COMP1008

An overview of Polymorphism, Types, Interfaces and Generics

Being Object-Oriented

- Exploiting the combination of:
  - objects
  - classes
  - encapsulation
  - inheritance
  - dynamic binding
  - polymorphism
  - pluggability

Polymorphism

- Where something has multiple forms.
  - A single function that can be applied to multiple types.
  - Generic methods/classes.
  - Ability of objects of different types to respond to same messages (method calls).
- Allows one section of code to work with multiple values and objects.
  - Share rather than duplicate.
- Wikipedia has some good articles on polymorphism.

Forms of Polymorphism

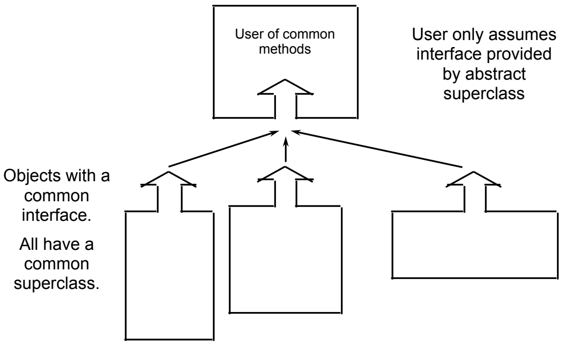
- Parametric Polymorphism - generic classes and methods.
- Subtyping Polymorphism - inheritance
- Ad-hoc polymorphism
  - Overloading
  - Coercion

Polymorphism and Inheritance

- A superclass can define a common interface.
- Subclasses inherit the common interface and specialise the corresponding methods.
- A subclass object can be used where a superclass object has been specified.
- Remember shapes:

```
        Shape myShape = new Square(4,4,10);
```

Pluggability



User of common methods

User only assumes interface provided by abstract superclass

Objects with a common interface.

All have a common superclass.

Old Code can Call New Code

- New pluggable components can be added *without* changing the users of the components.
- Code designed to use the common interfaces remains unchanged.
- For example,
  - BankAccount.
  - And specific kinds of bank account.

Objects and Types

- An Object has a state.
  - The values of its instance variables.
- The overall value of an object is determined by its state.
- An object also has a type.
  - An object's class determines its type.
  - A class is a user defined type.
- An object reference has a reference type.
  - Determines what kind of objects it can refer to.

Type Conformance

- But all classes are subclasses of Object (except Object),
- and a class can have other superclasses (the inheritance chain),
- so an object can have multiple types.
  - Or to be precise an object can *conform* to multiple types.
  - Type conformance means that any method declared by a type can be called on an object that conforms to the type.
  - Object <- Shape <- Square
    - Square conforms to both type Shape and type Object. Any public methods declared in Shape and Object can be called on a Square object (and may be overridden).

All types conform to type Object

- Hence, a reference of type *reference to Object* can refer to any object that conforms to type Object.
  - i.e., all objects
- And all objects inherit (and may override) the methods declared in class Object.
- For example, toString overridden in class Square:

```
public String toString() {
  return “This is a square of size” + size;
}
...
System.out.println(square); // toString called here
```

Type Hierarchies

- Types also have supertype and subtype relationships (like superclass/subclass).
- The class hierarchy defines part of the type hierarchy.
- But it gets more interesting...

Enter the Interface

- Types can also be declared using an *interface*:

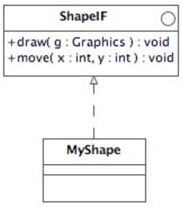
```
public interface ShapeIF
{
   void draw(Graphics g);
   void move(int x, int y);
}
```

Specifies the methods a type declares. No method bodies, no instance variables.

Implements

- A class can implement an interface.

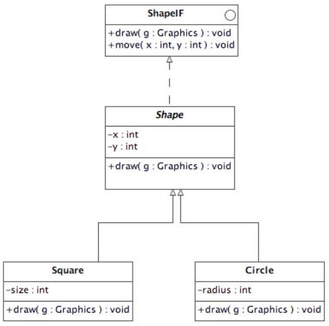
```
class MyShape implements ShapeIF
{
   // Class must override draw and move methods
   // or be abstract.
}
```


ShapeIF
+draw(g : Graphics) : void
+move(x : int, y : int) : void
MyShape

Circle denotes interface. Can also use «interface» below name.

Dashed line with open triangle denotes implements.

Combining Interface and Abstract Class


ShapeIF
+draw(g : Graphics) : void
+move(x : int, y : int) : void
Shape
-x : int
-y : int
+draw(g : Graphics) : void
Square
-size : int
+draw(g : Graphics) : void
Circle
-radius : int
+draw(g : Graphics) : void

Interface (common methods)

Abstract class (shared code)

Concrete class (complete code)

Using an interface

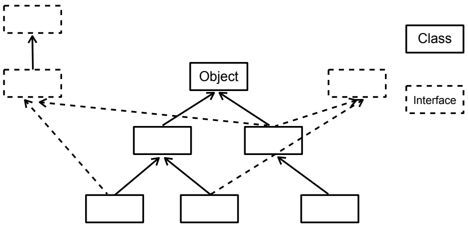
- Can now use the interface type with Shape objects:

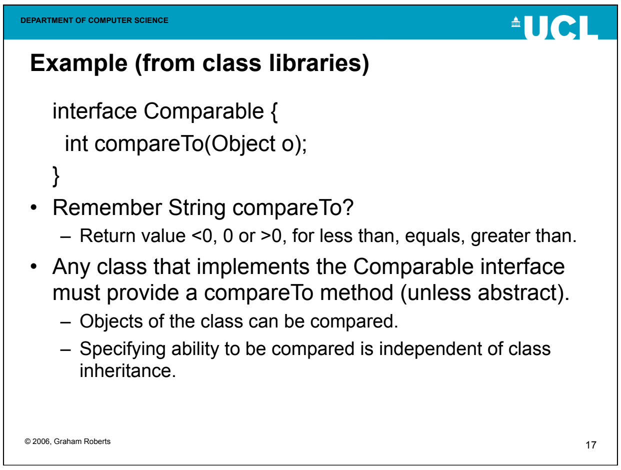
```
public void drawPicture
  (ArrayList<ShapeIF> shapes, Graphics g)
{
    for (ShapeIF shape : shapes)
    {
        shape.draw(g);
    }
}
```

Majority of code written using interface type(s). Can use any object of a class that implements the interface.

Interfaces and Class hierarchy

- Interfaces allow types to be declared independently of classes and the class hierarchy.

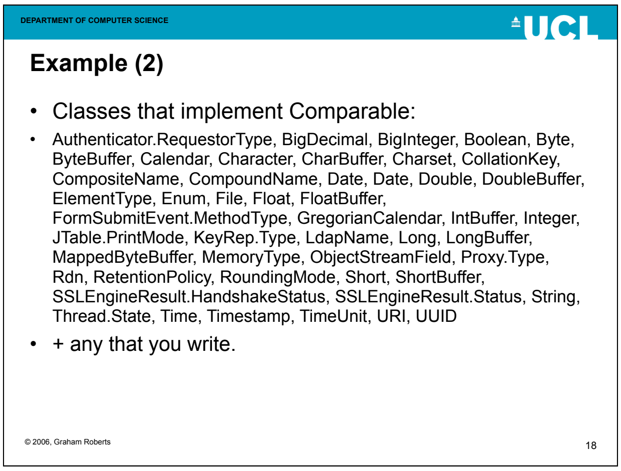




Example (from class libraries)

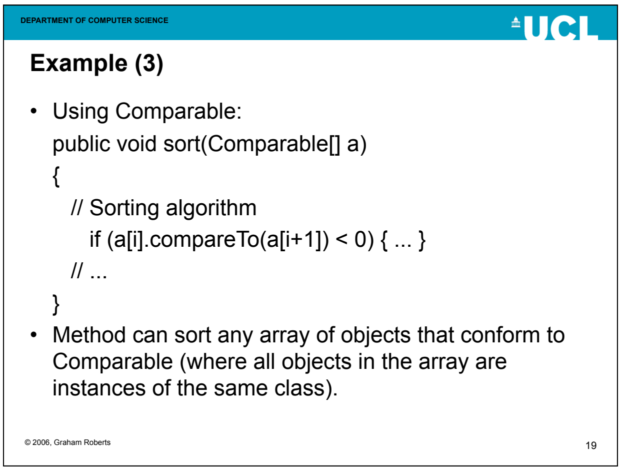
```
interface Comparable {
  int compareTo(Object o);
}
```

- Remember String compareTo?
  - Return value <0, 0 or >0, for less than, equals, greater than.
- Any class that implements the Comparable interface must provide a compareTo method (unless abstract).
  - Objects of the class can be compared.
  - Specifying ability to be compared is independent of class inheritance.



Example (2)

- Classes that implement Comparable:
- Authenticator.RequestorType, BigDecimal, BigInteger, Boolean, Byte, ByteBuffer, Calendar, Character, CharBuffer, Charset, CollationKey, CompositeName, CompoundName, Date, Date, Double, DoubleBuffer, ElementType, Enum, File, Float, FloatBuffer, FormSubmitEvent.MethodType, GregorianCalendar, IntBuffer, Integer, JTable.PrintMode, KeyRep.Type, LdapName, Long, LongBuffer, MappedByteBuffer, MemoryType, ObjectStreamField, Proxy.Type, Rdn, RetentionPolicy, RoundingMode, Short, ShortBuffer, SSLEngineResult.HandshakeStatus, SSLEngineResult.Status, String, Thread.State, Time, Timestamp, TimeUnit, URI, UUID
- + any that you write.

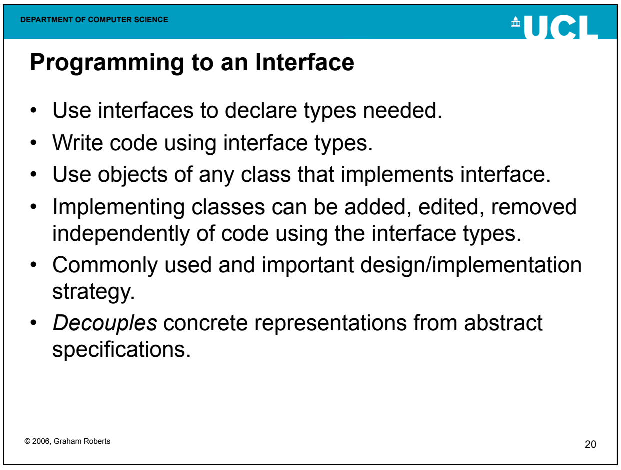


Example (3)

- Using Comparable:

```
public void sort(Comparable[] a)
{
  // Sorting algorithm
    if (a[i].compareTo(a[i+1]) < 0) { ... }
  // ...
}
```

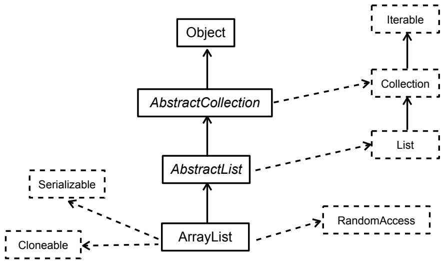
- Method can sort any array of objects that conform to Comparable (where all objects in the array are instances of the same class).



Programming to an Interface

- Use interfaces to declare types needed.
- Write code using interface types.
- Use objects of any class that implements interface.
- Implementing classes can be added, edited, removed independently of code using the interface types.
- Commonly used and important design/implementation strategy.
- *Decouples* concrete representations from abstract specifications.

Remember ArrayList...



ArrayList conforms to many types.

ArrayList declaration

```
class ArrayList extends AbstractList
  implements Serializable, Cloneable, List, RandomAccess
{ ... }
```

- A class can both extend and implement.
  - One superclass only (extend).
  - But multiple implements.
- ArrayList is a concrete class so must override all inherited abstract methods and all methods declared in the interfaces.

List and ArrayList

- Often see this:

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

- Create an ArrayList object but access it via the List type.
- Code using list does not depend on ArrayList directly.
- Can substitute different concrete class:

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

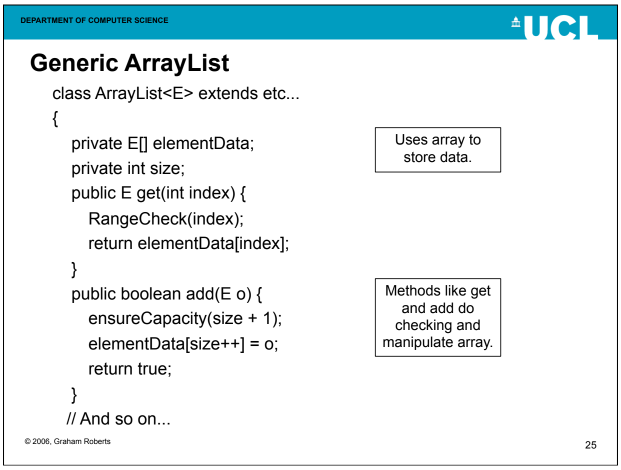
  - Discover linked list is a better data structure for current application.
  - Create different object but code using List type remains same.

Generic Interface

- Actually many of the classes/interfaces associated with ArrayList are *generic*.
- public interface List<E> // A generic interface

```
{
    boolean add(E obj);
    E get(int index);
    boolean isEmpty();
    // etc...
}
```

E is a type variable, *instantiated* during type checking.

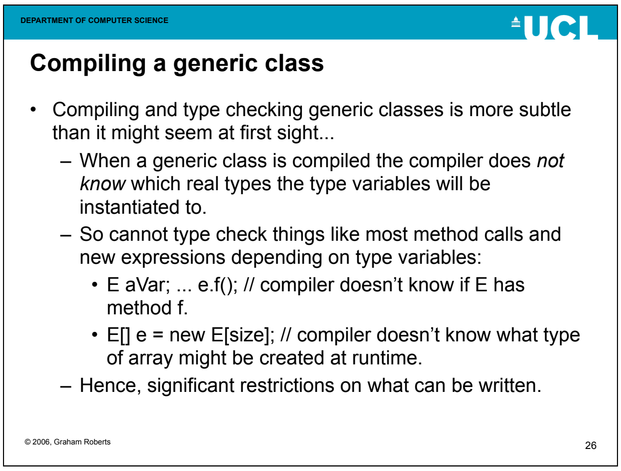


Generic ArrayList

```
class ArrayList<E> extends etc...
{
   private E[] elementData;
   private int size;
   public E get(int index) {
      RangeCheck(index);
      return elementData[index];
   }
   public boolean add(E o) {
      ensureCapacity(size + 1);
      elementData[size++] = o;
      return true;
   }
  // And so on...
```

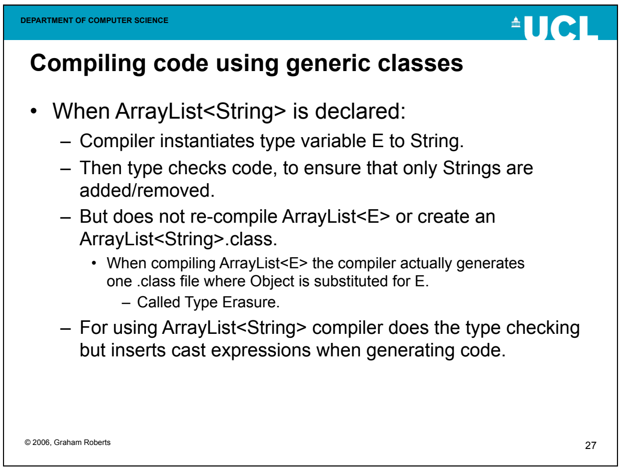
Uses array to store data.

Methods like get and add do checking and manipulate array.



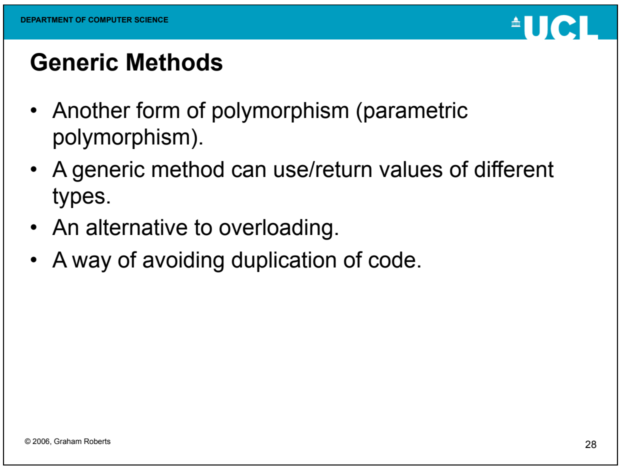
Compiling a generic class

- Compiling and type checking generic classes is more subtle than it might seem at first sight...
  - When a generic class is compiled the compiler does *not know* which real types the type variables will be instantiated to.
  - So cannot type check things like most method calls and new expressions depending on type variables:
    - E aVar; ... e.f(); // compiler doesn't know if E has method f.
    - E[] e = new E[size]; // compiler doesn't know what type of array might be created at runtime.
  - Hence, significant restrictions on what can be written.



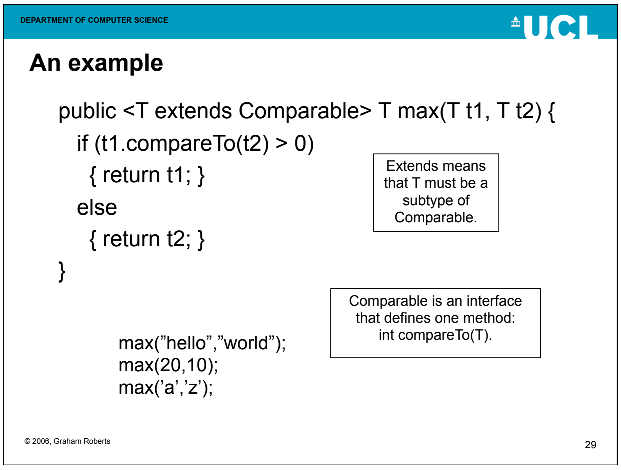
Compiling code using generic classes

- When ArrayList<String> is declared:
  - Compiler instantiates type variable E to String.
  - Then type checks code, to ensure that only Strings are added/removed.
  - But does not re-compile ArrayList<E> or create an ArrayList<String>.class.
    - When compiling ArrayList<E> the compiler actually generates one .class file where Object is substituted for E.
      - Called Type Erasure.
  - For using ArrayList<String> compiler does the type checking but inserts cast expressions when generating code.



Generic Methods

- Another form of polymorphism (parametric polymorphism).
- A generic method can use/return values of different types.
- An alternative to overloading.
- A way of avoiding duplication of code.



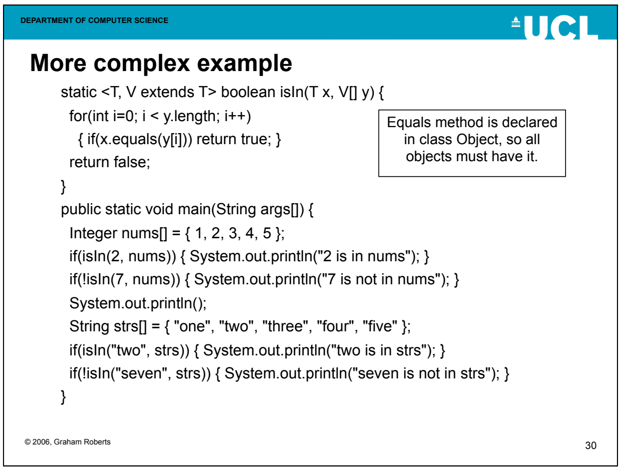
An example

```
public <T extends Comparable> T max(T t1, T t2) {
   if (t1.compareTo(t2) > 0)
     { return t1; }
   else
     { return t2; }
}


          max(”hello”,”world”);
          max(20,10);
          max(’a’,’z’);
```

Extends means that T must be a subtype of Comparable.

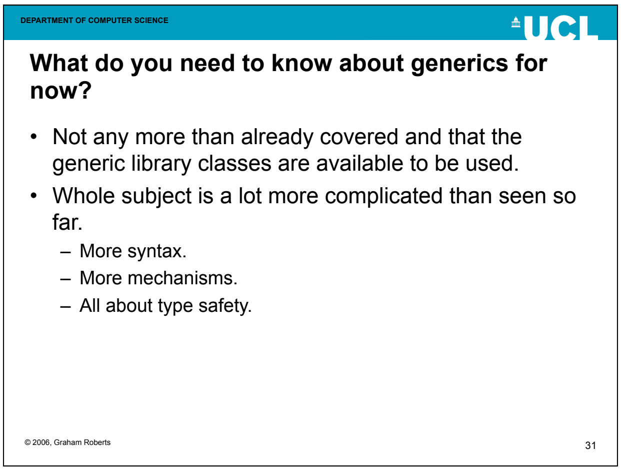
Comparable is an interface that defines one method: int compareTo(T).



More complex example

```
static <T, V extends T> boolean isIn(T x, V[] y) {
  for(int i=0; i < y.length; i++)
    { if(x.equals(y[i])) return true; }
  return false;
}
public static void main(String args[]) {
  Integer nums[] = { 1, 2, 3, 4, 5 };
  if(isIn(2, nums)) { System.out.println("2 is in nums"); }
  if(!isIn(7, nums)) { System.out.println("7 is not in nums"); }
  System.out.println();
  String strs[] = { "one", "two", "three", "four", "five" };
  if(isIn("two", strs)) { System.out.println("two is in strs"); }
  if(!isIn("seven", strs)) { System.out.println("seven is not in strs"); }
}
```

Equals method is declared in class Object, so all objects must have it.



What do you need to know about generics for now?

- Not any more than already covered and that the generic library classes are available to be used.
- Whole subject is a lot more complicated than seen so far.
  - More syntax.
  - More mechanisms.
  - All about type safety.