Object – Oriented Design with UML and Java

Part IV: Core Java

(with just enough C++ to compare and contrast with Java, and to understand the examples in the *Design Patterns* book)

Version 2011

OO Programming Languages

- Simula / Smalltalk / C++ / Java / Python / Lisp / Eiffel / C# / JADE / ...

This section introduces OO programming using **C++ & Java**.

- Java lives and breathes on the web. Primary and current sources of documentation are available on-line.

Simula

- First OO Programming Language, *1967*.
- Designed by Dahl, Myhrhaug, Nygaard at Norwegian Computing Center.
- Programming language for “event based simulation.”
- Also a general-purpose language.
- Bjarne Stroustrup started with Simula as motivation for C++.
- Extention/Modification of Algol 60.
- Class concepts and reference-variables (pointers to objects).

Smalltalk

- Developed at Xeroc PARC in late ’70s by Adelle Goldber, et al.
- Smalltalk 80 was the first publicly distributed version.
- Lots of interest from late 80s to mid 90s.
- Pure OO - everything is an object.
- First generally available integrated development system.
- Has a garbage collector and a virtual machine.

C++

- Developed at Bell Labs in mid 80s by Bjarne Stroustrup.
- Built on wide acceptance of the **C** programming language.
- The big, general purpose language of the 90s.
- Not completely object-oriented.
- Not a good language for novice programmers (gotchas!)
- Don’t let the Java hype fool you; there is a lot of C++ being written.
- Because it is based on **C**, it can work with an huge number of existing applications, utilities, devices, etc.
- Is suitable for system programming.
- Good, reliable tools.
- Mature.
- Fast.

Java

- Developed at Sun Microsystems in early 90s by James Gosling.
- Originally call *Oak* (object application kernel), and designed for use in embedded consumer electronic applications.
- Syntax looks a lot like C++.
- Good language for novice programmers (much simpler than C++).
- Semantics are more like Smalltalk.
- Slower than C++ due to the overhead of the JVM (virtual machine).
- Evolving quickly. Tool vendors are hard-pressed to keep up with new releases of the JDK (the Java Development Kit).
- The first language designed for Internet *applets*.
- Dynamic class loading (no link step required as in C++).
- 100% Thread and GUI portability can be challenging to achieve.

C#

- More similar to Java than to C++
  - VM, garbage collection, reflection, no multiple inheritance, Object, arrays checked, . . .
- Developed by Microsoft (as part of .NET) - for Windows.
- Compiles into .NET *instruction language* for the CLR (Common Language Runtime) a cross-language variant of Java bytecode.
- Many keywords from Java have been gratuitously changed:
  - super ... base
  - import … using
  - synchronized … lock
  - package … namespace
- New: stackAlloc(), delegates, memory pointers, LINQ, lambda expressions, …

Python

- Popular and useful… worth learning.
- Concise syntax is easy to learn and enhances productivity.
- No compile-time type checking.
- Interpreted “scripting” language.
- Jython is an implementation of Python for the Java Virtual Machine.

“Hello World” in Java

Goal: Get code running in an ***Integrated Development Environment***

with a ***Visual Debugger*** such as ***ECLIPSE***...

`http://www.softwarefederation.com/cs4448.html`

- Click on Beginning Java. Download Eclipse. Follow the instructions.

A few Java rules:

- For every file Foo.java, there must be a *public* class in the file whose name matches the name of the file (a public class named Foo).
- Java packages are hierarchical in the same way that directory structures are hierarchical; Java package structures must have parallel directory structure. If there is a bar package, there must be a bar directory.
- If your PATH and CLASSPATH ***environment variables*** are not set up right, your JDK (Java Development Kit) will not work properly.
- To compile a .java file by hand, use: "javac -g *.java" (the –g specifies debug). This will create .class files.

Java Packages

- Java ***`packages`*** help to organize code, provide additional name scoping, and additional access control. The package structure is designed to mimic the file system's hierarchical directory structure.
- C++ provides ***`namespaces`***, which provide similar name scoping benefits.

Suppose all of my Java development code resides under the **`J:\java`** directory, and my environment's ***`CLASSPATH`*** **`= D:\jdk1.1.8\lib\classes.zip;J:\java;`**

Suppose I have the **`J:\java`** subdirectories **`projectFoo`** and **`projectBar`**.

Further suppose that **`projectFoo`** has the subdirectories **`client`** and **`server`**.

Finally, suppose the **`client`** package has the classes **`X, Y,`** and, **`Z.`**

The fully specified name of class **`X`** is **`projectFoo.client.X`**

- It is not always necessary to provide the fully specified name; Java provides the ***`import`*** statement for this purpose (*not* the same as the C++ ***`#include`***).

```
import projectFoo.client.X;
X x = new X();  // projectFoo.client is optional
```

Naming Convention

- The de facto Java naming convention (*which you should always use for all code that you turn in for this course*) states that ClassNames, all ClassNames, and only ClassNames, begin with a capitol letter. allOtherVariableNamesShouldBeMixedCaseLikeThis.
- Example: What is happening with the following line of code?

```
java.lang.System.out.println( “print to the Java Console” );
```

- We know that **`java.lang`** must be a package.
- We know that **`System`** is a class, with a *public & static* member named **`out`**.
- We know that **`out`** is a non-null reference to an object.
- We *don’t know* the type of object **`out`** refers to; we’re going to have to look it up.
- We know that the class for which **`out`** is an instance has a public method **`println()`**.
- By the way, **`out`** refers to an instance of class **`PrintStream`**.

Access Control (Degree of Encapsulation)

Java and C++ both use the same keywords for access control:

`public` = Interface stuff -- Can be accessed by anyone.

`private` = Encapsulated stuff -- Can only be accessed by the class' own member functions, initializers, & (in C++) by the class' ***`friends`***.

`protected` = Same as private, except that the stuff is also available to subclasses. Additionally, in Java, if you grant access to your subclasses, you grant it to every class in the same ***`package`*** as well. This is often referred to as package visibility.

- Java's default access control is none of the above; if the level of access for a member function or field is not specified, it can be accessed by any class in the same ***package*** (not necessarily subclasses).
- Rule of thumb: make everything as inaccessible as possible. Make things private unless there is a good reason not to. Encapsulation is good.

Inheritance

In C++, base classes are almost always declared *public* or *protected*, as in:

```
class Derived : public Base {}
```

- Make sure you say `:public` or `:protected` since the default is `:private`, and `:private` is very rarely what you want.
- The *public* here means that *public* members of the Base class get inherited as *public* members of the Derived class; likewise, *protected* members of the Base class get inherited as *protected* members of the Derived class.
- If the Base class were declared with the *protected* access specifier, then *public* and *protected* members of the Base class would be *protected* in the Derived class (this is used with so-called "mix-in" classes).

Java does not make this distinction:

```
class Derived extends Base {}
```

Java however, makes a different distinction:

```
class Derived implements AnInterface {}
```

Memory Models

There are three types of memory: static, heap & stack.

- C++ programmers choose between ***heap or stack*** memory for new objects.

```
{                          // Construct 2 Objects ( C++ )
  X x;                     // new X using Stack Memory
  X* xp = new X();         // new X using Heap Memory
}
```

- This code has a ***memory leak*** because the *heap* memory pointed to by `xp` never gets destructed (needs an explicit call to `delete xp`).
- Note that the *stack* object `x` gets destructed automatically (no explicit call to `delete` required) upon leaving the { block of code }.

- **Java** programmers create new objects on the ***heap only*** with `new`.
- Java's ***garbage collector*** would catch such a memory leak.
- *Static* memory is reserved for class static attributes, described later.

C++ Memory Models & Syntax

```
X x;                    // Constructs a new X on the stack
X* xp = new X();        // Constructs a new X on the heap
xp = &x;                // "&" means "address of"
x = *xp;                // "*" means "dereference" (opposite of "&")
class X {
  Y* yp = null;                // "*" means "type: pointer to"
  Y& barY( const Y& yy ) {     // "&" means "type: reference"
    yp = yy.y;          // Use "." for references and stack objects
    yp->foo();          // Use "->" syntax for pointers
    return *yp;         // "*" means "dereference"
  }
  Y barY( Y yy ) {      // Copies yy (pass by value)
     return yy;         // Copies yy again (return by value)
  }
};
```

Constructors

- Both Java and C++ have *Constructors*, which are necessary to ensure that an object is in a consistent and useful state before any other object can utilize it. When you create an object, memory is allocated and the constructor code automatically runs.

```
X* xp = new X();    // new C++ object “on the heap”
X x;                // new C++ object “on the stack”

X x = new X();      // new Java objects are always “on the heap”
X x;                // In Java, this defines the type of x,
                    // but does not create a new object.
```

Java Primitive Types

- Everything in Java is an Object, except the following "primitive types" …

  `boolean byte char float double int long short void`

- Objects in Java are ***passed by reference***; primitive types are ***passed by value***.
- Unlike C++, the definitions of the primitive types are identical for all platforms.
- C++ primitive types are passed by value (***copied***); C++ reference parameters are passed by reference; other C++ parameters are passed by value.
- Java provides *immutable classes*: **`Boolean, Byte,`** `etc…` (they may not change state once initialized).

```
try // Java syntax to convert a string to an int:
{
  anInt = Integer.valueOf( numberString.trim() ).intValue();
}
catch( NumberFormatException t ) {}
```

Pass by Reference / Value Example

```
class IntWrapper // Trivial class to wrap primitive int
{
  public int i;
  public IntWrapper( int i ) { this.i = i; }
  public String toString() { return "" + i; }
  // To be most useful, should override equals() & hashcode()
}

public class JavaTypes // Example: int vs Integer vs IntWrapper
{
  public static void main( String[] args )
  {
    JavaTypes jt = new JavaTypes();
    jt.begin();
  }
```

Pass by Reference / Value (cont.)

```
public void begin()
{
  int primitiveInteger = 1;  // Passed by value
  Integer integerObject = new Integer( 1 );  // Immutable
  IntWrapper intWrapperObject = new IntWrapper( 1 );

  addOne( primitiveInteger );
  System.out.println( "Primitive int => " + primitiveInteger );

  addOne( integerObject );
  System.out.println( "Integer Object => " + integerObject );

  addOne( intWrapperObject );
  System.out.println( "IntWrapper Object => " + intWrapperObject );
}
```

Pass by Reference / Value (cont.)

```
  private void addOne( int primitiveInteger ) {
    primitiveInteger++;
  }
  private void addOne( Integer intObject )
  { // Can't change the given (immutable) Integer object.
    // Note: The reference itself is passed by value (copied).
    intObject = new Integer( intObject.intValue() + 1 );
  }
  private void addOne( IntWrapper intWrapperObject ) {
    intWrapperObject.i++;
  }
}
```

- **Primitive int => 1**
- **Integer Object => 1**
- **IntWrapper Object => 2**

Java’s class String vs StringBuilder

- Instances of **`java.lang.String`** are *immutable.*
- This is not obvious because the language provides operators (such as + for **`String`** concatenation) that make it look like the **`String`** changes state. In fact, new objects are created for each such operation. This can be very inefficient.

```
String s = “New String Without new.”;   // 1 new String object.
s += “more” + “ and more String data.”; // 2 more new objects.
```

- Java introduced **`StringBuffer`** as a substitute for **`String`** to provide better performance whenever you need a *mutable* **`String`**. All access is **`synchronized`** for thread safety, which introduces another performance issue. We recommend using **`StringBuilder`** when performance matters.

```
StringBuilder sb = new StringBuilder( “New StringBuilder.” );
sb.append( “more and more SrtingBuilder data.” );
String s2 = sb.toString();  // All objects support toString().
```

String vs StringBuilder

```
private void doFoo()
{
  String s = new String( “S” );
  StringBuilder sb = new StringBuilder( “SB” );
  foo( s );
  foo2( sb );
  System.out.println( s + “ : ” + sb ); // !! ==> S : SB_foo2.
}
private void foo( String s )
{
  s += “foo.”; // This does NOT affect the input string reference!!
}
private void foo2( StringBuilder sb )
{
  sb.append( “_foo2.” );
}
```

this

- Every object in both C++ and Java has a reference to itself called ***this***
- ***this*** is an object’s way of saying “me”
- ***this*** can also be used (in Java) for one constructor to invoke another

```
class Bar {          // Java
  private int value = 46;
  public Bar() {
    this( 711 );
  }
  public Bar( int value ) {
    this.value = value;
  }
  public boolean equals( Object o ) {
    return ( o == this ); // identical instance?
  }
}
```

Constructor Chaining

Suppose class Derived is a subclass of class Base...

- For both C++ and Java, upon constructing a new Derived, the Base constructor automatically executes first, followed immediately by the Derived constructor.
- C++ destructors execute in the reverse order - with the Derived destructor executing first, followed immediately by the Base destructor.
- Both C++ and Java provide a mechanism for the Derived constructor to choose which Base constructor to execute.
- In all cases, code in the Base constructor executes before code in **`Derived()`**.

Constructor Chaining (C++)

```
class Base {
public:
  Base( ) { …; }                        // Default constructor
  Base( String* s ) { …; }
};
class Derived : public Base {
public:
  const int foo;
  Derived( ) : foo( 24 ) { …; }  // Default constructor
  Derived( String* s ) : Base( s ), foo( 38 ) { …; }
};
```

- The first Derived constructor implicitly invokes Base’s default constructor.
- The second Derived constructor invoke the second Base constructor.
- Note the use of the `:` “initialization operator” (required to initialize the const).

Constructor Chaining (Java)

```
class Base {
  public Base( ) { …; }    // Default constructor
  public Base( String s ) { …; }
}
class Derived extends Base {
  public Derived( ) {
    super( “From Derived” );
    …;
  }
  public Derived( String s ) {
    super( s );  // Invoke the chosen Base constructor
  }
}
```

- The call to **`super()`** must be the very first line of code in the Derived constructor, because the Base class constructor code must run first.

Java’s super keyword

- Java has the keyword ***super***, which is an objects way of saying “my base class”
- ***super*** is required for “constructor chaining” as we just saw.
- ***super*** can also be used to call a base class’ overridden method:

```
class Bar {         // Java
  public void x( List list ) { …; }
}

class Foo extends Bar {
  public void x( List list ) {
    super.x( list );  // would be this->Bar::x( list ) in C++
    list.add( this ); // Add myself to the List.
  }
}
```

Abstract Classes (Java)

```
abstract class Player {
   public abstract Move   getMove();
   public          String getName { return name; }
   protected       String name = “default name”;
}
class ComputerPlayer extends Player {
   public final    Move   getMove() { …; }
}
```

- Abstract classes may define some implementation.
- A class must be declared to be abstract if it has one or more abstract method.
- Concrete subclasses must implement all inherited abstract methods.
- Note: an entire class in Java can be *final*, which means it allows no subclasses. A final class is kind of the opposite of an abstract class. Declare classes *final* unless they are designed to be sub-classed.

Interfaces (Java)

- An interface simply defines a set of method signatures.
- An "interface" in C++ is a class with nothing but "pure virtual" methods.
- Java interfaces may also define "constants" (which must be public, final & static).
- An interface may *extend* another interface.

```
interface Player {
   Move   getMove(); // public abstract Move getMove();
   String getName();
   public final static String CONSTANT = "foo";
}

class ComputerPlayer implements Player {
   public       Move   getMove() { ...; }
   public final String getName() { ...; }
}
```

Multiple Inheritance

- Java rules prohibit multiple *implementation* inheritance; a class can only inherit code from one base class; but multiple *interface* inheritance is OK.

```
class DrawingCanvas extends Canvas implements MouseListener,
  MouseMotionListener
```

- Be careful with C++ multiple implementation inheritance! These notes do not cover all of the many issues.

```
class SchedulableBackup extends Backup implements Schedulable
```

- Note that alternative designs using delegation instead of inheritance are not only viable, but often superior. This is a topic for section V – OO Design.

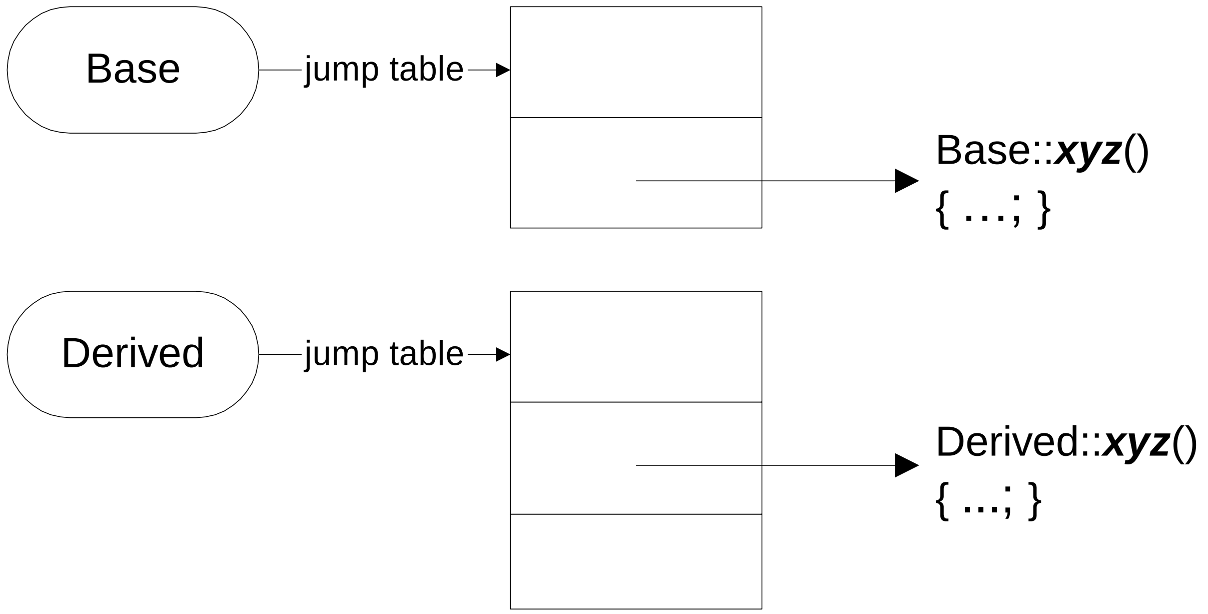
Abstract Classes & Virtual Functions (C++)

```
class Player  // Abstract: has a "pure virtual" method
{
public:
                 Player( );
   virtual      ~Player( );          // Note virtual destructor
   virtual Move* getMove( )  = 0;    // "Pure virtual"
   virtual char* getName( )   { return "default name"; }
};
```

- Class Player cannot be instantiated because it has a *pure virtual* function.
- Abstract classes defer part of their implementation to concrete subclass(es).
- C++ functions must be declared to be virtual for *polymorphism* to be possible; otherwise the decision as to which implementation to use will be made at compile-time, based on the declared (not the run-time) type of the object.
- All methods in Java are "virtual" (polymorphic, bound at run time) unless they are final, static, or private.

Polymorphism

- Polymorphic methods ("virtual" functions with dynamic, run-time binding) work "behind the scenes" using **Jump Tables**, which contain pointers to the virtual function code. If a class has one or more virtual functions, then each instance carries around a pointer to its class' jump table. Every class with a virtual function has a different jump table. Polymorphism requires this run-time "indirection".



Constructor Difficulties

If you need to do something complex within a constructor, consider having a trivial constructor and a complex **init()** method (two-phase construction). Also consider lazy initialization, delegation, and creational design patterns such as Builder.

- There are two reasons for this:
  - There can be subtle initialization problems with polymorphic method calls from within base class constructors.
  - Constructors do not have a normal return type to indicate success or failure; instead they return a reference (pointer) to the new object.

- If there is not enough memory to create the **new** heap object...
  - Java will throw an **OutOfMemoryError**.
  - C++ will return 0 from the call to **new Foo()**, essentially a *null pointer*. The programmer can optionally write a function that gets automatically called when this happens by calling **set_new_handler()**.

Constructor Difficulties (cont.)

Java and C++ differ in the class construction process…Upon constructing an instance of a Derived class, both languages construct the Base part's code first. However...

- C++ uses the Base class *jump table* while executing the Base class construction code, and then replaces that jump table with the Derived class jump table upon entering the Derived class constructor code. Therefore, any calls to a virtual method from within the Base class constructor code will call the Base class implementation of the method, not getting the desired(?) intuitive(!) polymorphic behavior.
- Java only ever uses the Derived class jump table. Thus, calls to virtual methods from within the Base class constructor code will call the Derived class implementation of the methods.

C++ plays it safe, ensuring that all initialization code defined by a class gets executed before methods may be called on behalf of an instance of the class.

Java allows Derived class methods to be called before the Derived class constructor code gets executed; this can lead to subtle bugs in class initialization.

- Avoid these problems by design.

Java Class Initialization

- First … default initializers are performed on *all* member fields (**`0`**, **`0.0`**, **`null`**, **`false`**)
- Second… field initializers are performed, in declaration order, for base class
- Third… base constructor runs
- Fourth… derived class field initializers are performed, followed by derived constructor

```
public class InitTest {
  public static void main( String[] args ) {
    System.out.println( "N = " + new InitTestDerived().getValue() );
  }
}
```

- ***outputs**:* **`N = 0`**
- ***`!?!?!`*** See next slide . . .

Java Class Initialization (cont.)

```
class InitTestBase
{
  protected int baseValue = 46; // field initializer
  public        InitTestBase() { baseValue = foo();  }
  protected int foo() { return 98;  }
}

class InitTestDerived extends InitTestBase
{
  private   int derivedValue = 711;
  public        InitTestDerived() { derivedValue = foo(); }
  protected int foo() { return derivedValue; }
  public    int getValue() { return baseValue; }
}
```

Type Checking

- Java and C++ both have strong compile-time type checking.
- Types include: **`int`**, **`long`**, **`char*`**, **`String`**, **`Object`**.
- In C++ a class defines a type.
- In Java, an interface may also define a type.
- A class may implement more than one interface, and so a class may have more than one type.

```
Foo foo = new Bar();  // Compiler error !!!  Type mismatch.
                      // Unless Bar is a kind of Foo.
```

- Smalltalk does not provide compile-time type checking.
- This lack of type checking in Smalltalk provides greater run-time flexibility, as well as greater risk of run-time errors that might otherwise have been caught by a compiler.

Casting

- A variable of a base class type may refer to an instance of any of its derived classes. This is a very common and useful thing to do, as it encourages generalizations.

```
Derived d = new Derived();  // Java
Base b = ( Base ) d;        // It is OK to up-cast.
b.polymorphicMethod();      // Derived supports Base interface.
Derived d2 = ( Derived ) b; // Legal down-cast.
```

- But some *down-casting is dangerous* and can result in undefined behavior:

```
Derived d;
Base b = new Base();
d = ( Derived ) b;    // Illegal - throws ClassCastException
d.derivedMethod();    // This would fail at run-time (if C++)
                      // because the Base class does not
                      // support the derivedMethod()
                      // No exception would be thrown in C++
```

Casting (cont.)

- Casting can also be used to convert primitive types

```
double pi = 3.14159;
int three = ( int ) pi;
```

- Java provides an **`instanceof`** operator which can be used to prevent an improper downcast.

```
if( b instanceof Derived ) {
  d = ( Derived ) b;
}
```

- C++ has a **`dynamic_cast`** operator which returns **`null`** in the case of an otherwise improper downcast.

```
Derived* d = dynamic_cast< Derived* > ( b );
if( d != null ) ...
```

Virtual Functions (C++)

```
#include <iostream.h>              // C++ example
class Base {
  public:
    Base() {}
    virtual void foo() { cout << "Base Foo" << endl; }
            void bar() { cout << "Base Bar" << endl; }
};
class Derived : public Base {
  public:
    int x = 0;
    Derived() : x( 2 ) {}
    void foo() { cout << "Derived Foo" << endl; }
    void bar() { cout << "Derived Bar : ″ << x << endl; }
};
```

Virtual Functions (cont.)

```
int main( int argc, char** argv ) {
    Base* b = new Base();
    b->foo();                       // Base Foo
    b->bar();                       // Base Bar
    b = (Base*) new Derived();      // Note: "upcast" optional
    b->foo();                       // Derived Foo
    b->bar();                       // Base Bar
    Derived* d = new Derived();
    d->foo();                       // Derived Foo
    d->bar();                       // Derived Bar : 2
    d = (Derived*) new Base();      // Note: "downcast" required
    d->foo();                       // Base Foo *
    d->bar();                       // Derived Bar : 96622347 *
}
```

* *In general, improper downcasts in C++ have undefined behavior and might crash !!!*

Destructors

- C++ also has *Destructors*, which were designed primarily for memory management, allowing a class to free up whatever additional memory it “owns.” Destructors are also useful whenever there is other logic to be executed at object-destruct-time, such as closing open files or database connections.
- C++ destructors can be a bit tricky:
  - Objects created “on the stack” will get automatically destructed.
  - Objects created “on the heap” will get destructed upon calling **`delete`**.
  - Arrays of objects created on the heap (with **`new Foo[ n ]`**) should be destructed by calling **`delete [] foos;`** `// Note the` **`[].`**
  - If you do not do this right, your program will have a *memory leak*.
- Java provides an (asynchronous) **`finalize()`** method instead of a destructor. **`finalize()`** gets automatically called just before the object’s memory gets garbage collected, which means maybe never. **`finalize()`** is provided not for memory management, but rather, for destruct-time logic only. Be careful, however, since there is no guarantee that this method will ever get called.

Virtual Destructors (C++)

- For any C++ class with subclasses, the destructor needs to be declared to be virtual, or else it will be *bound* at compile-time based on the declared type. What is the program’s output? What if **`~Base`** were declared to be *virtual*?

```
class Base {
public:
  Base() { cout << “Base::Base” << endl; }
 ~Base() { cout << “Base::~Base” << endl; }
};
class Derived : public Base {
public:
  Derived() { cout << “Dervide::Derived” << endl; }
 ~Derived() { cout << “Dervide::~Derived” << endl; }
};
Base* pd = new Derived();
delete pd; // The declared type of pd is pointer to Base.
```

Java Field Shadowing

```
public class SuperShadow {
  protected int value = 7;
  protected int getValue() { return value; }
  public static void main( String[] args   {
    SubShadow sub = new SubShadow();
    SuperShadow ss = sub;
    System.out.println( ″ss.getValue() = " + ss.getValue() );
    System.out.println( "sub.getValue() = " + sub.getValue() );
    System.out.println( ″ss.value = " + ss.value );
    System.out.println( "sub.value = " + sub.value );
  }
}
class SubShadow extends SuperShadow {
  protected int value = 11; // shadows super’s value
  protected int getValue() { return value; }
}
```

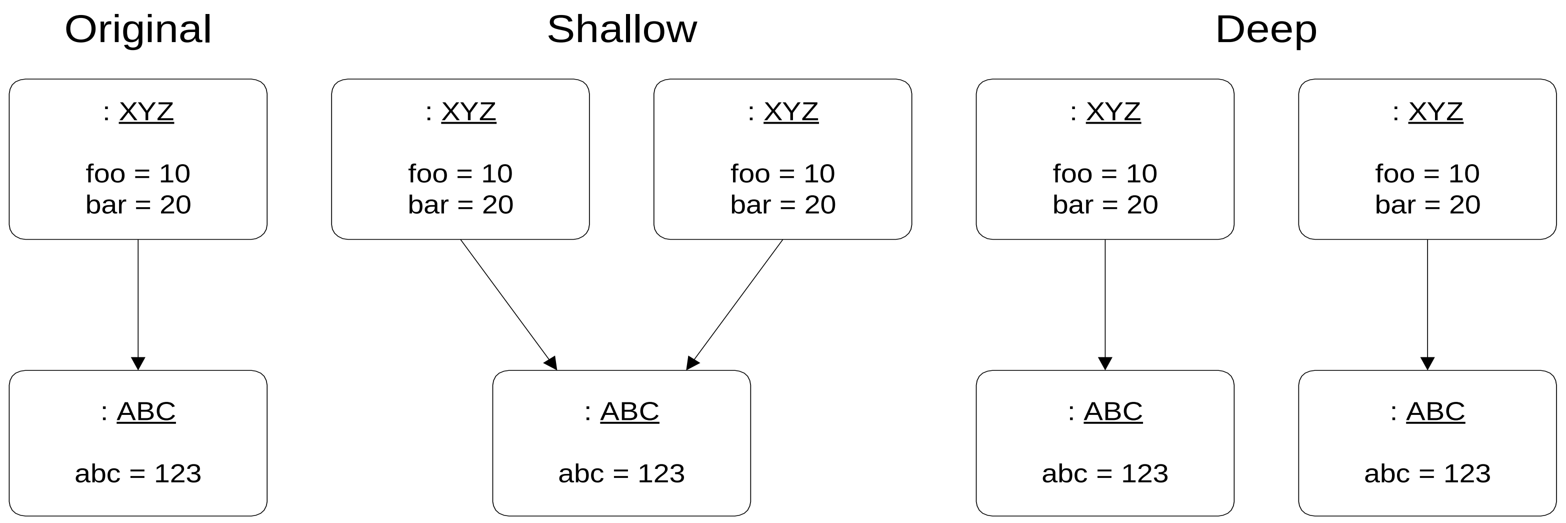
Java Field Shadowing (cont.)

```
// The program outputs:
super.getValue() = 11
sub.getValue() = 11
super.value = 7
sub.value = 11
```

- Java fields do not behave the same as methods with regard to polymorphism. Instead, Java fields (much like C++ *non-virtual* functions) are bound at *compile-time* based on the *declared* type, rather than at run-time based on the actual type.
- In general, it is not a good idea to exploit field shadowing; avoid it!
- This is another good reason to *make fields private*, with get() and set() methods.
- Also note: this is less likely to be an issue if you use semanticallyMeaningfulVariableNames (avoid field names such as foo, i, and x).

Shallow vs. Deep Copy

- C++ provides a default "copy constructor" (and **`operator=()`**).
- Java provides a default *clone* implementation.
- These default implementations provide ***shallow*** copies.
- If a ***deep*** copy is desired, you must provide the code yourself.
- Beware: it is easy to have bugs by getting this wrong!



Copy Constructors

```
class Foo
{
  public:
    Foo() // Default Constructor
    {
      this->bar = new Bar(); // this-> is optional
    }
    Foo( const Foo& copy ) // Copy Constructor
    {
      this->bar = new Bar( *(copy.bar) ); // Deep Copy …
      // NOT: this->bar = copy.bar; (would be Shallow Copy).
    }
  private:
    Bar* bar;
};
```

Java's class Object

- **`java.lang.Object`** is the root class of the class hierarchy for all Java classes. If a Java class does not declare a superclass, it automatically extends **`Object`**. Methods defined by **`Object`** are thus inherited by all classes.
- For more on this topic, refer to Chapter XIV: Java Reflection.

```
public class Object
{
  protected Object clone(); // copy self (or use copy constructor)
  public boolean equals( Object o ); // not the same as ==
  protected void finalize(); // for garbage collection
  public final Class getClass(); // refer to Java Reflection notes
  public String toString(); // frequently overridden
  . . . // refer to Java Reflection notes
}
```

Java's class Object (cont.)

- The class **`Object`** is the ultimate generalization.
- Before Java 5, **`Object`** was frequently used in "collection classes" such as **`java.util.Vector`** (essentially a *dynamic* array):

```
Vector v = new Vector();
v.addElement( new Foo() );
Object obj = v.elementAt( 0 );
Foo foo = (Foo) obj; // downcast required
```

- Post Java 5, there is a better way to implement collection classes using "generics." But the class **`Object`** is still useful in the context of "Reflection."
- C++ does not have an equivalent base class. If you want a *type* that you intend to mean "this can be a reference to anything" it is typical in C++ to use **void***.

Arrays in C++

- C++ arrays are not objects; instead they use low-level “pointer arithmetic”.

```
char* chars = new char[ 4 ]; // uses indices 0..3
// char[ 1 ] is equivalent to (char)(chars + sizeof( char ))
delete [] chars; // note: the [] is required, else memory leak.
```

- C++ arrays do not work well for arrays of *objects* (especially for heterogeneous collections); the simple pointer arithmetic used is too simple, leading to problems.
- C++ arrays do not perform bounds checking, so you can “walk off the end” of the array and not know (until after you pay your dues inside your debugger).
- C++ programmers should use a String class instead of **`char*`**, and dynamic ***collection classes*** (like those provided by the ***Standard Template Library***) instead of these error-prone arrays.

Arrays in Java

- Java Arrays are *objects*:

```
Question[] questions = new Question[ 11 ]; // One new Array object
questions[ 0 ] = new YesNoQuestion( “Are we having fun yet?” );
// An attempt to access questions[ 11 ] would throw an
// ArrayIndexOutOfBoundsException
```

- Java Arrays have special initialization syntax:

```
int[][] ints = { {1}, {2,2}, {3,3,3} }; // ints.length is fixed at 3
makeChoice( new String[] { “pick me”, “run away” } );
```

- Q: What if you don’t know the number of elements that the Array should hold at the time when you construct the Array?
- A: Use some other *dynamically sized* **collection class** (such as *ArrayList* or *HashMap*).

Class Members & Methods

- What if every instance of a class needed to know how many instances there are?
  Solution: make a *class variable* shared by all instances of the class.
  Static members act as global variables *for a class*.

```
class Person  // C++
{
public:
             Person( ) { numPeople++; }
  virtual   ~Person( ) { numPeople--; }
  static int getNumPeople( );
protected:
  static int numPeople;
};

int Person::numPeople = 0;  // C++ static initialization syntax
int Person::getNumPeople( ) { return numPeople; }
```

Class Members & Methods (cont.)

Static methods are *class* methods that can be invoked without a reference to an instance of the class. They cannot refer to any *nonstatic* member variables; they have no (implicit) *this* pointer; they cannot be called polymorphically as with non-static methods. Use static, class members and methods:

- whenever all instances of a class share data... Keeping one copy of that data, in one place, can save memory and ensure consistency. Bonus question: under what circumstances will that one copy ever get garbage collected?
- whenever you need access to a class, but you do not have a reference to any instance of the class... Statics provide global access.

```
Foo f = MyClass.bar();     // Java
Foo* f = MyClass::bar();  // C++
```

- The C++ code uses the `::` "scoping operator" for invoking class methods.
- Note that Java provides an optional static initialization block, providing the programmer with more control over the initialization of static members.
- Also note: any class in Java can have a *static* **`main()`** method.

Static Polymorphism (lack of)

```
public class StaticPoly {
  public static void main( String[] args ) {
    StaticPoly staticPoly = new StaticPoly();
    StaticPoly subStaticPoly = new SubStaticPoly();

    System.out.println( “A=“ + staticPoly.foo() );
    System.out.println( “B=“ + subStaticPoly.foo() );
    System.out.println( “C=“ + StaticPoly.foo() );
    System.out.println( “D=“ + SubStaticPoly.foo() );
  }
  public static int foo() {
    return 1;
} }
```

Static Polymorphism (lack of, cont.)

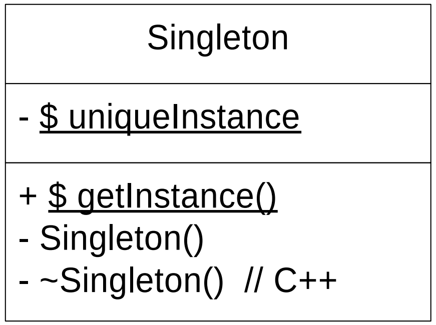
```
class SubStaticPoly extends StaticPoly {
  public static int foo() {
    return 2;
  }
}

// The program outputs:
A=1
B=1
C=1
D=2
```

- Note: this problem can be worked around using Java reflection.

Example: Singleton

```
class DatabaseServerProxy {  // C++
public:
   static DatabaseServerProxy* instance();
private:
   DatabaseServerProxy() {}
  ~DatabaseServerProxy () {}  // why not virtual?
   static DatabaseServerProxy* theInstance;
};
DatabaseServerProxy* DatabaseServerProxy::theInstance = 0;
DatabaseServerProxy* DatabaseServerProxy::instance() {
  if( theInstance == 0 )
  {
     theInstance = new DatabaseServerProxy(); // where's delete?
  }
  return theInstance;
}
```

Design Pattern: *Singleton*



Intent: Ensure that one instance of a class exists, and provide global access to it.

- There are classes that should have only one instance. Examples: the File System, the Database Proxy, perhaps the Security Manager...
- Make the constructor *private*. Give the class one *static* instance of itself.
- Ensure that the implementation is *thread-safe*.
- Can be a performance bottleneck if thread locking is used.
- What about destroying (deleting) the instance? The class is responsible for its creation, and therefore should also be responsible for its demise.
- *Variation:* Java's **`Class.forName( String className ) -`** There is one instance of class **`Class`** per class per ***`ClassLoader`*****`.`**
- Given the *disadvantages* of the pattern, consider alternatives. Is it reasonable to have one instance of the class per User-Session, say?

Singleton (cont.)

```
// Example Singleton class from the Video Store:
class Inventory { // Java
  private static Inventory instance = new Inventory();
  private Inventory() {}
  public static Inventory instance() { return instance; }
  ...
}
```

- A Variation of this Singleton formulation is to have a class with nothing but static methods. But this approach is limited because neither Java nor C++ allows static methods to behave polymorphically. What about defining a subclass of the *Singleton* class? One work around for this is to apply the *Bridge* pattern to separate the Singleton's abstraction from its implementation; the Singleton class can delegate to some other abstract implementation class that does the real work.

Memory Management

Memory management (especially in C++, but also in Java to a lesser degree) is a serious issue, which can consume a lot of programming effort - to prevent bugs and ensure optimal performance (for Java's garbage Collection).

- In C++, whenever memory is dynamically allocated (on the heap), 1 (one) object, class, or method must be *responsible* for freeing (deleting) that memory.
- In Java, set references to `null` when done, especially statics and collections.
- Avoid creating gratuitous new objects, especially inside a loop.

Some Patterns:

- You create it, you delete it…
- Ownership transfer
- Memory Manager
- Reference Counting
- Garbage Collection (implemented by Java, C#, & Smalltalk virtual machines).
- Note: memory is not the only resource that must be managed.

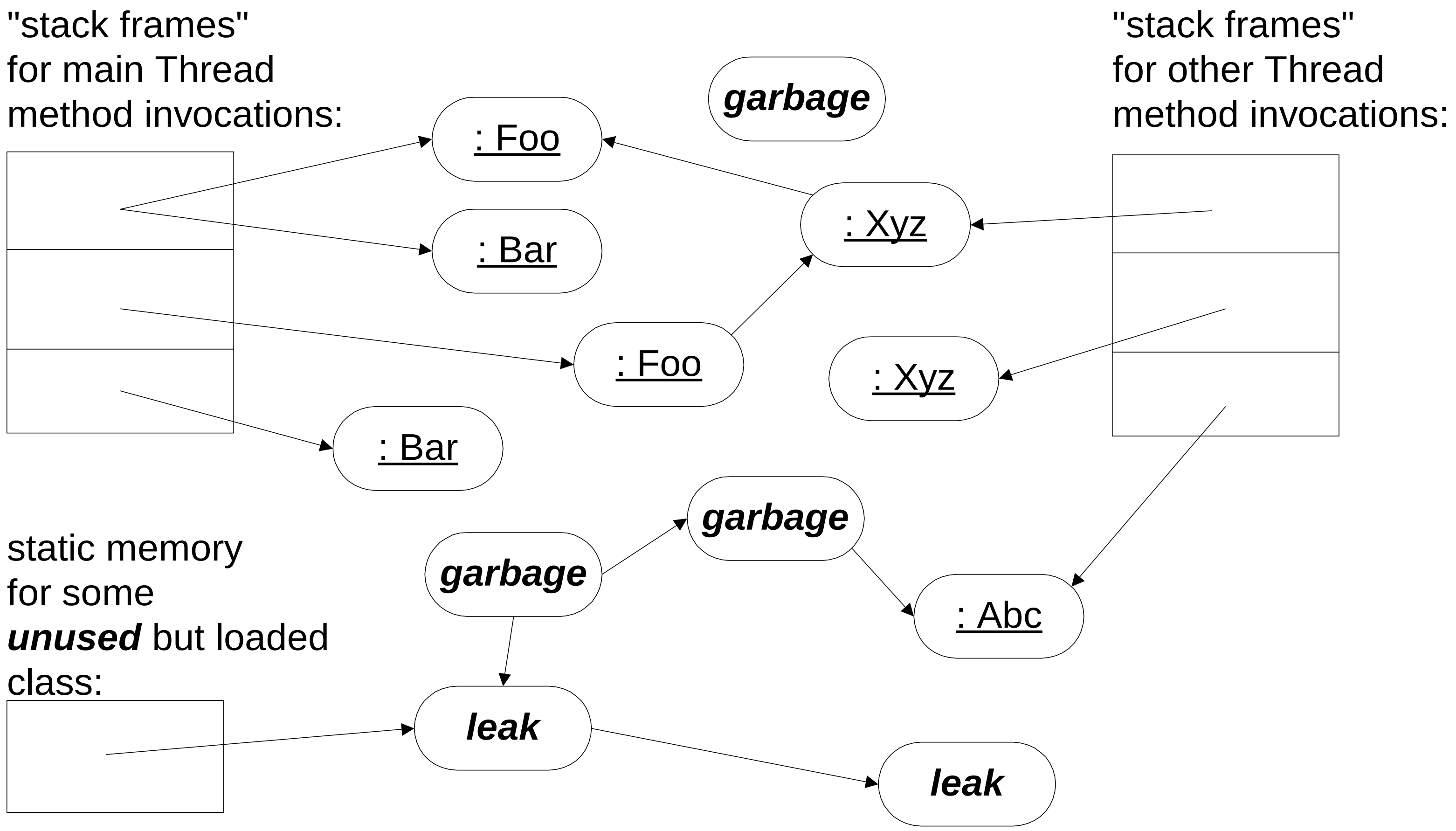
Java Garbage Collection

Pass 1: Find and mark all the objects that can be reached by direct or indirect reference from all variables in scope, for all active Threads.

Pass 2: Return the memory for all unmarked objects back to the heap's free memory pool, *maybe* calling **`finalize()`** on those objects.

- Note: In Java, Garbage Collection happens asynchronously, in a Virtual Machine process (Thread) outside of your control, and it can be slow.
- Memory leaks can happen in Java if you have non-**`null`** references to unwanted objects. If you want an object to be garbage collected, you have to be sure that all references to it are **`null`**. Such memory leaks in Java are common with non-**`null`** static references, and also non-Java memory, and other system resources.
- To increase the performance of your garbage collector, set references to **`null`** when you're done with them, breaking the reference chains; also clean out your collections.
- Sometimes you need to call special functions for built-in Java classes to free up system resources efficiently. Examples:
  - Call **`close()`** for java.sql.Connection, java.sql.ResultSet, ...
  - Call **`dispose()`** for java.awt.Graphics, java.awt.Frame, java.awt.Dialog, ...

Java Garbage Collection (cont.)



Reference Counting

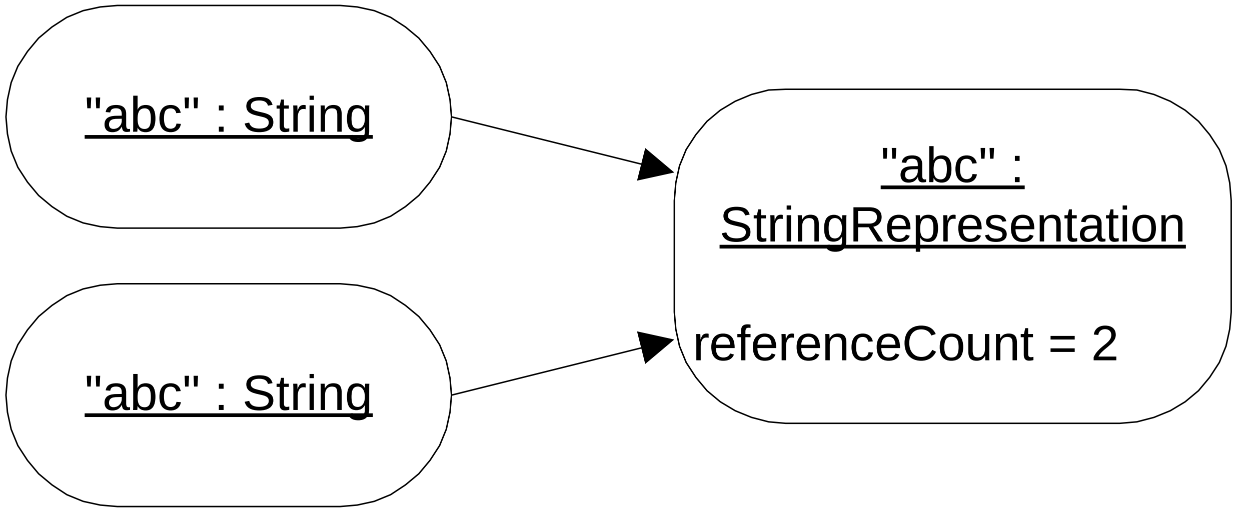
- Make the object responsible for its own timely demise.

```
class Shared // C++
{
public:
  Shared() { refCount = 1; }
  void adopt() { refCount++; }
  void disown()
  {
    if( 0 == --refCount ) delete this; // object suicide
  }
private:
  int refCount;
}
```

Reference Counting (cont.)

Example from a C++ String class:

- Instances of String share an instance of a StringRepresentation object.
- StringRepresentation has “copy-on-update” semantics.



C++ Operator Overloading

- C++ allows operator overloading with the “operator” keyword.
- Methods are *overloaded* when they have the same name but different signatures (ignoring the return type):

```
class Foo
{
  public:
    X* bar( const X& x );
    Y* bar( String* initializer );          // Overloaded bar()
    boolean operator==( const Foo& other ); // Overloaded ==
    Foo&    operator=( const Foo& other );  // Overloaded =
  private:
    Y* managedMemory;
};
```

C++ Operator Overloading (cont.)

- C++ provides overloading for many operators, including: **`+ - * / % ^ & | ~ ! = == != < > += -= << >> && || ++ -- -> () [] new delete`**
- Also: **`operator long()`**, **`operator int()`**, **`operator const char*()`**, ...

```
Foo& Foo::operator=( const Foo& other ) {
  if( &other != this ) // Do this or self assignment crashes !!
  {
    delete managedMemory; // Do this or have a memory leak !!
    managedMemory = other.managedMemory;
  }
  return *this; // This allows assignment chaining: a = b = c...
}

Foo f( 1 );
Foo* f2 = new Foo( 2 );
f = *f2;   // Identical to: f.operator=( *f2 );
```

C++ Templates (Parameterized Types)

- Motivation: Write a ***type-safe*** C++ “collection class” (without **`void*`**). . .

```
template < class X >      // template is a C++ keyword.
class LinkedList          // X is the template parameter.
{
  public:
    X*    getFirst( );    // Refer to Iterator design pattern
    X*    getNext( );
    ...
};
LinkedList< MyClass > myClassList;      // Usage in code...
MyClass* foo = myClassList.getFirst( ); // No cast required
```

- This mechanism eliminates the need for a class called LinkedListOfX.
- Learn to use C++’s Standard Template Library (STL)…

Java Templates (Generics)

- As of Java 1.5, Java also has “generics” providing increased type safety and expressiveness. (Collections will be dealt with in more detail later).

```
// Bad example - avoid this usage:
private final Collection foos = ...; // collection of Objects
foos.add( new Bar() ); // Wrong, but compiles and runs
for( Iterator i = foos.iterator(); i.hasNext(); ) {
  Foo foo = (Foo) i.next(); // throws ClassCastException

// A better way:
private final Collection< Foo > foos = new ArrayList< Foo >();
foos.add( new Bar() ); // Will not compile (unless Bar is-a Foo)
for( Foo foo : foos ) { // No downcast required.
  ... // Easy syntax, type safety, all good :-)
```

Inner Classes

- Java provides an “inner class” mechanism, allowing classes to be contained within other classes, just like methods and fields.
- The motivation for this is programmer convenience, intended to be used for small or trivial jobs where a class is required.
- An inner class may see the private methods and fields of its containing class (if static) and/or object (if non static).
- We will see numerous examples later with the Java AWT. Inner classes are great for AWT event handling :^)

```
class Outer {
  private class Inner1 {}
  public static class Inner2 {} // Outer.Inner2
}                               // Outer$Inner2.class
```

Inner Classes (cont.)

```
public class Outer {
  public Outer() {
    new Inner();
  }
  private String io = “OUT”;
  private class Inner {
    private String io = “in ”;
    public Inner() {
      System.out.print( “” + this.io );
      System.out.println( “” + Outer.this.io ); // !
  } }
  public static void main( String[] args ) {
    new Outer();
} }
```

- Outputs: ***in OUT***
- The inner class’s .class file is named: **`Outer$Inner.class`**