Bharathidasan University

Centre for Differently Abled Persons

Khajamalai Campus

Tiruchirappalli-620 023

Tamilnadu

Bachelor of Computer Applications

(For Students with Speech and Hearing Impairment)

Course: Java Programming

unit-3

CC BY

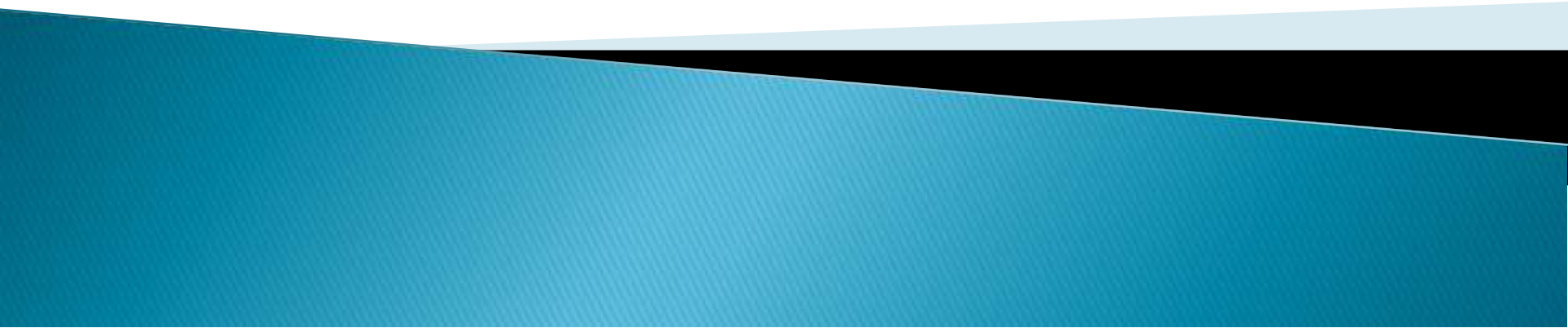
Compiled By

Dr.M.Prabavathy

(Assistant Professor)

Ms.V.Vijayalakshmi

Object-Oriented Programming



Different Programming Paradigms

- Functional/procedural programming:
  - program is a list of instructions to the computer
- Object-oriented programming
  - program is composed of a collection *objects that communicate with each other*

Main Concepts

- Object
- Class
- Inheritance
- Encapsulation

Objects

- identity – unique identification of an object
- attributes – data/state
- services – methods/operations
  - supported by the object
  - within objects responsibility to provide these services to other clients

Class

- “type”
- object is an **instance** of class
- class groups similar objects
  - same (structure of) attributes
  - same services
- object holds values of its class’s attributes

Inheritance

- Class hierarchy
- Generalization and Specialization
  - subclass inherits attributes and services from its superclass
  - subclass may add new attributes and services
  - subclass may reuse the code in the superclass
  - subclasses provide specialized behaviors (overriding and dynamic binding)
  - partially define and implement common behaviors (abstract)

Encapsulation

- Separation between internal state of the object and its external aspects

- How ?
  - control access to members of the class
  - **interface** “type”

What does it buy us ?

- Modularity
  - source code for an object can be written and maintained independently of the source code for other objects
  - easier maintainance and reuse
- Information hiding
  - other objects can ignore implementation details
  - security (object has control over its internal state)
- but
  - shared data need special design patterns (e.g., DB)
  - performance overhead


JAVA

Why Java ?

- Portable
- Easy to learn

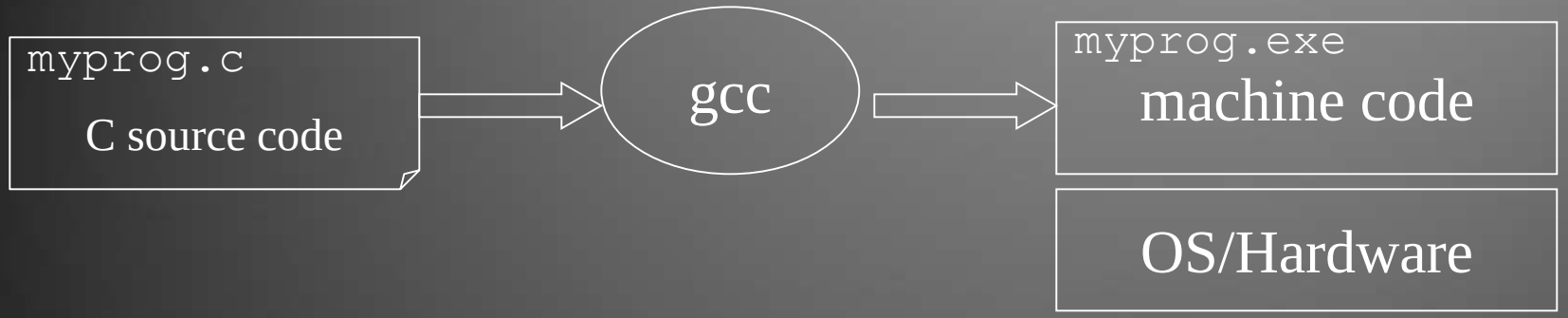
- [Designed to be used on the Internet]

JVM

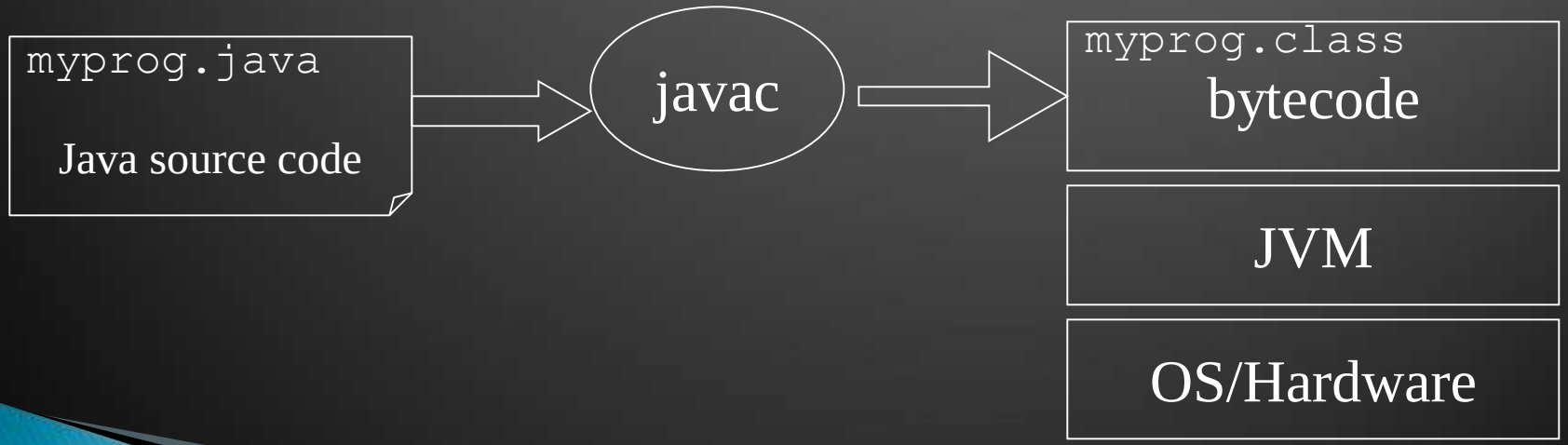
- JVM stands for
  Java Virtual Machine

- Unlike other languages, Java “executables” are executed on a CPU that does not exist.

Platform Dependent



Platform Independent



Primitive types

- int 4 bytes
- short 2 bytes
- long 8 bytes
- byte 1 byte
- float 4 bytes
- double 8 bytes
- char Unicode encoding (2 bytes)
- boolean {true,false}

Behaviors is exactly as in C++

Note:
Primitive type always begin with lower-case

Primitive types

- **Constants**

  37 integer

  37.2 float

  42F float

  0754 integer (octal)

  0xfe integer (hexadecimal)

Wrappers

Java provides Objects which wrap primitive types and supply methods.

Example:

```
Integer n = new Integer(“4”);
int m = n.intValue();
```

Hello World

Hello.java

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

C:\javac Hello.java *(compilation creates Hello.class)*

C:\java Hello *(Execution on the local JVM)*

More sophisticated

Default C'tor

Copy C'tor

```
class Kyle {
        private boolean kennyIsAlive_;
        public Kyle() { kennyIsAlive_ = true; }
        public Kyle(Kyle aKyle) {
                kennyIsAlive_ = aKyle.kennyIsAlive_;
        }
        public String theyKilledKenny() {
                if (kennyIsAlive_) {
                        kennyIsAlive_ = false;
                        return "You bastards !!!";
                } else {
                        return "?";
                }
        }
        public static void main(String[] args) {
                Kyle k = new Kyle();
                String s = k.theyKilledKenny();
                System.out.println("Kyle: " + s);
        }
}
```

Results

```
javac Kyle.java      ( to compile )

java Kyle            ( to execute )

Kyle: You bastards !!!
```

Arrays

- Array is an object
- Array size is fixed

```
Animal[] arr; // nothing yet …

arr = new Animal[4]; // only array of pointers

for(int i=0 ; i < arr.length ; i++) {
      arr[i] = new Animal();

// now we have a complete array
```

Arrays – Multidimensional

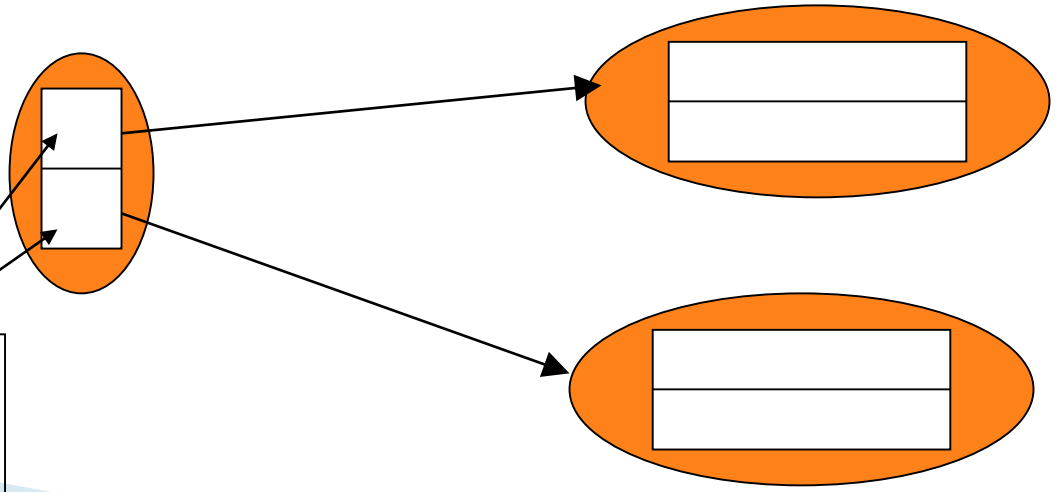
- In C++

```
Animal arr[2][2]
```

Is:

- In Java

```
Animal[][] arr=
    new Animal[2][2]
```



What is the type of the object here ?

Static

- **Member data** – Same data is used for all the instances (objects) of some Class.

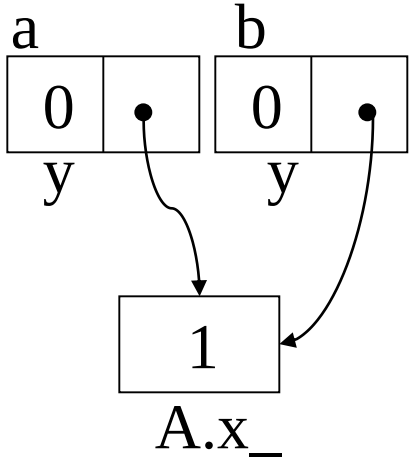
```
Class A {
    public int y = 0;
    public static int x_ = 1;
};

A a = new A();
A b = new A();
System.out.println(b.x_);
a.x_ = 5;
System.out.println(b.x_);
A.x_ = 10;
System.out.println(b.x_);
```

Assignment performed on the first access to the Class.
Only one instance of 'x' exists in memory

Output:

```
1
5
10
```



Static

- **Member function**
  - Static member function can access only static members
  - Static member function can be called without an instance.

```
Class TeaPot {
        private static int numOfTP = 0;
        private Color myColor_;
        public TeaPot(Color c) {
                myColor_ = c;
                numOfTP++;
        }
        public static int howManyTeaPots()
                { return numOfTP; }

        // error :
        public static Color getColor()
                { return myColor_; }
}
```

Static

Usage:

```
TeaPot tp1 = new TeaPot(Color.RED);

TeaPot tp2 = new TeaPot(Color.GREEN);

System.out.println("We have " +
     TeaPot.howManyTeaPots()+ "Tea Pots");
```

Static

- **Block**
  - Code that is executed in the first reference to the class.
  - Several static blocks can exist in the same class (Execution order is by the appearance order in the class definition).
  - Only static members can be accessed.

```
class RandomGenerator {
   private static int seed_;

   static {
        int t = System.getTime() % 100;
        seed_ = System.getTime();
        while(t-- > 0)
          seed_ = getNextNumber(seed_);
        }
   }
}
```

String is an Object

- Constant strings as in C, does not exist

- The function call `foo("Hello")` creates a String object, containing "Hello", and passes reference to it to `foo`.

- There is no point in writing :

```
String s = new String("Hello");
```

- The String object is a constant. It can't be changed using a reference to it.

Flow control

Basically, it is exactly like c/c++.

if/else

```
If(x==4) {
   // act1
} else {
   // act2
}
```

do/while

```
int i=5;
do {
   // act1
   i--;
} while(i!=0);
```

for

```
int j;
for(int i=0;i<=9;i++)
{
   j+=i;
}
```

switch

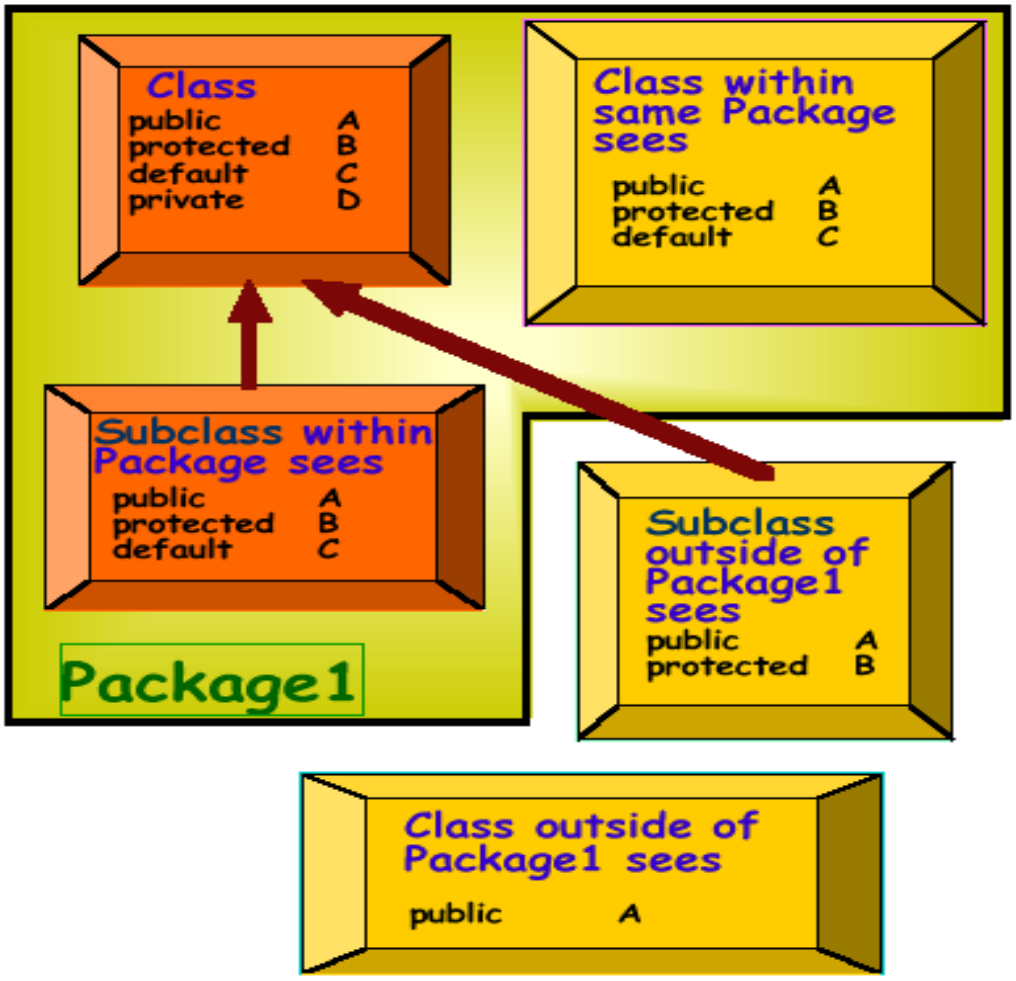
```
char
c=IN.getChar();
switch(c) {
   case 'a':
   case 'b':
      // act1
      break;
   default:
      // act2
}
```

Packages

- Java code has hierarchical structure.
- The environment variable CLASSPATH contains the directory names of the roots.
- Every Object belongs to a package (‘package’ keyword)
- Object full name contains the name full name of the package containing it.

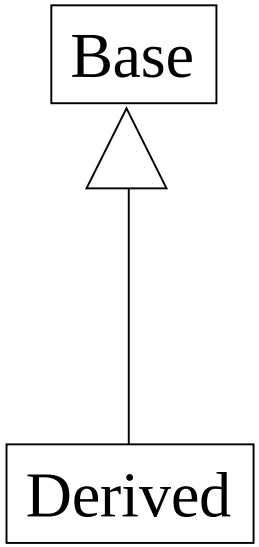
Access Control

- ***public*** member (function/data)
  - Can be called/modified from outside.
- ***protected***
  - Can be called/modified from derived classes
- ***private***
  - Can be called/modified only from the current class
- ***default (if no access modifier stated)***
  - Usually referred to as “Friendly”.
  - Can be called/modified/instantiated from the same package.



A summary of Java scoping visibility

Inheritance




```
class Base {
    Base(){}
    Base(int i) {}
    protected void foo() {…}
}

class Derived extends Base {
    Derived() {}
    protected void foo() {…}
    Derived(int i) {
      super(i);
       …
      super.foo();
    }
}
```

As opposed to C++, it is possible to inherit only from ONE class.

Pros avoids many potential problems and bugs.

Cons might cause code replication

Polymorphism

- Inheritance creates an “is a” relation:

For example, if B inherits from A, than we say that “B is also an A”.

Implications are:

- access rights (Java forbids reducing access rights) – derived class can receive all the messages that the base class can.
- behavior
- precondition and postcondition

Inheritance

- In Java, all methods are virtual :

```
class Base {
  void foo() {
    System.out.println("Base");
  }
}
class Derived extends Base {
  void foo() {
    System.out.println("Derived");
  }
}
public class Test {
  public static void main(String[] args) {
    Base b = new Derived();
    b.foo();  // Derived.foo() will be activated
  }
}
```

Inheritance

```
class classC extends classB {
   classC(int arg1, int arg2){
     this(arg1);
     System.out.println("In classC(int arg1, int arg2)");
   }
   classC(int arg1){
     super(arg1);
     System.out.println("In classC(int arg1)");
   }
}
class classB extends classA {
   classB(int arg1){
     super(arg1);
     System.out.println("In classB(int arg1)");
   }
   classB(){
     System.out.println("In classB()");
   }
}
```

Inheritance

```
class classA {
   classA(int arg1){
        System.out.println("In classA(int arg1)");
   }
   classA(){
     System.out.println("In classA()");
   }
}

class classB extends classA {
   classB(int arg1, int arg2){
     this(arg1);
     System.out.println("In classB(int arg1, int arg2)");
   }
   classB(int arg1){
     super(arg1);
     System.out.println("In classB(int arg1)");
   }

  class B() {
    System.out.println("In classB()");
  }
}
```

Abstract

- ***abstract*** member function, means that the function does not have an implementation.
- ***abstract*** class, is class that can not be instantiated.

```
AbstractTest.java:6: class AbstractTest is an abstract class.
It can't be instantiated.
        new AbstractTest();
        ^
1 error
```

NOTE:
An abstract class is not required to have an abstract method in it.
But any class that has an abstract method in it or that does
not provide an implementation for any abstract methods declared
in its superclasses must be declared as an abstract class.

Abstract – Example

```
package java.lang;
public abstract class Shape {
       public abstract void draw();
       public void move(int x, int y) {
          setColor(BackGroundColor);
          draw();
          setCenter(x,y);
          setColor(ForeGroundColor);
          draw();
        }
}
```

```
package java.lang;
public class Circle extends Shape {
       public void draw() {
         // draw the circle ...
       }
}
```

Interface

Interfaces are useful for the following:

- Capturing similarities among unrelated classes without artificially forcing a class relationship.
- Declaring methods that one or more classes are expected to implement.
- Revealing an object's programming interface without revealing its class.

Interface

- abstract “class”
- Helps defining a “usage contract” between classes
- All methods are public
- Java’s compensation for removing the multiple inheritance. You can “inherit” as many interfaces as you want.

Interface

```
interface IChef {
   void cook(Food food);
}
```

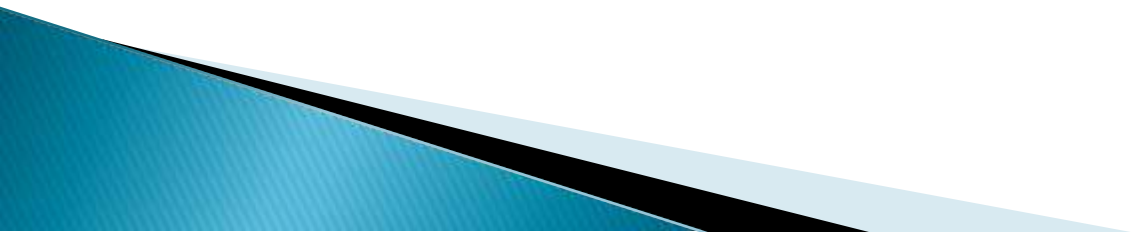
```
interface BabyKicker {
   void kickTheBaby(Baby);
}
```

```
interface SouthParkCharacter {
   void curse();
}
```

```
class Chef implements IChef, SouthParkCharacter {
       // overridden methods MUST be public
       // can you tell why ?
       public void curse() { … }
       public void cook(Food f) { … }
}
```

When to use an interface ?

Perfect tool for encapsulating the classes inner structure. Only the interface will be exposed



Collections

- Collection/container
  - object that groups multiple elements
  - used to store, retrieve, manipulate, communicate aggregate data
- Iterator – object used for traversing a collection and selectively remove elements

- Generics – implementation is parametric in the type of elements

Java Collection Framework

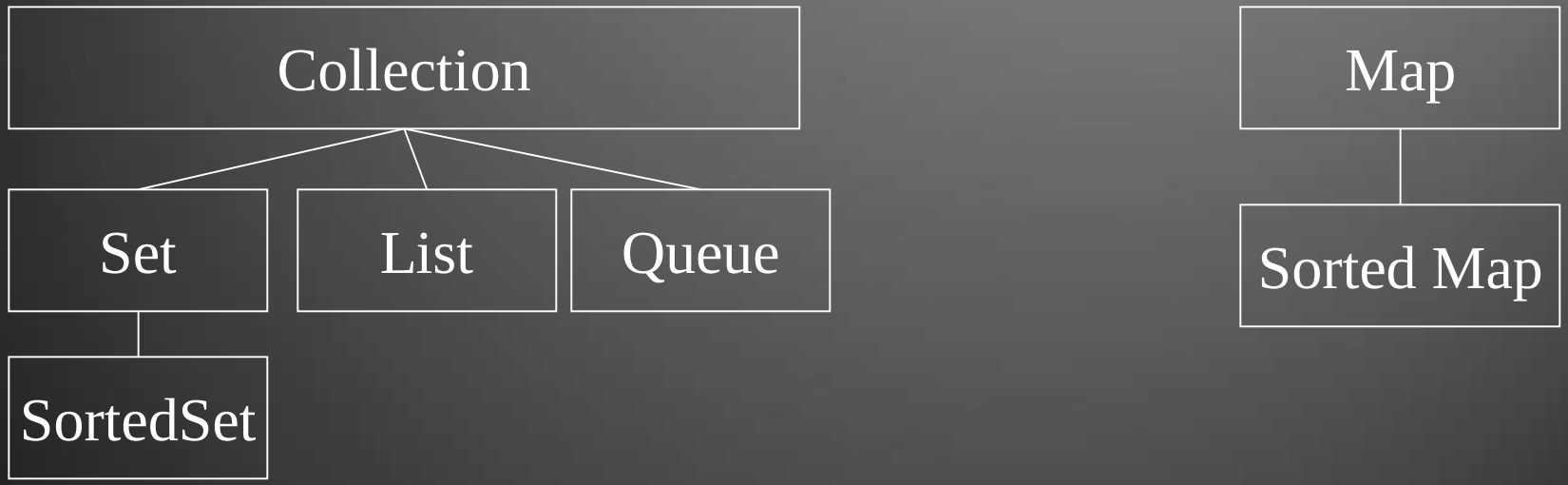
- Goal: Implement reusable data-structures and functionality

- Collection interfaces – manipulate collections independently of representation details
- Collection implementations – reusable data structures

```
List<String> list = new ArrayList<String>(c);
```

- Algorithms – reusable functionality
  - computations on objects that implement collection interfaces
  - e.g., searching, sorting
  - polymorphic: the same method can be used on many different implementations of the appropriate collection interface

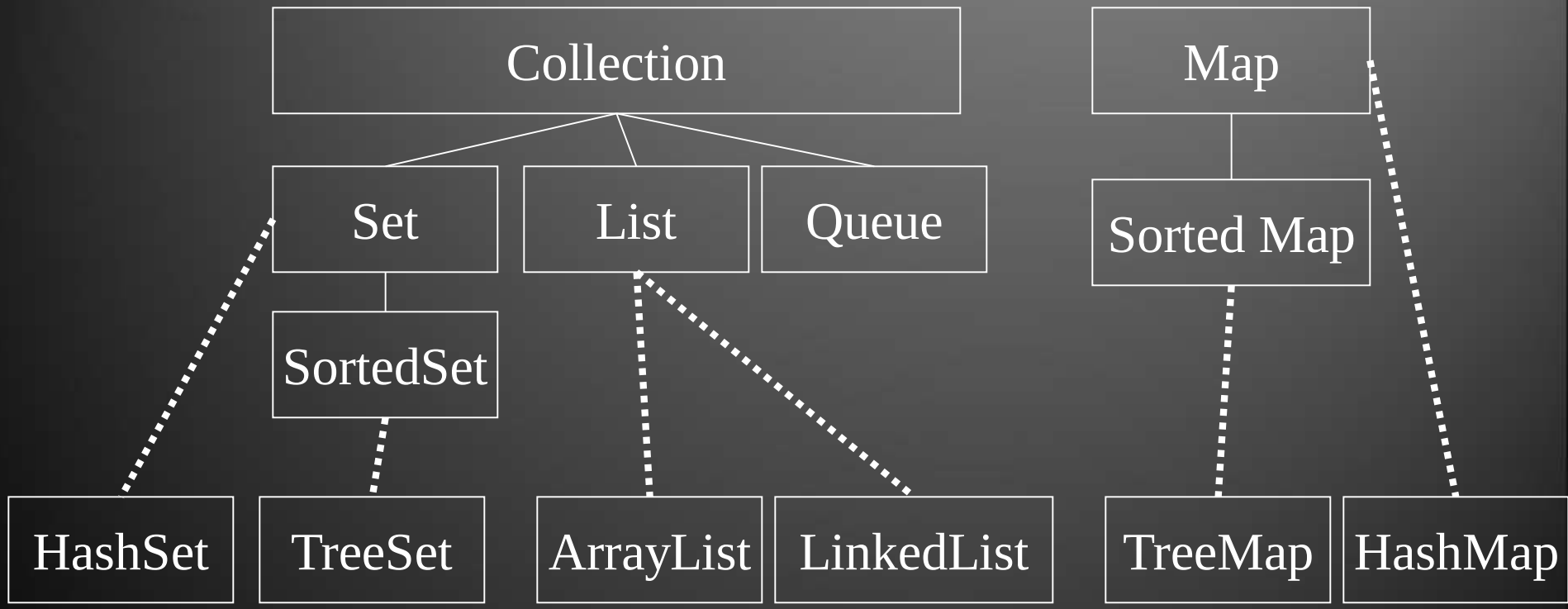
Collection Interfaces



Collection Interface

- Basic Operations
  - int size();
  - boolean isEmpty();
  - boolean contains(Object element);
  - boolean add(E element);
  - boolean remove(Object element);
  - Iterator iterator();
- Bulk Operations
  - boolean containsAll(Collection<?> c);
  - boolean addAll(Collection<? extends E> c);
  - boolean removeAll(Collection<?> c);
  - boolean retainAll(Collection<?> c);
  - void clear();
- Array Operations
  - Object[] toArray(); <T> T[] toArray(T[] a); }

General Purpose Implementations




```
List<String> list1 = new ArrayList<String>(c);
List<String> list2 = new LinkedList<String>(c);
```

final

- ***final* member data**
  Constant member
- ***final* member function**
  The method can't be overridden.
- ***final* class**
  'Base' is final, thus it can't be extended

(String class is final)

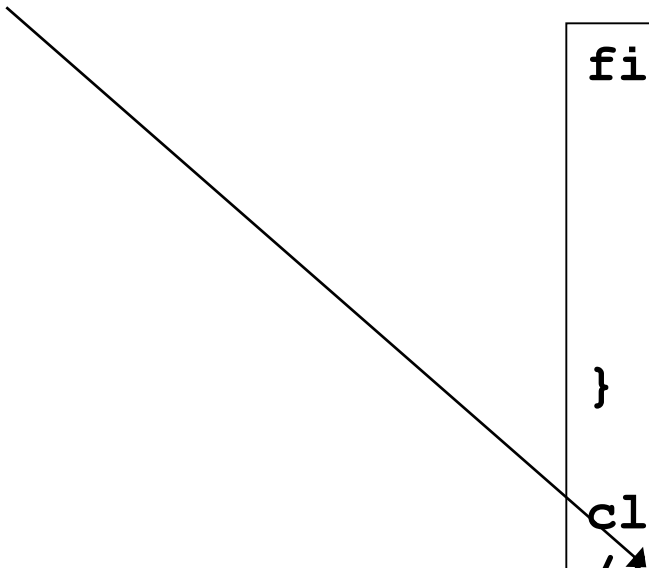
```
final class Base {
  final int i=5;
  final void foo() {
     i=10;
//what will the compiler say
about this?
   }
}

class Derived extends Base {
// Error
  // another foo ...
  void foo() {

  }
}
```

final

```
Derived.java:6: Can't subclass final classes: class Base
class class Derived extends Base {
      ^
1 error
```



```
final class Base {
  final int i=5;
  final void foo() {
    i=10;
  }
}

class Derived extends Base {
// Error
  // another foo ...
  void foo() {

  }
}
```

IO – Introduction

- Definition
  - **Stream** is a flow of data
    - characters read from a file
    - bytes written to the network
    - …
- Philosophy
  - All streams in the world are basically the same.
  - Streams can be divided (as the name "IO" suggests) to **Input** and **Output** streams.
- Implementation
  - Incoming flow of data (characters) implements "Reader" (InputStream for bytes)
  - Outgoing flow of data (characters) implements "Writer" (OutputStream for bytes –eg. Images, sounds etc.)

Exception – What is it and why do I care?

Definition: An *exception* is an event that occurs during the execution of a program that disrupts the normal flow of instructions.

- Exception is an Object
- Exception class must be descendent of Throwable.

Exception – What is it and why do I care?

By using exceptions to manage errors, Java programs have the following advantages over traditional error management techniques:

1: Separating Error Handling Code from "Regular" Code
2: Propagating Errors Up the Call Stack
3: Grouping Error Types and Error Differentiation

1: Separating Error Handling Code from "Regular" Code

```
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

1: Separating Error Handling Code from "Regular" Code

```
errorCodeType readFile {
    initialize errorCode = 0;
    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
        close the file;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
}
```

1: Separating Error Handling Code from "Regular" Code

```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
        doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

2: Propagating Errors Up the Call Stack

```
method1 {
     try {
          call method2;
     } catch (exception) {
          doErrorProcessing;
     }
}
method2 throws exception {
     call method3;
}
method3 throws exception {
     call readFile;
}
```

Comments are almost like C++

- **The java doc program generates HTML API documentation from the "java doc" style comments in your code.**

```
/* This kind of comment can span multiple lines */
// This kind is to the end of the line
/**
  * This kind of comment is a special
  * 'javadoc' style comment
  */
```

An example of a class

```
class Person {
    String name;
    int age;
```

Variable

```
    void birthday ( ) {
```

Method

```
        age++;
        System.out.println (name + ' is now ' +
age);
    }
}
```

Scoping

- As in C/C++, scope is determined by the placement of curly braces {}.
- A variable defined within a scope is available only to the end of that scope.

```
{  int x = 12;
   /* only x available */
      { int q = 96;
         /* both x and q available */
      }
      /* only x available */
      /* q “out of scope” */
    }
```

This is ok in C/C++ but not in Java.

```
{   int x = 12;
    { int x = 96; /* illegal */
     }
 }
```

An array is an object

- `Person mary = new Person ( );`
- `int myArray[ ] = new int[5];`
- `int myArray[ ] = {1, 4, 9, 16, 25};`
- `String languages [ ] = {"Prolog", "Java"};`
- Since arrays are objects they are allocated dynamically
- Arrays, like all objects, are subject to garbage collection when no more references remain
  - so fewer memory leaks
  - Java doesn't have pointers!

Scope of Objects

- Java objects don't have the same lifetimes as primitives.
- When you create a Java object using **new**, it hangs around past the end of the scope.
- Here, the scope of name s is delimited by the {}s but the String object hangs around until GC'd

```
{
    String s = new String("a string");
} /* end of scope */
```

Methods, arguments and return values

- Java methods are like C/C++ functions. General case:
  - *returnType methodName* (*arg1*, *arg2*, … *argN*) {
  - *methodBody*
  - }
- The return keyword exits a method optionally with a value

```
int storage(String s) {return s.length() * 2;}
boolean flag() { return true; }
float naturalLogBase() { return 2.718f; }
void nothing() { return; }
void nothing2() {}
```

The static keyword

- Java methods and variables can be declared static
- These exist **independent of any object**
- This means that a Class's
  - static methods can be called even if no objects of that class have been created and
  - static data is "shared" by all instances (i.e., one rvalue per class instead of one per instance

```
class StaticTest {static int i = 47;}
StaticTest st1 = new StaticTest();
StaticTest st2 = new StaticTest();
// st1.i == st2.I == 47
StaticTest.i++;          // or st1.I++ or st2.I++
// st1.i == st2.I == 48
```

Array Operations

- Subscripts always start at 0 as in C
- Subscript checking is done automatically
- Certain operations are defined on arrays of objects, as for other classes
  - e.g. myArray.length == 5

Echo.java

```
C:\UMBC\331\java>type echo.java
//  This is the Echo example from the Sun
tutorial
class echo {
  public static void main(String args[]) {
    for (int i=0; i < args.length; i++) {
      System.out.println( args[i] );
    }
  }
}

C:\UMBC\331\java>javac echo.java

C:\UMBC\331\java>java echo this is pretty
silly
this
is
pretty
silly

C:\UMBC\331\java>
```

Factorial Example

```
/**
 * This program computes the factorial of a number
 */
public class Factorial {                        // Define a class
  public static void main(String[] args) {      // The program starts here
    int input = Integer.parseInt(args[0]);      // Get the user's input
    double result = factorial(input);           // Compute the factorial
    System.out.println(result);                 // Print out the result
  }                                             // The main() method ends here

  public static double factorial(int x) {       // This method computes x!
    if (x < 0)                                  // Check for bad input
      return 0.0;                               //   if bad, return 0
    double fact = 1.0;                          // Begin with an initial value
    while(x > 1) {                              // Loop until x equals 1
      fact = fact * x;                          //   multiply by x each time
      x = x - 1;                                //   and then decrement x
    }                                           // Jump back to the star of
loop
    return fact;                                // Return the result
  }                                             // factorial() ends here
}                                               // The class ends here
```

JAVA Classes

- The *class* is the fundamental concept in JAVA (and other OOPLs)
- A class describes some data object(s), and the operations (or methods) that can be applied to those objects
- Every object and method in Java belongs to a class
- Classes have data (fields) and code (methods) and classes (member classes or inner classes)
- Static methods and fields belong to the class itself
- Others belong to instances

Constructors

- Classes should define one or more methods to create or construct instances of the class
- Their name is the same as the class name
  - note deviation from convention that methods begin with lower case
- Constructors are differentiated by the number and types of their arguments
  - An example of overloading
- If you don’t define a constructor, a default one will be created.
- Constructors automatically invoke the zero argument constructor of their superclass when they begin (note that this yields a recursive process!)

Constructor example

```
public class Circle {
    public static final double PI = 3.14159;  // A
constant
    public double r;   // instance field holds circle's
radius

    // The constructor method: initialize the radius field
    public Circle(double r) { this.r = r; }

    // Constructor to use if no arguments
    public Circle() { r = 1.0; }
    // better: public Circle() { this(1.0); }

    // The instance methods: compute values based on
radius
    public double circumference() { return 2 * PI * r; }
    public double area() { return PI * r*r; }
}
```

Overloading, overwriting, and shadowing

- **Overloading** occurs when Java can distinguish two procedures with the same name by examining the number or types of their parameters.
- **Shadowing** or **overriding** occurs when two procedures with the same signature (name, the same number of parameters, and the same parameter types) are defined in different classes, one of which is a superclass of the other.

On designing class hierarchies

- Programs should obey the *explicit-representation principle*, with classes included to reflect natural categories.
- Programs should obey the *no-duplication principle*, with instance methods situated among class definitions to facilitate sharing.
- Programs should obey the *look-it-up principle*, with class definitions including instance variables for stable, frequently requested information.
- Programs should obey the *need-to-know principle*, with public interfaces designed to restrict instance-variable and instance-method access, thus facilitating the improvement and maintenance of nonpublic program elements.
- If you find yourself using the phrase *an X is a Y* when describing the relation between two classes, then the X class is a subclass of the Y class.
- If you find yourself using *X has a Y* when describing the relation between two classes, then instances of the Y class appear as parts of instances of the X class.

Data hiding and encapsulation

- Data-hiding or encapsulation is an important part of the OO paradigm.
- Classes should carefully control access to their data and methods in order to
  - Hide the irrelevant implementation-level details so they can be easily changed
  - Protect the class against accidental or malicious damage.
  - Keep the externally visible class simple and easy to document
- Java has a simple access control mechanism to help with encapsulation
  - Modifiers: public, protected, private, and package (default)

Abstract classes and methods

- Abstract vs. concrete classes
- Abstract classes can not be instantiated
  - public abstract class shape { }
- An abstract method is a method w/o a body
  - public abstract double area();
- (Only) Abstract classes can have abstract methods
- In fact, any class with an abstract method is automatically an abstract class

Syntax Notes

- No global variables
  - class variables and methods may be applied to any instance of an object
  - methods may have local (private?) variables
- No pointers
  - but complex data objects are “referenced”
- Other parts of Java are borrowed from PL/I, Modula, and other languages

Thank You