CO536 Introduction to Java

January 2017

What Is Java?

Java is:

- A high-level, strongly-typed, object-oriented, "C-like" language
- Very widely known and used (a must-know according to my UTA)
- A "platform" for running software
  - code runs in Java Virtual Machine (JVM)
  - code can use Java API (huge library)

Is used extensively on:

- CO528 Concurrency
- CO580 Algorithms

What You Will Learn

On this course you will learn the basics of Java:

- How to write simple Java programs using objects, loops, arrays etc.
- How to define your own Java objects
- How to use Java API types in your code
- One or two other features: nested classes, exceptions

Recommended further information:

- The Java Tutorials (Oracle)
  - Trail: Learning the Java language
  - https://docs.oracle.com/javase/tutorial/java/TOC.html
- The Java API Documentation (Oracle)

Aims for Today

ToDo

1. Write and run Hello World! in Java
2. Implement a `WholeNumber` type to represent integers
   - should be printable and comparable by value
3. Implement a super type `Expression`
   - can be evaluated to an int

Aims for Today

ToDo

1. Write and run Hello World! in Java

A First Java Program

Example *(Hello.java)*

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

- Every Java program is written as a set of types, e.g. a class
- A type MyType is defined in a file called MyType.java
- Java does not have separate header files
- Execution begins with the main() method of a class
- `System.out` is standard output

Compilation

Example *(Hello.java)*

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

- The Java compiler is the `javac` program
- A type MyType is compiled with `javac path/to/MyType.java`
- The result of successful compilation is a set of `.class` files
- The `.class` files can be executed by the `java` interpreter

Execution

Example *(Hello World)*

```
tk106@matrix01% java -cp src Hello
Hello World!
```

- The interpreter ('Java virtual machine') is the `java` program
- The argument is the name of the class to run (no `.java`)
- The `-cp` option supplies the classpath
- Directories on the classpath are searched for code (.class files)

Packages

Example *(tk106/Hello.java)*

```
package tk106;

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

- Java types are organised into packages
- A package provides a namespace and affects visibility
- Packages can be grouped under a "parent" package
- Use a unique name for your top level package

Package Statements

Example *(a/fully/qualified/name/Hello.java)*

```
package a.fully.qualified.name;

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

- The package statement applies to all types in the file
- The statement must give the full package name
- The subpackage separator is .
- The directory structure must match the package hierarchy

Exercise

In Pairs

- Write two Java classes: Howdy and Doody
- Each should have a main method that prints a message
- Put each class into a different subpackage
- Compile and run both programs

Pair Programming Roles

- The Driver operates the keyboard
- The Navigator decides what is written
- Drivers should not head off on their own
- Exchange roles frequently

Comments

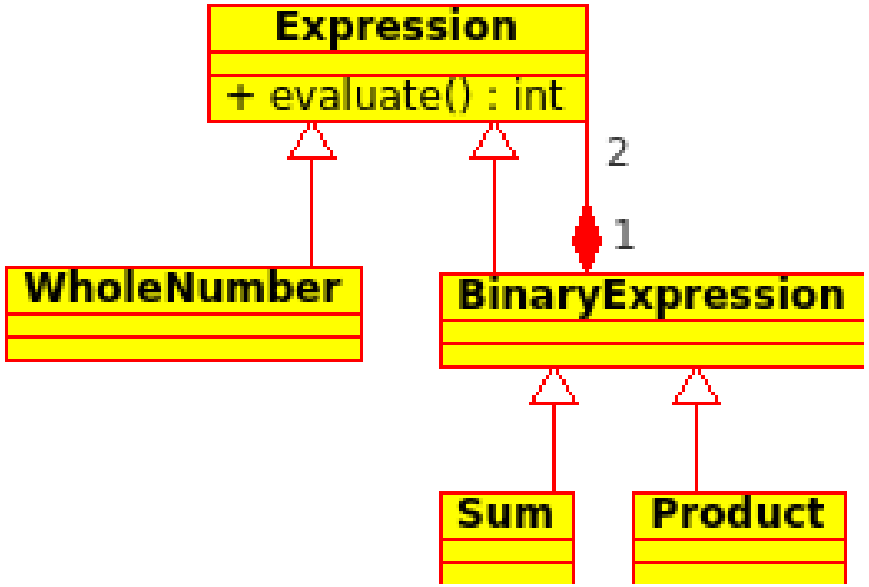
Example *(Hello)*

```
/**
 * This javadoc comment will appear in documentation
 */
public class Hello {
  public static void main(String[] args) {
    System.out.println("Hello World!");
  }
}
```

- As well as C++ style comment syntax: `//` and `/*...*/`
- Java has its own javadoc style comments
- The `javadoc` program converts these comments into html docs

A Project

Our aim is to implement types representing arithmetic expressions



Aims for Today

ToDo

1. ~~Write and run `Hello World!` in Java~~
2. Implement a `WholeNumber` type to represent integers
   - should be printable and comparable by value

Java Identifiers

Example *(ExpressionProg.java)*

```
public class ExpressionProg {
  public static void main(String[] args) {
    WholeNumber first = new WholeNumber(5);
  }
}
```

- By universal convention, Java identifiers for variables, methods and types are written in camel case
- `TypeNames` start with a capital
- `methodNames` and `variableNames` start with a lower case letter

Java Objects

Example *(ExpressionProg.java)*

```
public class ExpressionProg {
  public static void main(String[] args) {
    WholeNumber first = new WholeNumber(5);
  }
}
```

- Java objects are only created using the `new` keyword
- `new` invokes a constructor
- If no constructor is defined, a class is given a no args one automatically
- `new` returns a pointer just like C++
- Object variables are always pointers so have no * form

Java Objects

Example *(ExpressionProg.java)*

```
public class ExpressionProg {
  public static void main(String[] args) {
    WholeNumber first = new WholeNumber(5);
  }
}
```

- Java has automatic garbage collection of objects
- The programmer does not have to deallocate memory
- An object is eligible for garbage collection when all pointers to it go out of scope

Member Variables

Example *(WholeNumber.java)*

```
public class WholeNumber {
  private int value;

  public WholeNumber(int val) {
    value = val;
  }
}
```

- Recall encapsulation: keep member variables private
- private is not the Java default so must be declared

Primitive Types

Java's 8 Primitives

Type	Size	Literals
boolean	1 bit	true, false
byte	8 bit	1, 0b11, 0x2A
short	16 bit	1, 0b11, 0x2A
int	32 bit	1, 0b11, 0x2A
long	64 bit	1, 0b11, 0x2A, 44L
char	16 bit, unicode	'a', '\u0123'
float	32 bit	6.21f, 6.21e23f
double	64 bit	6.21, 6.21e23

- All integer types are signed
- No other values are treated as `true` or `false`

Constructors

Example *(WholeNumber.java)*

```
public class WholeNumber {
  private int value;

  public WholeNumber(int val) {
    value = val;
  }
}
```

- Java has no specialised constructor syntax
- Member variables are initialised in the body
- Declaring any constructor removes the "default" no args one

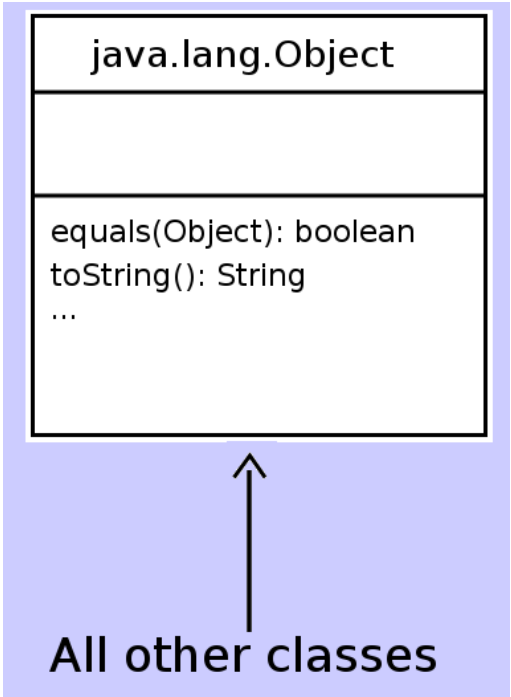
Properties of Objects

Example *(ExpressionProg.java)*

```
WholeNumber first = new WholeNumber(5);
System.out.print("The first number is ");
System.out.println(first);
...
System.out.print("The first number is ");
equal = first.equals(second) ? "equal" : "not equal";
System.out.printf("%s to the second\n", equal);
```

- Java objects can be printed
- They can be compared using `equals()`
- There is no operator overloading or friend functions
- The behaviour is via methods inherited from the `Object` superclass

java.lang.Object



- Every Java class is a subclass of `Object`
- The methods `equals`, `toString`, ... have simple implementations
- User-defined types should override them

Exercise

In Pairs

- Write your own ExpressionProg and WholeNumber classes
- In WholeNumber, override:
  - `public boolean equals(Object ob)`
  - `public String toString()`
- Compile and run the program

Challenges (see Java Tutorial for help):

- The parameter of `equals(...)` has type `Object`
- The return type of `toString()` has type `String`

java.lang.String

A Java string is an (immutable) object of type `java.lang.String`

- The String type is not an alias for a char array
- Java has special support for Strings

Example *(Strings)*

```
System.out.print("The first number is ");
System.out.println(first);
System.out.println("The second number is " + second);
System.out.println("The number after four is " + 5);
```

- String objects can be created using `"quoted text"` or `new`
- Strings can be concatenated (with any type) using `+`
- The `String` class provides other methods

java.lang.String

Example *(Strings)*

```
String equal =
  first.equals(second) ? "equal" : "not equal";
System.out.printf(
  "%s is %s to %s\n", first, equal, second);
System.out.printf("%d is one less than %d\n", 4, 5);
```

- `toString()` is called automatically when a String is required
- *Format* strings are templates that avoid excessive concatenation

Arrays

- Arrays are also objects, created on the heap
- Arrays have a single public member variable: `length`

Example *(Arrays)*

```
boolean[] someTruths = new boolean[8];
int[]          myInts = {1, 2, 3, 4};
```

- Appending `[]` modifies any type `T` to be an array of `T`
- Create a zeroed array using `new Type[size]`
- Or use literal form with $\{\}$

Arrays

Example *(Arrays)*

```
int[] myNums = new int[8];
System.out.printf(
  "A new array contains all %ds\n", myNums[4]);
System.out.printf("My new array is %s\n",
  java.util.Arrays.toString(myNums));
```

- Access element `i` using `arr[i]`
- No pointer arithmetic with array pointer
- The java.util.Arrays class has lots of utility methods for arrays

Importing Code

- Types can be imported from other packages
- All types in `java.lang` automatically imported

Example *(Arrays)*

```
import java.util.Arrays;
...
System.out.printf(
  "My new array is %s\n", Arrays.toString(myNums));
```

- Importing avoids using the full name in the code
- All types in a package can be imported with wildcard (*)
- Package `tk106.games` needs separate import to package `tk106`
- Wildcarding not recommended (possible collisions)

Constants

- Constants are declared with keyword `final`
- Constant names in `UPPER_SNAKE_CASE` by convention
- Classes, methods and parameters can be final too

Example *(Constant)*

```
public static void main(String[] args) {
  final int NUM_COUNT = 10;
  WholeNumber[] nums = new WholeNumber[NUM_COUNT];
  ...
```

- Constants are often class members
- Class members are declared using `static`

Exercise

In Pairs

- In ExpressionProg.main(...)
  - Declare an array of `WholeNumbers`
  - Use a class constant to define the size
  - Loop through the array to populate it with objects
  - Print the array using `printf` and `Arrays.toString(...)`
- Compile and run the program

`Arrays.toString(...)` works by ...

Aims for Today

ToDo

1. ~~Write and run `Hello World!` in Java~~
2. ~~Implement a `WholeNumber` type to represent integers~~
   - ~~should be printable and comparable by value~~
3. Implement a super type `Expression`
   - can be evaluated to an int

Interfaces

- A Java interface is an abstract type
- It is a set of behaviours

Example *(Expression.java)*

```
public interface Expression {
  public int evaluate();
}
```

- Interfaces are usually just a list of unimplemented methods
- Can also contain constants, static methods and "default methods"
- They correspond to C++ pure abstract classes
- An interface cannot be instantiated

Implementing an Interface

- A class that implements an interface guarantees to provide the interface behaviours

Example *(WholeNumber.java)*

```
public class WholeNumber implements Expression {
  ...
  public int evaluate() {
    return this.value;
  }
}
```

- Must define all interface methods or be declared abstract
- Now safe to treat a `WholeNumber` object as an `Expression`
- Can implement multiple interfaces (comma separated list)
- Java's version of multiple inheritance

Apparent Types

- An object's declaration defines its apparent type (on the LHS)

Example *(ExpressionProg.java)*

```
Expression   five = new WholeNumber(5);
Expression[] exps = new Expression[NUM_COUNT];
exps[0] = new WholeNumber(10);
int v   = exps[0].evaluate();
```

- Objects can be declared with type of implemented interface
- The consequences of declaring these objects using the actual type and the apparent type are:

Aims for Today

ToDo

1. Make a linked list data structure
   - should be able to hold any type

Java Generics

- Generics are classes, interfaces or methods with type parameters
- Somewhat like C++ templates but only types can be parameter

Example *(List.java)*

```
public class List<T> {
  public void add(T value) { ... }
}
```

- This class can be instantiated with any type as `T`
- "Any type" does not include primitives
- Before Java 1.5 such a class would have used `Object`
- This used to lead to the use of ... (avoid)

Nested Classes

- Nested classes are declared inside other classes

Example *(List.java)*

```
public class List<T> {
  private Element<T> front;
  private Element<T> back;
  private class Element<S> {
    S          content;
    Element<S> next;
  }
```

- The nested class can be static or non-static (an inner class)
- The inner class has access to any variables of the outer class
- An inner class can be private to the outer class

Using Generics

Example *(ListProg.java)*

```
public class ListProg {
  public static void main(String[] args) {
    List expList = new List<Expression>();
    ...
  }
}
```

- This declares a `List` of `Expressions`
- Any type implementing `Expression` can be used with this list
- The `Expression` on the right can be omitted if it can be inferred

Pair Programming Exercise

Implement a constructor for the `Element<S>` inner class:

- `Element(S value)`

Implement and test these methods in your `List<T>` class:

- `public String toString()`
  - Should contain all list values, space separated
- `public void add(T value)`
  - New element goes at end of list

Test in `ListProg.main`

Exceptions

- Use of exceptions is common in Java
- `java.lang.RuntimeException` (and subclasses) is used to indicate something a programmer could have prevented

Example *(List.java)*

```
public T first() {
  if (front == null) {
    throw new RuntimeException("The list is empty!");
  }
  return front.content;
}
```

- No sensible value to return (null?)
- Exception object can give info about the problem

Checked Exceptions

Exceptions other than `RuntimeExceptions` and `Errors` are called checked exceptions

Example *(List.java)*

```
public T first() throws Exception {
  if (front == null) {
    throw new Exception("The list is empty!");
  }
  return front.content;
}
```

- Methods must declare if they throw checked exceptions
- Rules also apply when overriding the method (see Java Tutorial)
- Checked exceptions are for unavoidable situations

Handling Exceptions

Example *(ListProg.java)*

```
Expression expr;
try {
  expr = expList.first();
} catch (Exception e) {
  System.out.println("Oh no!");
  e.printStackTrace();
}
```

- Code calling a method throwing checked exceptions must handle them
- Either:
  - Put the call inside a `try` block and catch the exception
  - Or declare that this code also throws the checked exception
- In this case a Runtime/Checked exception is more appropriate ...?