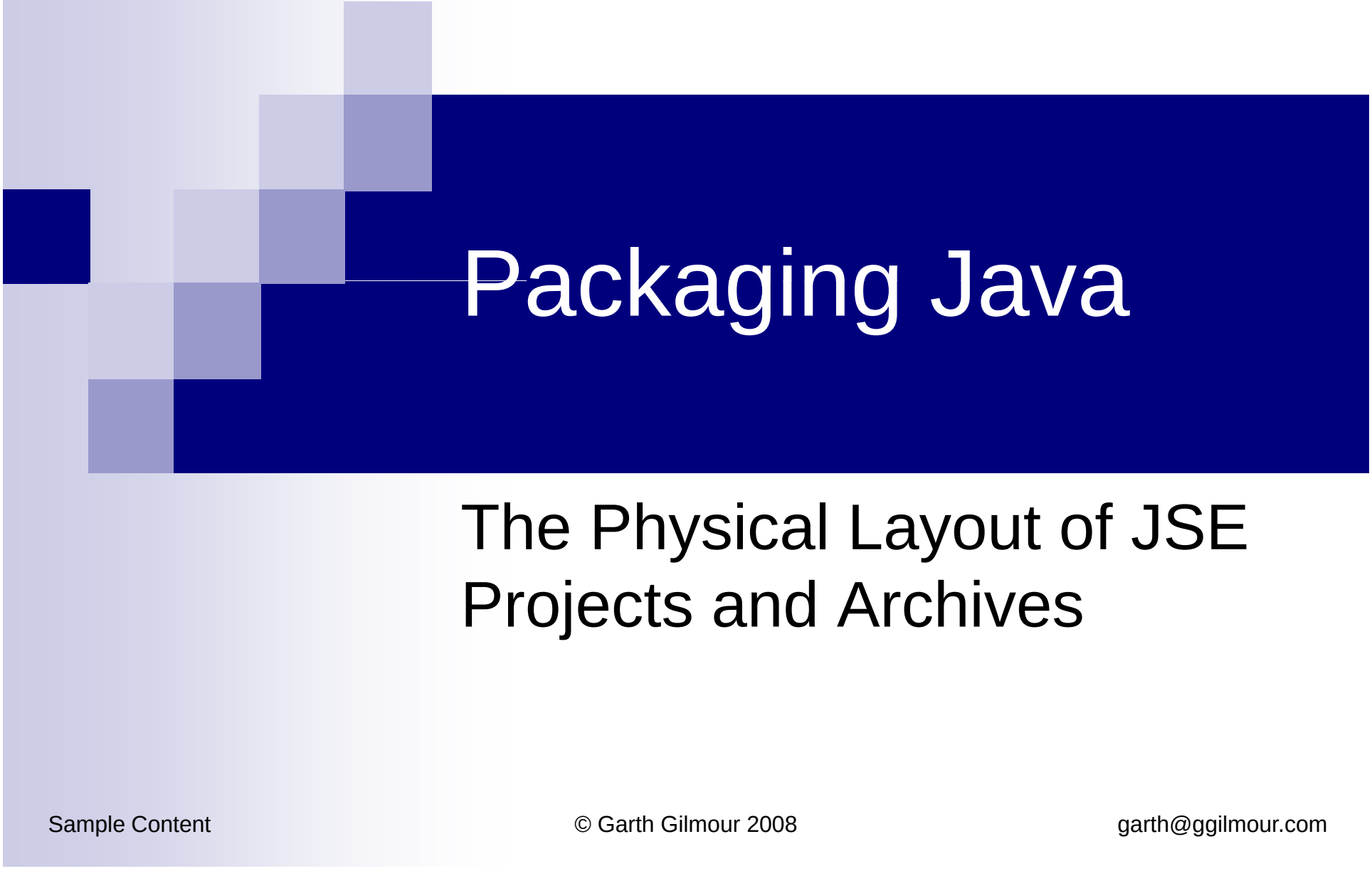
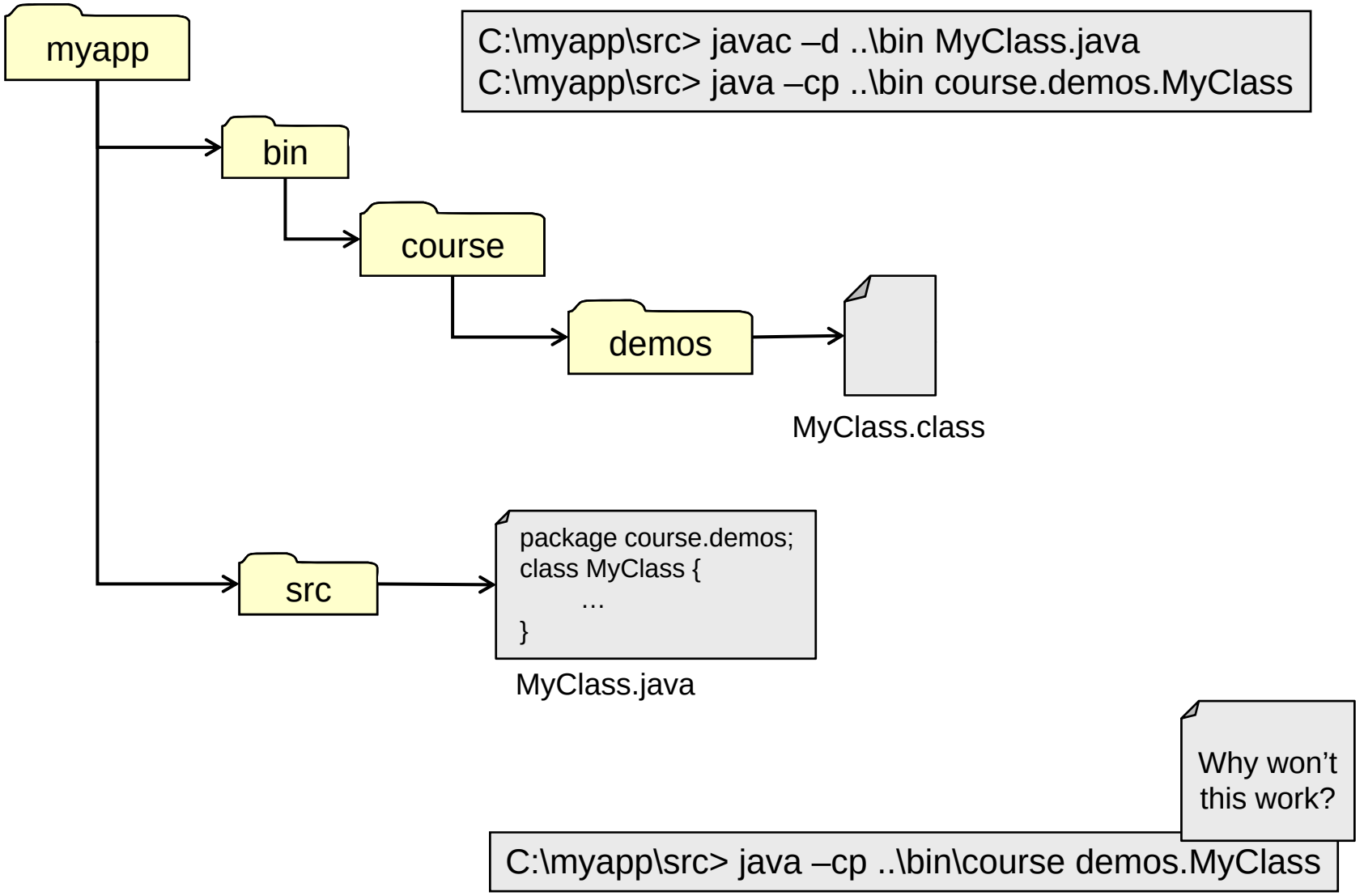


The Physical Layout of JSE Projects and Archives

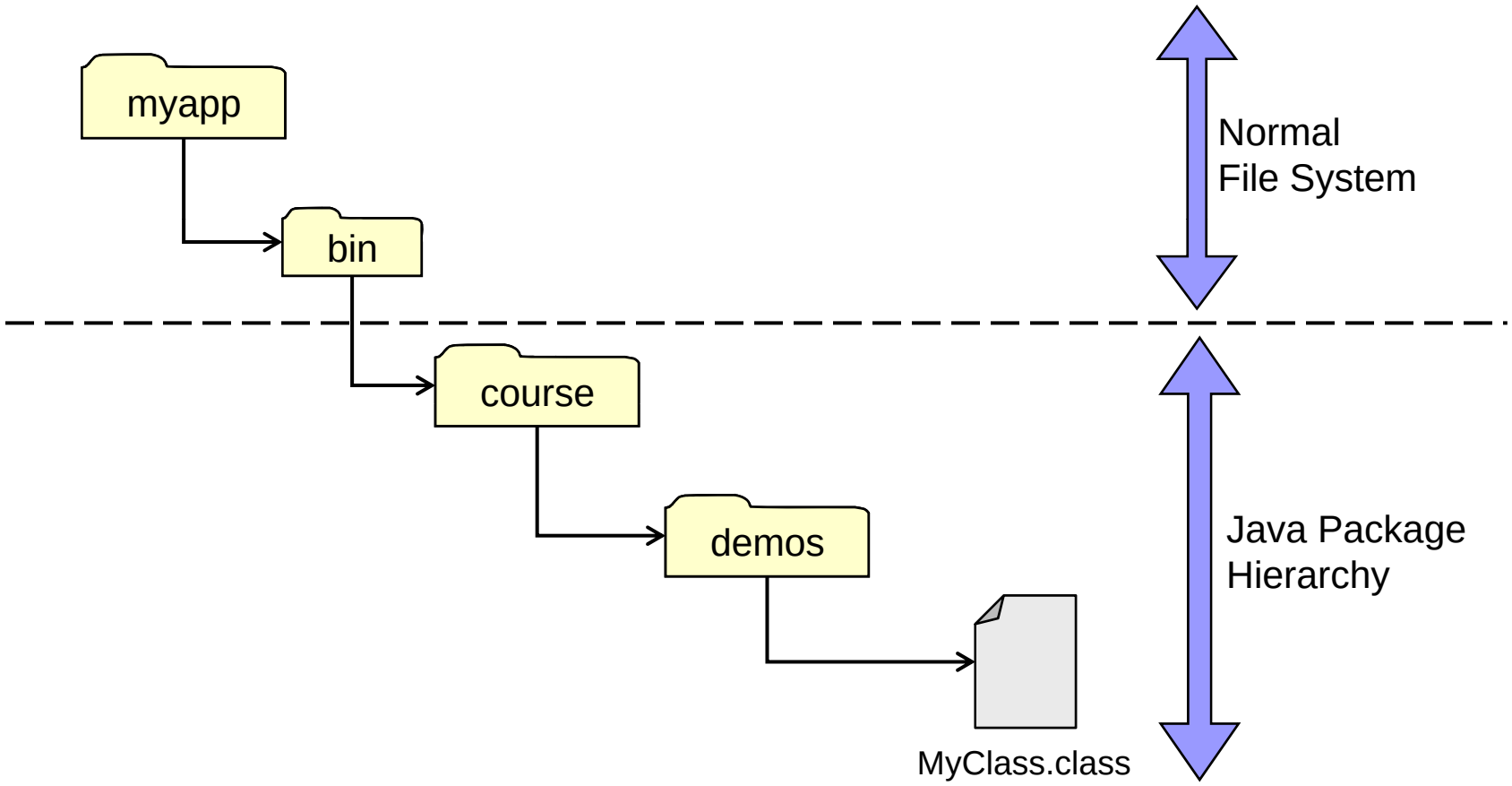
garth@ggilmour.com

Java Packages

- In Java a class is always compiled into a package
  - The ‘default package’ is for classes belonging to none
- Packages are represented as nested directories
  - Enabling synonymous classes to occur in different packages
- The VM sees a class by its fully qualified name
  - The package name plus the class name
  - The full name is encoded inside the class file
- Packages themselves are not nested
  - Classes in the package ‘com.megacorp’ have no special access rights to classes in (for example) ‘com.megacorp.gui’

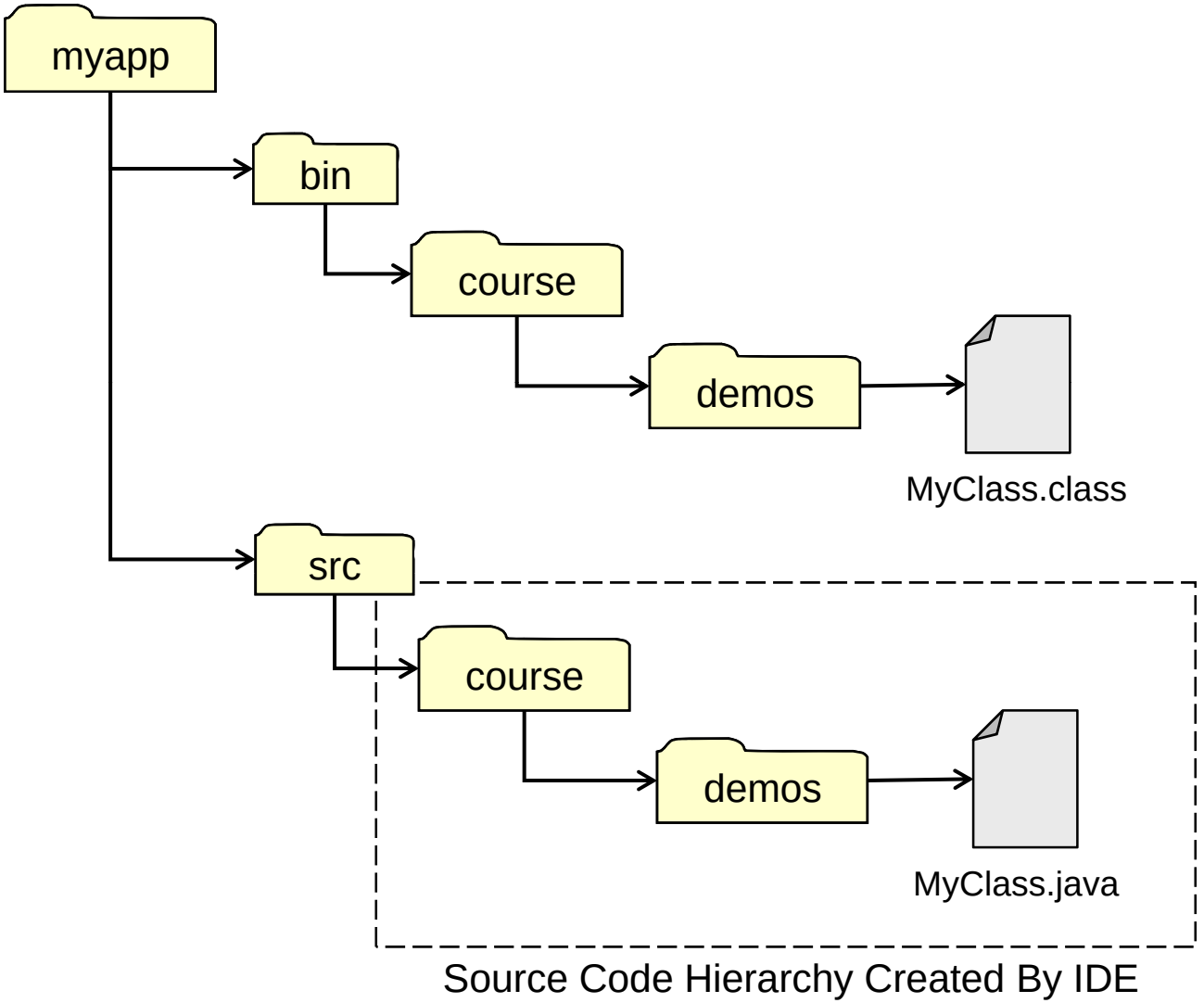


The Package Hierarchy



Organizing Packages

- Packages are arranged as a tree of subfolders
  - So ‘com.megacorp.Employee’ would be stored as ‘com/megacorp/Employee.class’
    - Otherwise the compiler would place ‘com.megacorp.Employee’ and ‘com.widgetsrus.Employee’ into the same folder
- The Virtual Machine expects to find this tree
  - So when copying class files between drives or machines you should always preserve the structure of subfolders
- Most IDE’s store source files in the same way
  - So ‘Employee.java’ would be found inside ‘com/megacorp’
  - There is no requirement in the language to do this

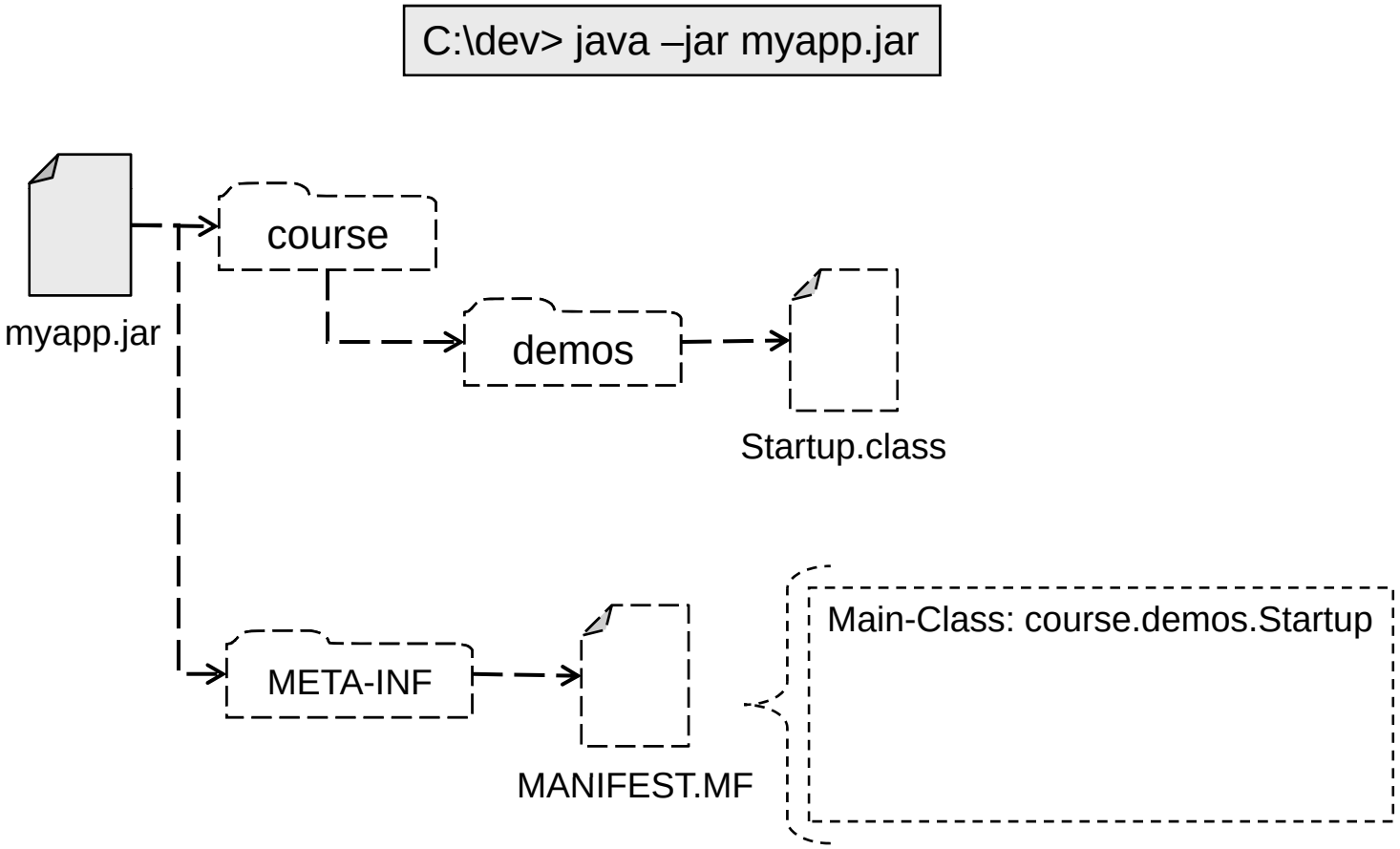


Source Code Hierarchy Created By IDE

Java Archives (JAR Files)

- Java Archives provide a way of bundling classes
  - They were originally designed to avoid web browsers making multiple trips to the server when loading applets
- Files are archived using the ZIP compression algorithm
  - You can create JAR's using tools like 'WinZip'
- A JAR normally has its own configuration file
  - Called 'MANIFEST.MF' and located in the 'META-INF' folder
- The JDK comes with a 'jar.exe' tool for building archives
  - The options are very similar to the UNIX 'tar' command
  - The tool can also add entries to the manifest file

Java Archives (JAR Files)



Understanding The Classpath

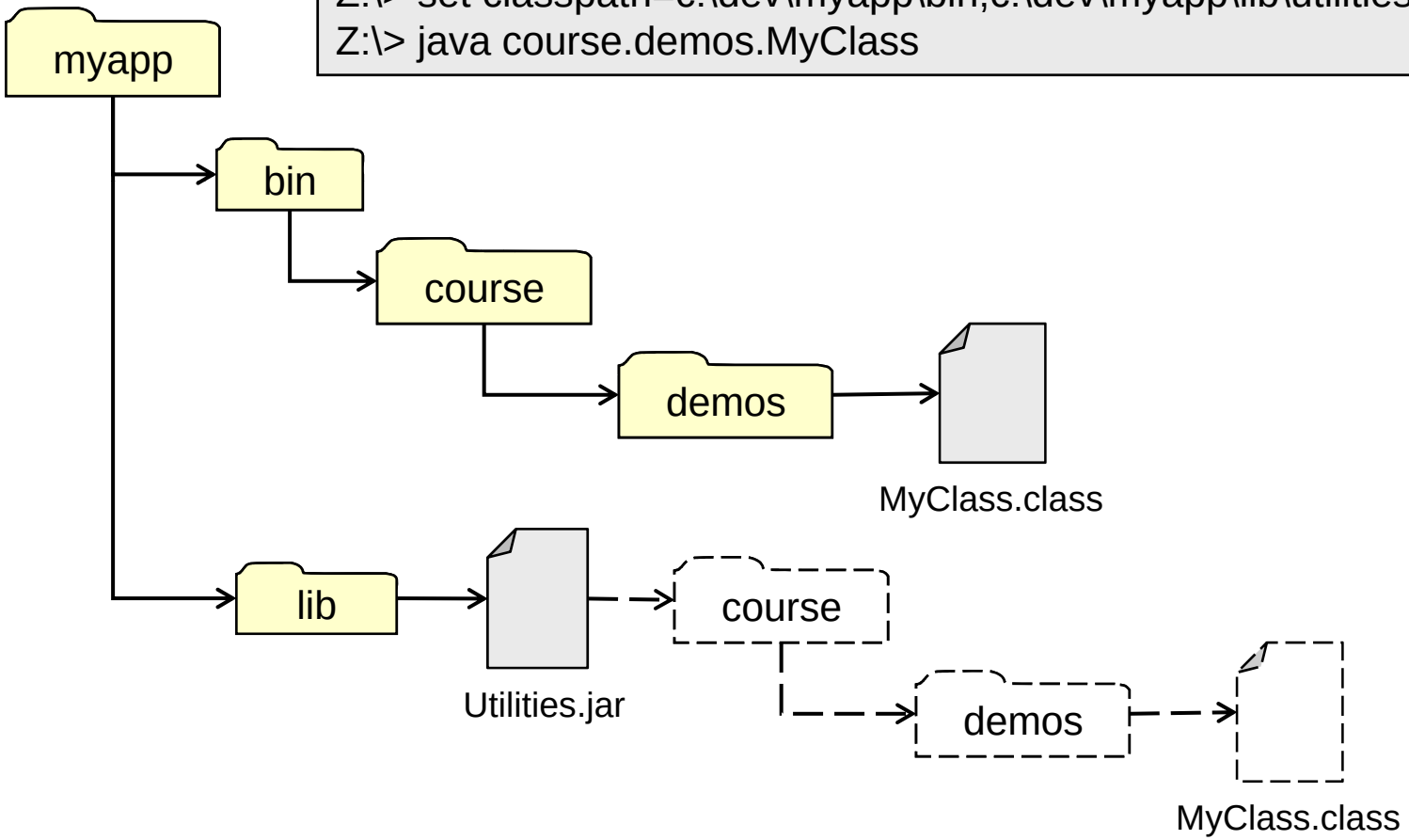
- How do the Java compiler and VM find classes?
  - By searching the entries in the ‘classpath’
- The term ‘classpath’ means two things:
  - The set of locations known to the compiler and/or virtual machine from where class files can be located and loaded
  - The OS environment variable which provides a set of locations
- These two meanings are related but not identical
  - The locations actually used are a superset of those listed in the environment variable (especially in JEE)

Understanding The Classpath

- The classpath environment variable contains
  - The path to a directory on the file system
    - For example ‘c:\dev\myapp\bin’
  - The path to a JAR file (including the JAR filename)
    - For example ‘c:\dev\myapp\lib\utilities.jar’
- We always specify a location inside which there is a hierarchy of folders representing package names
  - Class files always need to be stored in a hierarchy of directories
  - If the VM needs to load ‘com.megacorp.MyClass’ it will:
    - Look for ‘com/megacorp/MyClass.class’ inside ‘bin’
    - Look for ‘com/megacorp/MyClass.class’ zipped into ‘utilities.jar’

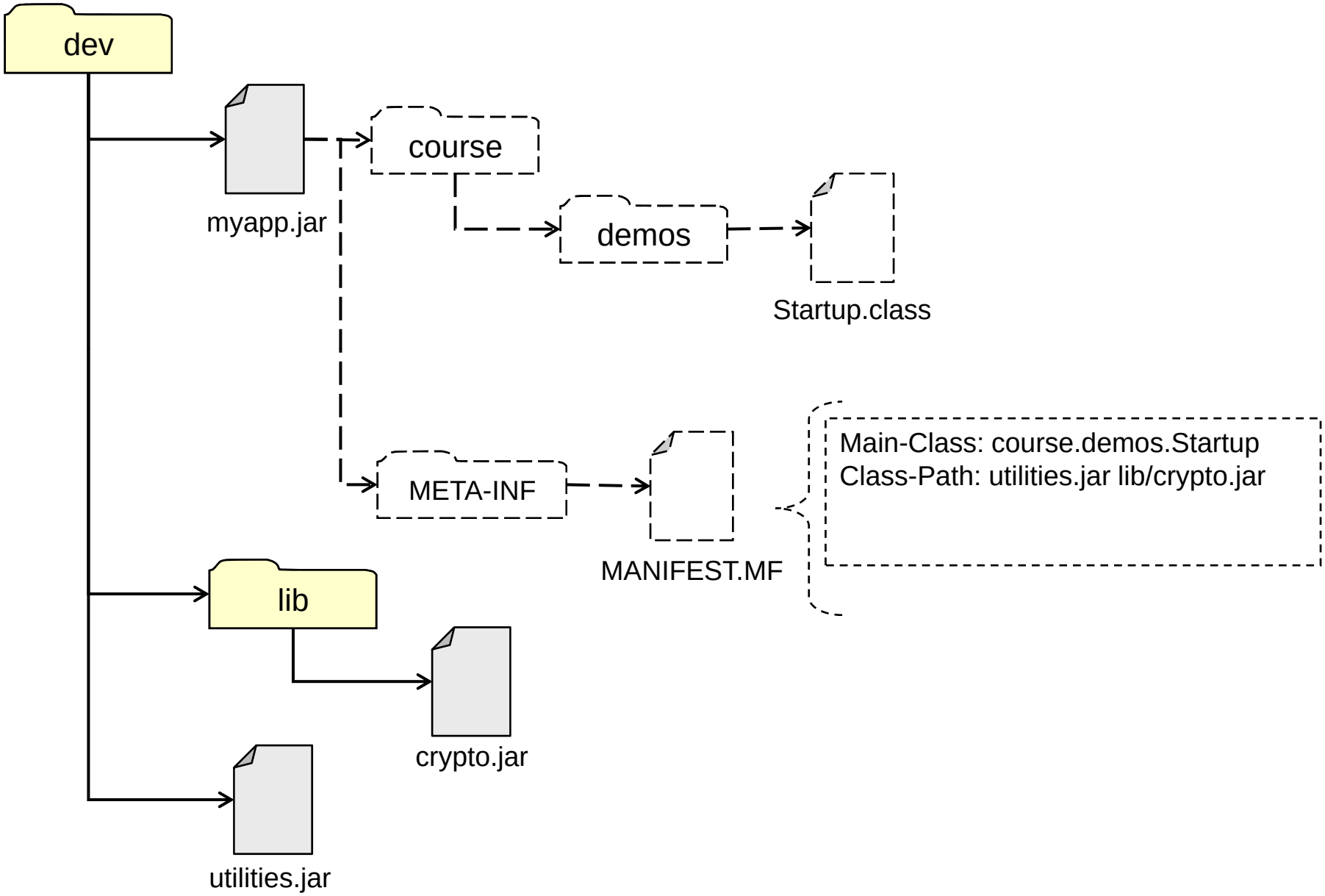
Understanding The Classpath

```
Z:\> set classpath=c:\dev\myapp\bin;c:\dev\myapp\lib\utilities.jar
Z:\> java course.demos.MyClass
```



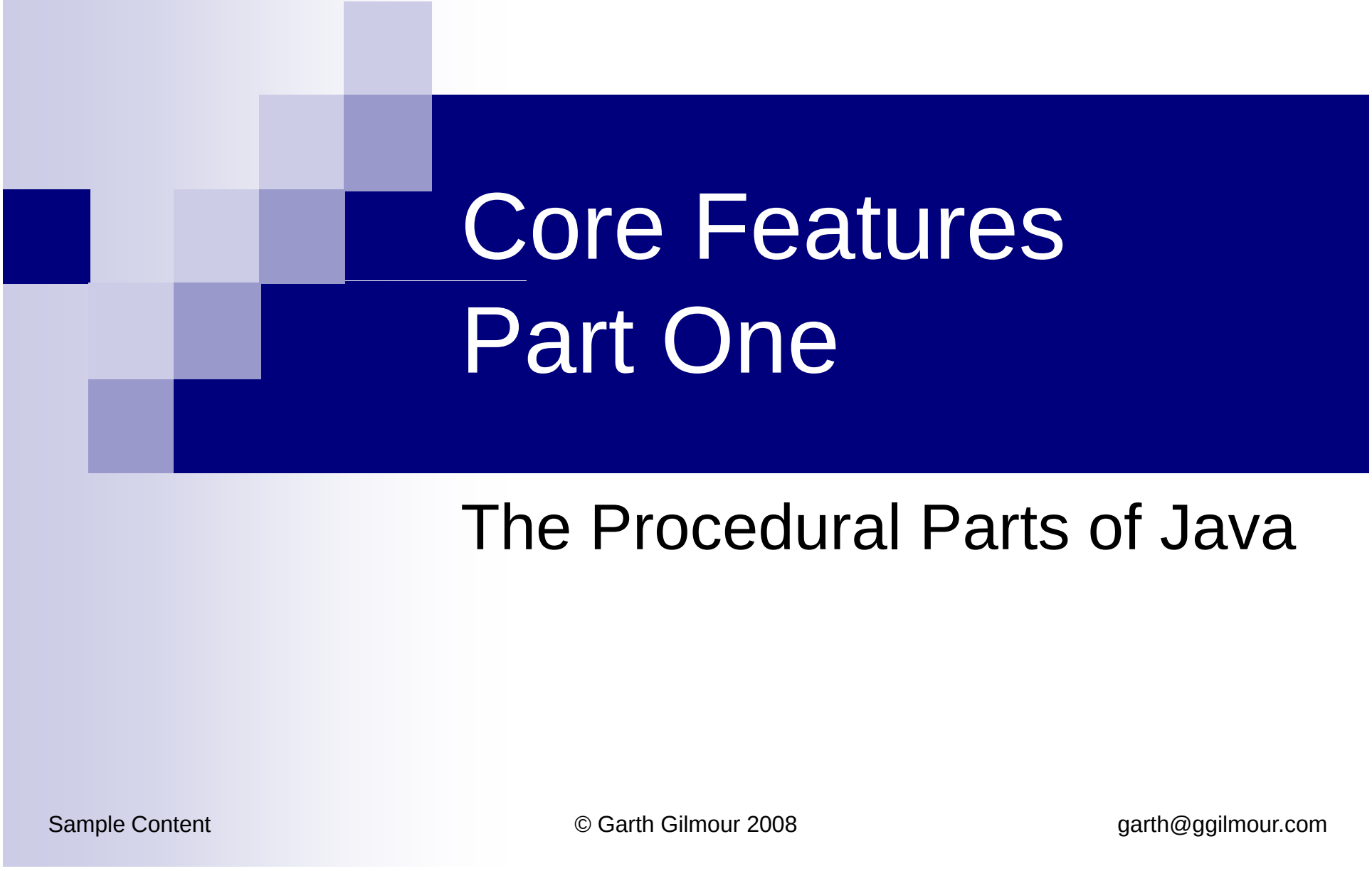
Understanding The Classpath

- The classpath environment variable can be overridden
  - By using the ‘-classpath’ or ‘-cp’ command line option
  - This works with both the compiler and virtual machine
- Note that this option is ignored when using ‘-jar’
  - The compiler and virtual machine assume that all application specific classes will be within the archive
- A JAR file can specify additional archives
  - To hold dependencies of classes found in the current JAR
  - This is done by adding a ‘Class-Path’ entry to the manifest file
    - Paths discovered in this entry are interpreted as relative file paths



The Java Build Process

- References to other classes are resolved twice
  - Firstly when the compiler checks that dependencies exist
    - And contain the symbols that you are using in your code
  - Secondly when VM loads the dependencies at runtime
- Locations are never hard-coded
  - A class file does not contain any information about the locations of the other classes it is dependant on
  - Hence you could compile and test your program, bundle the classes into multiple JAR files and then rerun the program
    - As long as you remembered to update the CLASSPATH
- Classes don't have to be loaded from the file system
  - A custom class loader can load class files from anywhere



garth@ggilmour.com

Hello World in Java

- This is the simplest possible Java program:
  - All classes live in packages
  - All functions (methods) live in classes
  - The entry point is a method called ‘main’ which:
    - Can be called without creating an object (static)
    - Takes an array of strings as a parameter
    - Returns nothing (void)

```
package demos.basic;
public class HelloWorld {
        public static void main (String args[ ]) {
                System.out.println ("Hello World");
        }
}
```

Identifiers

- Java is fully case sensitive
  - As is the compiler and virtual machine
  - Watch out for this on Windows platforms
    - Especially if you have previously developed in a case insensitive language…
- An identifier is made up of:
  - A letter plus zero or more letters or numbers
  - The letters are those defined by Unicode
  - Underscore and currency symbols count as letters
    - Currency symbols are traditionally reserved for compiler generated names

Coding Conventions

- Package names are always lowercase
  - E.g. ‘com.megacorp.mousetrap.gui’
- Class names are always capitalized
  - E.g. ‘Employee’ or ‘PurchaseOrder’
- Function (method) names are in camel case
  - E.g. ‘calculateInterestRate’ or ‘findOrder’
- Constants are always capitalized
  - E.g. ‘PI’ or ‘MANDATORY_RETIREMENT_AGE’
- There is no convention for variable names
  - You may see ‘myvar’, ‘myVar’, ‘my_var’ or ‘_myvar’

Basic Syntax

- Braces are used to delimit a block of code
  - A variable is ‘scoped’ inside the block it belongs to
- A semi-colon terminates each line
  - A logical line of code does not have to be identical to a physical line in the file, which makes formatting easier

```
class MyClass {
        void someMethod() {
                while(op1()) {
                        if(op2()) {
                                op3();
                        }
                }
        }
}
```

```
int result = someMethodCallWithLongName();
```

```
int result =
      someMethodCallWithLongName();
```

```
int i = 7;
```

```
int
i
=
7
;
```

Syntax and Grammar

abstract	assert	boolean	break	byte	case	catch	char
class	const	continue	default	do	double	else	extends
final	finally	float	for	goto	if	implements	import
instanceof	int	interface	long	native	new	package	private
protected	public	return	short	static	strictfp	super	switch
synchronized	this	throw	throws	transient	try	void	volatile
while							

- Procedural
- OO Support
- Exceptions
- Threading
- Unused

Basic Types

- Java has three basic types
  - Primitive types are boolean and numerical
    - They are allocated on the stack as in C/C++
    - Their size and format is fixed across platforms
  - Reference types are disguised pointers
    - They point to a class, interface or array instance
    - You cannot dereference a Java reference
  - Objects have no type but belong to a class
    - A reference refers to an object or to the null type
    - All arrays and strings are objects

Primitive Types

Type	Size (Bytes)	Examples
byte	1	3
short	2	4
char	2	‘A’,’\u0000’
int	4	5
float	4	6.1f
long	8	7
double	8	8.1
boolean	N/A	true,false

Primitive Types

- Primitive types are signed
  - Except for the boolean and char types
  - You cannot create unsigned versions
- The char type uses the Unicode character set
  - UTF-8 encoding is used by the VM
  - Hence the need for a separate byte type
- The size of the boolean type is irrelevant
  - No equivalence between booleans and numbers
  - You cannot cast booleans to numbers or vice versa

Narrowing Conversions

- Use an explicit cast to store
  - A double in a float, long, int, char, short or byte
  - A float in a long, int, char, short or byte
  - A long in an int, char, short or byte
  - An int in a char, short or byte
  - A char in a short or byte
  - A short in a byte or char
  - A byte in a char
- An explicit cast will always succeed
  - But your original value may be badly mangled…

Narrowing Conversions

```
public class Conversions {
    public static void main(String[] args) {
        double d = -129.456;

        float f = (float)d;          //a float is smaller than a double
        System.out.println(f);       //prints '-129.456‘

        int i = (int)f;              //an int does not support fractions
        System.out.println(i);       //prints '-129‘

        short s = (short)i;          //a short is smaller than an int
        System.out.println(s);       //prints '-129‘

        byte b = (byte)s;            //a byte is smaller than a short
        System.out.println(b);       //prints '127'
    }
}
```

Casting in Detail

- We cast when the range of the receiving type is smaller
  - Possible loss of precision does not mean a cast is required

Conversion	Casting	Reason
char into short	explicit	A char can hold a greater range of positive numbers than a short
short into char	explicit	The sign may be lost
long into float	implicit	A float has a greater range than a long (precision may be lost)
int into float	implicit	As above
double into long	explicit	Loss of fractional part

Narrowing Conversions

- A cast is used to store a value in a smaller type
  - Floating point literals are double by default
  - On a VM register types <32 bits become integers
    - Because the width of a register is fixed at 4 bytes
    - This can produce some surprising results

```
byte b1 = 1;
byte b2 = 2;
//Wont work! Use (byte) (b1 + b2)
byte b3 = b1 + b2;
```

Making Choices

- If statements work the same as in C/C++
  - The condition must be a boolean expression
- Switch statements are also very similar
  - Each case value must be unique
  - Case statements should end with a break
  - The default case statement is optional
  - You can switch on a char, byte, short or int
    - The compiler will check that the values used in case statements are in the correct range for the type

Making Choices

```
if (password.equals(“wn1hgb”)) {
      //do stuff
}  else if (password.equals(“admin”)) {
      //do other stuff
} else {
      //do default
}
```

```
switch ( obj.func() ) {
      case 1:
      //do stuff
      break;

      case 12:
      //do stuff
      break;

      default:
      //do stuff
}
```

Iteration

- Java supports the same loops as C/C++
  - The ‘for’ loop for bounded iteration
  - The ‘while’ loop for unbounded iteration
  - The ‘do…while’ loop which executes at least once
- The condition must be a boolean expression
  - You can ‘continue’ past the current iteration
  - You can ‘break’ out of one or more loops
- Break and continue statements may be labeled
  - Which transfers control to an arbitrary enclosing block

Iteration

```
while (i < 7) {
      if(someCondition) {
            i++
      }
      //do stuff
}
```

```
do {
      if(someCondition) {
            i++
      }
      //do stuff
} while (i < 7);
```

```
for (int i=0; i< LIMIT; i++) {
      //do stuff
}
```

Iteration

```
while (i < 7) {
    if(conditionA) {
        //exit the loop
        break;
    } else if (conditionB) {
        //skip the rest of the body
        continue;
    }
    //do stuff
}
```

```
OUTER:
while (i < 7) {
    while (x > 10) {
        if(conditionA) {
            //exit the inner loop
            break;
        } else if (conditionB) {
            //exit both loops
            break OUTER;
        }
    }
}
```

Core Features Part Two

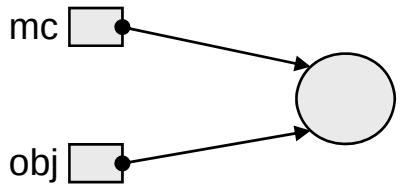
The Built In Object Types

garth@ggilmour.com

References and Objects

- A variable declaration with a class name on the left hand side always denotes a reference
  - A reference is a link or pointer to an object
  - A reference of type ‘Object’ can point to any object
- Objects are always used indirectly in Java
  - Although we may say a method is ‘passed an object’ this is always shorthand for ‘passed a reference to an object’
- The only operator you can use on references is ‘==’
  - It always tests if both references refer to the same object

```
MyClass mc = new MyClass();
Object obj = mc;
```

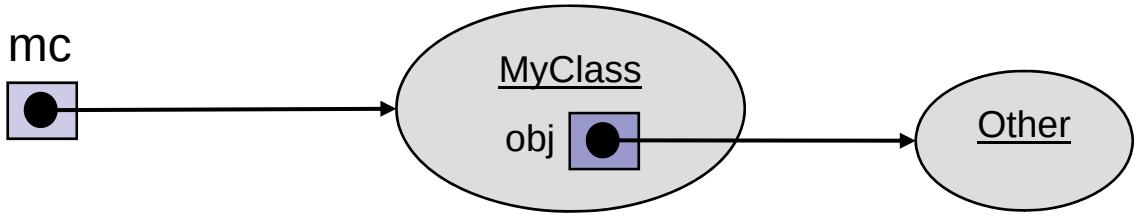


A variable declaration with a class name on the left hand side always denotes a reference

Objects never contain other objects - instead one object references another

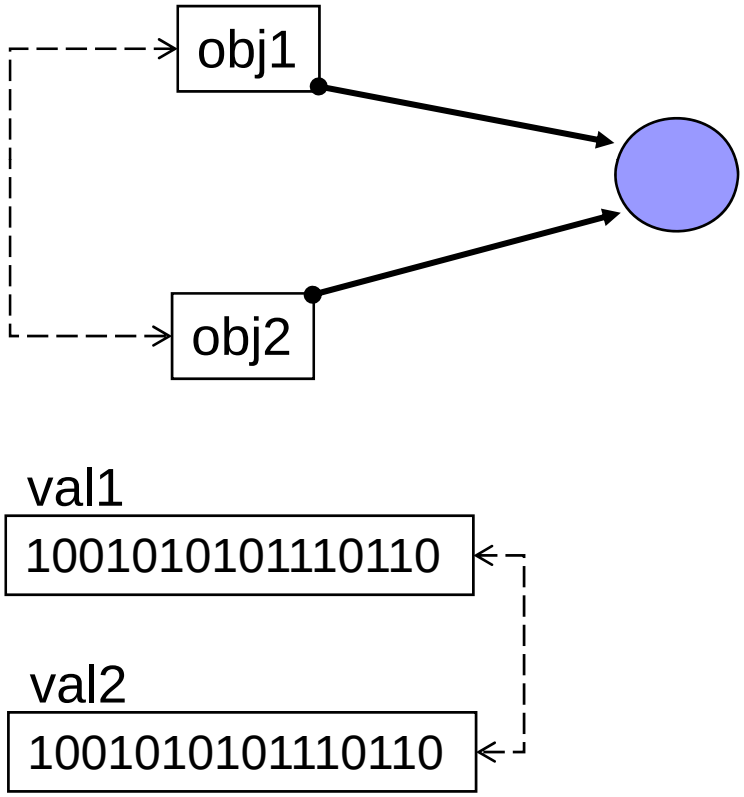
```
class MyClass {
    private Other obj = new Other();
}
```

```
MyClass mc = new MyClass();
```



Equality in Java

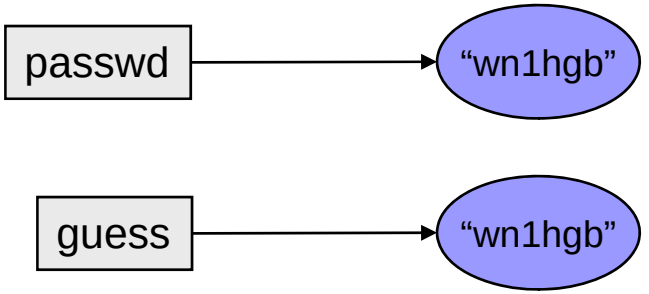
```
Object obj1 = func1();
Object obj2 = func1();

int val1 = func2();
int val2 = func2();

//variables are references
if(obj1 == obj2) {
      //do stuff
}

//variables are primitive types
if(val1 == val2) {
      //do stuff
}
```



A Common Mistake…

```
String passwd = readPasswdFromConfigFile();
String guess = readInputFromGui();
```

```
//WRONG - CAN NEVER MATCH!
if(passwd == guess) {
      logInUser();
} else {
      denyAccess();
}
```



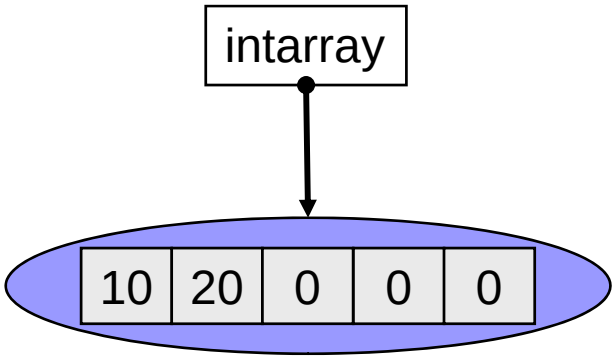

```
//CORRECT
if(passwd.equals(guess)) {
      logInUser();
} else {
      denyAccess();
}
```

Arrays In Java

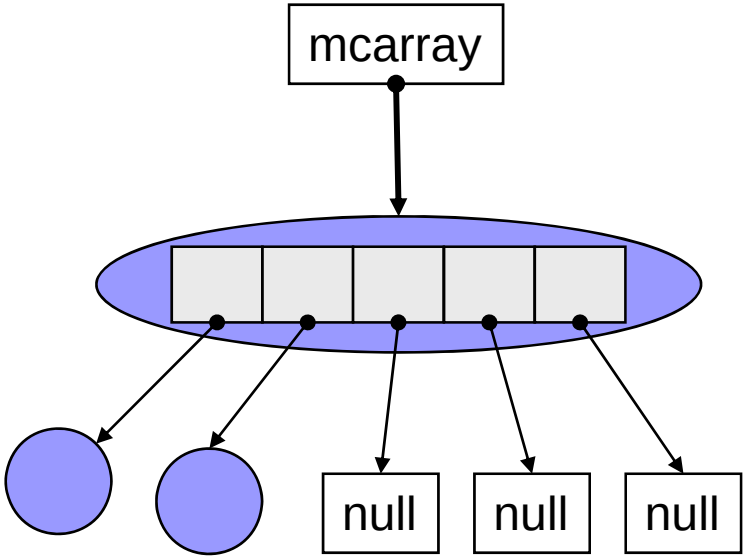
- Arrays in Java are objects
  - As with all objects they are created on the heap
- This has many advantages
  - Arrays know their own length
    - Accessed via a final field e.g. ‘myarray.length’
  - Access is always bounds checked
    - An exception is thrown if the index supplied is invalid
    - Java continues the C tradition of counting from zero
  - Values are automatically initialized
    - To zero, false or null

Arrays in Java

```
int [ ] intarray = new int[5];
intarray[0] = 10;
intarray[1] = 20;
```




```
MyClass [ ] mcarray = new MyClass[5];
mcarray[0] = new MyClass();
mcarray[1] = new MyClass();
```

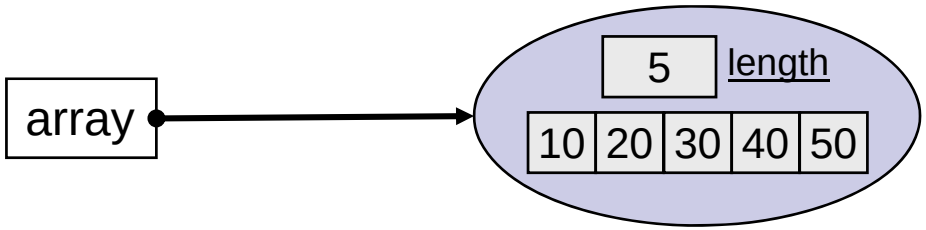


Arrays in Java

- Array classes cannot be used directly
  - They have illegal names, such as [I for int[]
- There is no way to resize an array
  - However the ‘System.arraycopy’ method provides a convenient way of efficiently copying values between arrays
- Note that a reference can point to an array of any size
  - E.g. a reference of type ‘int [ ]’ can point to any integer array
- There is no way to make the contents of an array final
  - When an array reference is passed into a method the code within can change the contents of elements within the array

Arrays In Java

```
public void printArray(int[] array) {
      //Iterate over the array
      for(int i=0; i<array.length; i++) {
            System.out.println("Value " + i + "is " + array[i]);
      }
      //This would cause an 'ArrayIndexOutOfBoundsException'
      //  to be thrown (arrays are zero indexed)
      int i = array[array.length];
}
```



Arrays of Objects in Depth

- It is impossible to have an array of objects
  - In Java one object never contains another
  - Instead objects reference each other
- An array of objects is really an array of references
  - Each box within the array contains a reference to an object
    - As opposed to a nested object
  - By default the boxes contain the null reference
  - The term ‘array of objects’ is still used as a shortcut
- The object referred to may itself be an array
  - This is how Java implements multidimensional arrays

Arrays of Literals

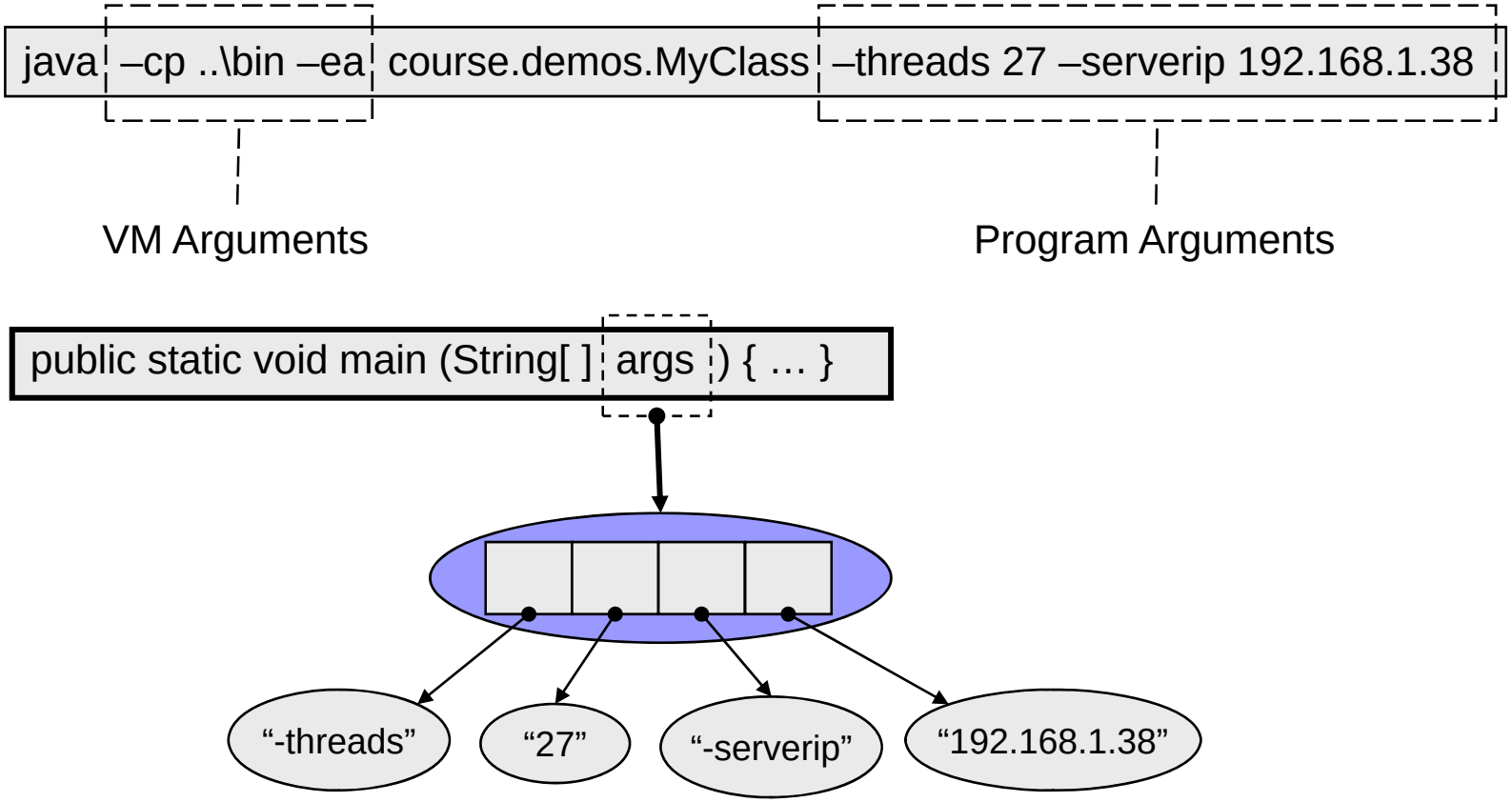
- Arrays can be created out of literals
  - The new operator can be omitted in definitions
  - Specifying the size of the array is a compiler error

```
//Create arrays out of integer literals
int [] intarray1 = new int[]{10,20,30,40,50 };
int [] intarray2 = {100,200,300,400,500 };
//Compiler Error !!
int[] intarray3 = new int[3] { 23,34,45 };
//Create arrays out of String literals
String [] strarray1 = new String[] { “abc”,”def”,”ghi”,”jkl” };
String [] strarray2 = { “abc”,”def”,”ghi”,”jkl” };
//Create arrays out of objects
Object [] objarray1 = { new Object(), new String(“xxx”), new MyClass() } ;
Object [] objarray2 = new Object[]{ new Object(), new String(“xxx”), new MyClass() } ;
```

Command Line Arguments

- You can specify two kinds of arguments
  - Arguments to the Virtual Machine are passed between the call to the ‘java’ executable and the class name
  - Arguments to the program itself are passed after the class name, separated by whitespace
- Program arguments are converted into a String array
  - Which the VM then passes into the main method
- Unlike C/C++ the first argument is not the class name
  - You always know this when writing the class
  - Or you can determine it at runtime via reflection

Command Line Arguments



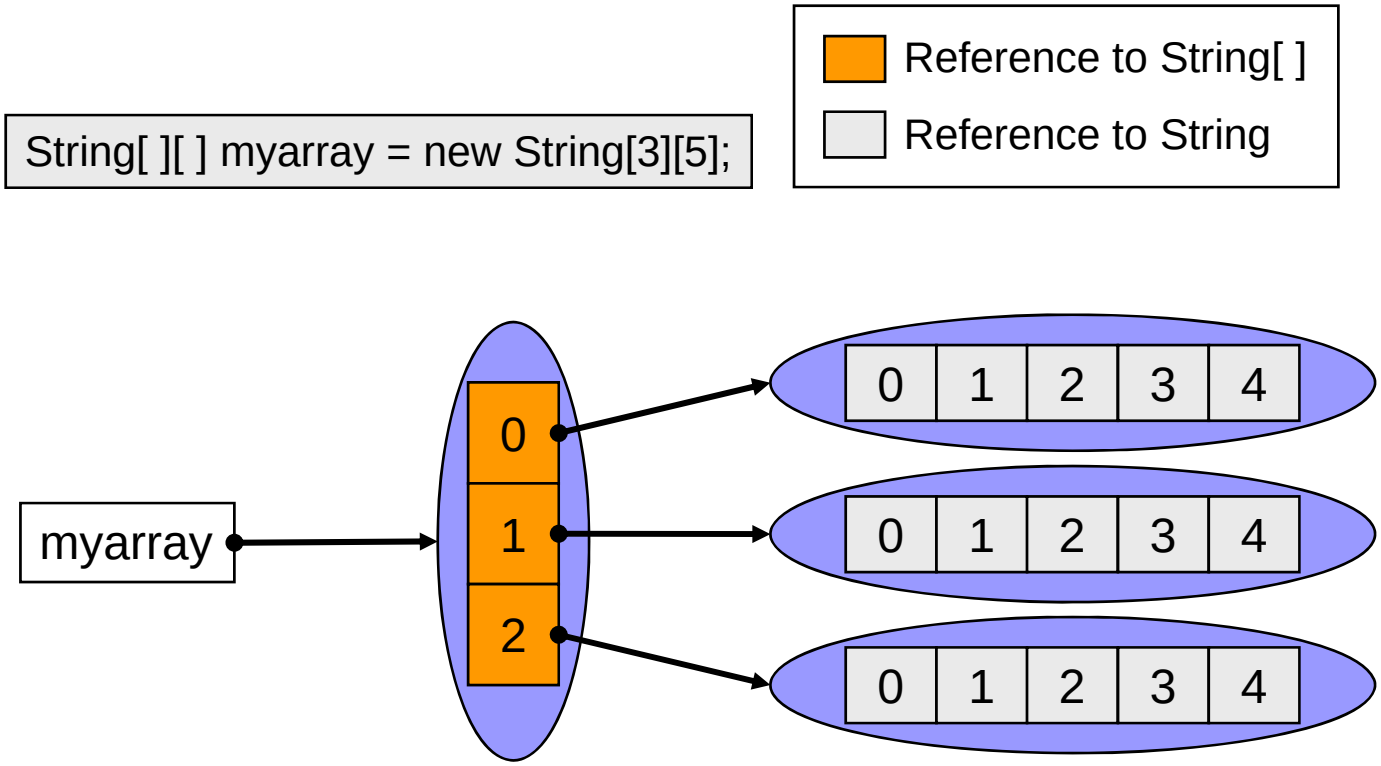
The For-Each Loop

- Iteration in Java is a little untidy
  - Arrays have a ‘length’ field, strings have a ‘length’ method and collections have both iterators and a ‘size’ method
  - The code to loop over these is tedious boilerplate
- Java 1.5 introduces a new version of the ‘for’ loop
  - The syntax is ‘for(MyClass mc : someCollection)’
    - There is no new ‘foreach’ keyword
    - The colon is read as meaning ‘in’
  - The compiler generates the code to loop over the collection and set ‘mc’ to refer to the current item
    - There is no built-in way to obtain the current index

Multi Dimensional Arrays in Java

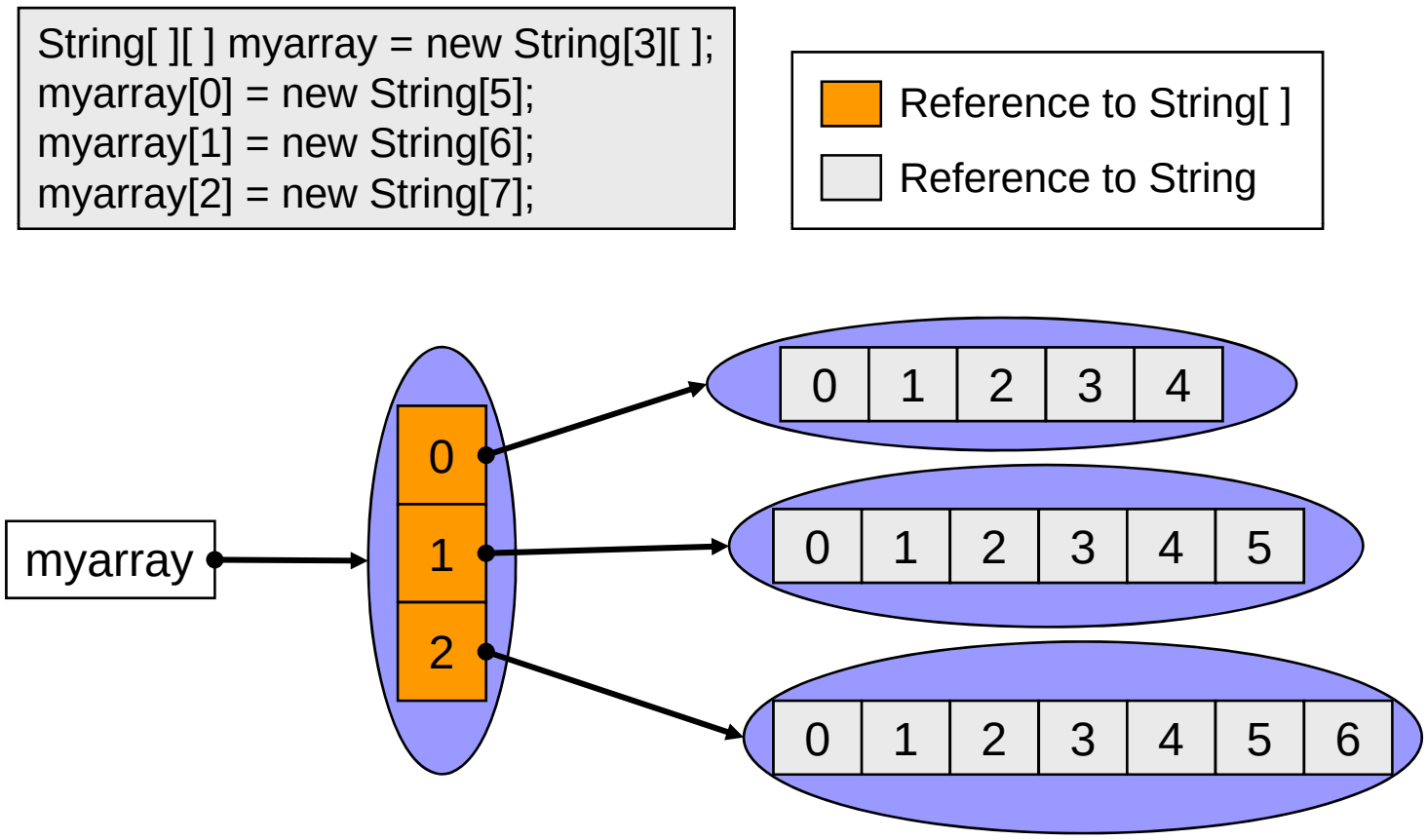
- Java supports multidimensional arrays
  - But not as a C/C++ programmer would expect
- They are implemented as arrays of arrays
  - A 2D array of Strings is an array object which contains references to normal array objects
    - These then contain references to strings
  - Similarly a 3D array would be an array object that contained references to 2D arrays
- Multidimensional arrays can be ragged
  - You can build the final dimension at runtime
  - Only the first dimension is mandatory

Two Dimensional Arrays



Two Dimensional Jagged Arrays

```
String[ ][ ] myarray = new String[3][ ];
myarray[0] = new String[5];
myarray[1] = new String[6];
myarray[2] = new String[7];
```

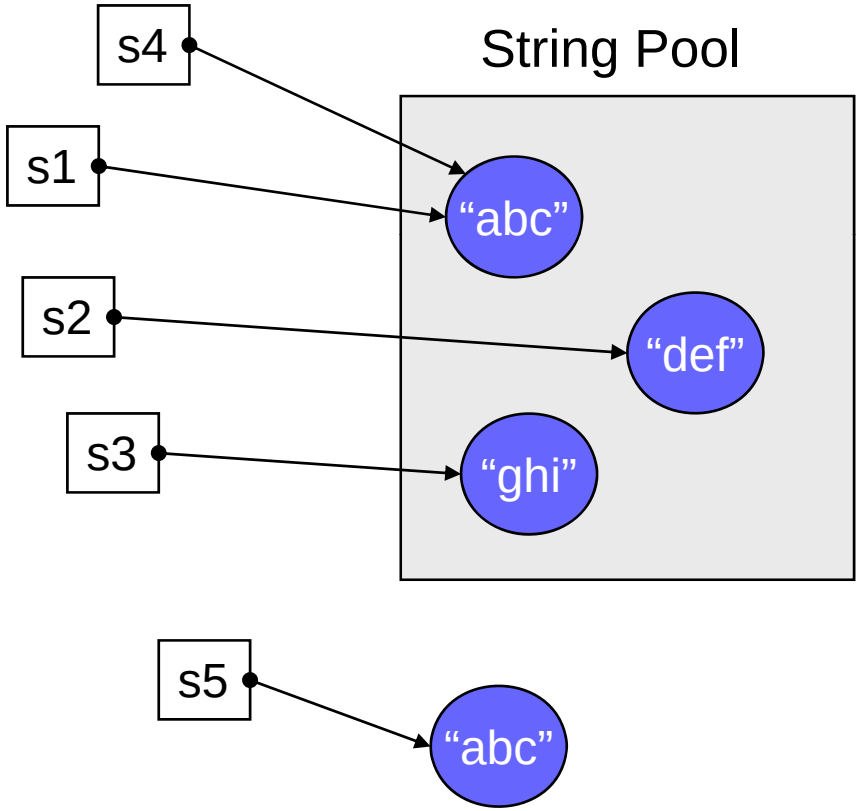


Strings

- String literals in Java are not arrays of char
  - The compiler wraps literals into String objects
  - Identical literals are represented by the same object
    - Expression “abc” == “abc” is always true
  - Using ‘new’ always creates a new object
    - Expression “abc” == new String(“abc”) is always false
- Strings objects are immutable
  - Content cannot be changed after initialization
  - Concatenation creates a new String object

String Literals

```
String s1 = “abc”;
String s2 = “def”;
String s3 = “ghi”;
String s4 = “abc”;
String s5 = new String(“abc”);
```



Conversions To/From Strings

- Each of the basic types has a wrapper class
  - Integer for int, Double for double etc…
- These wrapper classes have two functions
  - To allow a basic type to be passed as an object
  - To provide a location for static conversion functions
- Each wrapper class has a static ‘parseXXX’ method
  - Which extracts a basic type value from a String
  - These functions only accept number characters
- To convert in the other direction use ‘String.valueOf’
  - This method is overloaded for all the basic types
  - ‘+’ and ‘+=’ handle conversions automatically

Conversions To/From Strings

```
String doubleStr = "250.6";
String floatStr = "2.5";
String intStr = "500";
String booleanStr = "true";

double d = Double.parseDouble(doubleStr);
float f = Float.parseFloat(floatStr);
int i = Integer.parseInt(intStr);
Boolean b = new Boolean(booleanStr);

if(b.booleanValue()) {
      double result = d + f * i;
      System.out.println("Result is: " + result);
}
```

```
int i = 7;
float f = 8.1f;
double d = 9.2;
long l = 10;
boolean b = true;

String intStr = String.valueOf(i);
String floatStr = String.valueOf(f);
String doubleStr = String.valueOf(d);
String longStr = String.valueOf(l);
String boolStr = String.valueOf(b);
```