

© nabg

Java

classes
objects
~~pointers~~ *object reference variables*

Assuming some familiarity with C++ classes.

© nabg

A touch of class

- Why class?
 - encapsulation
 - packaging together data structures and the code that manipulates those structures
 - information hiding
 - implementation details are hidden; “client” code cannot become dependent on current structures (facilitating future changes and extensions)

© nabg

Encapsulation & information hiding

- Typically:
 - Private data members
 - Class instance (object) takes responsibility for all operations on data that it owns
 - Public function members (methods)
 - Program can ask an instance of this class to perform certain operations on the data that it owns
 - Private auxiliary functions
 - Public functions define an operation but implementation broken down (standard functional decomposition style) into a number of auxiliary functions
 - These only called from within body of public function, never from client code, so they are private

© nabg

Object-based programming

- Object-based programming:

A technique for simplifying overall programming problem by splitting it up into separately analysed parts.

 - Identify distinct types of data used in your program and operations done on those data
 - Define classes for these data types and implement member functions
 - Test these individual components
 - Only then do you write overall program that uses instance of these tested classes

© nabg

Are classes better?

- It depends on your application.
- Some C traditionalists still seem to dislike the idea of classes.
- Some student programmers claim to prefer C++ (which supports but does not require classes) over Java (which requires classes) on various largely spurious grounds of “not having to use classes”

© nabg

If your application problem is traditional “scientific” application ...

- Scientific application
 - Read in some numeric data characterizing some physical system (e.g. geological data relating to oil field) and fill in arrays (matrices)
 - Invoke a series of functions that transform the numeric data according to various physics formulae
 - Print out some derived value
- *Then you don't gain much from classes.*
- Use functional decomposition, break down into simple functions, work with global data

If your application is “limited access to database” or something similar ...

- Applications like
 - Receive input from user, validate input data, add record to database or file
 - Receive query from user, search data table or file, print selected report data
- Then you probably won't need to invent any classes of your own.
- Program hybrid style
 - Your code procedural
 - *Making use of existing classes in libraries*

... for everything else

- You are better off with classes.
- Classes allow a more complete decomposition of problem into small separately testable parts.

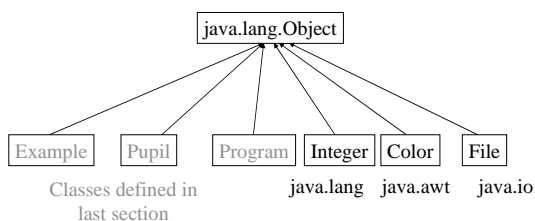
Decomposability ...

- DIVIDE AND CONQUER
 - Caesar
 - Good programmers
- Instead of thinking
 - How do I solve this big problem?
- Think
 - What identifiable kinds of data are there? What operations get done on those data?
 - Separate those out as an independent testable class
 - Compose overall program as “world of interacting objects”

A touch of class

- How class?
 - Generally similar to C++
 - *Where there are differences, Java has gone back to other older models (like original Simula67, Apple's Object Pascal (1980s), Eiffel, and to lesser extent Smalltalk)*
 - A pervasive change from C++
 - as in Smalltalk and Eiffel, all Java classes regarded as being extensions of a “base class” provided by the language (inevitably called `class Object`)

Class Hierarchy



Every object in a Java program is an instance of a class that extends `java.lang.Object` (either directly as here, or indirectly in more complex hierarchies)

Class Hierarchy

- Dealt with in more detail in subsequent lectures.
- Some terms (most should be familiar from C++)
 - “**concrete class**”: class for which your program can instantiate an object
 - Defines data members
 - Defines methods (member functions)
 - “**abstract class**”:
 - Often defines data members
 - Defines some methods
 - Declares other methods – leaving definition to subclass(es)
 - In C++ is class with some virtual methods defined as `abstract (=0)`
 - “**interface**” or “**pure abstract class**”
 - Simply declares methods – leaving definitions to subclasses (in C++, `class` with only virtual methods all defined as `=0`)
 - In Java, can define constant data members as well

Example of Java interface

```
interface Collection {
    boolean add(Object o)
    boolean addAll(Collection c)
    void clear()
    boolean contains(Object o)
    boolean containsAll(Collection c)
    boolean equals(Object o)
    int hashCode() Any collection object that
    boolean isEmpty() you encounter will be able to
    Iterator iterator() perform any of these operations
    boolean remove(Object o)
    boolean removeAll(Collection c)
    boolean retainAll(Collection c)
    int size()
    Object[] toArray()
    Object[] toArray(Object[] a)
}
```

Aside – terminology!

- You must be familiar with words that have one spelling but multiple meanings
 - match: something used to strike a light and set fire to things
 - match: person who is an equal
 - match: contest of skill
 - ...
- “Interface”

Interface

- interface
 - *formal technical term in Java*
 - Java construct that declares an abstract data type – a set of functions (and, rarely, constants)
- interface
 - *informal technical term, any OO language*
 - the set of all public functions of a class (those it declared and those it inherited)
- Graphical user interface (GUI)
 - A picture drawn on the screen of a computer with subareas that are monitored for mouse-inputs (controls)

interface

- You would think that these were reasonably distinct uses of a word, and understandable
- But, in a previous exam, some students tried to equate Java interface (abstract type declaration) and GUI.
- Sigh.

C++ v Java hierarchies

- Java has its base Object class from which every concrete class is derived; C++ does not have equivalent feature.
- C++ has very sophisticated model for creating subclasses that combine the features of more than one base class (*multiple inheritance*)
 - Too sophisticated, it is challenging for compilers
 - Too complex, average programmer cannot use C++ multiple inheritance effectively (or even correctly)
- Java syntax for declaring things like abstract classes is clearer than C++

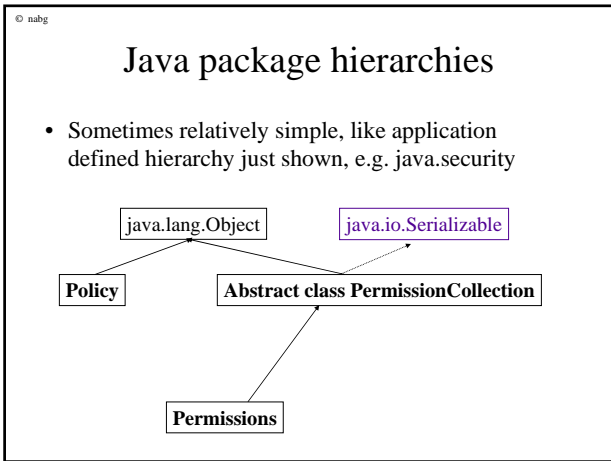
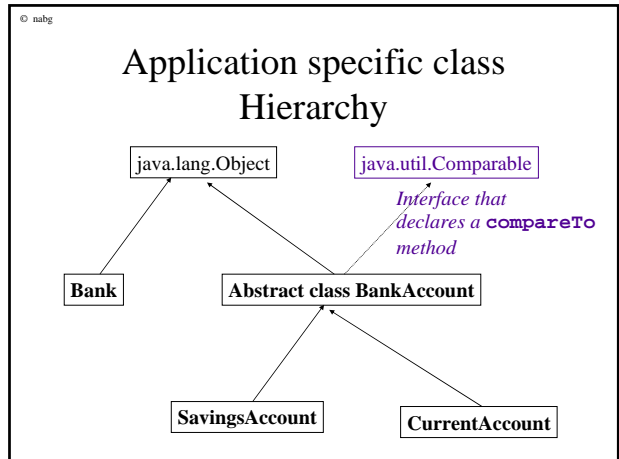
Inheritance versus *implements interface*

- Sometimes, all you want to say is that a class that you will define has a set of functions that have standard signatures (result type, function-name, argument types, exceptions)
- C++
 - Can use inheritance, your class inherits from a C++ pure abstract class that declares the functions
 - **May not be explicitly declared at all!** C++ template style, as in Standard Template Library (which you will meet in CSC1204), often depends on objects (used as arguments to template functions) supporting certain functions. C++ compiler checks the class definition when generating code.
- Java:
 - **implements interface**
 - Can implement several interfaces (superficially looks like multiple inheritance)

© nabg

Application class Hierarchies

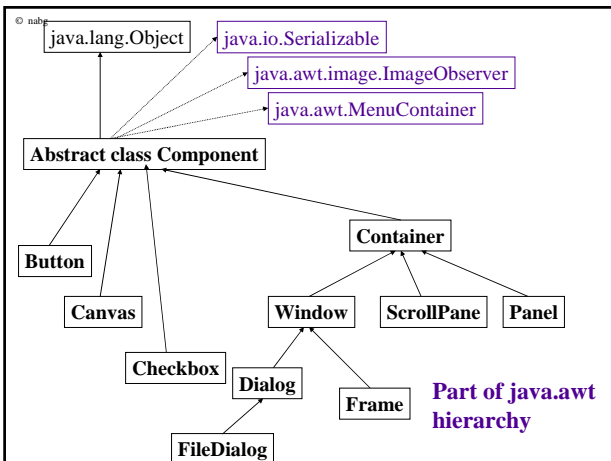
- Application defined classes –
 - Often just implicitly extending class Object
 - May implement interfaces defined in one or other Java package
 - Less commonly – have some application defined class hierarchy that captures commonalities (favourite text book example is “bank account” hierarchy with base class BankAccount and specializations like SavingsAccount and CurrentAccount)
 - So
 - Simple, shallow hierarchies



© nabg

Java package hierarchies

- Sometimes complex.
 - e.g. Classes in graphics package
 - Graphics objects have to work together in complex ways (e.g. when a “window” must be redrawn, it must tell any enclosed components to redraw themselves; when a component is resized, it may need to inform an enclosing window)
 - These behaviours can be defined in methods of abstract classes that form structure of complex hierarchy



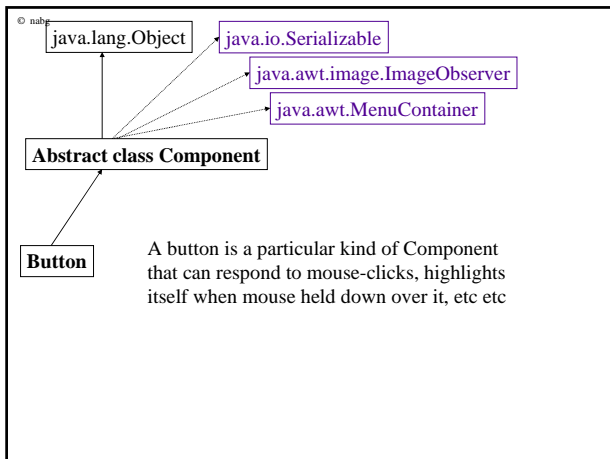
© nabg

```

classDiagram
    class java_lang_Object["java.lang.Object"]
    class java_io_Serializable["java.io.Serializable"]
    class java_awt_image_ImageObserver["java.awt.image.ImageObserver"]
    class java_awt_MenuContainer["java.awt.MenuContainer"]
    class Component["Abstract class Component"]

    java_lang_Object <|-- Component
    java_io_Serializable <|.. Component
    java_awt_image_ImageObserver <|.. Component
    java_awt_MenuContainer <|.. Component
  
```

- A Component:
 - extends java.lang.Object, so has everything an object has (like a lock, a hashcode etc)
 - Implements ImageObserver, so provides actual imageUpdate function
 - Implements getFont() and remove() as declared in MenuContainer
 - Defines lots and lots more functions! setEnabled(), getHeight() + 50 more




© nabg

Class hierarchies

- More in next lecture segment
rest of this lecture will simply be classes that extend `java.lang.Object` (extension of `java.lang.Object` is implicit, never explicitly stated)
- You will never have to design anything as complex as the `java.awt` hierarchy (*so stop worrying*)
- You will find that you can use classes from a complex hierarchy like `java.awt` without having to understand the structure and definition of the hierarchy

© nabg



A universe of classes

- One new contribution in Java
 - A systematic way of uniquely naming every class invented by any Java programmer on the network!
 - OK, not routinely used;

© nabg

Universe of classes

- Example
 - Oracle and IBM both have databases and must supply classes that allow Java programs to use their databases
 - Very similar classes needed; e.g. both need to supply a class that implements the `java.sql.Connection` interface
 - Have to be uniquely named

`oracle.jdbc.OracleConnection`
`com.ibm.db2.jdbc.app.DB2Connection`

© nabg

Universe of classes

- Qualified class name
 - Packages
- Package names supposedly based on URL of provider (IBM obeyed rules, Oracle didn't)
- *Student ead432 at Wol? Presumably something like:*
`au.edu.uow.ead432.assignment2.MyClass`

© nabg

Java classes

Class declaration



class declaration

- A class declaration will
 - specify any use of inheritance, if nothing specified then “extends Object” is implicit
 - Specify any interfaces for which this class will define implementation functions
 - define constants (if any)
 - define instance data members (type, access, initial value)
 - define class data members (shared static)
 - define instance and class member functions (Java prefers the Smalltalk term “methods”)

class declaration

- A class declaration is a single syntactic unit (following example of earlier languages Eiffel and Simula67) --- **it must contain the definitions of all member functions.**
- No restrictions on order of declaration of different members of a class.
- Each member declaration should include its “access specification” (public, private, protected)
 - (a default value applies if nothing specified, you don't continue with most recent specification as you do in C++)

C++ style – separate files, etc

```
class Demo {
public:
    Demo();
    void functionA(
    ...
private:
    ...
}
```

Demo.h

```
#include Demo.h

Demo::Demo()
{
    ...
}

void Demo::functionA(
{
    ...
}
```

One or more implementation files (Demo.cpp etc)

Java style: single module (file) with everything

```
public class Demo
{
    // Constructor
    public Demo()
    {
        ...
    }

    public void functionA(...)
    {
        ...
    }

    private void subA1 ( ) { ... }

    ...
}
```

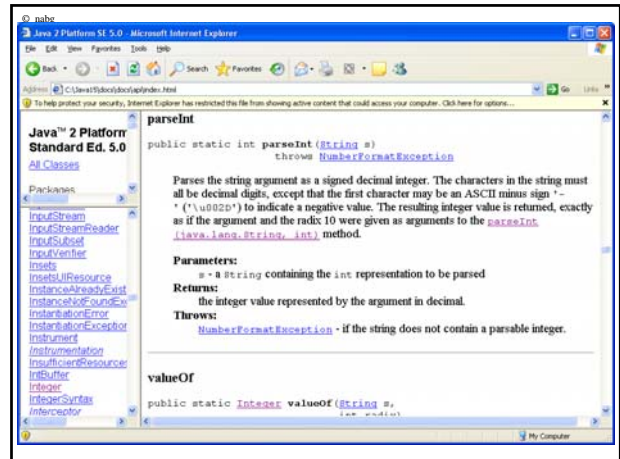
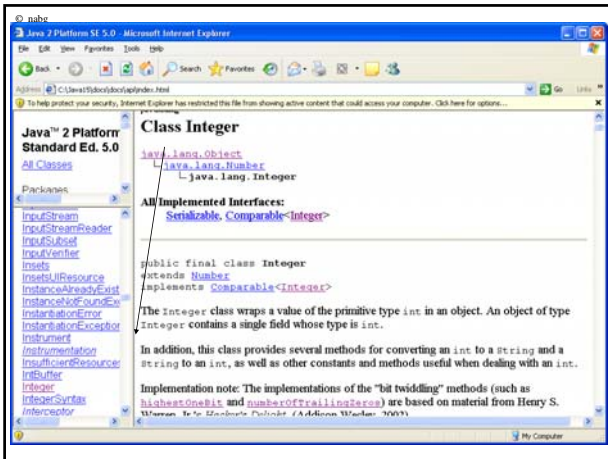


static qualifier

- “static” has same meaning in class declaration as it did in C++.
 - a “static” data member is one that is shared by all instances of the class
 - a “static” member function (method) is one that uses only static data members (or one that doesn't use any data members)

static qualifier

- As in C++, static member functions are invoked via the class (syntactic difference, in Java use
 - <class name>.<static function name>
 - rather than C++'s
 - <class name>::<static function name>)
- Had lots of examples already:
 - `int Integer.parseInt(String)`



Static member function

- Uses data supplied in arguments and:
 - either
 - only static data members (and constants)
 - Or
 - No other data (e.g. Integer.parseInt(arg))

Examples of shared static data

- Class Box (example associated with A1)


```
class Box implements Comparable {
    static int idCount = 0;
    int id;
    int width;
    int length;
    int height;
    String color;
    public Box(int w, int l, int h, String c) {
        width = w; length = l;
        height = h; color = c;
        id = ++idCount;
    }
}
```

Examples of shared static data

```
class Overdraft extends BankAccount {
    private static BigDecimal interestRate;
    public static getInterestRate() { return interestRate; }
    public static setInterestRate(BigDecimal newrate)
    { interestRate = newrate; }

    private BigDecimal customerLimit;
    private BigDecimal currentOverdraft;
    public BigDecimal monthlyInterest() {
        // some function of currentOverdraft and
        // interestRate
        ...
    }
}
```

Access controls

- Java has controls similar to, but not identical to those of C++.
- Access controls depend in part on new Java concept of a “package”
 - package --- a group of classes that in some way belong together, and whose instances often interact

Access controls

- Where C++ distinguished
 - class,
 - subclasses,
 - clients,
 - friends
- Java has
 - class,
 - other classes in same package,
 - subclasses in same package,
 - other subclasses,
 - clients.
- “package” access has some similarities to C++’s friend relations

	Public	Default	Protected	Private
“client”	Yes	No	No	No
Package	Yes	Yes	No	No
Subclass in package	Yes	Yes	Yes	No
Other subclass	Yes	No	Yes	No

C++ equivalent would be to make all classes of “Package” to be mutual friends.

Packages, files, “modules”, classes, ...

- File: something that your operating system understands, a unit for editing, copying etc
- Module: a unit for compilation (if C++ then typically two files - header and implementation; for Java it is one file); so, for Java “*a file is a module*”

Packages, files, “modules”, classes, ...

- If you want your classes to be reusable components, then tempting to think in terms like “*a file is a module is a class*”¶
- Each file then contains just one class
 - class Queue
 - class Structure
 - class MyCanvas
 - ...

¶a slogan from the OO language Eiffel

Packages, files, “modules”, classes, ...

- “*a file is a module is a class*”
In practice, that isn’t really appropriate.
- Your class List needs an auxiliary Link class
 - Although some other people might want to use your List class, no one else needs to know about ‘Links’.
 - Having separate file for Links just complicates life.

Java “module” (file)

- Generally, a Java module will contain some number of auxiliary classes along with a single principal class.
- A source file can only contain one ‘public’ class
 - if only one class defined in file, it is public¶
 - otherwise, one and only one class must be declared as public

¶Don’t rely too much on this, declare class as public, particularly if “Applet”

Inner classes

- Not covered in CSCI213
 - Bit of a “hack”
 - Can make code less lengthy, if you need a little helper class at one spot, you can define the class “inline”
 - Generally, makes code slightly more obscure
- Where might you see inner classes used?
 - Most likely in code relating to graphical user interfaces (where tend to need things like little helper classes that define how to handle something like “a window closing event”)
 - Textbooks will give examples
- Advice, *don't bother about inner classes for now, write the long winded but clearer code.*

package

- Java's packages provide an organizational unit greater than a single file
- Packages provide a defined and therefore clearer model of what we've been referring to as “*class libraries*”
 - a package is a set of modules, each module defines one (publicly known) class (and possibly some associated auxiliary classes)
 - the classes defined in the different modules in a package “belong together”

package

- Typical packages
 - java.math
 - classes for multi-digit arithmetic (BigDecimal etc)
 - java.io
 - classes for i/o streams
 - java.awt
 - Graphics User Interface classes

package access

- The awt classes provide examples of where the “package” access mechanisms might be useful
- GUI classes typically work closely together
 - windows interact with subwindows and menus etc
 - so, there may be places where want a Window to have more access to a Menu than would really wish to allow to an ordinary client

package and directory

- Just as a file must have the same name as the (principal) class that it contains,
- so a package name must match a directory name.

package and classnames

- It is this naming convention that provides the basis for the scheme that assigns a unique name for every class.
- Unique class name: computer domain name (actually reversed), pathname down to package directory, class name.

packages and imports

- Your classes can specify data members that are instances of other classes - specifying qualified class name

```
class Communications {
    java.net.Socket fMySocket;
    java.net.URL fURL;
    ...
    void Setup(String host, String file) {
        ...
        fURL = new java.net.URL ("http:", host, 80, file);
        ...
    }
}
```

packages and imports

- An import statement saves you from having to specify such qualified class names

```
import java.net.URL;
...
class Communications {
    URL fURL;
    ...
}
```

- You can import individual classes, or all classes in a package (import java.net.*;)

packages and imports

- If you do manage to import two class URL s from different packages, you will again have to use qualified class names.
- Example
 - java.util and java.sql both define Date classes
 - If you are using both collection classes and database classes in your class (quite a reasonable thing to do), you will import both java.util and java.sql
 - You then have to specify java.util.Date or java.sql.Date if you define any Date variables!

java.sql.Date is actually a specialization of java.util.Date that omits information that cannot be used in standard SQL databases

packages and imports CLASSPATH

- Most Java development environments require that you set a “CLASSPATH” environment variable.
- This essentially lists the directories where the Java compiler is to look for package subdirectories.

Your classes and packages

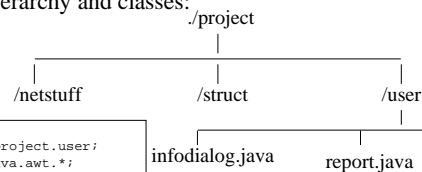
- If you don't specify anything about a package for your class, it goes into a default package. This is OK for small scale individual development (as needed in CSCI213).
- If you are developing something more substantial, or working in a group on a large assignment for a subject like CSCI311, then you should use packages.



NetBeans looks after defining packages for you, making their use easy enough for CSCI213

Your classes and packages

- A file must start with a “package” statement if its class is to be part of a specific package.
- Example, a larger program with data structures, networking and interfacing components; directory hierarchy and classes:



```
package project.user;
import java.awt.*;
public class infodialog {
```

Back to class declarations! Member functions

- From C++, you should remember:
 - accessor functions
 - allow “client” code read access to information belonging to an object
 - mutator functions
 - make changes to an object
 - constructors
 - initialize an object
 - destructors
 - release resources held by an object

Example : class Pupil

```
class Pupil
{
    private String name;
    private int mark;
    ...
}
```

Constructor

- Constructor – initialize data members when a new object of this class is created:

```
public Pupil(String aName, int aMark)
{
    name = aName; mark = aMark;
}
```

- Constructor(s)
 - No return type
 - Name same as class name
 - Argument list

Accessor

- Accessor – get data from object but don't change it:

```
public String getName() { return name; }
public int getMark() { return mark; }
public String toString()
{
    return name + "\t:+" + mark;
}
```

Mutator

- Change object

```
public void setName(String newName)
{
    name = newName;
}
```

Java

- Accessors and mutators
 - not much change from C++ (Except without the const tag that C++ can use to flag accessors – silly silly omission!)
- Constructors
 - Mostly the same in Java and C++
 - Obscure detail
 - Java changes working of overloaded constructors (one can call another, which you can't do in C++ --- this())
- Destructor?



Java & destructors

- **Destructor**
 - Which C++ classes had destructors? The resource manager classes.
 - Why did they have them? So that instances could free any resources that they had acquired.
 - What kinds of resources? Most commonly, the resource was memory (heap space used for auxiliary data); other resources were things like files, sockets, ...

Never heard of a destructor?

- If you took the new healthier “low-class” CSCI124, you may not have encountered “destructors”.
- You will enjoy meeting them in your later C++ studies.

Java & destructors

- Java has automated garbage collection so it is not necessary to explicitly free memory.
- So, “*Java doesn’t need destructors.*”
- *But what about other types of resource?*

finalize()

- finalize() serves roughly the same role as a destructor.
- When Java’s garbage collector decides to reclaim space for a discarded object, it first calls that object’s finalize() method (provided by class Object as a “do nothing”, and possibly redefined in subclass).
- The finalize() method can free other resources.

finalize() ?

- But, no guarantee of when garbage collector will get to an object!
- If the objects in your class do use system resources like files, you shouldn’t really rely on finalize().
- Put your resource release code in a method that you define (tidy_up(), dispose(), or follow class Applet and have a destroy() method)
- Most likely to encounter
 - Files and network connections – “close” as soon as possible
 - Database related objects – similar “close” result-sets, statements and connections

“dispose()”

- When you’ve finished with a ‘system resource manager’ object, then don’t simply forget it and leave it to the garbage collector.
- Remember to tell it to perform its tidy_up() or dispose() function.

Another C++/Java difference (relating to class hierarchies really)

- C++
 - Class defines a member function
 - **Can only be overridden** in subclass with changed definition if the member function declared **virtual**
- Java
 - Class defines a method
 - **Can be overridden** by in any subclass **unless** method originally declared as **final**
- C++ “**you can only redefine my functions if I give you permission**”
- Java – arbitrary redefinition (potentially cause of problems in large systems with many classes)

Mutual access (same as C++)

- Sometimes, instance methods of a class take arguments that represent other instances of same class. e.g.
`BigDecimal max(BigDecimal val)`
Returns the maximum of this BigDecimal and val.
- Class BigDecimal has some private data member that holds its value.
- Code for max() function can access the value fields of both BigDecimal objects – `this` and `val`
- Don't need to supply accessor functions!



Exception specifications

- In C++, a function declaration can optionally include an exception specification.
- In Java, specification is **not** optional. If a member function can throw an exception (or pass on an exception thrown by some other function that it calls) then this **must** be stated in the declaration.

Exception specifications

- Hence declarations like following:

```
class Integer ... {
    ...
    public static int parseInt(String s)
        throws NumberFormatException
    ...
}
```

Java's exceptions

- Java has a defined exception hierarchy
 - Class Throwable
 - Error
 - AssertionError, ... VirtualMachineError
 - Exception
 - RuntimeException – things like array subscript out of bounds (don't have to use try... catch for these)
 - About 50 immediate subclasses, e.g. IOException (which has itself some 20 subclasses)

Your exceptions

- Define a class that extends Exception

```
public class MyException extends Exception
{
    public MyException(String reason)
    {
        super(reason);
    }
}
```

Why define MyException?

- MyException doesn't do anything that could not have been done with a vanilla Exception.
- But it can be caught explicitly making try ... catch blocks more understandable

You want classes?
We've got classes!

Java supplied classes

- There are now several thousand classes distributed over more than 100+ package in the standard Java release.
- Look in these libraries before you start coding – what you want may already exist!

Java supplied classes

- Anything “standard” is already there!
- Standard?
 - Collections (lists, dynamic arrays, hashtable, etc)
 - Date, time, ...
 - Networking
 - Database access
 - Graphics User Interface
 - Random access and other files
 - Image manipulation
 - Data encryption and other security requirements
 - ...

Classes used in CSCI213

- java.util Collection classes –
 - Vector, TreeSet, Hashtable, possibly others
- java.io
 - BufferedReader, PrintWriter, FileReader, ...
- java.net
 - (probably) Socket, ServerSocket
- java.sql
 - (probably) Connection, Statement, ResultSet
- java.awt and javax.swing
 - Graphical user interface classes

You are NOT taught how to use class X for X=...

- Classes are supposed to be adequately documented so that you determine how to use object by looking at interface description
- e.g. Vector
 - Simple dynamic array for storing objects
 - Look it up in the on-line Java documentation
 - Find its methods
 - size(), add(), elementAt(), remove(), ...
 - If have problems, use the other links in the online documentation
 - Sun's tutorial at <http://java.sun.com/docs/books/tutorial/index.html>
 - Annotated reference to (some of) classes

For the future ...

- In future, you may be using
 - Security
 - 2D- graphics
 - 3D- graphics
 - Sound
 - Printing
 - ...
- Need to learn now how to exploit documentation to find use of classes

Your classes ...



Creating your own classes

- Firstly
 - *owns, does analysis for problem domain*
 - leading to “fuzzy blob” model for class
 - List of principal data members
 - List of public functions

Creating your own classes

- Secondly, detailed design
 - choice of data types for data members (remembering that there are lots of classes readily available in libraries (packages) so you should expect to use things like “Date”s and, “String”s, rather than lots of data members of primitive types - integers, byte arrays etc)
 - expansion of functionality into pseudo code outlines
 - development of function prototypes
 - decisions on accessibility of members
 - consideration of any shared (class) data (static members)
 - will you want a constructor (often no, default initialization of data members may suffice)
 - do you need any “tidy up” function (mostly no)

Creating your own classes

- Decide
 - *Is this class a major component of system, or a simple auxiliary thing?*
 - *If major component, create a new file for the class*
 - *If a simple auxiliary thing, put class declaration in with the more major component that uses it – well that would be normal but with NetBeans simpler to use a separate file*

Creating your own classes

- Start any new file with:
 - package declaration (if required)
 - import statements for
 - individual classes
 - entire packages (if using many classes from same package)
 - append class declaration(s)
 - although there are no language restrictions, it is worth following some fixed style e.g. having “public” parts appear first, followed by “private” implementation details

Your classes ...

- Design
 - Identify data owned
 - Determine functions supported (public interface)
- Detailed design
 - Determine implementation of public functions (private auxiliary functions)
- Implement
- Devise a test program

Design and testing of a class

- Example, suppose you didn't know that `java.util.Vector` existed and you had decided to implement `DynamicArray`
- Design
 - A dynamic array owns:
 - Pointer to stack based array, count of number elements present, size limit variable
 - A dynamic array supports public functions:
 - `Size()`, `add(...)`, `getElementAt(...)`,
- Detailed design
 - ? Probably all methods sufficiently simple that don't require auxiliary private member functions!

```
public class DynamicArray
{
    private static final int kDEFAULTSIZE = 10;
    private int _count;
    private int _size;
    private Object[] _data;

    public DynamicArray() {
        _data = new Object[kDEFAULTSIZE];
        _size = kDEFAULTSIZE; _count= 0;
    }
    ...
    public int size() { return _count; }
    public int capacity() { return _size; }
    ...

    public static void main(String[] args)
    { System.out.println("Testing dynamic array");
      ...
    }
}
```

Including a main() that does simple testing?

- If a class represents a simple easily tested data type, then quite common for there to be a `main()` defined in the class.
- This `main()` instantiates the class and runs some simple tests.
- (Java doesn't mind many classes declaring `main()` methods, when you start the run-time with the **java** command you specify a class-name; it is this class whose `main()` will be run.)

Separate test program

- If class is more elaborate, create a separate test program
 - Instantiate the class
 - Invoke each member function in appropriate context – test results; e.g.
 - Create `DynamicArray`
 - Add three objects
 - Ask if capacity is now 10 and size is now 3, if either fail then report failure
 - If functionality more complex
 - Have standard input files of data associated with test program
 - Have files that show expected output
 - Run test – do file comparison;

- Classes
- Objects
- Object Reference Variables



Objects

- All objects (class instances and arrays) are dynamic structures
 - created on the heap (using **new** operator)
 - accessed via a **pointer** object reference variable
- Objects are garbage collected when no longer referenced

Objects

- All classes derive from **class Object**, so all their instances possess a few standard properties
 - one property is an individual (one per object) synchronization lock
 - will meet later when consider multithreaded programs
 - it is another example of a Java convenience feature: you can have multithreading in C++ (using a more comprehensive thread support system from OS) but you have to do a lot of work by hand (work that is automated by a Java compiler) and you can have no expectation of consistency for threading as implemented in different libraries.
 - several of methods defined in class Object relate to this lock (they are **final** methods, subclasses can not change locking behaviors)

Objects

- Other behaviors possessed by all Objects include
 - `toString()` String representation of state of object, mainly for debugging; but can override in your class and define a function that returns a String representation of data in object – can be useful as can then simply use `println()` on instance of class
 - `finalize()` Tidy up when memory gets repossessed
 - `clone()` May duplicate memory representation of object (but only in subclasses that explicitly agree to support cloning)
 - `hashCode()` Return an int that represents content
- If you don't define these in your own class, you will get a minimal (“do nothing”) version.

Objects

- Java's collection classes work with Objects (bit like `void*` collections in C++)
 - *dictionary, stack, hashtable, and vector.*
- You put something into collection, you get back “an Object”.
- Type casts are needed when working with such collections (of course, with Java 1.5 have option of defining a type secure “template” collection)

Objects

- “Identity”
 - Some text books will go on at length about another characteristic of objects --- their “possession of identity”.
 - All that it means is that different instances of same class (even if have identical values in their data fields) are distinguishable.
 - Of course different instances of same class will be distinguishable, *they will be located at different points in heap.*

Objects : identity

- Object identity is (in practical terms) synonymous with object address.
- If you have two object reference variables


```
Phone a;
...
Phone b;
... a = new Phone(); ...
b = theDirectory.PhoneWithNumber(71571); ...
if(a == b) ...
```
- a test (`a == b`) tests the addresses held in these variables (i.e. do they refer to the same object?)

Objects : equality

- If you want to check whether two objects (instances of the same class) are “equal”, you have to provide an `equals()` function in your class
 - this function would compare the values in successive data members

`equals()`

- Many Java standard classes define `equals`
 - `String` (character-by-character comparison)
 - `Date` (same timestamp)
- You can define `equals` for any class you create –
 - Compare all the data members checking equality
- *Relatively rare to need to define `equals()`*



Garbage collection

- Very old trick – invented in 1950s, first standardized with Lisp (from MIT in 1960)
- Garbage collection
 - Depends on being able to identify pointer variables at run-time (could do in Lisp, can do in Java, cannot really do in C++)
 - These pointers can be found on stack (local variables of functions in current call chain) and within structures allocated on heap (pointers to other structures – e.g. Java instance data members of Object types)

Garbage collection (contd)

- Mark and sweep approach
 - Keep running until get low on heap space
 - Mark heap structures in use
 - Start with pointers in stack
 - Follow pointer to structure, mark as in use
 - If structure has pointer members
 - Follow each of these to other structures, mark these in use
 - Recurse
 - Once all “live” structures marked, sweep through heap freeing up space associated with unmarked structures.

Garbage collection

- Java uses more sophisticated algorithms than mark and sweep.
 - Generational garbage collection
 - Separate gc thread

Garbage collection

- C++:
 - Too much responsibility
 - Must remember to delete objects when no longer require them
 - If forget to delete – die from memory leak
 - If delete to early (while object still being used) – die with “dangling pointer”
 - Error prone
 - Can be efficient (get rid of memory at clearly defined points in code when you wish to clean up)
- Java with garbage collector
 - Easy
 - Somewhat expensive

Garbage collection

- In Java you basically don't worry about creating and destroying objects (in C++, you had better worry!)
- Most of time, garbage collection works fine and you don't have to think about it
 - “incremental garbage” collection means that very rarely get situation where program stops doing useful processing because it needs to devote all time to cleaning up heap
- *Note: memory leaks still possible (you keep live references to objects in some collection long after you ceased to use those objects)*

- Classes
- Objects
- Object Reference Variables

Object reference variables

- Variables that are declared as being of class types are “object reference variables”
- Essentially identical to pointers
 - contain either null or address returned by **new** operator
- Unlike pointers in C/C++, the values in Java's “object reference variables” cannot be used in calculations - only use is for accessing an object.

Object reference variables

- Like built-in types int, float, double
 - can be data members of a class
 - can be local (STACK) variables of a function

```
void process(Student s)
{
    String name = null;
    double scaledmark = 0.0
```

pointer arg ↗
return address
pointer local variable name
double (8byte) local variable

Stack frame

Object reference variables

- Access to referenced object uses a syntax similar to C/C++ “member access operator” for a structure.

```
URL aURL = new URL("http:", "www.java.com", 80,
                  "info.html");
...
if (aURL.equals(anotherURL)) ...;
...
thePort = aURL.getPort();
...
```

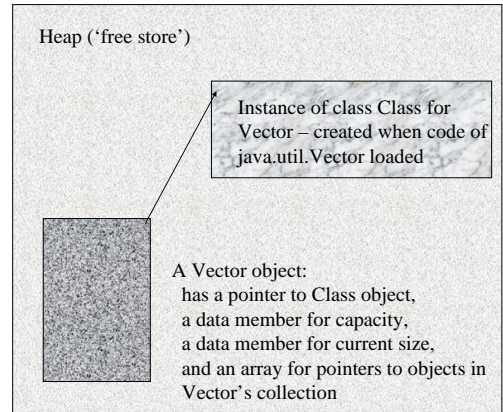
Object access involves pointer de-referencing

- C++ has
 - “.” operator to access fields (data and function) of structs and class instances created in static data space or in stack
 - “->” operator to access fields of structs and class instances created in heap
- Java uses “.” when accessing members of an object variable – but think more in terms of analogy to C++'s “->” operator.

Object reference variables

- Mechanism for invoking a member function for an object is essentially identical to that used for virtual functions in C++
 - use value in ~~pointer~~ object reference variable to get to data structure and thence to its “virtual table”
 - access table for entry for required function, getting its address
 - push object address onto stack, push values of other arguments onto stack
 - call function at address just determined
 - on return, clean up stack

While C++ uses a fairly simple table of function pointers as “virtual table” for a class, Java uses an elaborate object, an instance of class `Class`, that contains lots of data that are used to check function calls.



class `Class`

- Much more elaborate than C++ virtual table
- Lots of information for run-time checking
- Slower but safer than C++

“this”

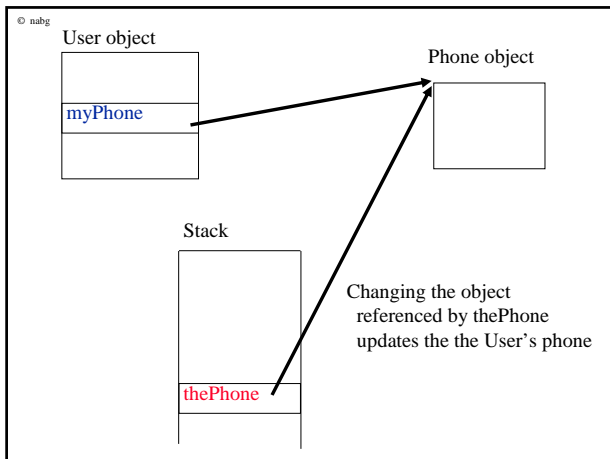
- As in C++, all instance member functions have an implicit “this” parameter (an object reference variable holding address of object that is executing the function).
- As in C++, you *can* qualify accesses to data members and member functions with `this.`, but it is not a common style.

```
class Point {
    int h, v;
    public void
        Move(int dh, int dv) {
        this.h += dh;
        v += dv;
    }
}
```

Passing object references to functions

- Java passes arguments by value
 - For primitive types (int, double etc), get copy of value pushed onto stack
 - For object reference types, get an address value pushed on stack
- You can pass an object reference variable to a function
 - while the function is being executed, you will have (at least two) object reference variables referring to same object (the one in calling environment, the other in the stack frame of the called function)

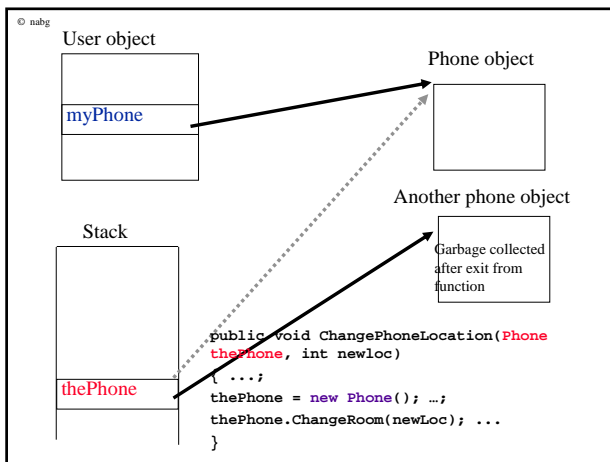
```
class Phone {
    ...
    public void ChangeRoom(int newRoom)
        { fRoom = newRoom; }
    ...
    private int fRoom;
}
class Directory {
    ...
    public void ChangePhoneLocation(Phone thePhone,
        int newLoc)
        { ...; thePhone.ChangeRoom(newLoc); ... }
}
class User {
    Phone myPhone;
    void Move() { ...;
        mainDirectory.ChangePhoneLocation(myPhone, rnum)
    }
}
```



© nabg

Function arguments

- Called function changes value of primitive type argument – no change in value of variable in calling environment
- Called function changes object referenced by object reference variable – after return, calling environment sees that argument object has been changed
- Called function changes object referenced by argument object reference variable – no effect on object reference in calling code.



© nabg

Initialization of object reference variables

- Often see code in student's program like the following:

```

String aName = new String();
aName = input.readLine();
...
Date dueDate = new Date();
dueDate = someFunctionThatReturnsADate();

```

- Such code is not actually erroneous (program runs still) but it is inappropriate

© nabg

Don't create an object just to throw it in the garbage a moment later

```

String aName = new String();
Date dueDate = new Date();

```

- Both those statements create new objects in the heap and invoke default initialization code.
- These objects are immediately thrown away when the object reference variables (pointers) get assigned new values that are the results from function calls.
- Object creation, and garbage collection, is not free. If you have such code in some loop, your program will be wasting a lot of time.

© nabg

Initialization – done right

```

String aName = null;
...
aName = input.readLine();
...
Date dueDate = null;
...
dueDate = someFunctionThatReturnsADate();

```

- Or, simply

```

String aName = input.readLine();
...
Date dueDate = someFunctionThatReturnsADate();

```

Example

(Depending on time constraints, this material may not be explicitly reviewed in the lectures)

DynamicArray

- Create a DynamicArray class for storing Java objects
 - Constructors
 - Default, Initial size
 - Methods
 - Size - current number of elements
 - Capacity - number of elements that could be stored without resizing
 - Add - add an element (must not be null – if null, throw exception)
 - getElementAt - access an element (should throw defined exception if asked for non-existent element)
 - Merge - create new DynamicArray that contains all elements from two DynamicArrays

Classes needed ...

- (Class specification pretty hopeless! This is just a tiny demonstration.)
- DynamicArray
 - As previously illustrated
- MyException
 - Runs DynamicArray through its paces

```
public class DynamicArray
{
    private static final int kDEFAULTSIZE = 10;
    private static final int kDEFAULTINCREMENT = 10;
    private int _count;
    private int _size;
    private Object[] _data;

    // Methods
    ...
}
```

```
public class DynamicArray
{
    ...
    public DynamicArray() {
        _data = new Object[kDEFAULTSIZE];
        _size = kDEFAULTSIZE; _count = 0;
    }
    public DynamicArray(int size) throws MyException {
        if(size<0) throw
            new MyException(
                "Negative size not permitted");
        try {
            _data = new Object[size];
            _size = size;
            _count = 0;
        }
        catch(java.lang.OutOfMemoryError oome) {
            throw new MyException(
                "Can't have dynamic arrays that large");
        }
    }
}
```

```
public class DynamicArray
{
    ...
    public int size() { return _count; }
    public int capacity() { return _size; }
    public void add(Object o) throws MyException
    {
        if(o==null) throw
            new MyException(
                "No nulls in DynamicArrays!");
        if(_count==_size) {
            Object[] tmp = new
                Object[_size+kDEFAULTINCREMENT];
            for(int i=0;i<_count;i++)
                tmp[i] = _data[i];
            _data = tmp;
            _size += kDEFAULTINCREMENT;
        }
        _data[_count++] = o;
    }
    ...
}
```

© nabg

Static (class) method merge

- Creates new DynamicArray
- Class method, operating on two dynamic array arguments; could have had instance method that that took single argument for other DynamicArray
- Code of class method can access private data members of instance arguments
- Must have try ... catch ... for adding null data elements (though know these cannot occur)

© nabg

```
public class DynamicArray
{
    ...
    public static DynamicArray merge(
        DynamicArray d1, DynamicArray d2)
    {
        DynamicArray result = new DynamicArray();
        try {
            for(int i=0;i<d1._count;i++)
                result.add(d1._data[i]);
            for(int i=0;i<d2._count;i++)
                result.add(d2._data[i]);
        }
        catch(MyException me) {
            // Can confidently say this won't occur
            // Neither array can contain nulls!
        }
        return result;
    }
}
```

© nabg

Test program

- Try sequence of tests
 - Test constructors first, are the arrays of size and capacity specified
 - Test if can add some data, and then check whether data added are there
 - Test some of conditions that should cause exceptions to be thrown (like adding null)
 - Test merge operation
- Example program continues test sequence even if some fail; when testing more sophisticated classes, better to test simple functionality first and terminate as soon as get a failure

© nabg

```
public class Test
{
    public static void main(String[] args)
    {
        test1();
        test2();
        test3();
        test4();
        test5();
    }
    private static void test1() { ... }
    private static void test2() { ... }
    private static void test3() { ... }
    private static void test4() { ... }
    private static void test4() { ... }
}
```

```
private static void test1()
{
    System.out.println("Trying constructors");
    System.out.println("Default constructor - should give
DynamicArray d1 = new DynamicArray();
System.out.println(d1.toString());
System.out.println("Capacity " + d1.capacity());
System.out.println("Size " + d1.size());
System.out.println("Trying other constructor - should
DynamicArray d2 = null;
try {
    d2 = new DynamicArray(15);
}
catch(MyException me) {
    System.out.println(me.getMessage());
    System.out.println("Shouldn't have an exception
System.exit(1);
}
System.out.println(d2.toString());
System.out.println("Capacity " + d2.capacity());
System.out.println("Size " + d2.size());
}
```

© nabg

```
private static void test2()
{
    String[] data = {
        "zero", "one", "two", "three", "four", "five",
        "six", "seven", "eight", "nine", "ten"
    };
    System.out.println("Trying to create a DynamicArray
System.out.println("Should report size 11 capacity 2
DynamicArray d1 = new DynamicArray();
for(int i=0;i<data.length;i++) {
    try {
        d1.add(data[i]);
    }
    catch(MyException me) {
        System.out.println(me.getMessage());
        System.out.println("?. shouldn't have got
    }
}
System.out.println(d1.toString());
System.out.println("Capacity " + d1.capacity());
System.out.println("Size " + d1.size());
...
}
```

```

private static void test2()
{
    ...
    System.out.println(d1.toString());
    System.out.println("Capacity " + d1.capacity());
    System.out.println("Size " + d1.size());
    System.out.println("Listing elements, should be words");
    for(int i=0;i<d1.size();i++) {
        try {
            System.out.println(d1.elementAt(i));
        }
        catch(MyException me) {
            System.out.println("Uhm, shouldn't happen");
            System.out.println(me.getMessage());
        }
    }
}

```

```

private static void test3()
{
    System.out.println(
        "Try creating illegal sized arrays" +
        " - should get reports from two exceptions");

    DynamicArray d1 = null;
    try {
        d1 = new DynamicArray(-27);
    }
    catch(MyException me) {
        System.out.println(me.getMessage());
    }
    try {
        d1 = new DynamicArray(800000000);
    }
    catch(MyException me) {
        System.out.println(me.getMessage());
    }
}

```

```

private static void test4()
{
    DynamicArray d1 = new DynamicArray();
    System.out.println("Seek exception from adding null");
    try {
        d1.add("Hello");
        d1.add(" world");
        // Following elements should not be added
        d1.add(null);
        d1.add("Hi mom");
    }
    catch(MyException me) {
        System.out.println(me.getMessage());
    }
    System.out.println(
        "DynamicArray should have two elements - Hello and world");
    System.out.println(d1.toString());
    System.out.println("Capacity " + d1.capacity());
    System.out.println("Size " + d1.size());
    for(int i=0;i<d1.size();i++) { ... }
}

```

```

private static void test5()
{
    System.out.println("Trying the static merge function");
    String[] data = {
        "zero", "one", "two", "three", "four", "five",
        "six", "seven", "eight", "nine", "ten"
    };
    DynamicArray d1 = new DynamicArray();
    for(int i=0;i<data.length;i++) {
        /* add data to d1 */
        ...
    }
    String[] data2 = {
        "hello", "world", "hi", "mom"
    };
    DynamicArray d2 = new DynamicArray();
    for(int i=0;i<data2.length;i++) {
        /* add data2 to d2 */
        ...
    }
}

```

```

private static void test5()
{
    ...

    DynamicArray merged = DynamicArray.merge(d1, d2);
    System.out.println("Should have an array capacity 20, "
        + " size 15");
    System.out.println(merged.toString());
    System.out.println("Capacity " + merged.capacity());
    System.out.println("Size " + merged.size());
    System.out.println("Contents should be strings zero..");
    for(int i=0;i<merged.size();i++) {
        ...
    }
}

```

© nabg

JUnit

JUnit test framework

- JUnit
 - Semi-automated system for
 - creating test programs for your classes
 - Running tests
 - Too sophisticated for CSCI213, maybe later