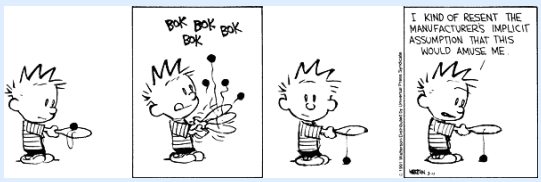


COMP2007
A Quick Review of Object-Oriented Programming and Java
+
An Overview of Java 6

Agenda

- Things you should know about Object-Oriented development.
- Things you should know about Java.

- If in doubt ask questions now!

Warning!

- These slides list the things you should know.
- Mostly I will go through them very quickly and NOT go into detail.
- Your job to spend the time filling in the gaps.

More Information

- Read the text book Developing Java Software.
- Use the online Java tutorials.
- Use the reading list.

Java Development

- There is an entire culture or body of knowledge programmers need to be familiar with.
  - Not just the programming language
- Tools
- Testing
- Idioms, patterns, architectures
- Design strategies
  - e.g., dependency injection
- Project organisation and management

Tools (1)

- Editor – JEdit.
  - www.jedit.org.
- NetBeans
  - Full Java Integrated Development Environment (IDE).
  - www.netbeans.org
- Eclipse
  - Another IDE
  - Open source, www.eclipse.org.
- Ant build tool - v.1.7
  - See ant.apache.org

Tools (2)

- TestNG unit test framework
  - testng.org
  - you must become proficient at using this.
    - Also JUnit unit test framework
      - www.junit.org
- Subversion
  - svn
  - Version control tool
  - subversion.tigris.org
- Also find out about tools like Maven, Hudson, FindBugs.

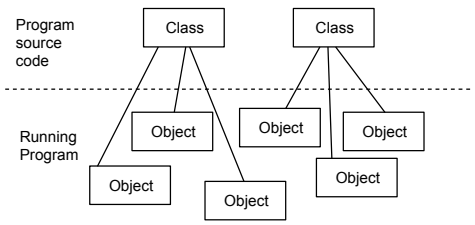
Class

- An abstraction.
- Public interface, private implementation.
- Instance objects.
- Scope, Encapsulation.
- Structural.
- Source code construct.

Object

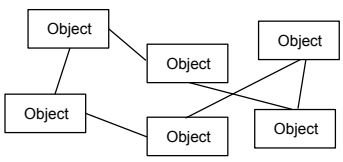
- Instance of a class.
- Exists when program runs.
- Contains private instance variables.
- Used by calling public methods.

Classes and Objects



Communicating Objects

- Object-oriented programs consist of communicating objects:

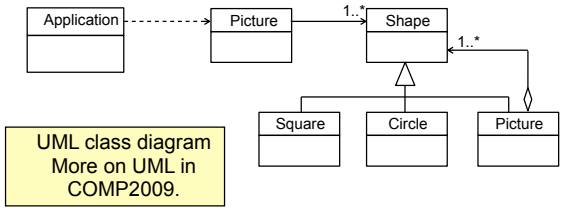


Classes…

- Have responsibilities
  - to describe behaviour (methods)
  - to know things (variables)
- Can collaborate with other classes to describe more complex behaviours.
  - Associations and dependencies.

Class relationships

- Association
- Inheritance
- Dependency



UML class diagram
More on UML in COMP2009.

Inheritance

- A superclass is a generalisation.
  - Shape defines the abstract properties of shapes in general.
- A subclass is a specialisation.
  - Square represents a specific kind of concrete shape.

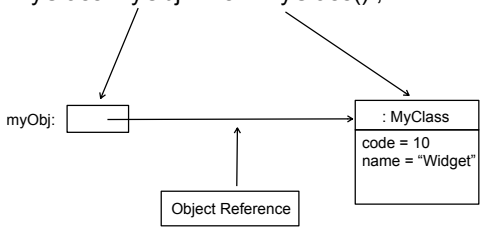
Levels of detail

- Classes and objects provide the components, or building blocks, to construct a program from.
- Statements in methods provide the detail describing how objects perform operations.
- Think about levels of detail, or abstraction.

Questions?

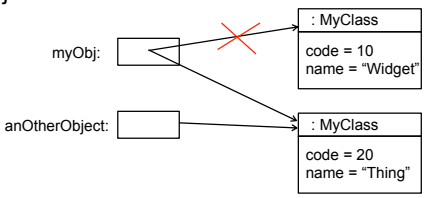
Object References

- MyClass myObj = new MyClass() ;



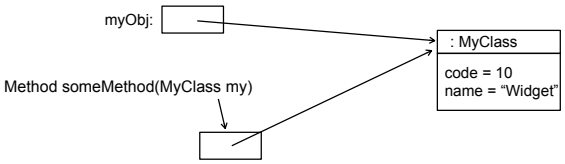
Reference Assignment

- Assignment means storing a reference to a different object:



Evaluate: myObj = anOtherObject ;

References and Parameters



Evaluate: someMethod(myObj) ;

Public, Private

- Specify encapsulation properties.
- All instance variables should be private.
- Public methods define the public interface of an object.
- Also protected.

Inheritance mechanisms

- class A extends class B
- All instance variables and methods inherited by B
  - B is an extension of A
- Private variables/methods not accessible.

Protected

Change Shape:

```
class Shape
{
    protected int x, y ;
    ...
}
```

A protected variable can be accessed from subclasses but not from any unrelated classes.

Overloading

- Two or more methods or constructors can have the same name.
- But must have different arguments.
  - String()
  - String(byte[])
  - String(char[])
  - String(String)
  - String(byte[], int)
- Return types don't matter.

Overriding

- A subclass can override an inherited method.
- A new method body is provided, specialised to the needs of the subclass.
- The method name, argument types and return type must be the same.

Dynamic binding

- Binding is the term used for the process of mapping a method call to a method body that can be executed.
- Dynamic binding means that the method body is determined at runtime by looking at the class of the object the method is called for.

Casting

- An exact type can be recovered using a cast expression:
- String s = (String)(a.get(0)) ;

Type of this expression is Object.

Explicitly convert the type to String

- A lot of the need for casting has been removed by Generics (more later).

final

- A final method cannot be overridden by a subclass.
  - public final String f(int i)
- The value of a final variable cannot be changed.
  - private final int n = 10 ;
  - public static final double PI = 3.141 ;

this

- this is special variable that is automatically declared in an instance method.
- It is a reference to the object the method was called for.
- Allows you to refer directly to the current object.

this (2)

- Can also be used to call a different overloaded constructor:

```
// This constructor does the real work
T(int x, int y, String z) { ... }

T() // Supply default values
{
  this(0,0,"Hello") ; // no duplication of
}                     // init code
```

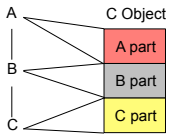
Super and Constructors

```
public Square(int px, int py, int sz)
{
  super(px,py) ;
      size = sz ;
}
```

- super allows reference to the superclass.
- When used in a constructor it results in a call to the superclass constructor with the matching argument list.

Subclass object initialisation - the general

- A constructor must be called for each inherited part of a C object.
- And the constructor bodies executed in the order A, B, C.



Abstract methods

- Declare the method abstract.
- public abstract void draw(Graphics g) ;
- No method body is given.
- Put down a marker that the method should exist in subclasses.
- A concrete subclass must override an abstract method.

Abstract class

- Declaring an abstract method forces the class to be declared abstract as well.

```
abstract class Shape
{
      ...
}
```

- An abstract class can have no instances.
- It is a partial description that can be inherited.

Interfaces

- An interface declares a new type:

```
public interface Queue<T>
{
      void addBack(final T x) ;
      T removeFront() ;
}
```


Interfaces (2)

- Declares public methods, but no method bodies.
- Can declare static variables but not instance variables.
- Cannot be used to instantiate objects.

Implementing an interface

- A class can implement one or more interfaces:

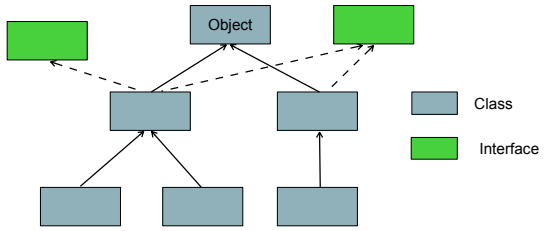
```
class QueueImpl<T> implements Queue<T>
{
    public void addBack(final T x)
    { ... }
    public T removeFront()
    { ... }
    ...
}
```


Interfaces (3)

- Many classes can implement the same interface.
- Objects of the classes conform to the type.
- A reference of the interface type can refer to any of the objects.
- Any methods declared by the interface can be called (dynamic binding).

Type Conformance

- Interfaces declare types independently of subclass relationships.



"Programming to an interface"

Inner and Member Classes

- It is possible to nest a class declaration within another class declaration.
- This opens up interesting possibilities at the cost of additional complexity…
- The following slides will give a quick overview.

Top-level Classes

- Name for classes that are not nested inside another class.

```
class X
{
}

class Y
{
}
```

```
X x = new X();
Y y = new Y();
```


Nested Top-level Classes

```
class X
{
  public static class Y
  {
  }
}
```

```
X x = new X() ;
X.Y y = new X.Y() ;
```

Why have a nested class?

- To remove the nested (inner) class name from the top level package scope.
- If private, to provide infrastructure for the implementation of the enclosing class.

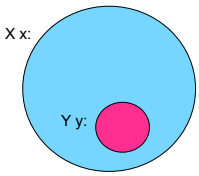
Member Classes

```
class X
{
      public class Y       // No static keyword
      {
      }
}
                  X x = new X() ;
                  X.Y y = new X.Y() ;
```

Looks the same as on the nested top-level slide but the semantics are different...

Member Classes (2)

The member class object is effectively contained inside the enclosing class object.



Containment Hierarchy

- An instance of a member class always has a reference to its parent object.
- A member class can access any methods and variables of the parent that are in scope.
- The member class object acts as an extension of the parent object.

Member Classes (3)

```
class memberTest
{
   public static void main()
   { X x = new X() ; }
}
```

Can now do this

```
class X
{
  private int z = 0 ;
  public class Y
  {
     public Y()
     { z = 3 ; }
  }
  public X()
  { Y y = new Y();}
}
```


Member Classes (4)

```
class memberTest
{
  public static void main()
  { X.Y y = new X.Y() ; } // Error here
}
```

Can only instantiate a member class from within the scope of the nesting class.

OK here

```
class X
{
   public class Y
   { }
}
```

Member Classes (5)

- The previous program generates the error message:
- No enclosing instance of class X is in scope; an explicit one must be provided when creating inner class X. Y, as in "outer. new Inner()" or "outer. super ()".

```
X.Y x = new X.Y() ;
      ^
```


Member Classes (6)

```
class memberTest
{
  public static void main()
  { X x = new X() ; }
}
```

Now OK

```
class X
{
  public class Y
  { ... }
  public X()
  { Y y = new Y();}
}
```


Beyond member classes

- There are also local classes
  - declared within a compound statement.
- Anonymous classes
  - classes that have no name.

Why???

- What use are these mechanisms?
- They allow one object to temporarily be an extension of another existing object.
- But in a safe way.

Yes, but why???

- Think of a method call on an object:
  - It can change the state of the object, take arguments, return a result.
  - But has a limited lifetime and internal structure.
- A member class object provides greatly extended behaviour:
  - unlimited lifetime
  - complex internal structure
  - bundle of methods and state

Exceptions

- try, catch, finally

```
try
{  statements that might throw exception }
catch (Exception e)
{ handle exception }
finally
{ always do this }
```

Exceptions (2)

- throw, throws

```
public void f(int i) throws MyException
{
  if (i < 10)
   { throw new MyException("Invalid parameter");}
  ...
  otherwise carry on
}
```


Library Classes and Javadoc

- Many reusable classes, providing wide range of services.
- Use the Javadoc!

More?

- Any other features to revise?
- A lot more information is in the text book (Developing Java Software)
- Get started on the exercises!

Java 6

- Java 6 now well established.
  - Java 5 continues in widespread use.
- Java language extended.
- Java class libraries modified and extended.
- Following slides give an overview but omit detail.
- See Javadoc for more information.
- Or visit java.sun.com or java.net.

Enhanced for Loop - should be familiar

```
public int sum(int[] a)
{
   int result = 0;
   for (int i : a)
    { result += i; }
   return result;
}
```

for (type var : collection)
{ loop body }
In each iteration, var is assigned next value in collection and can be used in loop body.

Generics

- Extended type system to allow type safety for collections, generic methods and classes.
- For example:
  - List<String> list = new ArrayList<String>();
  - list.add("hello"); // OK
  - list.add(new Integer(123)); // error
  - String a = list.get(0); // No cast needed
- Collection type specified in angle brackets.
- (Java Generics are not C++ templates.)

Autoboxing/Unboxing

- Automatic conversion between primitive and equivalent wrapper class types.
  - int <-> Integer

```
List<Integer> intList = new ArrayList<Integer>();
intList.add(1); // Adding type int
int n = intList.get(0); // int assignment
```

No cast expressions, no compile errors.

Generic Interface

```
public interface List<T>
{
  void add(T x);
  Iterator<T> iterator();
}
public interface Iterator<E>
{
  E next();
  boolean hasNext();
}
```

T, E are type variables/ parameters.

Generic Class

Generic class declared in same way, e.g.:

```
public class List<T>
{
  private T[] members = (T[]) new Object[20] ;
  public void add(T item) { ... }
  public T getHead( ) { ... }
   ...
}
```

This is important to understand.

Why T[] members = (T[]) new Object[20] ; ?

- Class List and code using List are compiled and type checked independently.
- The type(s) that T can be instantiated to when List <T> is *compiled* are *not known*.
- Cannot use `new T[20]` as the actual type of T is not known and will differ depending on how a List is declared (List <Integer>, List <String>, etc.).
- *Type Erasure* takes place when List is compiled and Object substituted for T.

Using List

- When List is used, e.g., List <Double>
- The type parameter T is instantiated to Double to allow public method calls to be type checked:

  double d = l.getHead();
- Method bodies in List are not type checked.
  - List is not recompiled for each type.
  - A single .class file is created, using type Object via type erasure.
- Behind the scenes the compiler inserts casts to make runtime type checking work.

Generic Method

```
void printCollection(Collection<?> c)
{
  for (Object e : c)
  {
    System.out.println(e);
  }
}
```

? is a wildcard type to match any type

Using Collection<Object> won't work as match has to be exact.

enum

- Enumerated types - new kind of class.
- Allows constants to be declared in type safe way

public enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY }

Day d = Day.TUESDAY;

No more public static final int TUESDAY = 2;
Can't write Day d = 2; // Only values in enum

enum can have methods

```
public enum InnerPlanet {
   MERCURY (3.303e+23, 2.4397e6), VENUS   (4.869e+24, 6.0518e6),
   EARTH   (5.976e+24, 6.37814e6),  MARS   (6.421e+23, 3.3972e6);
   private final double mass;   // in kilograms
   private final double radius; // in meters
   InnerPlanet(double mass, double radius) {
      this.mass = mass;
      this.radius = radius;
   }
   private double mass()   { return mass; }
   private double radius() { return radius; }
   public static final double G = 6.67300E-11;
   double surfaceGravity() {
      return G * mass / (radius * radius);
   }
   double surfaceWeight(double otherMass) {
      return otherMass * surfaceGravity();
   }
}
```

```
public static void main(String[] args) {
      double earthWeight = Double.parseDouble(args[0]);
      double mass = earthWeight/EARTH.surfaceGravity();
      for (InnerPlanet p : InnerPlanet.values())
        System.out.printf("Your weight on %s is %f%n",
                          p, p.surfaceWeight(mass));
   }
```


Varargs

- Arbitrary number of method parameters.

```
public void write(String... records) {
  for (String record: records)
    System.out.println(record);
}
```

- String... treated as String[].
- Method can be called with any number of arguments
  - write("one","two","three");

Annotations (metadata)

```
public @interface RequestForEnhancement {
   int    id();
   String synopsis();
   String engineer() default "[unassigned]";
   String date();    default "[unimplemented]";
}

@RequestForEnhancement(
   id       = 2868724,
   synopsis = "Enable time-travel",
   engineer = "Mr. Peabody",
   date     = "4/1/3007")
public static void travelThroughTime(Date destination) { ... }
```

Method is tagged with annotation to show why it has been added to the program.
Annotation is *data* so can be processed automatically by tools.

- Allows metadata to be attached to declarations.
- Metadata can be accessed at runtime using reflection or checked by the compiler.

Java 6

- Only summarised main additions to language.
- There is a *lot* more detail.
  - Major addition to class libraries to improve concurrency support.
- See the Javadoc.

Summary

- Basically you need to know everything covered in this slide set...
- Read books/tutorials.
- Write code
  - Best way to learn details is to write code.
  - Learn from your mistakes.
  - Don't be timid.
  - Don't assume you can get away without spending a lot of time writing code.