COMP1002
Data Structures and Algorithms

Lecture 1a: Object-Orientation Concepts

Department of Computing

Last updated: February 20, 2017

Copyright Warning

This Week

- Standards
- Review: Primitives vs classes
- Data Structures and ADTs
- ADTs vs data structures vs objects
- Relationships between classes (UML)
- Inheritance
  - Polymorphism, singly-rooted inheritance hierarchies
- Interfaces
  - vs abstract classes
  - Implementation vs interface inheritance
  - Support for less-dangerous multiple inheritance
- Exceptions

Coding Standards

- OOPD rules apply!
  - see JavaCodingStandard.pdf
- Indentation, braces and spacing
  - Indentation is by far the most important for readability
    - » You will be marked on this in the assignment and prac submissions!
  - Examples:

```
//if you are writing a book to save paper (bad!)

public void ExampleFunc(Object param1, int param2) {
   if ( (param1 != null) && (param2 > 0) ) {
      for (int i = 0; i < param2; i++)
         DoSomething(i)
   }
   else if (param2 < 0) {
      DoSomethingElse(param2);
   }
}
```

```
//if you are coding (nice)...

public void ExampleFunc2(Object param1, int param2)
{
   if ((param1 != null) && (param2 > 0))
   {
      for (int i=0; i<param2; i++)
      {
         DoSomething(i)
      }
   }
   else if (param2 < 0)
   {
      DoSomethingElse(param2);
   }
}
```

Coding Standards

- Naming
  - Ensure you give variables and methods *meaningful* names!
  - It's usual to use mixed upper/lower case rather than '_'
    - » eg: int numElements; vs int num_elements;
  - Variable names commonly start with a lower case letter
  - In Java, methods start with a lower case letter too
    - » Some others (notably Microsoft) start functions with upper case
  - Be as brief as possible with names without losing meaning
    - » eg: int numElements; vs int theNumberOfElements;
  - Method names tend to be verb-based ('doing')

Coding Standards

- You may also wish to use 'Hungarian notation'
  - Prefix *variable* names with a letter indicating their data type
    - » i = int/long, f = float/double, b = boolean, s = String, etc
      - Only do this for common data types: o = object is useless!
      - Keep the list of letters you use short - simple is better
    - » eg: iNumElements - the 'i' indicates the variable is an integer
    - » Don't use it for method names - that just looks ugly
    - » Don't use it with method parameter names since it makes the methods (which are a class's public interface) look messy
- Whatever you decide, *be consistent!*

Coding Standards

- Loop counter variables
  - It's typical to use one-letter variables like i, j, or k as the loop counter variable
    - » A holdover from subscript variables in mathematics

```
for (int ii = 0; ii < 10; ii++) {
   sum += value[ii];
}
```

  - Use it when there's no obvious name for the loop counter
  - You may also wish to use 'doubled' variants like ii, jj, kk.
    - » Makes life a LOT easier when trying to find every case where the counter variable is being used inside the loop! In contrast, think of how many times the letter 'i' occurs (7 times in this sentence alone)

Synonyms

- Submodule = Procedure = Function = Method
  - Function and method are by far the most common
  - Method: Special case : "a function that is part of a class"
    - » Important since there is data (class fields) that is shared amongst all the methods of an object
    - » Class fields: private data items of a class. Each created object gets its own set of data items to work with
      - AKA member fields, class members, fields
- Imports = Parameters = Arguments
  - Imports is the pseudocode term
  - Parameters is the code term (the list of imports in the brackets of a method header)
  - Arguments are what we pass to the method parameters via the method call.

Declaring Variables vs Using Variables

- Declaring a variable is the line that defines the type and name of the variable
  - There are three places where variables can be declared:
    - » Local variable declared inside a method
    - » Parameter of a method (*i.e.*, imports)
    - » Class field of a class. Need to define if private or public
- Variables have *scope*
  - A variable is available from the line it is declared but only lasts within the bounds of the { … } it was declared in:
    - » Parameter: lasts for duration of the method call
    - » Local variable: lasts for duration of the { … } that it was defined within (typically a method, but can be only a for loop too)
    - » Class field: lasts for duration of the object

Declaring Variables vs Using Variables

- Class fields also have *visibility* (as do methods)
  - Defines who is allowed to 'see' the item
    - » private – Only methods of the class can access the variable
    - » public – Anyone can access the variable (or method)
    - » protected – Only methods of the class *or inheriting classes* can see it
  - Local variables and parameters don't have this (*always* private)

Primitives vs Classes

- Java, C#, C++ and many other O-O languages make a distinction between *primitives* and *classes*
  - For reasons of efficiency - primitives are smaller and faster
  - Primitives: basic data types built into the language
    - » *e.g.*, int, long, float, double, boolean, int[], float[] ([ ] = array)
  - Classes: data types composed of primitives/classes & with associated methods, all defined by the user (or in a library)
    - » *e.g.*, String, Math, PilotClass
- Java/C# also have classes that *wrap* a primitive
  - *e.g.*, Integer wraps int, Double wraps double
  - Useful for passing primitives where objects are required

Data Structures and ADTs

- Data structures and abstract data types (ADTs) are concepts that predate object-orientation
- A data structure is a particular way of organising the storage of data in a computer so as to make it efficient and easy to access/add/remove that data
  - *e.g.*, arrays, binary trees, linked lists
- An ADT is a data structure that only defines the operations for manipulating its data, but not how that structure is implemented
  - *e.g.*, stacks, queues
  - Stack can be done with an array OR with a linked list

Objects vs Data Structures/ADTs

- Objects have some similarities with these concepts
  - An object's fields can be considered to be a data structure
  - Objects can be used to implement data structures
- But objects ≠ data structures
  - Data structures can be implemented in any programming paradigm, not just O-O languages
  - Objects are more about the application's specific needs, data structures are more about general-purpose structures that could be useful in any application

Arrays

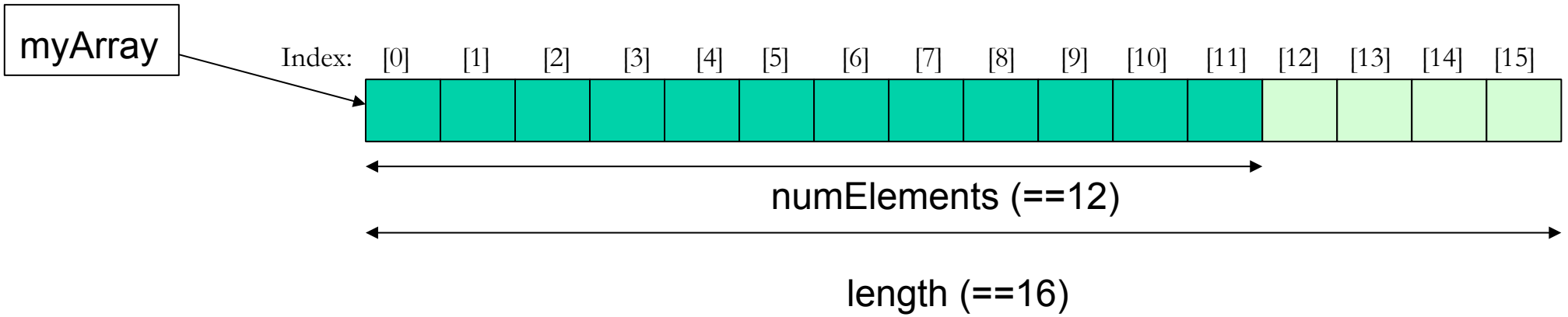
- We often work with *sets* of similar data
  - *e.g.*, how to handle the list of student marks in DSA?
  - double student1Mark, student2Mark, student3Mark, …?
- Arrays are one solution to this problem
- Simplest kind of data structure for storing data sets
  - Arrays are built-in to *all* programming languages
  - Instead of just one element, an array is a variable that contains *many* elements
  - The array variable itself is actually a reference to the first element of the array

Arrays - Properties

- Elements are located sequentially in memory
  - *i.e.*, the array is a *contiguous* block of memory
- All elements must have same data type (*e.g.*, double)
- Arrays can be initialised to any size (within memory limits)
- However, once initialised they cannot be resized
  - Must create a new array and copy over the contents of the old array in order to 'resize' an array

Arrays – Accessing Elements

- Once you have allocated an array, you need to be able to work with the elements inside the array
- Elements are accessed via an *index* (or *subscript*)
  - Java and C index from 0 to N-1 ('zero-based')
    - » In this case, the index is an *offset* from the first element



Arrays In Code (Java)

- Declaring: put '`[]`' on the end of the data type
  - » *e.g.*, `double[] myArray;`
  - » Any data type can be used with arrays, including classes
- Allocating: use new keyword with special `[]` syntax
  - » *e.g.*, `myArray = new double[16];`
- Indexing: myArray[index], index must be an int
  - » Negative indexes or indexes that are past the end of the array (*i.e.*, >= length) will cause an error during runtime.
  - » sample code to output contents of myArray

```
for(int ii = 0; ii < numElements; ii++)
{    //could use myArray.length
   System.out.println(myArray[ii]);
}
```

- We will cover arrays in detail in lecture 4

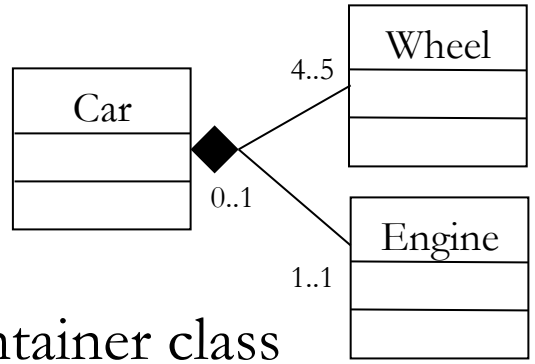
Goals of Object-Orientation

- Encapsulation / Information Hiding
  - Objects control access to their private fields and methods
  - Takes the emphasis off data (fields) and moves it onto publicly-available behaviour (methods)
- Abstraction / Polymorphism
  - Objects can be treated as if they were a simpler base class
    - » Inheritance means that any derived object can act as if it is an object of a base class (and the latter is always a simpler class)
  - To maximise this benefit, the specifics of *how* a method works should be hidden from the user
    - » *e.g.*, Car and Tank are both Vehicles. A Driver should only need to know Vehicle.accelerate() and not be concerned with how it works

Goals of Object-Orientation

- Reuse / Extensibility
  - Reuse: each class provides its functionality to other classes
  - Can inherit from a class to reuse/extend its functionality
- Modularization - low coupling, high cohesion
  - Objects should be responsible for their own data state
  - Objects should represent a single concept and all methods should relate to that concept (high cohesion)
  - Only the object's interface should matter to a user of that object, not the details of its implementation (low coupling)

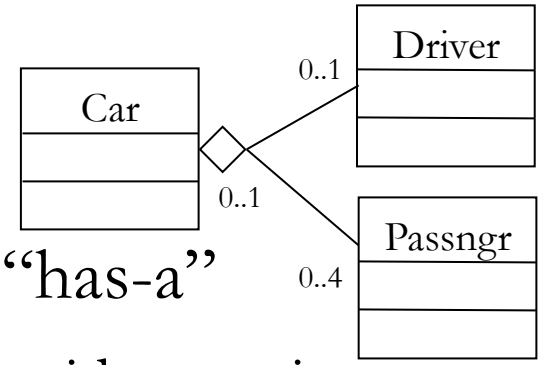
Class Relationships



– Composition

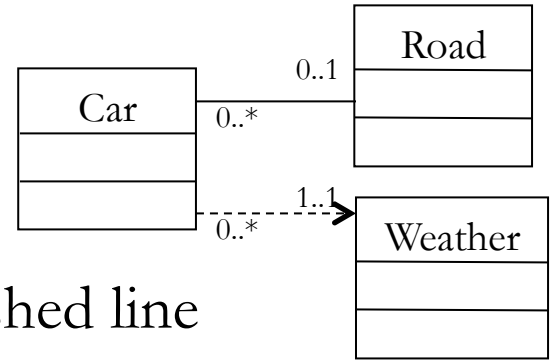
- "has-a" or "whole-part" relationship
  - » UML: Shown with solid diamond beside container class
  - » *e.g.*, Car "has-a" Wheel
- Strong lifecycle dependency between classes
  - » Car is not a car without four Wheels and an Engine
  - » When Car is destroyed, so are the Wheels and Engine
- In code:
  - » Car would have Wheel and Engine as private class fields

Class Relationships (2)



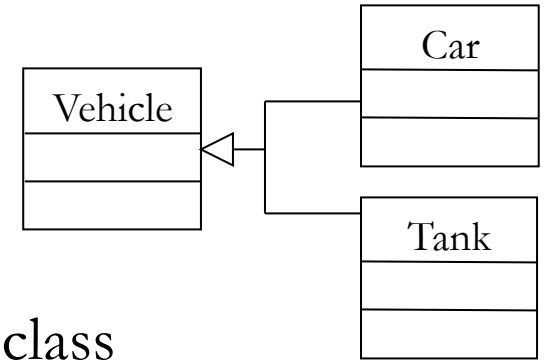
- Aggregation
  - Weaker form of composition, but is still "has-a"
    - » UML: Shown with open/unfilled diamond beside container
  - Lifecycle dependency usually not strong
    - » Car does not always have a driver
    - » When Car is destroyed driver and passengers are not
    - » Drivers can drive different cars
  - In code:
    - » Car would have Driver and Passenger as private class fields
    - » …exactly like composition!

Class Relationships (3)



- Association and Dependency
  - Indicates interaction between classes
    - » Association = solid line, Dependency = dashed line
    - » Difference between the two is murky: UML is a *guide*, not a *law*
  - Used to show that one class invokes methods on another
    - » … but that there is no other relationship beyond this
    - » With arrow, implies *unidirectional* (Car calls Weather, not vice-versa)
    - » No arrow implies *bidirectional* (Car and Road call each other)
  - In code: Any way that a method call can be set up and made
    - » *e.g.*, Weather object is passed as a parameter to a Car method
      - – *e.g.*, Car.setAggressiveness(Weather currentConditions)
    - » *e.g.*, Road has a class field of all Cars on that Road (aggregation?)

Class Relationships (4)



– Inheritance
  - "is-a" relationship
    - » Indicates one class is a sub-type of another class
    - » Shown with an open triangle arrowhead beside super-type
  - Implies the specialisation of the super-type
    - » Super-type synonyms: 'parent', 'base'
    - » Sub-type synonyms: 'child', 'derived'
  - In code: During class declaration; syntax is language-specific
    - » Java: public class Car **extends** Vehicle
    - » C++/C#: public class Car **:** Vehicle

Inheritance

- Inheritance allows us to specialise another class
  - *e.g.*, Tank "is-a" Vehicle, Car "is-a" Vehicle
  - Both inherit from Vehicle, but specialise it differently
- A Tank object can then be passed anywhere a Vehicle is required, since it *is* a type of Vehicle
  - This is called polymorphism - the ability for an object of one class to also act as an object of *any* of its base classes
- In Java, *all* classes ultimately inherit from Object
    - » if you don't explicitly inherit from anything, you inherit from Object
  - *i.e.*, Object is the 'most-base class' - singly-rooted hierarchy
  - A class marked as final cannot be extended
    - » *e.g.*, String, Math

Polymorphism and Method Overriding

- An important aspect of inheritance for polymorphism is the ability to override methods of the base class
  - Consider passing a Tank to a method void drive(Vehicle veh)
  - A call to veh.accelerate() will actually call Tank's accelerate()
    - » Which will behave differently to Car's accelerate()
- What is happening here?
  - Tank somehow becomes Vehicle. How?
  - What if you wanted to get back to Tank from Vehicle?
    - » Since it really is a Tank, surely you can do it

Overloading vs Overriding

- Overloading is when many methods share the same name but differ in their parameters
  - Constructors are a good example: default, alternate and copy constructor all have the same name, different parameters
    - » Uniqueness is defined by name + parameter *types*
      - – This is called the method's *signature*, or *prototype*
    - » *e.g.*, Car(String model) and Car(int numSeats) are different
    - » But: Car(String model) and Car(String ownerName) cannot be disambiguated - will cause compiler error
    - » Note that return type is *not* part of the method signature
  - Most modern languages support overloading
    - » C and Fortran are a couple that don't support overloading

Overloading vs Overriding

- Overriding is where a method has exactly the same signature as a method in a *super/parent/base* class
  - *i.e.*, the child class is overriding the behaviour of the parent
  - Only applies to object-oriented languages, and all O-O languages support it
    - » Overriding = specialisation, one of the cornerstones of O-O
- A method can be an overload *and* an override
  - Overloads the name of another method *in the current class*
  - Overrides the signature of a method *in the parent class*

this, super keywords

– Keyword 'this' is a reference to the *current object*

e.g.,
```
public Tank clone() {
    return new Tank(this); // Use copy constructor to make copy of ourselves
 }
```

– Keyword 'super' is a 'reference' to the current object's *parent class*

  • Use it to force a call to the parent class's code

e.g.,
```
public Tank(Tank otherTank) {    // Copy constructor
    super(otherTank);  // Call parent's copy constructor code first
    // Now do our own copy constructor code... }
```

e.g.,
```
public void doSomething() {    // A method
    super.doSomething();          // Call parent's doSomething() code
    // Now do our own code... }
```

– super and this are relative to the current object/class

  • this = current object

  • super = current class's direct parent class

Casting Between Types

- Changing from one type to another is called casting
  - You can also cast between numeric primitive types
    - » *e.g.*, ints to floats and vice-versa, but not int to String.
    - » C/C++ let's you cast *anything* - it's your problem if its wrong!

```
float fNum = 1.01;
int iNum = (int)fNum;          ← Cast by placing target data type in brackets
```

  - Java (and pretty much every language) will implicitly do casts for you when it knows that the cast is 'safe'
    - » Since Tank is-a Vehicle, casting Tank to Vehicle is safe

```
Tank t = new Tank();
Vehicle v1 = t;                ← Implicit cast
Vehicle v2 = (Vehicle)t;       ← Explicit cast, same result as implicit cast
```

    - » There's no need to explicitly do the casting here

Casting Between Types (2)

- So when do you have to cast? And why?
  - When you are casting between numeric types
    - » because loss of information can occur, *e.g.*, float 1.01 → int 1
  - When you are attempting to downcast to a derived class
    - » *e.g.*, casting Vehicle to Tank is not safe since the compiler cannot be sure that the object (of known type Vehicle) is a Tank or not
      - Tank is-a Vehicle does not mean Vehicle is-a Tank!
- If you know the cast is OK you can do it explicitly
  - *e.g.*, You know that the Vehicle really is a Tank
  - Compiler then leaves it to run-time to try the cast
    - » Fails at run-time with a ClassCastException if it's not a Tank

Casting Between Types (3)

```
Vehicle v = new Tank();          ← Implicit cast is happening here
Tank t1 = v;                     ← Compiler error
Tank t2 = (Tank)v;               ← OK, and will work at run-time too
Car c1 = (Car)v;                 ← Will compile, but fails at run-time
```

- Some notes on casting
  - Primitives:
    - » Casting from floats to ints will truncate the decimal places
    - » Casting from ints to floats may lose some numerical precision
  - Classes
    - » Object is a handy class to use for making general-purpose containers - simply contain an Object and you can contain *anything*
      - – You have to explicitly cast back to the right class later though

Checking Class Type

- Downcasting sounds a bit risky
  - What if you aren't totally sure of the object's true class?
    - » Downcasting could cause a ClassCastException
    - » Could catch this exception and try again, but that's ugly
  - Java provides you with a solution: instanceof keyword
    - » Let's you check if object A is really an instance of class X

```
Vehicle v = new Tank();
if (v instanceof Tank) {        ← Check if it really is a Tank
   Tank t1 = (Tank)v;
}
```

    - » **Warning:** try to limit your use of instanceof since it can be an indication of bad design and makes polymorphism redundant
      - – Plus, if you are certain that the cast is OK, instanceof is a waste

The 'Class' Class

- instanceof lets you check if an object is a given class
  - What about if you have two objects and you want to see if they are compatible, but don't particularly care about type?
  - Java has a type called 'Class' (!!), which describes every other type (classes, primitives, even void)
    - » Access it via the getClass( ) method that every Object has
    - » Then use Class.instanceof( ) to check instanceof

```
Object o1 = new Tank();
Object o2 = new Tank();
...
Class c = o1.getClass();        ← Gets the Class object for Tank, not Object
if (c.instanceof(o2)) {         ← True if o2 is a Tank or a sub-type of Tank
   System.out.println("Classes are compatible"); ← We don't actually know o1's type!
}
```

    - » As you can guess, this should be used even less than instanceof

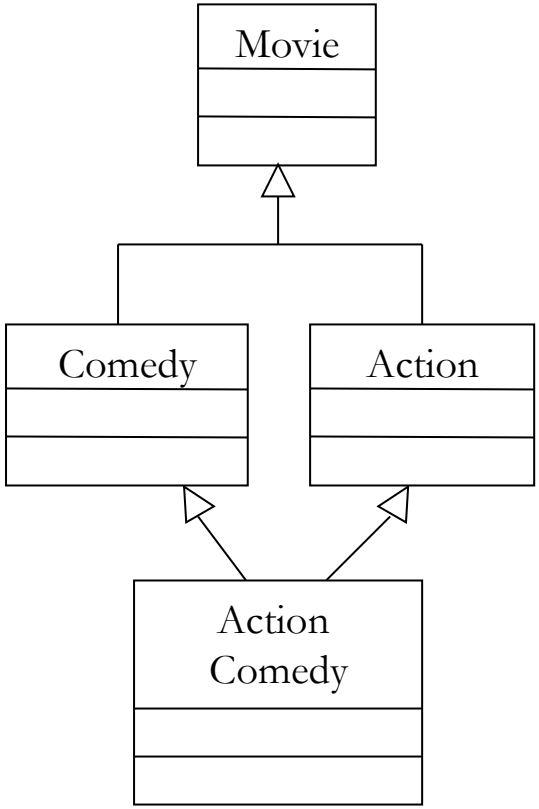
Multiple Inheritance

- So Tank "is-a" Vehicle
  - But Tank "is-an" Artillery as well, not just a Vehicle
    - » and Artillery is not always a Vehicle, so can't put Artillery in between Tank and Vehicle
  - *i.e.*, Tank really has more than one base class
- One solution: allow multiple inheritance (*e.g.*, C++)
  - Tank inherits from *both* Vehicle and Artillery
  - Tank is then polymorphic with both Vehicle and Artillery

Multiple Inheritance - Problems

- Theoretically, multiple inheritance is fine
- But in practice (in the code), things get messy
  - Say both Vehicle and Artillery define a method getSize()
    - » If Tank does not override getSize(), which getSize() version should the compiler call? Vehicle's? Artillery's?
    - » Worse, what if Artillery.getSize() refers to the size of the *shells* it fires, but Vehicle.getSize() refers to the *vehicle's* size?
  - In more complicated inheritance hierarchies, you can even inherit from the same class more than once!
    - » The next slide shows an example of this

Multiple Inheritance - Problem Example



Interfaces

- Interfaces are used as a solution to resolve (some of) the problems with multiple inheritance
  - An interface is essentially an abstract class where:
    - » All methods are abstract (ie: have no implementation)
    - » All methods are public
    - » No class fields exist
- In other words, an interface class only defines a set of public methods that its child classes must implement
  - Note that interfaces cannot have a constructor
    - » There's nothing to construct, so what would be the point?
  - Interfaces can inherit from (extend) other interfaces, but do not have to (unlike classes, which extend at least Object)

Interfaces and Multiple Inheritance

- Many multiple inheritance issues can then be resolved
  - Allow inheritance from as many interfaces as required
    - » Interface inheritance
  - BUT only allow inheritance from a single class, which includes abstract classes
    - » Implementation inheritance
- Why does this help?
  - Because interfaces cannot have any code
  - Thus there is never any confusion as to which base class's method should be invoked - there is only ever one base class with an implementation (all others are interfaces)

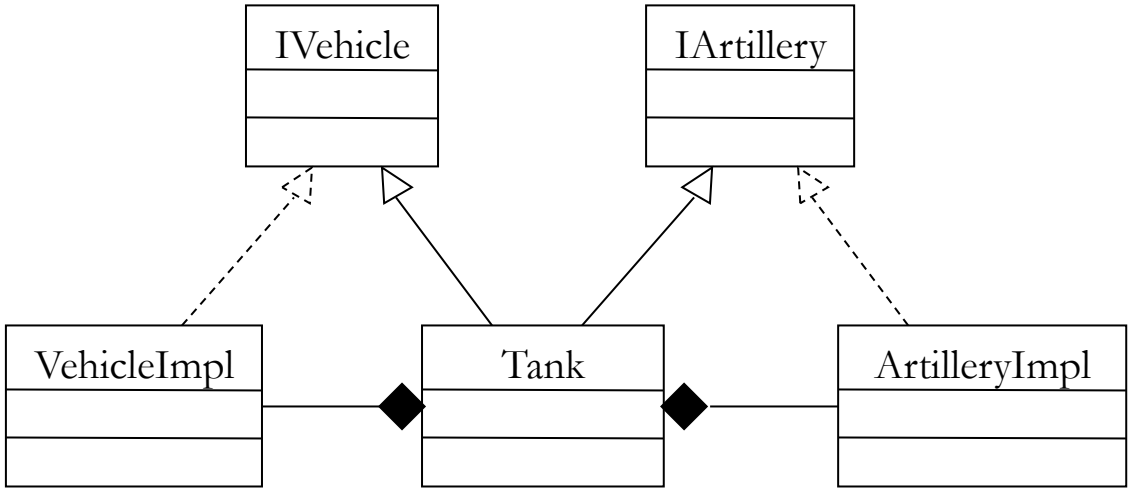
Interfaces and Multiple Inheritance

- Interfaces are not a magic cure-all
  - *e.g.*, If Vehicle and Artillery are both made into interfaces, but getSize( ) has different meanings for both:
    - » Tank still can't properly choose how to override getSize( )
    - » C# has the ability to define different methods, one per interface
  - *e.g.*, Action Comedy
    - » Action and Comedy aren't abstract, and so can't be interfaces
    - » Could make *all* movie genres into interfaces, and have separate implementation classes inheriting from these. Messy!
  - Limits code reuse potential
    - » Interfaces have no implementation (code) to reuse!

Emulating Multiple Impl Inheritance

- Ideally Tank would inherit from Vehicle and Artillery
  - …and both would have code that Tank can reuse
    - » *i.e.,* they are not interfaces, probably abstract classes instead
- The aforementioned issues with M.I. are in our way
  - But we can *emulate* M.I. with interfaces and composition
    - » Have Tank inherit from interface IVehicle
    - » Have Tank compose with (contain) a class VehicleImpl that implements all the would-be-non-abstract methods of IVehicle
      - – VehicleImpl might also inherit from IVehicle, but will have to bomb out on any truly-abstract method - a bit messy
    - » Have Tank 'delegate' calls to equivalent methods in VehicleImpl
      - – VehicleImpl code can then be shared (re-used) with other classes
    - » Then do the same with IArtillery

Emulating M.I. - Example



Class vs Abstr Class vs Interface Inheritance

- Inherit from classes…
  - …when you need to specialise behaviour of existing class
- Inherit from abstract classes…
  - …where a lot of the code in derived classes is common among most/all of the derived classes
  - The abstract class is then a 'repository' for shared code
- Use interfaces and composition+delegation…
  - …everywhere else
    - » It avoids wasting your precious single base class
    - » It also helps you get around integrating with or reusing existing classes - inherit from one, compose+delegate with others

Interfaces in Code

- Naming:
  - A prefixed capital 'I' is common for interfaces, eg: IVehicle
- In code:
  - Declaring: Almost identical to declaring a class
    - » Java: public **interface** IVehicle { … methods here … }
    - » C#: public **interface** IVehicle { … methods here … }
  - Inheriting from:
    - » Java: public class Tank **implements** IVehicle, IArtillery
    - » C#: public class Car **:** IVehicle, IArtillery
  - Can use extends and implements keywords together:
    - » public class Tank **extends** MilitaryObject **implements** IVehicle, IArtillery

Exceptions

- Error handling is a necessary task, but how can you do it elegantly?
  - Errors aren't 'normal' - you don't make a system that *expects* errors! But you must handle error situations
  - One solution: return an error code. Issues:
    - » Wastes the method's return value on errors
      - – Cumbersome; destroys similarity to maths equations:
        x = Add(y,z) becomes errCode = Add(y, z, x) (?what's the export?)
      - – And Java doesn't let you return exports in the parameter list!
    - » What error codes to use? Integers? Unix C does this with errno
      - – Problem: What does each number mean?

Exceptions (2)

- O-O languages solve error handling with exceptions
  - An independent ‘return path’ designed specifically for notifying the caller of an exceptional situation (=error)
  - On an error, a method ‘throws’ an exception
    - » Similar to a method returning a value, but a different pathway
  - The calling method can ‘catch’ the exception
    - » Conceptually similar to assigning the method’s return value to a variable when no error, but implemented differently
    - » If caller doesn’t catch it, the exception is thrown to the next-higher caller
      - see Checked vs Unchecked Exceptions
    - » If no-one catches it, the exception causes the program to crash

Exceptions (3)

- Java only lets objects of type Exception to be thrown
  - » (actually, Exception inherits from interface Throwable, but Exception is the typical starting point for most programmers)
  - Java has a range of classes inheriting from Exception
    - » eg: IllegalArgumentException, ArrayIndexOutOfBoundsException
  - You can define your own exception class, as long as it inherits from Exception (or one of it's subclasses)
    - » C# is similar. C++ let's you throw pretty much anything (!!)
- Each exception should require a message (a String)
  - Why throw an error if you don't say what the problem is?
  - also use getMessage() or getLocalizedMessage()
    - » may or may not be detailed enough

Checked vs Unchecked Exceptions

- Unchecked Exceptions
  - code will compile if not caught anywhere
  - RuntimeException, Error, and their subclasses
- Checked Exceptions
  - code will NOT compile if not caught somewhere
  - all other Exceptions
  - can use throws in method header to catch in calling method
    - » see multiple catch example coming up

Writing your own exceptions

- inherit from Exception
  - this will make it a checked Exception

```
public class MyException extends Exception
{
    private String error;

    public MyException(String message)
    {
        super(message);
        this.error="Something bad Happened";
    }

    public MyException(String message, String error)
    {
        super(message);
        this.error=error;
    }

    public String getError(){
        return this.error;
    }
}
```

Throwing Exceptions

- Java uses the throw keyword to throw exceptions
  - So do most other O-O languages

```
public double Divide(double numer, double denom) {
   if (denom == 0) {
      throw new IllegalArgumentException("denom must be non-zero");
   }

   return numer / denom;
}
```

- Note that we throw a *new* IllegalArgumentException
  - ie: We create an object and then throw it.
  - Could also have done the following (same result):

```
IllegalArgumentException e = new IllegalArgumentException("denom must be non-zero");
throw e;
```

Catching Exceptions

- Exceptions from different methods are often all caught at the one place in the calling method
  - Convenient: all error handling happens in one place
- Most languages use try .. catch .. [finally] blocks:
  - try: define the set of statements whose exceptions will all be handled by the catch block associated with this try
  - catch: processing to do if an exception is thrown in the try
  - finally: processing to always do regardless of whether an exception occurs or not.
    - » Good for clean-up, *e.g.*, closing open files
    - » This block is optional and executes after the try and catch blocks

Catching Exceptions - Example

```
public static void Main() {
      double fFirst, fSecond, fSum;
      try {                                  ← Try the following few statements
        fFirst = Divide(10,1);
        fSecond = Divide(2,0);               ← Will cause an exception to be thrown
        fSum = fFirst + fSecond;
        System.out.println("Sum of the divides is: " + fSum);
      }
      catch (Exception e) {                  ← Catch any exception that occurs
         System.out.println("Couldn't perform divides. Reason: " + e.getMessage());
      }
      finally {
         fSum = 0;                           ← Pointless clean-up - this is just an example!
      }
 }
```

– It's possible to *only* catch IllegalArgumentExceptions:

```
      catch (IllegalArgumentException e) {
         System.out.println("Couldn't perform divides. Reason: " + e.getMessage());
      }
```

- … but then other exceptions won't be caught, and if they occur they will cause the program to crash

Multiple catch clauses

- previous example caught all exceptions
- what if we want to handle various exceptions in a different way?
  - *e.g.*, if a file doesn't exist, ask the user for a different filename, but if anything else goes wrong with the file, terminate the program.
- every try clause must be followed by one or more catch clauses
  - order is important!

multiple catch example

```
private void readFileExample() throws IOException  {          ← Required as Checked Exception, must be caught in caller
   String inFilename;
   //file code omitted – see lecture 2
   boolean noFile;
   do {
      noFile = true;                                          ← file doesn't exist yet,
      inFileName = ConsoleInput.readLine("Please enter the filename");
      try {
       //file code omitted – see lecture 2                   ← Open file – if it doesn't exist, FileNotFoundException
          lineNum = 0;
        //file code omitted                                   ← Read the first line
          while (line != null){                               ← While not end-of-file, process and read lines
             lineNum++;                                        ← IOException may be thrown
             processLine(line);                               ← Some other exception may be thrown here
          }
         noFile = false;                                      ← if we get here, everything worked fine
      }//end try
      catch (FileNotFoundException e){                        ← can catch sub of IOExceptions
         System.out.println("Couldn't find your file " + e.getMessage() + "try again!");
      }                                                       ← will continue loop
      catch (IOException e){                                  ← MUST catch checked IOExceptions
         throw e;
      }                                                       ← re throw IOException to exit method
      catch (Exception e){                                    ← can catch all other Exceptions
         throw new Exception("I am a bad programmer: " , e);
                                                              ← re throw Exception to exit method
      }
   } while(noFile);                                           ← If the file doesn't exist, repeat the loop
}
```

Exceptions vs Error Codes

- Advantages over error codes:
  - ☑ Separate return path for separate issues
  - ☑ Caller can deal with all exceptions in batch & in one place
    - » Makes the code a lot easier to follow
  - ☑ Exceptions must be handled or the program will crash
    - » Stops errors from accidentally going unnoticed
  - ☑ Lets a constructor raise an error just like any other method
    - » Constructors don't have a return value!
- Disadvantages:
  - ☒ Need language support for throwing/catching exceptions

Exceptions in Real Life

- Exceptions aren't just merely 'error checking'
  - They are also incredibly useful as assertion checks, ensuring that the program is always in a valid state
  - Why bother? Because if you screw up somewhere in the code your screw-up will quickly result in an exception
    - » … alerting you to the existence of your screw-up
  - So PUT EXCEPTIONS EVERYWHERE
    - » The closer the exception is to the bug, the easier it is to find
    - » EXCEPTIONS WILL SAVE YOU OVER & OVER

Putting It All Together

< IVehicle.java: >

```
public interface IVehicle {
   public void accelerate();
   public void brake();
}
```

< IArtillery.java: >

```
public interface IArtillery {
   public void fire();
   public void reload();
}
```

Putting It All Together

< Tank.java: >

```
public class Tank implements IVehicle, IArtillery {
   private static final int MAX_SPEED = 20;          // Constant to limit speed

   private double m_fCurrentSpeed;
   private boolean m_bLoaded;

   public Tank() {
      m_fCurrentSpeed = 0;
      m_bLoaded = true;
   }

   public Tank(double initialSpeed) {
      if (initialSpeed < 0)
         throw new IllegalArgumentException("initialSpeed must be non-negative");

      m_fCurrentSpeed = initialSpeed;
      m_bLoaded = true;
   }
```

Putting It All Together

```
< Tank.java continued: >
   public void accelerate() {                ← From IVehicle
      changeSpeed(+2);
   }

   public void brake() {                     ← From IVehicle
      changeSpeed(-2);
   }

   public void fire() {                      ← From IArtillery
      if (m_bLoaded)
         m_bLoaded = false;
      else
         throw new IllegalStateException("Cannot fire when not loaded");
   }

   public void reload() {                    ← From IArtillery
      if (!m_bLoaded)
         m_bLoaded = true;
      // else we're already loaded - not an error situation!
   }

   private void changeSpeed(double changeInSpeed) {   ← Tank's own private method to handle accelerate()/brake()
      double fNewSpeed = m_fCurrentSpeed + changeInSpeed;
      if (fNewSpeed > MAX_SPEED)
         fNewSpeed = MAX_SPEED;          // We can't accelerate past our max speed
      else if (fNewSpeed < 0)
         fNewSpeed = 0;                  // We can't brake less than a complete stop
   }
}
```

Language Support in Java

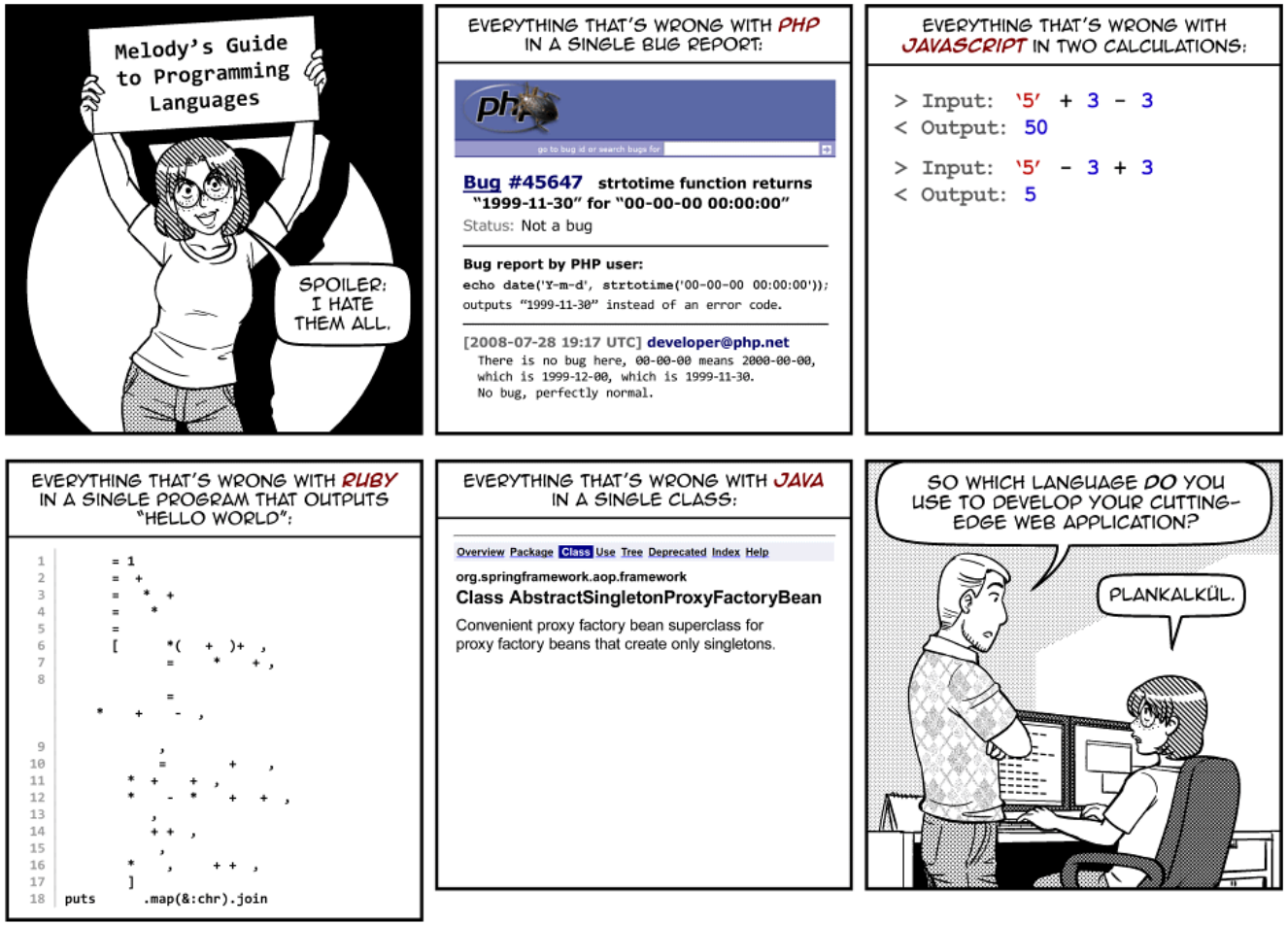
- Unicode
  - Characters are 2-bytes wide
  - Allows for international alphabets – Mandarin, Japanese, etc
  - Effects on space efficiency
  - Porting to C/C++ consequences
- Internationalisation
  - Support for multi languages for messages

A Final Note on Java

- Java has been around since the mid 1990s
  - In that time, it has gone through several versions, with many features and classes having been added
    - » Java 1.0, 1.1, 1.2
    - » Renamed Java2, versions 1.2, 1.3, 1.4, 1.5
    - » Renamed Java 5.0. Now up to Java 7.0
  - There are often multiple classes that perform a similar job
    - » *e.g.,* AWT (Java 1) vs Swing (Java 2) for GUI development
    - » Vector (Java 1) vs ArrayList (Java 2) for growable arrays
    - » Enumeration (Java 1) vs Iterator (Java 2)
  - Look for the "Since JDK X" in the Java help for versioning

End of Lecture

– Please be aware that labs start this week.



Sandra and Woo by Oliver Knörzer (writer) and Powree (artist) – www.sandraandwoo.com

Bonus

– The following is a Java Program:

\u0070\u0075\u0062\u006c\u0069\u0063\u0020\u0020\u0020\u0020
\u0063\u006c\u0061\u0073\u0073\u0020\u0055\u0067\u006c\u0079
\u007b\u0070\u0075\u0062\u006c\u0069\u0063\u0020\u0020\u0020
\u0020\u0020\u0020\u0020\u0073\u0074\u0061\u0074\u0069\u0063
\u0076\u006f\u0069\u0064\u0020\u006d\u0061\u0069\u006e\u0028
\u0053\u0074\u0072\u0069\u006e\u0067\u005b\u005d\u0020\u0020
\u0020\u0020\u0020\u0020\u0061\u0072\u0067\u0073\u0029\u007b
\u0053\u0079\u0073\u0074\u0065\u006d\u002e\u006f\u0075\u0074
\u002e\u0070\u0072\u0069\u006e\u0074\u006c\u006e\u0028\u0020
\u0022\u0048\u0065\u006c\u006c\u006f\u0020\u0077\u0022\u002b
\u0022\u006f\u0072\u006c\u0064\u0022\u0029\u003b\u007d\u007d

– Courtesy of http://www.javapuzzlers.com/