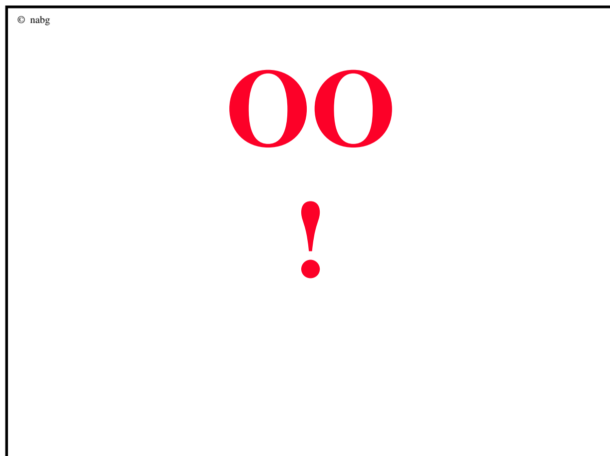


© nabg

- This stuff used to be reviewed briefly toward the end of CSCI192/CSCI124
- Inheritance and all its implications:
  - class hierarchies
  - virtual functions, dynamic binding and polymorphism
  - use of hierarchies in design
  - object oriented programming
  - use of inheritance as a “type specification” mechanism
- New “first year light” version of curriculum is light on class and will hardly have had time to mention these topics.

© nabg

Object orientation

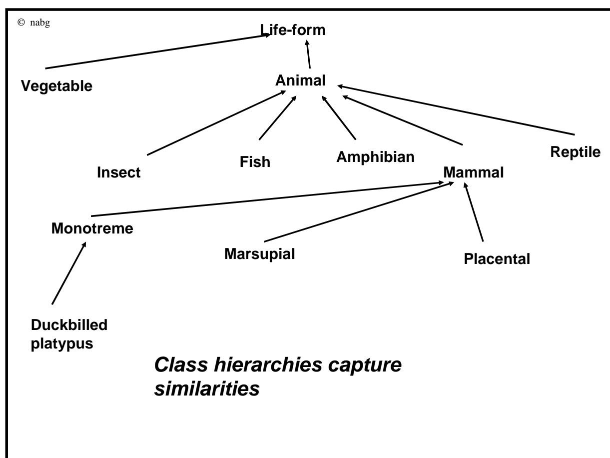
- Object oriented!
- Polymorphic!
- Dynamic binding.
- virtual methods
- inheritance.
- protected access.
- …
- *You must (may) have noticed these buzz words.*
- What do they all *mean*?

© nabg

class hierarchies

- Some languages (C++, Java, Smalltalk, Eiffel, …) allow you to define class hierarchies.
- Class hierarchy –
  - one class can be a more specialized version of another.

- So?
- Why?
- *Hierarchies are a means of describing similarities.*



© nabg

So?

- Similarity.
- You can exploit similarities:
  - clean up program design
  - save on coding
  - make it easier to allow for future extensions
  - capture complex behaviours

© nabg

Exploiting classes with inheritance

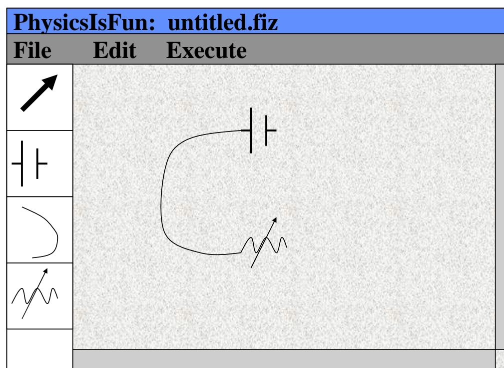
- Different ways of exploiting inheritance turn out to make a major difference in programming.
- Increasingly, program development processes are being adapted to take advantage of class hierarchies.

© nabg

Inheritance? Starting with a design problem

- Illustrate a difficulty with a program design.
- Show how application problem involves "similar objects".
- Simple design fails to exploit similarity – and ends up being rather clumsy.
- If could exploit similarities inherent in the application, then ...

© nabg



© nabg

"Physics is FUN"

- You are to write a "Macintosh/Windows" program for manipulating electrical circuits,
- Simple circuits made from:
  - wires,
  - switches,
  - batteries,
  - lamp-bulbs
  - resistors.
- A program to simulate such circuits would need
  - an editing component that allowed a circuit to be laid out graphically,
  - some part to do "Ohm's Law" and "Kirchoff's Law" calculations to calculate currents and "light up" the simulated bulbs.

© nabg

Physics is FUN ?

- Interface of program
  - a "palette of tools"
    - text (paragraphs describing the circuit),
    - circuit elements like the batteries and light bulbs.
  - The editor part would allow the user to
    - select a component,
    - move it onto the main work area
    - open a dialog window that would allow editing of text and setting parameters such as a resistance in ohms.
  - Obviously, the program would let the user save a partially designed circuit to a file from where it could be restored later.

© nabg

Obvious objects

- A "document" object
  - own all the data,
  - keep track of the components added
  - organize transfers to and from disk.
- Various collections, "lists"
  - These lists would be owned by the "document".
    - list of "text paragraphs" (text describing the circuit),
    - a "list of wires",
    - a "list of resistors"
- A "palette object".
- A "window" or "view" object used when displaying the circuit.
- …

© nabg

Obvious objects

- …
- Some "dialog" objects" used for input of parameters.
- Lots of "wire" objects.
- Several "resistor objects".
- A few "switch" objects".
- A few "lamp bulb" objects".
- At least one battery object.

© nabg

- class TextParagraph
  - Owns:
    - a block of text and a rectangle defining position in main view (window).
  - Does:
    - GetText() – uses a standard text editing dialog to get text changed;
    - FollowMouse() – responds to middle mouse button by following mouse to reposition within view;
    - DisplayText() - draws itself in view;
    - Rect() – returns bounding rectangle;
    - …
    - Save() and Restore() - transfers text and position details to/.from file.

© nabg

- class Battery
  - Owns:
    - Position in view, resistance (internal resistance), electromotive force, possibly a text string for some label/name, unique identifier, identifiers of connecting wires…
  - Does:
    - GetVoltStuff() – uses a dialog to get voltage, internal resistance etc.
    - TrackMouse() – respond to middle mouse button by following mouse to reposition within view;
    - DrawBat() - draws itself in view;
    - AddWire() – add a connecting wire;
    - Area() – returns rectangle occupied by battery in display view;
    - …
    - Put() and Get() – transfers parameters to/from file.

© nabg

- class Resistor
  - Owns:
    - Position in view, resistance, possibly a text string for some label/name, unique identifier, identifiers of connecting wires…
  - Does:
    - GetResistance() – uses a dialog to get resistance, label etc.
    - Move() – respond to middle mouse button by following mouse to reposition within view;
    - Display() - draws itself in view;
    - Place() – returns area when resistor gets drawn;
    - …
    - ReadFrom() and WriteTo() – transfers parameters to/from file.

© nabg

Pseudo code outlines

- As well as roughly characterising classes, your design process would involve sketching functions, and mapping object interactions, involved in handling major program events.

© nabg

```
Document::DoSave
    write paragraphList.Length()
    iterator i1(paragraphList)
    for i1.First(), !i1.IsDone() do
        paragraph_ptr = i1.CurrentItem();
        paragraph_ptr->Save()
        i1.Next();

    write BatteriesList.Length()
    iterator i2(BatteriesList)
    for i2.First, !i2.IsDone() do
        battery_ptr = i2.CurrentItem()
        battery_ptr->Put()

    …
```

© nabg

Document::Draw

```
iterator i1(paragraphList)
for i1.First(), !i1.IsDone() do
        paragraph_ptr = i1.CurrentItem();
        paragraph_ptr->DisplayText()
        i1.Next();
```

Draw paragraphs

```
iterator i2(BatteriesList)
for i2.First, !i2.IsDone() do
        battery_ptr = i2.CurrentItem()
        battery_ptr->DrawBat()
```

Draw batteries

…

© nabg

Document::LetUserMoveSomething(Point mousePoint)

```
iterator i1(paragraphList)
Paragraph *pp = NULL;
for i1.First(), !i1.IsDone() do
        paragraph_ptr = i1.CurrentItem();
        Rectangle r = paragraph_ptr->Rect()
        if(r.Contains(mousePoint) pp = paragraph_ptr;
        i1.Next();
if(pp != NULL)
        pp->FollowMouse()
        return
```

Did user pick a paragraph?

```
iterator i2(BatteriesList)
battery *pb
for i2.First, !i2.IsDone() do
        battery_ptr = i2.CurrentItem()
        Rectangle r = battery_ptr ->Area()
        if(r.Contains(mousePoint) pb = battery_ptr ;
        i2.Next();
if(pb != NULL)
        pb->TrackMouse()
        return
```

Did user pick a battery?

Etc. etc

© nabg

Clumsy?

- You should have the feeling that there is something amiss.
- The design with its "batteries", "wires", "text paragraphs" seems sensible.
- But the code is coming out curiously clumsy and unattractive in its inconsistencies.

each different component handled separately
inconsistencies, “put” / “WriteTo” / “Save”

© nabg

Aren’t they all similar?

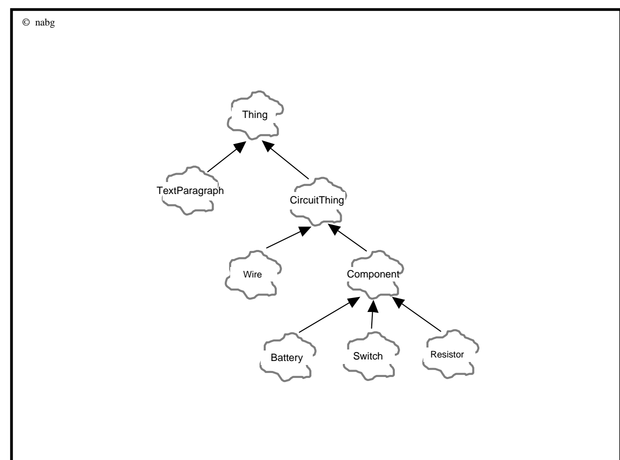
- Batteries, switches, wires, and text paragraphs may be wildly different kinds of things, but from the perspective of "document" they actually have some similarities.
- They are all "things" that perform similar tasks. A document can ask a "thing" to:
  - Save yourself to disk;
  - Display your editing dialog;
  - Draw yourself;
  - Track the mouse as it moves and reposition yourself;
  - …

© nabg

Some things are more similar than others

- Batteries, switches, and resistors will all have specific roles to play in the circuit simulation, and there will be many similarities in their roles.
- Wires are also considered in the circuit simulation, but their role is quite different, they just connect active components.
- Text paragraphs don't get involved in the circuit simulation part.
- All of them are "storable, drawable, editable" things, some are "circuit things", and some are "circuit things that have resistances".

© nabg



© nabg

class Thing

- Class Thing captures just the concept of some kind of data element that can
  - draw itself,
  - save itself to file
  - ….
- There are no data elements defined for Thing, it is purely conceptual, purely abstract.

Pure abstract class

© nabg

TextParagraph

- A TextParagraph *is a* particular kind of Thing.
- A TextParagraph owns
  - its text,
  - its position
  - ….
- You can also define actual code specifying exactly how a TextParagraph might carry out specific tasks like saving itself to file.
- Whereas class Thing is purely conceptual, a TextParagraph is something pretty real, pretty "concrete".
- You can "see" a TextParagraph as an actual data structure in a running program.

Concrete class

© nabg

class CircuitThing

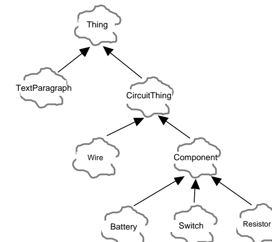
- ACircuitThing is somewhat abstract.
- You can define some properties of a CircuitThing.
- All circuit elements seem to need unique identifiers, they need coordinate data defining their position, and they need a character string for a name or a label.
- You can even define some of the code associated with CircuitThings – for instance, you could define functions that access coordinate data.

Partially abstract class

© nabg

others

- Concrete class Wire
- Partially Abstract class CircuitComponent
- Concrete classes Battery, Resistor, …



© nabg

class hierarchy

- OK, such a hierarchy provides a nice conceptual structure when talking about a program but how does it really help?
- Consistency
- Design simplification

© nabg

class hierarchy consistency

- One thing that you immediately gain is consistency.
- In the original design sketch, text paragraphs, batteries and so forth all had some way to display themselves, save themselves to file and so forth.
- But each class was slightly different: TextParagraph::Save(), Battery::Put() ….
- The hierarchy allows us to capture the concept of "storability" by specifying in class Thing the ability WriteTo().
- Each specialization performs WriteTo() in a unique way, but they can at least be consistent in their names for this common behaviour.
- But consistency of naming is just a beginning.

© nabg

Design simplification

- Suppose Document owned a list of “things” rather than lots of separate lists:

```
Document::DoSave(…)
    write thingList.Length()
    iterator i1(thingList)
    for i1.First(), !i1.IsDone() do
        thing_ptr = i1.CurrentItem();
        thing_ptr->WriteTo()
        i1.Next();
```

Save all different kinds of “thing”

```
Document::Draw
    iterator i1(thingList)
    for i1.First(), !i1.IsDone() do
        thing_ptr = i1.CurrentItem();
        thing_ptr->Draw()
```

Draw all different kinds of “thing”

© nabg

Advantages of simplified design

- Code is no longer obscured by all the different special cases, shorter and much more intelligible.
- Revised Document no longer needs to know about the different kinds of circuit component. Useful later if you decided to have another component (e.g. class Voltmeter); you wouldn't need to change the code of Document in order to accommodate this extension.

© nabg

Code sharing benefits

- Sometimes, you can define the code for a particular behaviour at the level of a partially abstract class.
  - e.g. you should be able to define the access function for getting a CircuitThing's identifier at the level of class CircuitThing while class Component can define the code for accessing a Component's electrical resistance.
- Defining these functions at the level of the partially abstract classes saves you from writing very similar functions for each of the concrete classes (like Battery, Resistor, etc.)

© nabg

C++ and Java class hierarchies

- Both have a way of specifying ideas like
  - "class Thing represents the abstract concept of a storable, drawable, moveable data element",
  - "class TextParagraph is a kind of Thing that looks after text and …".
- You start by defining the "base class", in this case that is class Thing which is the base class for the entire hierarchy.

© nabg

C++

Base class for a hierarchy

```
class Thing {
public:
    virtual ~Thing() { }
    /* Disk I/O */
    virtual void    ReadFrom(istream& i s) = 0;
    virtual void    WriteTo(ostream& os) const = 0;
    /* Graphics */
    virtual void    Draw() const = 0;
    /* mouse interactions */
    virtual void    DoDialog() = 0;     // For double click
    virtual void    TrackMouse() = 0; // Mouse select and drag
    virtual Rect    Area() const = 0;
    …
};
```

© nabg

Java

Base class for a hierarchy

```
interface Thing {
    /* Disk I/O */
    public abstract void    readFrom(BufferedReader in);
    ...
    /* Graphics */
    public abstract void    draw();
    /* mouse interactions */
    public abstract void    doDialog();     // For double click
    public abstract void    trackMouse();   // Mouse select and drag
    public abstract Rectangle    area();
    …
}
```

Judicious use of a few extra key words does help.
‘abstract’ --- don’t use with interface, all functions implicitly declared ‘public abstract’

© nabg

class Thing

- Class Thing represents just an idea of a storable, drawable data element and so naturally it is simply a list of function names.
- We know that all Things can draw themselves, but we can't say how.
- The ability to draw is common, but the mechanism depends on the specialized nature of the Thing that is asked to draw itself.
- In class Thing, we have to be able to say "*all Things respond to a Draw() request, specialized Thing subclasses define how they do this*".

© nabg

deferred functions

- We have to be able to say "*all Things respond to a Draw() request, specialized Thing subclasses define how they do this*".
- In Java, use keyword abstract.
- In C++
  - This is what the keyword virtual and the odd = 0 notation are for.
  - The keyword virtual identifies a function that a class wants to define in such a way that subclasses may later extend or otherwise modify the definition.
  - The =0 part means that we aren't prepared to offer even a default implementation. (Such undefined virtual functions are called "pure virtual functions".)

"Deferred" terminology is Eiffel's.

© nabg

class Thing

- The C++ and Java compilers knows that class Thing represents an abstraction.
- C++ compiler
  - will reject definitions of Thing variables:

    Thing aThing; // get compiler error msg.

  - You can define Thing* pointer variables – these are pointers that will point to an instance of some specialized subclass of class Thing.
- Java compiler
  - allows definitions of Thing object reference variables
  - Will not allow you to write something like Thing thing1 = new Thing();

© nabg

implements
extends

Java has two keywords;
implements goes with interface
extends when adding more functions

Subclasses

- Once you have declared class Thing, you can declare classes that are "based on" or "derived from" this class:

```
class TextParagraph implements Thing {
  TextParagraph(Point topleft)
        …
}
class CircuitThing implements Thing {
…
}
```

Java

```
class TextParagraph : public Thing {
  TextParagraph(Point topleft);
        virtual ~TextParagraph();
        …
};
class CircuitThing : public Thing {
…
};
```

C++

© nabg

```
class TextParagraph : public Thing {
public:
    TextParagraph(Point topleft);
    virtual ~TextParagraph();
    /* Disk I/O */
    virtual void      ReadFrom(istream& is);
    ...
    /* Graphics and mouse interactions */
    virtual void      Draw() const;
    ...
    virtual Rect      Area() const;
    // Member functions that are unique to TextParagraphs
    void  EditText();
    …
private:
    // Data needed by a TextParagraph
    Point fTopLeft;
    char   *fText;
    …
};
```

C++

© nabg

```
class TextParagraph implements Thing {
    public TextParagraph(Point topleft) { ... }
    /* Disk I/O */
    public void       readFrom(BufferedReader in) { ... }
    ...
    /* Graphics and mouse interactions */
    public void       draw()  { ... }
    ...
    // Member functions that are unique to TextParagraphs
    public void       editText()  { ... }
    …

     private Point   topLeft;
     private char[ ]  text;
    …
}
```

Java

© nabg

```
class CircuitThing : public Thing {
    CircuitThing(int ident, Point where);
    virtual ~CircuitThing();
    /* Disk I/O */
    virtual void     ReadFrom(istream& is);
    virtual void     WriteTo(ostream& os) const;
    …
    // Additional member functions that define behaviours
    // characteristic of all kinds of CircuitThing
    int              GetId() const { return this->fId }
    virtual Rect     Area() const;
    virtual double Current() const = 0;
    …
protected:
    // Data needed by a CircuitThing
    int     fId;
    Point  flocation;
    char   *fLabel;
    …
};
```

C++

© nabg

```
abstract class CircuitThing implements Thing {
    public CircuitThing(int ident, Point where);
    /* Disk I/O */
    public void      readFrom(BufferedReader in);
    …
    // Additional member functions that define behaviours
    // characteristic of all kinds of CircuitThing
    final public  int          getId() { return id }
    public Rectangle           area() { ... }
    abstract public double current();
    …
    // Data needed by a CircuitThing
     protected int    id;
     protected Point location;
     protected char[] label;
    …
}
```

Java

'abstract' keyword only used in context of 'partially implemented abstract class'

© nabg

"public derivation"

- Public derivation acknowledges that both TextParagraph and CircuitThing are specialized kinds of Things.
- So code "using Things" will work with TextParagraphs or CircuitThings.
- This is exactly what we want for the example where the Document object has a list of "pointers to Things" and all its code is of the form thing_ptr->DoSomething().

© nabg

C++ funnies

- Private derivation
- Protected derivation

- Use of these C++ forms generally indicative of a design problem!

© nabg

Java

- Java extends is similar to C++ public derivation
- Java has only this form.

© nabg

TextParagraph concrete class

- Text paragraph declaration has to be complete, and all the member functions will have to be defined:
  - constructor(s) and destructor
  - repeat the declarations from class Thing; so we again get functions like Draw() being declared
  - introduce some additional member functions describing those behaviours that are unique to TextParagraphs.
    - Some of these additional functions will be in the public interface; most would be private.
  - Class TextParagraph would also declare all the private data members needed to record the data possessed by a TextParagraph object.

© nabg

CircuitThing
a partially abstract class

- Class CircuitThing is an in between case, its main role is to introduce those member functions needed to specify the behaviours of all different kinds of CircuitThing and to describe those data members that are possessed by all kinds of CircuitThing.
- Class CircuitThing cannot provide definitions for all of those pure virtual functions inherited from class Thing;
  - ignore those you can't define at this level
  - define default or partial implementations for others

© nabg

protected access

- Class CircuitThing declares some of the data members – id, label, and location.
- These data members should not be public; you don't want the data being accessed from anywhere in the program.
- But if the data members are declared as private, they really are private, they will only be accessible from the code of class CircuitThing itself.
- You can see that the various specialized subclasses are going to have legitimate reasons for wanting to use these variables.

© nabg

Java's "interface" & "abstract" classes

- Interface is the more abstract
  - no data members
  - all functions simply names and argument lists with no body
- Interface is to capture some general concept.
- "abstract class"
  - can have data members
  - can have some member functions defined
  - but inherently "incomplete", still an abstraction

© nabg

Using the classes

```
void Document::DoSave(ostream& out)
{
    out << thingList.Length() << endl;

    iterator i1(thingList);
    i1.First();
    while(!i1.IsDone()) {
        Thing* thing_ptr = (Thing*) i1.CurrentItem();
        thing_ptr ->WriteTo(out);
        i1.Next();
        }
    }
```

C++

© nabg

Using the classes

```
public class Document {
    ...
    void DoSave(BufferedWriter out) {
    ..
    for(Thing aThing : thingList) {
        athing.writeTo(out);
        }
    }
```

Java

© nabg

Hey thing, write yourself! Do it your way!

```
for(Thing aThing : thingList) {
    athing.writeTo(out);
    }
```

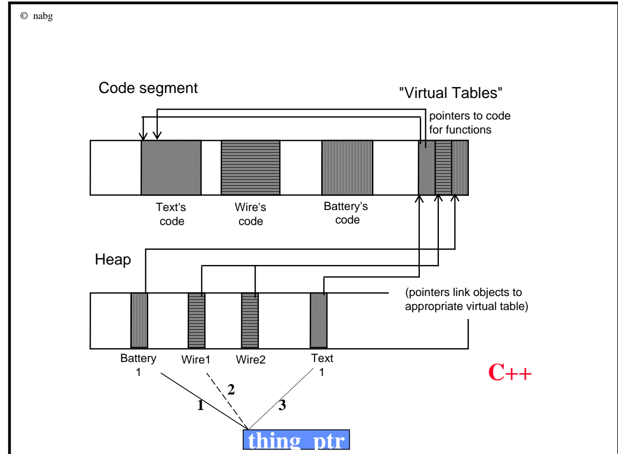
- Document has a list of Thing object reference variables (or, in C++, pointers).
- One pointer might reference a battery, next two might be wires, then a text paragraph, ...
- Code just says "Thing, write yourself".
- Need to execute Battery.writeTo(), then Wire.writeTo(), ...

© nabg

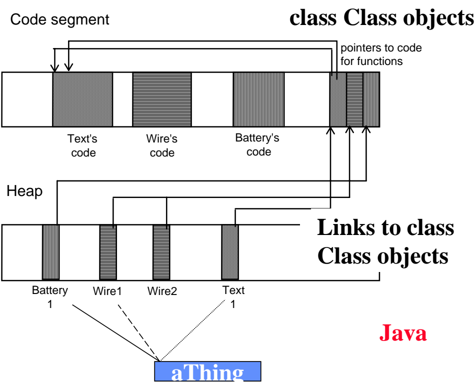
“Dynamic dispatch”

- Code is arranged so that the correct writeTo() function is sorted out at run-time.
- (Since it is done at run-time, it is “***dynamic***”.)
  - Code has pointer to object,
  - object will have been created as an instance of a specific class like Battery
  - when created, object “knew” what it was, and so “knew” that as it was a Battery it should use functions like Battery.writeTo
  - object should keep this information until needed
  - then can be asked to identify correct function

© nabg



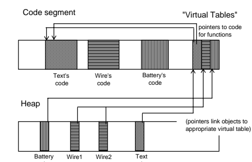
© nabg



© nabg

C++ virtual tables

- When compiler produces code for classes from a hierarchy, it also creates tables identifying the functions associated with the class.
  - could have a table with entries
    - 1 the WriteTo function
    - 2 the ReadFrom function
    - 3 the Draw function
    - …



© nabg

Virtual function pointer

- The compiler adds an extra data member to classes that use virtual functions.
- This extra (hidden) data member holds the address of the class’s virtual function table.
- When an instance of a class is created, this data member is automatically set.

This is standard C++ implementation (& compilers can vary)

© nabg

Java’s class Class

- Similar idea
- More elaborate, more secure, but slower implementation.
- For every class (data type) you use in a Java program you get an instance of class Class that is loaded with information describing your class.
- This class Class object is created by the class-loader that reads in the .class file.

© nabg

What is in class Class?

- Information about all the member functions, return types, arguments, location of byte code, etc etc
- Each method call handled by the JVM interpreter involves checking the data in class Class
  - C++ dynamic dispatch costs one extra memory cycle
  - Java's costs 100s of instructions
- But they achieve the same (with Java allowing lots of run-time security checks if you want them)

© nabg

Dynamic function dispatch C++

- The actual code generated for a call to a virtual function

```
thing_ptr ->WriteTo(out);
```

- is something like
  - use thing_ptr pointer to get to object
  - pick out address of table from hidden data member "virtual table address" (__vtbl)
  - WriteTo corresponds to entry 1, so index into the table of function addresses to get the address of the right function.
  - call that function

Polymorphic pointers

© nabg

Dynamic function dispatch Java

- The actual code generated for a call like

```
aThing.writeTo(out);
```

- is something like
  - use aThing object reference to get to object
  - pick out address of class Class object from a hidden data member
  - Check that signature of writeTo in class Class object matches use in call, get location of byte code for function
  - call that function

Polymorphic object reference variables

© nabg

Object-orientation

- Object-oriented programming:
  - use class hierarchies that capture similarities among objects
    - some are application specific (batteries, wires, ...)
    - some are generic:
      - class Application
      - class Window, subclass FrameWindow, ...
      - class Checkbox, class Menu, class Dialog
  - simplifies design and implementation of new components
  - allows "reuse of designs"
    - "*The XYZ application is based on the standard Windows framework design. The XYZ program opens a Framewindow and offers menu options File/New and File/Open. ...*"

© nabg