. The backing array grows and shrinks dynamically as elements are added to an `ArrayList` instance.

`ArrayList` operations are invoked by method calls, rather than by built-in Java syntax (unlike arrays). Below is an example source code snippet:

```
ArrayList<Integer> nums = new ArrayList<Integer>();
nums.add(1);
nums.add(1);
int i = 2;
int fib = 1;
while (fib < LIMIT && nums.size() < SIZE_LIMIT) {
  fib = nums.get(i-1) + nums.get(i-2);
  nums.add(fib);
  i++;
}
```

Elements can be removed from an `ArrayList` using the `remove()` call. If an element is removed from the middle of the list, elements to the right are shuffled down.

[1] http://docs.oracle.com/javase/tutorial/collections/
[2] Actually an interface, but we haven't learnt about these yet.
[3] http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html
[4] See http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html

Note that `ArrayList` structures can be converted to arrays using the `toArray()` method, and vice versa using the `java.util.Arrays`[5] helper methods.

Iterating over Collections

Collections are *Iterable*, which means that we can use a `for-each` loop to iterate over Collection elements.

```
ArrayList<String> words = new ArrayList<String>();
nums.add("antidisestablishmentarianism");
nums.add("monosyllabic");
int totalChars = 0;
for (String word : words) {
    totalChars += word.length()
}
System.out.printf("total number of characters: %d\n",
                  totalChars);
```

Generic Types

Collection classes are type-parameterized. The type specified in angle brackets after the Collection class name specifies the type of the elements stored in that Collection. This makes the Collections classes generic, in that they can be used with any type of element. The most general element is `java.lang.Object`. Note that subclasses of the specified element type are also valid element types. The element type is specified for the declaration and the construction of the Collection – e.g.

```
        ArrayList<Object> objects = new ArrayList<Object>();
```

Wrapper Classes

Only object references (i.e. pointers to objects in the heap) can be stored as elements in a Collection. Therefore primitive values cannot be stored directly as Collection elements. To get around this, Java uses wrapper classes for each primitive. `java.lang.Integer` is the wrapper class for the `int` primitive type.

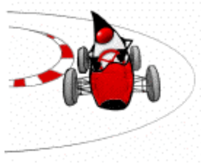
```
int iPrim = 42;
Integer iWrap = new Integer(iPrim);
int x = iWrap.intValue();
```

Question

Strings and Wrapper classes are immutable. What does this mean? Why is it useful?

[5] http://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html

Java & OO SE 2 – Lecture #11 – Jeremy.Singer@glasgow.ac.uk



Generic Classes

As we saw last week, the Java Collections Framework uses *type parameters* to allow the data structures to be specialized for particular element types, e.g. `ArrayList<String>`. It is also possible to define your own generic classes[1] with type parameters. In the example below, we will define a `Pair` class with a single type parameter *T*. Note how *T* is specified in angle brackets at the start of the class declaration. Then this type parameter can be used as a reference type within the scope of the class body.

```
public class Pair<T> {
  private T first;
  private T second;

  public T getFirst() {
    return this.first;
  }

  public void setFirst(T first) {
    this.first = first;
  }
}
```

Note that type parameters may be *constrained* in terms of the object-oriented inheritance hierarchy[2]. For instance, suppose the `Pair` generic class above should only be allowed to store `java.lang.Number`[3] types, we would specify this as:

```
public class Pair<T extends Number> {…}
```

Generics are a compile-time feature of Java. They are useful for compile-time type checking. However generics are *erased* before runtime. This has several important implications:

- `static` members are shared across all specialized versions of a generic class
- there is no way of distinguishing between types of generic classes using `instanceof` or Java reflection facilities at runtime – FIXME – check this!!

[1] See http://docs.oracle.com/javase/tutorial/java/generics/types.html for full details
[2] See http://docs.oracle.com/javase/tutorial/java/generics/bounded.html for details
[3] `Number` is a superclass of the library numeric types, see http://docs.oracle.com/javase/7/docs/api/java/lang/Number.html

Packages in Java

Java packages[4] are units of modularity. A package is used to group together a set of related resources (generally Java classes). Use the `package` keyword at the top of a Java source code file to specify the package to which a class belongs. Generally, a class in package `foo` should be stored in directory `foo` on the filesystem. If no package is specified, then the class belongs to the default package, which is the current working directory.

A fully-qualified classname includes its package, e.g. `java.lang.String` or `java.util.ArrayList`. However a class may be referred to without its package name if the `import` statement is used. This statement opens the namespace of the imported package to the current scope.

```
package a;
public class A { … }
```

```
package b;
public class B {  A.a … }
```

```
package c;
import a.A;
import b.B;
public class C { A … B … }
```

Classes in the current package do not require their fully-qualified names. Also the `java.lang` package is imported implicitly.

Package Naming Conventions

In theory, every Java class defined by any software developer should have a globally unique name. To accomplish this, there is a standard convention[5] for naming packages. Developers use their associated internet domain name in reverse form, followed by a locally unique suffix. So for instance, for JP2 example programs, I might use the package `uk.ac.glasgow.dcs.joose2`

Questions

To which packages do the following classes belong?

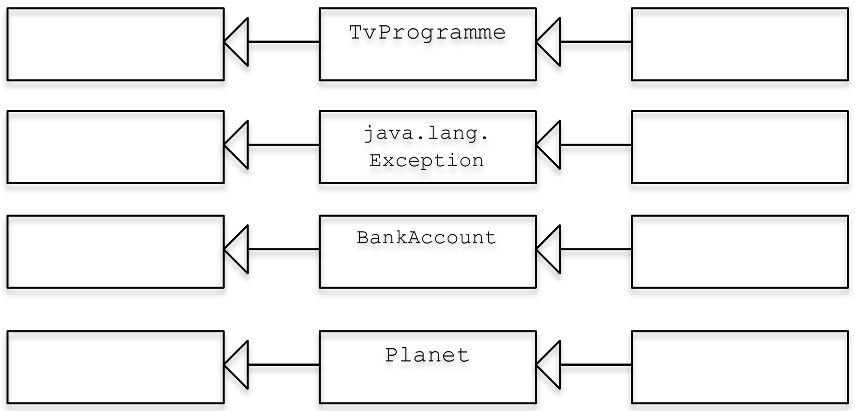
1. `Scanner`
2. `FileNotFoundException`
3. `List`
4. `Boolean`

[4] See http://docs.oracle.com/javase/tutorial/java/package/ for a good tutorial on packages
[5] See http://en.wikipedia.org/wiki/Java_package#Package_naming_conventions

Java & OO SE 2 – Lecture #12 – Jeremy.Singer@glasgow.ac.uk

Object-Oriented Class Hierarchies

Below are some example class hierarchies, with most general on the left, and most specific on the right.

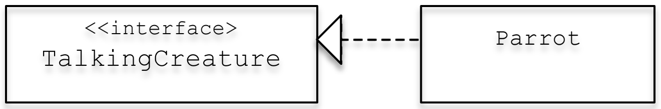


Motivating the need for Interfaces

A subclass specializes some feature of its superclass, as demonstrated above. However sometimes there are class features which run orthogonal to the inheritance hierarchy. For instance, `Human` and `Parrot` objects can both `speak()`, but in a typical inheritance hierarchy, they would not have a common superclass (other than `Vertebrate`, which does not have the `speak()` method since most other animals with a backbone are unable to talk).

The problem is that we want some classes *to inherit behaviour from multiple parent classes*. `Human` should be a subclass of both `Primate` and `TalkingCreature`. `Parrot` should be a subclass of both `Bird` and `TalkingCreature`.

The solution in Java[1] is to use *interfaces* to encapsulate these relationships that are orthogonal to the main inheritance hierarchy. An interface specifies a number of abstract methods (i.e. method signatures but no bodies). A class that `implements` an interface is obliged to provide an overriding method definition for the abstract methods inherited from the interface (unless the class is declared as abstract). Effectively, an interface is a form of *contract* that implementing classes must honour.



[1] More clunky solutions to this problem (e.g. C++) include *multiple inheritance*. More elegant solutions (e.g. Scala) include *traits* or *mixins*.

```
interface TalkingAnimal {
  void speak(String s);
}
```

```
public class Human extends Primate
                        implements TalkingAnimal {
  public void speak(String s) {
    // vocal chord vibrations…
  }
}
```

```
public class Parrot extends Bird
                         implements TalkingAnimal {
  public void speak(String s) {
    // stretch trachea and whistle…
  }
}
```

Note that a class may only extend one superclass, but it may implement many interfaces. Also, interfaces may extend other interfaces. Interfaces should only contain method signatures, which are implicitly `public` and `abstract`, and constant valued fields, which are explicitly `static` and `final`.

The Comparable Interface

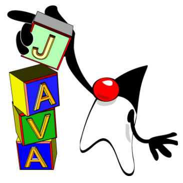
The Java standard library includes an interface `java.lang.Comparable<T>`[2] which requires implementing classes to provide a single method `compareTo()`. This interface enables the correct behaviour of the generic `java.util.Collections.sort()`[3] method.

```
public class Country implements Comparable<Country> {
  String name;
  int population; // in millions
  public int compareTo(Country other) {
    return (this.population-other.population);
  }
}
```

```
ArrayList<Country> cl = new ArrayList<Country<>();
cl.add(new Country("USA", 300);
cl.add(new Country("Scotland", 5);
cl.add(new Country("China", 1300);
Collections.sort(cl);
```

[2] See http://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html
[3] See http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html

Java & OO SE 2 – Lecture #13 – Jeremy.Singer@glasgow.ac.uk



JVM Memory Layout

Runtime memory in the Java virtual machine is organized into distinct areas[1] as you know from CS2.

The **stack**[2] is used to store local variables that belong to a single method. When the method is invoked, a stack frame for that method is pushed onto the stack. When the method returns, the corresponding stack frame is popped from the stack. This means that local variables are only alive from the time the method is called until the time the method returns.

The **heap** is used to store data that is dynamically allocated via the `new` keyword, such as objects and arrays. Heap-allocated data remains alive so long as it is reachable from a root reference (e.g. a local variable on a stack frame or a global variable in a static field). When heap-allocated data is no longer reachable, it may be garbage collected. Garbage collection is triggered when the heap fills up, or when the program invokes `System.gc()`[3]. If there is not enough space in the heap for new data to be allocated, then the virtual machine throws an `OutOfMemoryError`.

Each class file contains a **constant pool** area, which stores compile-time constants such as string literals and integer values. Also runtime linking information (method names, etc) is stored in the constant pool. When classes are loaded at runtime, the constant pool information is copied into the virtual machine constant pool area which is shared between all runtime threads.

Inserting into ArrayLists

Last week we saw that we could compare objects that implement the `Comparable` interface. Let's use this to insert objects into an `ArrayList` to maintain a sorted order. Effectively, this is the basis for the *insertion sort* algorithm which you will learn more about in ADS2 next semester.

```
list = new ArrayList<Comparable>();
for (String arg : args) {
  boolean inserted = false;
  for (int i=0; i<list.size() && !inserted; i++) {
    if (list.get(i).compareTo(arg)>0) {
      // first list element that's greater than arg
      list.add(i, arg);
      inserted = true;
    }
  }
  if (!inserted) {
    // add arg to end of list (largest element)
    list.add(arg);
  }
}
```

[1] See http://docs.oracle.com/javase/specs/jvms/se5.0/html/Overview.doc.html#1732 for full details.
[2] In fact there is one stack per thread, but we will not consider multi-threaded code just yet.
[3] See http://docs.oracle.com/javase/7/docs/api/java/lang/System.html

Functional Operations on Lists

Imagine you want to compute a sum of squares for a list of integers. Using the Java idioms we know already, the code might like similar to:

```
int sum = 0;
for (int i : list) {
  squareList.put(i*i);
}
for (int square : squareList) {
  sum += square;
}
```

It would be possible to fuse the two loops into a single loop body, although the code might look less clear and an optimizing compiler will probably do this anyway. In either case, the code looks messy. This kind of map / reduce computation is better expressed using a functional idiom.

The latest version of Java (i.e. Java 8) supports lambdas, which are anonymous functions that can be applied to streams of data. An anonymous function may be recognized by the right arrow syntax ->. The parameter or bracketed list of parameters is on the left hand side. The result expression or block is on the right hand side. The above sum-of-squares example might be rewritten as:

```
int sum = list.stream()
                 .map(x -> x*x)
                 .reduce((x,y) -> x+y)
                 .getAsInt();
```

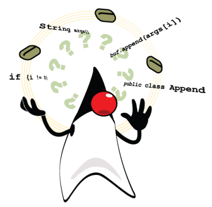
This code is simpler and more intuitive in every way. The programmer intention is explicitly seen in the program. This code is functional, elegant and parallelizable. Java 8 is available now[4] but may not be widely adopted yet. Other JVM languages (e.g. Scala[5], Clojure, Groovy) also have extensive support for functional operations on list-like data structures.

Question

What are the key differences between Collections objects and Streams objects?

[4] Download from http://java.com/en/download/manual.jsp
[5] Try for yourself at http://www.simplyscala.com/ or http://tryclj.com or https://groovyconsole.appspot.com/

Java & OO SE 2 – Lecture #14 – Jeremy.Singer@glasgow.ac.uk



Immutable Data Types

Once an immutable object[1] has been constructed, its internal state cannot be modified. Examples from the Java standard libraries include `String` and primitive wrapper classes like `Integer`. Operations on `String` objects like concatenation actually return newly constructed `String` objects and leave the original objects unmodified.[2] There are several key benefits with immutability:

- Immutable objects can be safely shared between threads or data structures.
- Deduplication optimizations can save memory. Two immutable objects with the same field values are effectively indistinguishable and can be mapped onto the same object at runtime.
- Immutable objects are ideal lookup values (keys) in `Map` data structures like hashtables.

Implementing Immutability

To define an immutable data type, all the instance fields need to be `private` and have associated getter methods but no setters. The constructor must set up all the internal state for the object, which cannot be subsequently modified. If any of the instance fields refer to mutable objects, then the associated getter should return a reference to a copy, rather than the original field. Look at the `Person` example below – the names field refers to an `ArrayList`, which is a mutable object so the underlying reference should not be returned directly.

```
public class Person {
  private ArrayList<String> names;

  // check out this interesting varargs syntax!
  public Person(String… names) {
    this.names = new ArrayList<String>();
    for (String name: names) {
      this.names.add(name);
    }
  }
  // returns a copy of the names list, not the
  // underlying reference
  public List<String> getNames() {
    return (List)(this.names.clone());
  }
}
```

[1] See http://www.javapractices.com/topic/TopicAction.do?Id=29 for full details on immutability

[2] This means the use of repeated `String` concatenation to build up a compound `String` is highly inefficient – it is better to use mutable objects like `StringBuilder` – see later.

Copying Objects

There are two ways to create copies of existing objects: copy constructors or clone methods. A copy constructor[3] for a class takes a single parameter, which has the same type as the class. The constructor simply copies the values of the fields from the supplied object into the new object. See the Pair example below.

```
public class Pair<T> {
  private T first;
  private T second;
  // copy constructor
  public Pair(Pair<T> other) {
    this.first = other.first;
    this.second = other.second;
  }
}
```

The clone() method is a general way of copying Objects. An object may only be cloned if its class implements the Clonable marker interface. All objects inherit a clone() method from java.lang.Object - but invoking this method on a non-Clonable object will throw the CloneNotSupportedException.

The default clone operation simply instantiates a new object of the appropriate type, and copies the values in the fields across to this new object. This is similar to the copy constructor outlined above. However sometimes this *shallow cloning* is insufficient. If a reference to a mutable object is copied in this way, then the two object will share this reference. In such cases, deep cloning is required. A *deep clone* requires overriding the inherited clone method with a custom method that creates a new object and copies/clones instance fields as appropriate.

Cloning is not recommended as good practice by many Java developers[4] – use of copy constructors appears to be more widely approved and supported.

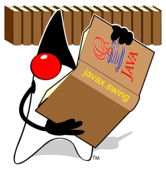
Mutating Strings

As outlined above, since Strings are immutable then repeated concatenation is inefficient. It is better to use a mutable class such as a StringBuilder[5] to update a character string representation.

```
StringBuilder [] bs = { new StringBuilder(“man”),
                        new StringBuilder(“califragilistic”) };
for (StringBuilder b : bs) {
  b.insert(0, “super”);
}
bs[1].append(“expialidocious”);
```

[3] See http://www.javapractices.com/topic/TopicAction.do?Id=12 for more details
[4] See http://en.wikipedia.org/wiki/Object_copy#In_Java for some reasons.
[5] See http://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html

Java & OO SE 2 – Lecture #15 – Jeremy.Singer@glasgow.ac.uk



Persistent Data in Files

Persistent data may be stored as files in a filesystem or using an alternative backing store (e.g. a database). The major Java library for file handling is the `java.io` package[1]. The basic abstraction for input and output is the *stream*, which is an ordered sequence of data (raw bytes or characters). For input, the `InputStream` is the basic abstract class. For output, there is a corresponding `OutputStream`. Concrete input classes include `BufferedReader` and `FileReader`. Concrete output classes include `BufferedWriter` and `FileWriter`.

```
public class FileSize {
  // take a single filename argument
  // count how many bytes of data file contains
  public static void main(String [] args) throws IOException {
    InputStream in = new FileInputStream(args[0]);
    int total = 0;
    while (in.read() != -1) {
      total++;
    }
    System.out.printf("size of file %s is %d bytes\n", args[0], total);
  }
}
```

Filesystem Operations

Actually there is a simpler way to calculate file size, via the `File` class[2], e.g. in above code.

```
      total = new File(args[0]).length();
```

Other standard filesystem interactions (e.g. setting permissions, listing directories, creating, renaming and deleting files) are all supported by methods in the File class[3].

Reading from a File

The most common use-case is reading data from plain text files in a line-by-line fashion. There are several library classes in Java to support this operation – we will use the `BufferedReader`[4] class. Notice how the *`currentLine`* variable is assigned as a side-effect in the `while` statement condition. Also notice the double checking for `IOException` – first when the `BufferedReader` is constructed and used, second when the `BufferedReader` is closed in the `finally` clause.

[1] Advanced features are available in the `java.nio` package.
[2] See http://docs.oracle.com/javase/7/docs/api/java/io/File.html
[3] Or by static methods in java.nio.file.Files in Java 7
[4] See http://docs.oracle.com/javase/7/docs/api/java/io/BufferedReader.html

```
String currentLine = null;
BufferedReader br = null;
try {
  br = new BufferedReader(new FileReader(FILENAME));
  while ((currentLine=br.readLine()) != null) {
    // echo line to standard output
    System.out.println(currentLine);
  }
}
catch (IOException e) {
  e.printStackTrace();
}
finally {
  try {
    if (br != null) {
      br.close();
    }
  }
  catch (IOException ee) {
    ee.printStackTrace();
  }
}
```

Java 7 try-with-resources construct

This double try/catch for `IOExceptions` is particularly inelegant. Java 7 introduces a new try-with-resources construct[5] as syntactic sugar to achieve the same effect without requiring an explicit `finally` clause.

Writing to a File

The `FileWriter`[6] is the standard library class used for generating text file output. To avoid constant byte-level filesystem access, these objects are generally wrapped in `BufferedWriter`[7] objects. The simple source code below writes a 'hello world' file in the current working directory. Note the use of a `try`-with-resources statement, which will automatically close the `BufferedWriter` object when it completes. Also note that the `newline()` method inserts the system-specific characters that encode a newline (different for DOS and Unix[8]).

```
    try(BufferedWriter bw =
              new BufferedWriter(new FileWriter("hello.txt"))) {
      bw.write("hello world");
      bw.newLine();
    }
    catch(IOException e) {
      e.printStackTrace();
    }
```

[5] See http://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html
[6] See http://docs.oracle.com/javase/7/docs/api/java/io/FileWriter.html
[7] See http://docs.oracle.com/javase/7/docs/api/java/io/BufferedWriter.html
[8] See http://en.wikipedia.org/wiki/Newline#In_programming_languages

Java & OO SE 2 – Lecture #16 – Jeremy.Singer@glasgow.ac.uk

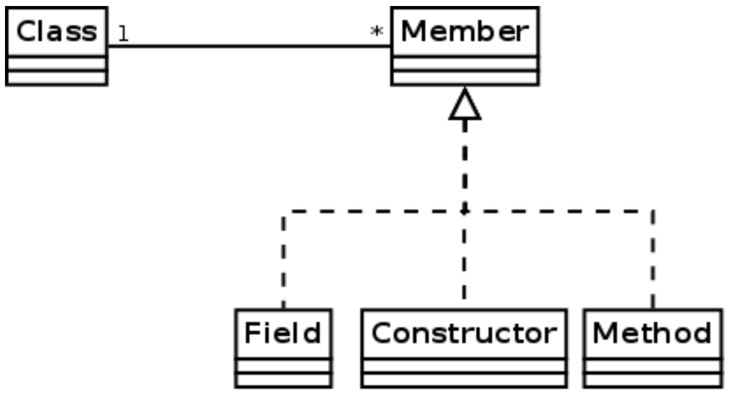
This lecture material is non-examinable

Reflection and Meta-level Analysis

So far in our Java studies, we have modeled real-world entities as objects, and real-world interactions as methods. We can take this to the meta-level by modeling Java programs as objects, and programmatic activity as methods. This concept is known as *reflection* or *introspection* – effectively we are using the facilities of Java to investigate the running program.

Reflection is a powerful tool. In this (non-examinable) lecture, we will give a basic overview of how the Java standard libraries support reflection.

Classes contain members, which are either fields, constructors or methods. Members have properties (e.g. their access visibility) and behaviours (e.g. getting and setting a value in a field, invoking a method). A UML diagram to describe this scenario is below:



Finding Out About Classes

Each Java class is represented (*reified*) as a java.lang.Class object. It is possible to get the Class object from an arbitrary Java object by calling its getClass() method. The Class class does not have a public constructor. Class objects are automatically constructed by the JVM when the corresponding classes are loaded.

```
void printClassName(Object o) {
  System.out.println("The class of " + o +
                     " is " +
                     o.getClass().getName());
}
```

It is also possible to acquire a Class object using the literal name of the class directly as a Java expression, e.g. `Object.class`, within a statement.

Finding Out About Members

Once we have a Class object, we can query it to get references to its members. Relevant methods are getDeclaredFields, getDeclaredConstructors and getDeclaredMethods. Each of these returns an array of Field, Constructor and Method references respectively. These classes belong to the java.lang.reflect package.

For Field references, we can get and set the value. There are lots of methods to do this, depending on the underlying type of the field. Exceptions may be thrown if we try to access the field using an incorrect type, or if we do not have permissions to access the field (e.g. it is private).

For Method references, we can query the annotations associated with the method and query the types associated with the method parameters and return value. We can also invoke the method, provided that we provide appropriate arguments. Again, Exceptions may be thrown.

```
Class<?> c = java.lang.Math.class;
double d = Math.random();
ArrayList<Method> doubleMethods = new ArrayList<>();
// filter methods: only want those that take a single double param
// and return a double value
for (Method m : c.getMethods()) {
    if (m.getParameterCount() == 1 &&
        m.getReturnType() == double.class) {
        Class<?> paramType = m.getParameters()[0].getType();
        if (paramType == double.class) {
            doubleMethods.add(m);
        }
    }
}
// invoke a sequence of 100 random method calls on a number
for (int i=0; i<100; i++) {
    Method invokingMethod =
        doubleMethods.get((int)(doubleMethods.size()*Math.random()));
    d = (double)(invokingMethod.invoke(null, d));
    System.out.printf("%s %d %f\n",
                      invokingMethod.getName().toString(), i, d);
}
```

Challenge

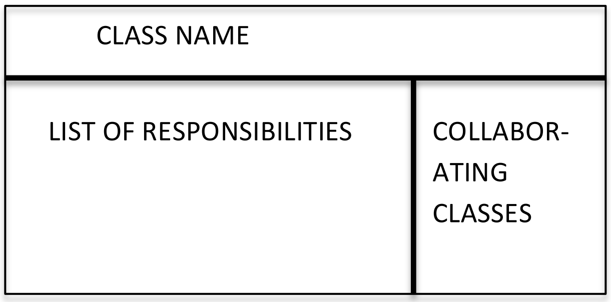
Could you use reflection to write a quine? (A quine is a program that, when executed, prints its own source code.)

Java & OO SE 2 – Lecture #17 – Jeremy.Singer@glasgow.ac.uk

Object-Oriented Domain Modeling

So far, we have looked at the mechanics of expressing object-oriented (OO) concepts in the Java language. At this stage, let's step back slightly and ask: 'where does an OO design come from?' In this lecture, we will consider how to extract an OO model from a problem description.

A general description will be in plain English, describing *entities* and *activities*. You will need to analyse the description. Highlight the nouns that will translate into classes and the verbs that will translate into responsibilities for classes. One popular technique for OO modeling is called Class-Responsibility-Collaboration (CRC) cards[1].



Each card is labeled with the name of a single class. Responsibilities might be fields or methods in the class. Collaboration will require one class depending on data or behaviour from another class.

Example Scenario: A book club has a catalogue of books. Books may be added or removed from the catalogue. There is a list of users who are members of the book club. Users may join or leave the membership list. Users buy books or borrow them. Books have individual prices. All loans are 2 weeks long. A user may have up to 4 books on loan at once. When a book is returned, it is no longer on loan. A history of loans and purchases is maintained for each user.

Exercise: Use CRC modeling to identify the classes involved in this problem description.

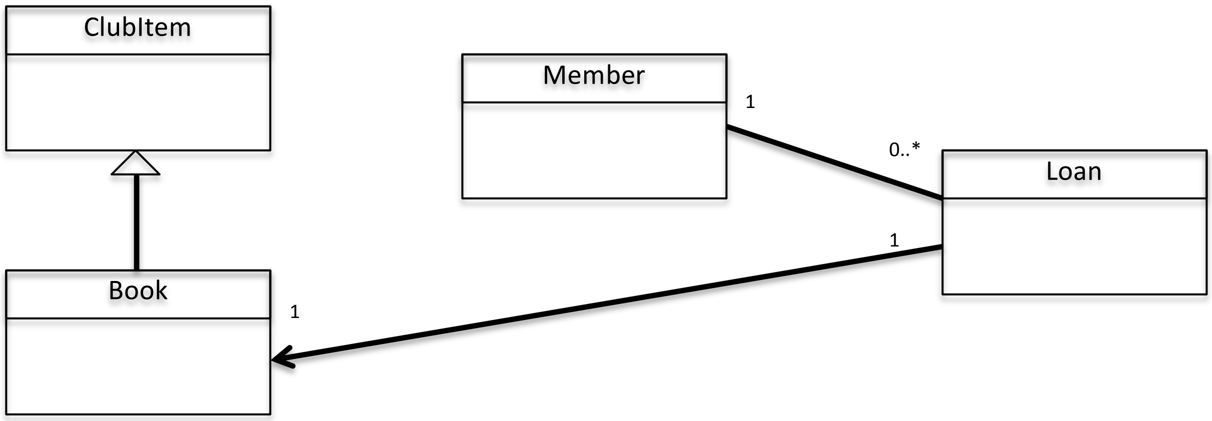
UML Domain Modeling

A UML class diagram allows us to define the relationships between classes (which may have been previously identified by CRC modeling). The two most important class relationships are **is-a** and **has-a**. Is-a refers to the inheritance relationship. Has-a or Knows-about refers to the association relationship[2].

Using the above domain model, we might say that `Book` is a `ClubItem`. We might say that a `Loan` has a `Book` and a `Member`. UML relationships would look like:

[1] More info at http://agilemodeling.com/artifacts/crcModel.htm

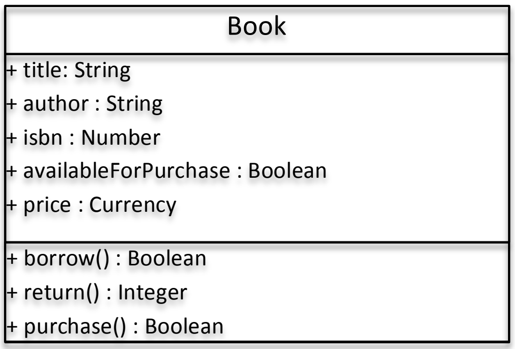
[2] I am over-simplifying things here. Actually in UML, class relationships that are modeled by Java reference fields include aggregation, composition and association. See http://agilemodeling.com/style/classDiagram.htm for more details, or read the *UML Distilled* textbook by Martin Fowler (in university library).



In these relationships, *directionality* and *multiplicity* are important. For inheritance relationships, the arrow points from the subclass to the superclass. For association relationships, the arrow (if the association is one-way) points from the knowing class to the known-about class. In the example above, the `Loan` knows about the `Book`, but not vice versa (perhaps).

Detailed UML Modeling

A UML class diagram may express high-level relationships between classes, as in the domain model examples above. A more comprehensive class diagram may give details about the attributes and behaviours of individual classes. For example, consider the `Book` class in the class diagram below. Notice the separation between attributes and behaviour, the type specifications and the visibility, which may be `public` (+) or `private` (-). Java-style `protected` visibility (#) is also possible.



Mapping UML into Java

There is a fairly straightforward correspondence from UML class diagram to Java source code. *Attributes* map onto *instance fields* of primitives (e.g. `int`, `char`) or scalar library types (e.g. `String`, `Date`). *Associations* map onto *instance fields of reference types*. *Behaviours* map onto *methods*. Some tools – like ArgoUML[3] – will automatically generate Java stub classes from UML class diagrams.

[3] See http://en.wikipedia.org/wiki/ArgoUML for details.

Java & OO SE 2 – Lecture #18 – Jeremy.Singer@glasgow.ac.uk

Object-Orientation in Python

Hopefully you encountered the Python scripting language in your CS studies last year. You will be using Python again in the Web Application Development course next semester. In this lecture, we will revisit basic OO concepts and see how they are expressed in Python. A simple class definition looks like:

```
class Employee:
   'represents an employee record in HR system'
   id = 0
   def __init__(self, name, salary):
       self.name = name
       self.salary = salary
       self.number = Employee.id
       Employee.id += 1
   def __str__(self):
       return '%s earns %d, num. %d' % (self.name, self.salary, self.number)
   def payRise(self, extra):
       self.salary += extra


x = Employee("fred", 200)
print x
```

Can you spot the correspondences with Java, in this OO code snippet? Look for:

- class definition
- static fields
- instance methods
- instance fields
- current object reference
- constructor
- string conversion
- object instantiation
- class-level documentation

Here is a subclass definition. Notice the inheritance hierarchy and virtual method overriding (which is the implicit default in Python).

```
class PartTimeEmployee(Employee):
    def __init__(self, name, salary, percent):
        Employee.__init__(self, name, salary)
        self.percent = percent

    def __str__(self):
        return '%s Part-Time earns %d, num. %d' % \
        (self.name, self.salary*self.percent/100, self.number)


y = PartTimeEmployee("barney", 200, 25)
print y
```

Differences between Python and Java

Since Python is a dynamic scripting language, it has some features that cannot be supported in Java. Instance fields (like ordinary Python variables) do not need explicit declarations – they can just be assigned values in the constructor. Further, fields are not statically typed, so the same variable could store a string or a number. Further, new instance fields may be added to objects dynamically during their lifetimes. Instance fields may also be deleted from objects, using the **del** keyword.

In terms of object-orientation, the most significant difference is that Python directly supports *multiple inheritance*, i.e. a class may have more than one parent class.

You will notice that Python scripts are much more concise than corresponding Java programs. This may be perceived as an advantage, although sometimes the extra information (e.g. types) required by Java is useful for large, complex software systems.

Running Python Scripts in Java

The Jython system implements the Python language interpreter directly on the Java virtual machine. It exhibits some differences from the default CPython interpreter although the basic syntax of the language is largely identical. Jython makes integration between Java and Python relatively straightforward, either by invoking Java from within a Jython application, or running Jython scripts from a Java application. Note that Jython is currently compatible with Python 2 but not Python 3.

Further Reading

- Details on OO Python: The official documentation at https://docs.python.org/2/tutorial/classes.html is clear although not the most interesting read. A more basic overview is at http://en.wikibooks.org/wiki/A_Beginner's_Python_Tutorial/Classes.
- Comparing and contrasting Java and Python. Guido van Rossum briefly compares Python with various other languages at https://www.python.org/doc/essays/comparisons/ . A graphical comparison is available at http://javarevisited.blogspot.co.uk/2013/11/java-vs-python-which-programming-laungage-to-learn-first.html .
- Details on Jython. See the official website at http://www.jython.org .

Java & OO SE 2 – Lecture #19 – Jeremy.Singer@glasgow.ac.uk

Windowing Libraries

So far, all our Java applications have been standalone, command-line based programs. In this lecture we consider GUI-based applications. There are several libraries for windowing GUIs in Java. These include java.awt (basic, platform independent), javax.swing (more powerful, native look-and-feel), and JavaFX (support for web and mobile with a common profile, media support).

Coding Style

GUI applications respond to user interactions. In terms of programming style, the application initializes the user interface, then registers callbacks or event handlers that will execute when triggered by particular events (e.g. mouse-clicks on buttons).

It is possible to construct a window-based user interface programmatically (as demonstrated below). However for larger applications, many developers prefer to use a drag-and-drop WYSIWYG style interface development. This is how Visual Studio produces GUI code; for Java, there are equivalent GUI builders in Netbeans and Eclipse[1]. The GUI builder will generate the raw Java code based on the designed GUI.

Simple Swing Example

Let's write a simple Java Swing application. Due to time and space restrictions, we will only cover the basics in this lecture. More advanced Swing tutorials are available online[2].

The key point is that the Swing library is completely object-oriented. Windowing components are grouped in classes. Instantiated objects have attributes that we can get and set. User interaction takes place via callbacks to methods.

Here is some simple code to pop an empty Window (known in Swing parlance as a JFrame[3]) onto the screen.

```
JFrame frame = new JFrame();
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setVisible(true);
```

[1] Eclipse plugins include WindowBuilder http://www.eclipse.org/windowbuilder/ and Jigloo http://marketplace.eclipse.org/content/jigloo-swtswing-gui-builder

[2] http://zetcode.com/tutorials/javaswingtutorial/ is a good tutorial, going through concepts step-by-step up to the construction of a Tetris-style GUI game.

[3] More comprehensive JFrame examples at https://docs.oracle.com/javase/tutorial/uiswing/components/frame.html

Note that we are calling setter methods to update attributes of the newly instantiated JFrame object. The JFrame is a basic window. We want to add GUI widgets to this window. The basic pattern is:

1) instantiate the appropriate GUI components
2) register callbacks for their interactive behaviour
3) add each component to the top-level window
4) make the window visible

Let's add a button to our window that does something interesting when we click it.

```
JButton button = new JButton("click me");
button.addActionListener(new BeepActionListener());
frame.getContentPane().add(button);
```

We need to specify the 'interesting' behaviour in a separate class that is an implementation of the ActionListener[4] interface.

```
public class BeepActionListener implements ActionListener {
	@Override
	public void actionPerformed(ActionEvent e) {
		java.awt.Toolkit.getDefaultToolkit().beep();

	}
}
```

Note that since the ActionListener is a functional interface (it only has one method signature) we can use Java 8 lambda expressions instead, to define the behaviour for the button object.

```
JButton button = new JButton("click me");
button.addActionListener(e -> java.awt.Toolkit.getDefaultToolkit().beep());
```

When the user interacts with the button object, the appropriate actionPerformed method is triggered by one of the Swing GUI background threads. Note that there are other events that may provide more information about the kind of event triggered[5].

[4] See https://docs.oracle.com/javase/tutorial/uiswing/events/actionlistener.html for details
[5] See https://docs.oracle.com/javase/tutorial/uiswing/events/api.html for details