THE UNIVERSITY OF AUCKLAND
NEW ZEALAND

CompSci 230
Software Construction

Java Implementation: Part 1 S1 2015

Agenda

- Topics:
  - Interfaces in Java
  - Reference data types
  - Abstract classes in Java
  - Java syntax: five important keywords
- Reading
  - In The Java Tutorials:
    - What is an Interface?, in the Object-Oriented Programming Concepts Lesson
    - The Interfaces and Inheritance Lesson

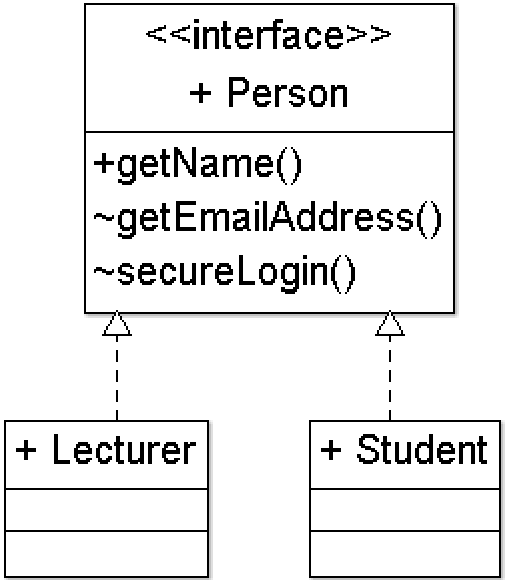
Learning objectives: Java Implementation

- Students will be competent at implementing OO designs in Java
  - Interfaces, reference data types, abstract classes, intro to generics
  - Visibility, packages, static & dynamic typing, conversion & casting

- The lectures will give you the basic "theory", but they won't give you a "working understanding" – you have to do the hard-yards of putting these ideas into practice.
  - You won't even understand the theory, if you listen passively to lectures. I'll try to help you "learn how to learn" from the Java tutorials.
  - You'll get many chances to develop your understanding in your lab assignments for this course.

Interfaces, in UML

- Interfaces specify behaviour (a public contract), without data or implementation.
- Interfaces are drawn like classes, but without attributes, and with the keyword `<<Interface>>`
- A dotted open-triangle arrow, from a class to an interface, means that "the class implements this interface".
  - We also say that "the class fulfils the contract specified by this interface", or that it "realizes the interface."



- Note that interfaces define methods but not attributes.
  - A password allows a secureLogin().

Interfaces in Java 7

- An **`Interface`** is like a **`Class`**, with no bodies in the methods. It may define constants (**`public static final`**) but no runtime variables.
  - Usually, an **`Interface`** is **`public`**.
  - An interface provides a standard way to access a class which could be implemented in many different ways.
- *The Java Tutorials*:
  - "There are a number of situations in software engineering when it is important for disparate groups of programmers to agree to a 'contract' that spells out how their software interacts."
  - "Each group should be able to write their code without any knowledge of how the other group's code is written."
  - "Generally speaking, *interfaces* are such contracts."

Interfaces in Java 8

- In Java 8, an interface may contain
  - `default` implementations of instance methods, and
  - implementations of `static` methods.
- In any OO language, an interface
  - cannot be instantiated, and
  - defines a "contract" which any realization of the interface must fulfil.
- Java is a strongly-typed language.
  - Java compilers *can* enforce contracts, by refusing to compile classes whose implementations might "partially realize" an interface.
- Java is a tightly-specified language.
  - If a compiler allows instantiations of incompletely-implemented interfaces, then it is *not* a Java compiler.

Implementations as contracts

- A class which realizes an interface **must** provide an implementation of **every method** defined within the interface
  - A class may implement some additional methods (but these extra methods aren't accessible through this interface)
  - Beware: adding another method to an existing Interface will "break" every current implementation of this Interface!
- A class can implement many interfaces.
- An **`Interface`** can extend other Interfaces.
  - Extension is the preferred way to add new methods to an Interface.
    - (Do you understand why?)
  - In Java, classes are less extendible than interfaces, because a **`Class`** can extend at most one other **`Class`** ("single inheritance").

```
class MountainBike extends Bicycle { … }
```

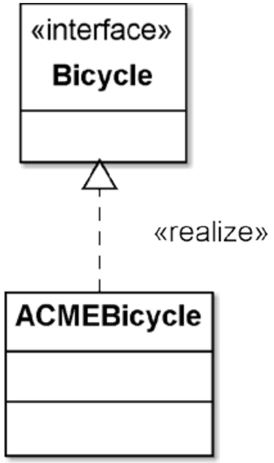
Interfaces in Java 8

- In Java 8, an interface may contain
  - `default` implementations of instance methods, and
  - implementations of `static` methods.
- In any OO language, an interface
  - cannot be instantiated, and
  - defines a "contract" which any realization of the interface must fulfil.
  - In Java, a realization is denoted by the keyword **`implements`**.

Example 1

```
public interface Bicycle {
  void changeCadence(int newValue);
  void changeGear(int newValue);
  void speedUp(int increment);
  void applyBrakes(int decrement);
}

class ACMEBicycle implements Bicycle {
  int cadence = 0; \\ an implementation may have variables
  void changeCadence(int newValue) {
    cadence = newValue;
  }
  \\ note: an implementation may be incorrect!
  void changeGear(int newValue) {}
  void speedUp(int increment) {}
  void applyBrakes(int decrement) {}
}
```



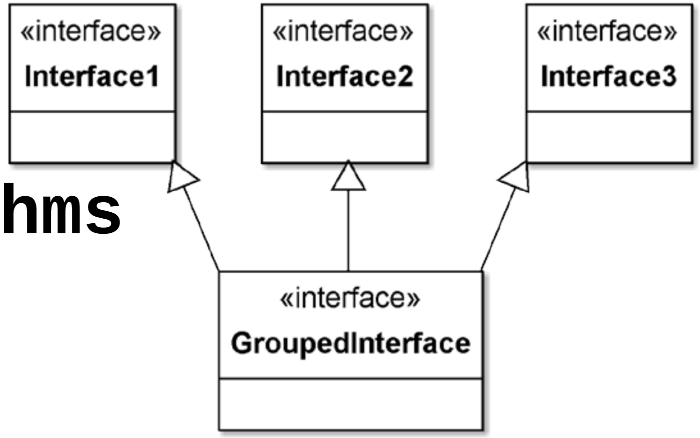
Example 2

```
public interface GroupedInterface extends
   Interface1, Interface2, Interface3 {

   // constant declarations
   // base of natural logarithms
   double E = 2.718282;

   // method signatures
   void doSomething( int i, double x );
   int doSomethingElse( String s );

}
```



Example 3

```
public interface EventListener {
   // No constants
   // No method signatures!
}
```

«interface»

EventListener

- “A tagging interface that all event listener interfaces must extend.” [http://docs.oracle.com/javase/6/docs/api/java/util/EventListener.html]
- Why?
  - At first glance, this is worse than useless! One more name for the Java programmer to remember…
- This interface allows programmers, and the Java compiler, to distinguish event-listeners from all other types of classes and interfaces.
  - Event-listeners are important, and they behave quite differently to a regular class. (Later, you’ll learn about inversion of control…)

MouseListener in java.awt.event

`public interface MouseListener extends EventListener`

The listener interface for receiving "interesting" mouse events (press, release, click, enter, and exit) on a component. (To track mouse moves and mouse drags, use the MouseMotionListener.)

```
public interface MouseListener
  extends EventListener {
    mouseClicked( MouseEvent e );
    mouseEntered( MouseEvent e );
    mouseExited( MouseEvent e );
    mousePressed( MouseEvent e );
    mouseReleased( MouseEvent e );
}
```

All Known Subinterfaces:

MouseInputListener

All Known Implementing Classes:

AWTEventMulticaster, BasicButtonListener, BasicComboPopup.InvocationMouseHandler, BasicComboPopup.ListMouseHandler, BasicDesktopIconUI.MouseInputHandler, …

```
public interface MouseMotionListener extends EventListener {
  mouseDragged( MouseEvent e );
  mouseMoved( MouseEvent e );
}
```

```
public interface MouseInputListener
  extends MouseListener, MouseMotionListener {
  // this interface has 7 method signatures, can you list them?
}
```

Using an Interface as a Type

- "When you define a new interface, you are defining a new reference data type.
  - "You can use interface names anywhere you can use any other data type name.
  - "If you define a reference variable whose type is an interface, any object you assign to it *must* be an instance of a class that implements the interface." [http://docs.oracle.com/javase/tutorial/java/IandI/interfaceAsType.html]
- Example on the next slide:
  - A method for finding the largest object in a pair of objects, for *any* objects that are instantiated from a class that implements **`Relatable`**.

```
public interface Relatable {
   public int isLargerThan( Relatable other );
}
```

Using an Interface as a Type

```
public Object findMax(Object object1, Object object2) {
  Relatable obj1 = (Relatable)object1;
  Relatable obj2 = (Relatable)object2;
  if( (obj1).isLargerThan(obj2) > 0 )
    return object1;
  else
    return object2;
}
```

- If comparisons are important in your application, then you'll be able to write very elegant code!
  - You can write **`z.findMax(x, y)`**, if **`x`** and **`y`** are instances of any class which extends **`Relatable`**.

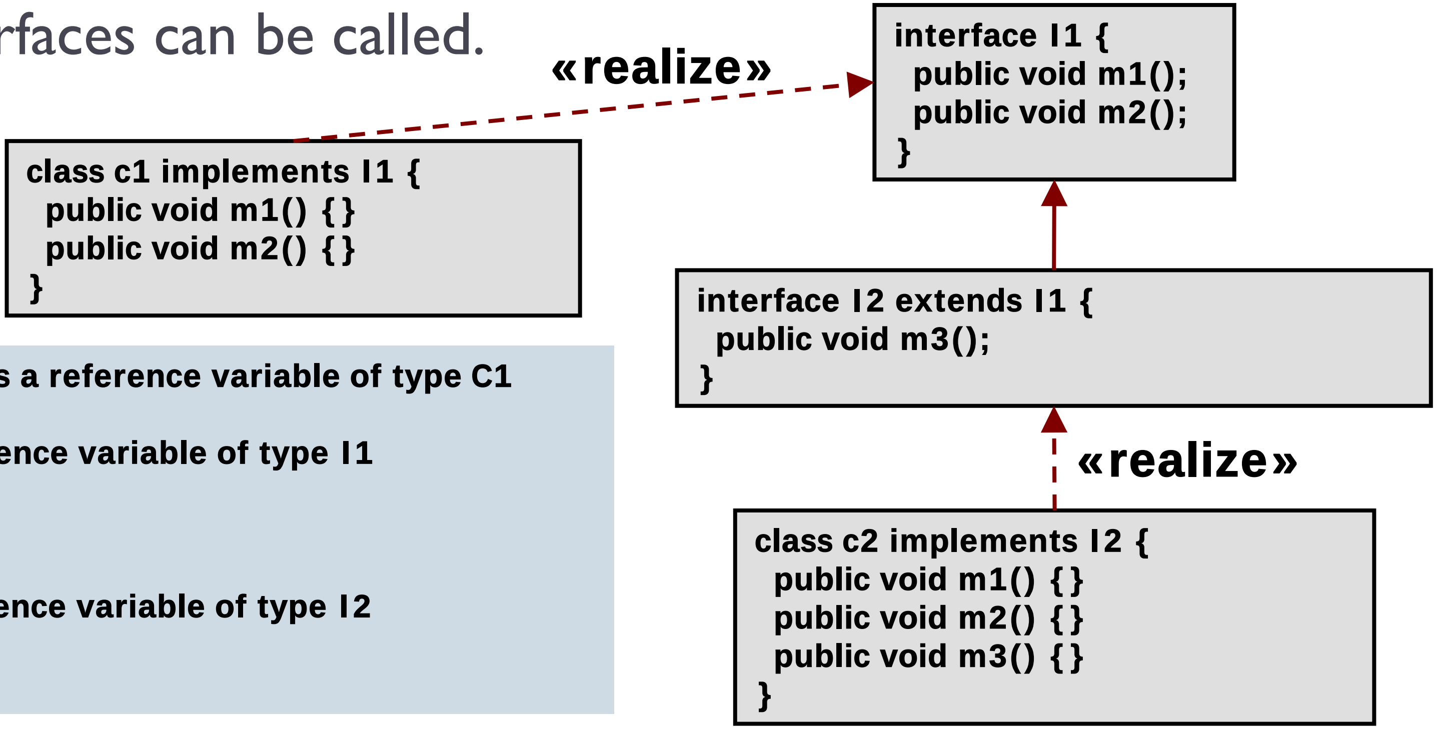
Using an Interface as a Type: Mismatches

```
public Object findMax( Object object1, Object object2 ) {
  Relatable obj1 = (Relatable)object1;
  Relatable obj2 = (Relatable)object2;
  if( (obj1).isLargerThan(obj2) > 0 )
      return object1;
    else return object2;
}
```

- We'd get errors at compile-time (or at runtime) if
  - `(object1).isLargerThan(object2)` were in the body of this method, if
  - we invoked it as `z.findMax(x,y)`, for any instance `x` of a class that doesn't extend `Relatable`, or if
  - we invoked it as `x.findLargest(y,z)`, if `y.isLargerThan()` does not accept `z` as a parameter.
- Typing is complex… we'll keep looking at it, in different ways…

Typing Rules

- The typing rules for interfaces are similar to those for classes.
  - A reference variable of interface type T can refer to an instance of any class that implements interface T or a sub-interface of T.
  - Through a reference variable of interface type T, methods defined by T and its super interfaces can be called.



«realize»

```
interface I1 {
  public void m1();
  public void m2();
}
```

```
class c1 implements I1 {
  public void m1() {}
  public void m2() {}
}
```

```
interface I2 extends I1 {
  public void m3();
}
```

«realize»

```
class c2 implements I2 {
  public void m1() {}
  public void m2() {}
  public void m3() {}
}
```

```
C1 a = new C1();  \\ a is a reference variable of type C1
C1 b = new C1();
I1 p = a; \\ p is a  reference variable of type I1
p.m1();

C2 c = new C2();
I2 q = c;  \\ q is a reference variable of type I2
q.m1();
q.m2();
```

`instanceof`

- You can use the `instanceof` operator to test an object to see if it implements an interface, **before** you invoke a method in this interface.
  - This *might* improve readability and correctness.
  - This *might* be a hack.
    - Where possible, you should extend classes and interfaces to obtain polymorphic behaviour, rather than making a runtime check.

```
if( b instanceof Bounceable ) {
  b.hitWall( "Wall A" );
} else { \\ abort, with an error message to the console
  throw new AssertionError( b );
}

Date somedate = new Date();
\\ throw an exception if somedate is not Relatable.
assert( Date instanceof Relatable );
\\ See http://docs.oracle.com/javase/1.4.2/docs/guide/lang/assert.html
```

Abstract Classes

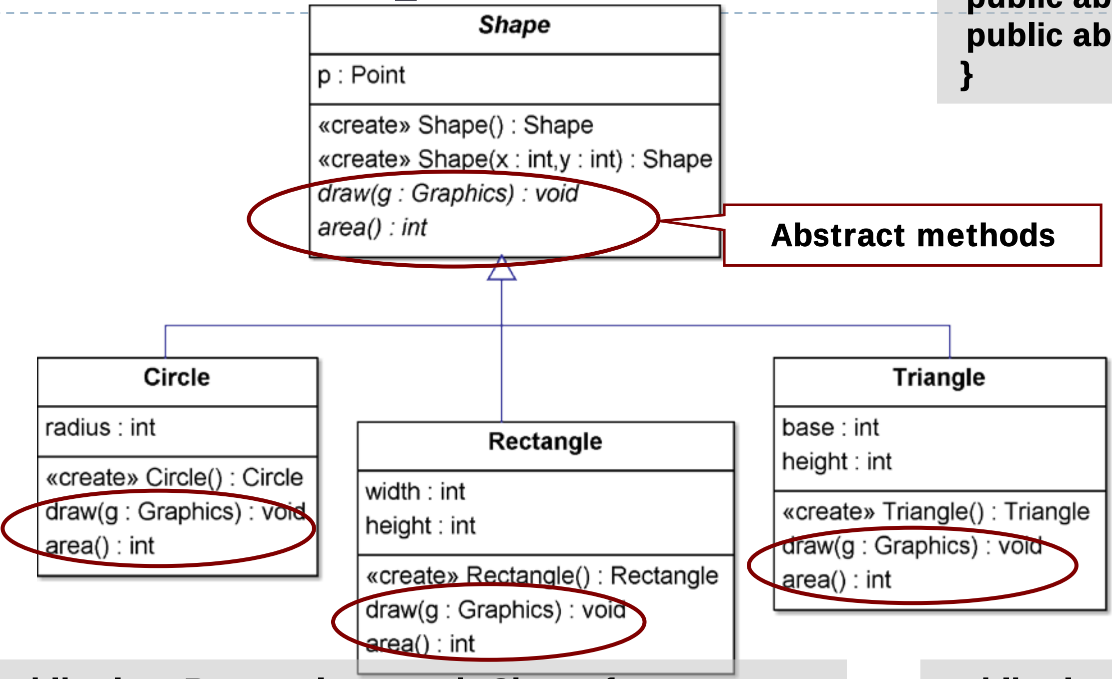
- Sometimes, it's appropriate to partly-implement a class or interface.
  - Abstract classes allow code to be reused in similar implementations.
- Abstract classes may include some abstract methods.
  - If there are no abstract methods, then the class is usually (but not always) implemented fully enough to be used by an application.
    - Sometimes it's helpful to have multiple implementations that differ only in their type, but this is quite an advanced concept in design.

```
public abstract class MyGraphicObject {
  // declare fields – these may be non-static
  private int x, y;
  // declare non-abstract methods
    // (none)
  // declare methods which must be implemented later
  abstract void draw();
}
```

Example

```
abstract class Shape {
 Point p;
 Shape(){ this(0, 0); }
 Shape(x, y){ p = new Point(x, y); }
 public abstract void draw(Graphics g);
 public abstract int area();
}
```

An abstract method is defined with a signature but no implementation.



Shape

p : Point

«create» Shape() : Shape

«create» Shape(x : int,y : int) : Shape

draw(g : Graphics) : void

area() : int

Circle

radius : int

«create» Circle() : Circle

draw(g : Graphics) : void

area() : int

Rectangle

width : int

height : int

«create» Rectangle() : Rectangle

draw(g : Graphics) : void

area() : int

Triangle

base : int

height : int

«create» Triangle() : Triangle

draw(g : Graphics) : void

area() : int

Abstract methods

Concrete subclasses must implement all abstract methods.

```
public class Rectangle extends Shape {
  private int width, height;
   public int area() {
    return (width * height);
  }
  ...
```

```
public class Circle extends Shape {
  private int radius;
  public int area() {
    return (int) (Math.PI * radius * radius);
  }
  ...
```

```
public class Triangle extends Shape {
  private int base, height;
  public int area() {
    return (base * height) / 2;
  }
  ...
```

Super!

- If your method overrides one of its superclass's methods, you can invoke the overridden method through the use of the keyword `super`.
  - You can also use `super` to refer to a hidden field (although hiding fields is discouraged).
- Example below.
  - Can you determine what will be printed to `System.out` when `main()` is executed?

```
public class Superclass {
  public void printMethod() {
    System.out.println("Printed in Superclass.");
} }
public class Subclass extends Superclass {
  public void printMethod() { // overrides super.printMethod
    super.printMethod();
    System.out.println("Printed in Subclass");
} }
public static void main(String[] args) {
  Subclass s = new Subclass();
  s.printMethod();
} }
```

```
Printed in Superclass.
Printed in Subclass
```

Hiding vs overriding

- If a subclass defines a static method with the same signature as a static method in the superclass, then
  - the method in the subclass *hides* the one in the superclass.
- The distinction between hiding a static method and overriding an instance method has important implications:
  - The version of the overridden instance method that gets invoked is the one in the subclass.
  - The version of the hidden static method that gets invoked depends on whether it is invoked from the superclass or the subclass.
    - Hmmm… this could be confusing! So … I don't encourage you to hide methods.
- Overriding methods is an important part of OO design.

this

- Within an instance method or a constructor, `this` is a reference to the current object —
  - the object whose method or constructor is being called.
- You can refer to any member of the current object
  - from within an instance method or a constructor
  - by using `this`.
- The most common reason for using the `this` keyword is
  - because a field is shadowed by a method or constructor parameter.

Is shadowing a good idea?

- A parameter can have the same name as one of the class's fields.
  - If this is the case, the parameter is said to *shadow* the field.
- Shadowing fields can make your code difficult to read and is conventionally used
  - only within constructors and methods that set a particular field.
- For example, consider the following `Circle` class …
  - Source: http://docs.oracle.com/javase/tutorial/java/javaOO/arguments.html

Example: using this.x

```
public class Point {
    public int x = 0;
    public int y = 0;
    public Point(int a, int b) {
        x = a;
        y = b;
    }
}
```

- Equivalently:

```
public class Point {
    public int x = 0;
    public int y = 0;
    public Point(int x, int y) {
        this.x = x; // this.x refers to the shadowed instance variable
        this.y = y;
    }
}
```

Using `this()`

- From within a constructor, you can also use the `this` keyword to
  - call another constructor in the same class.
- Doing so is called an *explicit constructor invocation*.

[https://docs.oracle.com/javase/tutorial/java/javaOO/thiskey.html]

(Let's look at an example of this in Eclipse.)

(I also want to show you how to import a JARfile.)

Final

- The final keyword can be applied to prevent the extension (over-riding) of a field, argument, method, or class.
  - Final field: constant
  - Final argument: cannot change the data within the called method
  - Final method: cannot override method in subclasses
  - Final class: cannot be subclassed (all of its methods are implicitly final as well)

```
class ChessAlgorithm {
. . .
  final void nextMove(
    ChessPiece pieceMoved, BoardLocation newLocation ) {
       \\ body of nextMove – can’t be overriden
  }
}
```

Review

- Interfaces in Java
- Types in Java
- Abstract classes in Java
- Six important keywords:
  - `interface`
  - `implements`
  - `abstract`
  - `super`
  - `this`
  - `final`