Object-Oriented Programming

Partly adapted with permission from Eran Toch, Technion

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

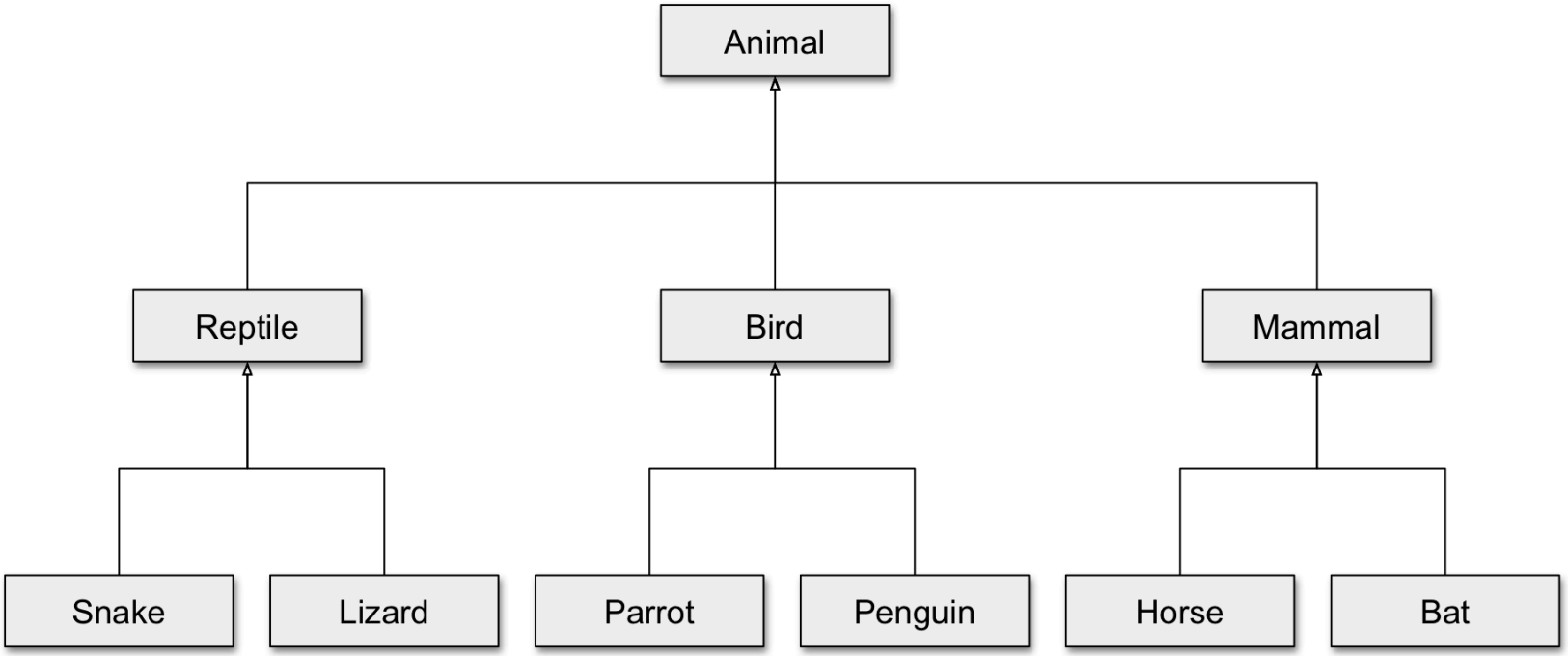
- identity – unique identification of an object
- attributes – data/state
- services – methods/operations
  - supported by the object
  - within objects responsibility to provide these services to other “clients” (objects)

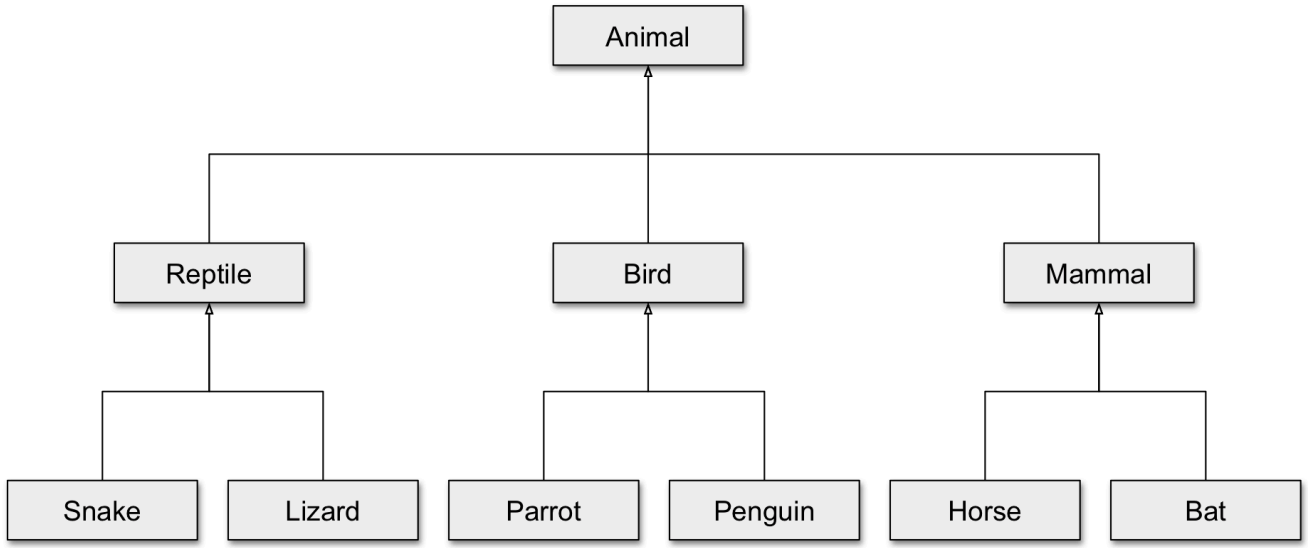
Class

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

Inheritance

- Class hierarchy

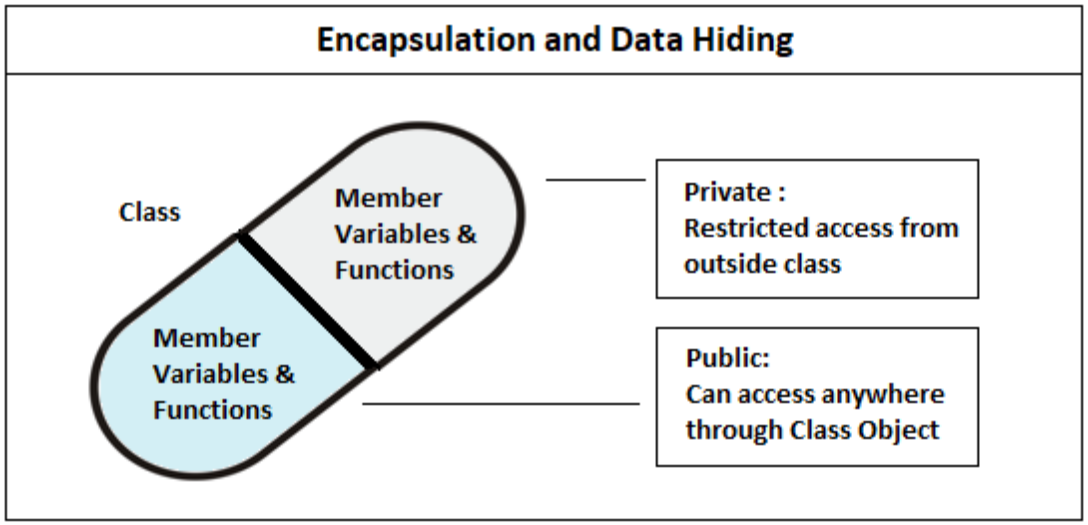




- Generalization and Specialization
  - Subclass *inherits attributes* and methods from its superclass
  - Subclass may *add new attributes and services*
  - Subclass may *reuse the code* in the superclass
  - Subclasses provide *specialized* behaviors (overriding and dynamic binding)
  - Superclass (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”



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


JAVA

Why Java ?

- Portable (*as Python*)
- Easy to learn (*less then Python*)
- Robust (*more than Python*)

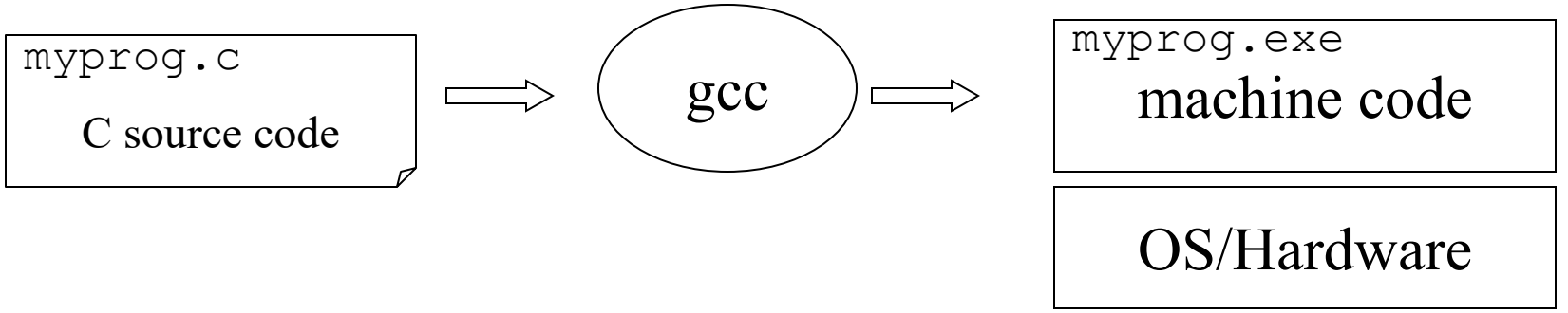
JVM

- JVM stands for

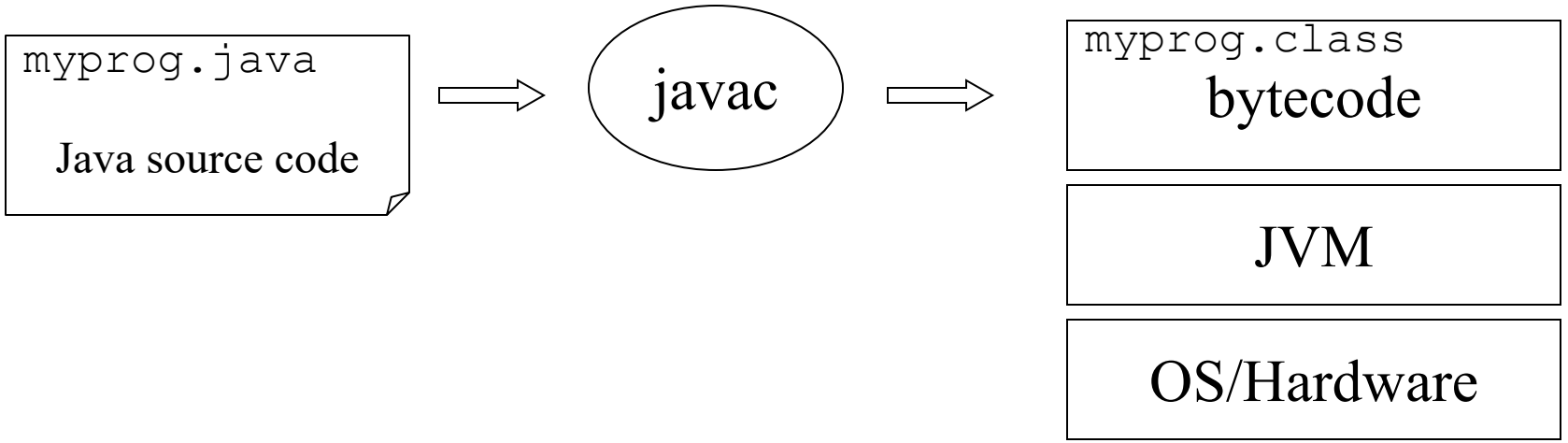
  Java **V**irtual **M**achine

- 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

Wrappers

Java provides Objects which wrap primitive types and supply methods.

Example:

Wrapper

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

Primitive

```
int p = 4;
```

Read more about Integer in JDK Documentation

Hello World

Hello.java

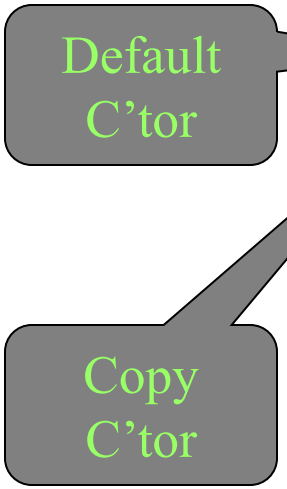
```
class Hello {
        public static void main(String[] args) {
                System.out.println("Hello World !!!");
        }
}
```

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

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


OH, MY GOD,
THEY KILLED KENNY !

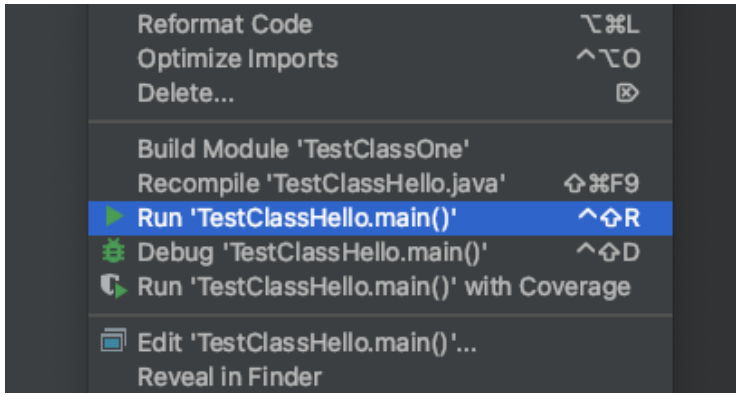
More sophisticated




```
class Kyle {
        private Boolean kennyIsAlive_;

        public Kyle() { kennyIsAlive_ = true; }
        public Kyle(Kyle aKyle) {
                kennyIsAlive_ = aKyle.kennyIsAlive_;
        }

        public String theyKilledKenny() {
                if (kennyIsAlive_) {
                        kennyIsAlive_ = false;
                        return "Oh Noooooo!!!";
                } 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 )
```



Result: **Kyle: Oh Noooooo!!!**

Advanced Example

```
import java.util.Scanner;

public class God {
	public static void main(String args[]) {

	Scanner scan = new Scanner(System.in);
	Kyle k = new Kyle();
	String text;

	while(true){
		text = scan.nextLine();

		if (text.equals("resurrection")) {
			k = new Kyle();
		}

		String s = k.theyKilledKenny();
		System.out.println("Kyle: " + s);
	}
  }
}
```

Arrays

```
if (kennyIsAlive_) {
        kennyIsAlive_ = false;
        return "Oh Noooooo!!!";
} else {
        return "?";
}
```

- Array is an **object**
- Array size is **fixed**

```
Kyle[] arr; // nothing yet …

arr = new Kyle [4]; // only array of pointers (empty)

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

String s = arr[0].theyKilledKenny();

s = arr[1].theyKilledKenny();
arr[0] = arr[2];

s = arr[0].theyKilledKenny();
s = arr[2].theyKilledKenny();
```

Arrays - Multidimensional

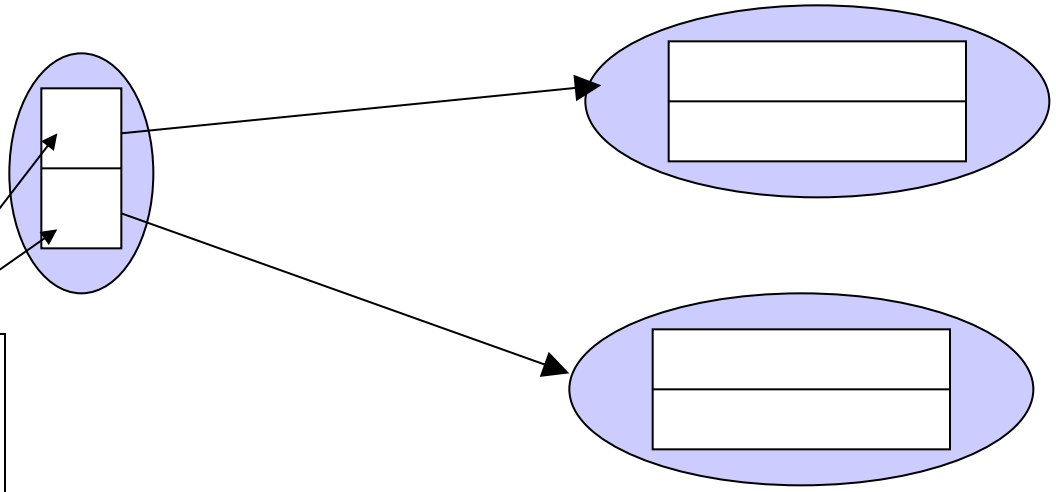
- In C++

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

Is:

- In Java

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



What is the type of the object here ?

Static - [1/4]

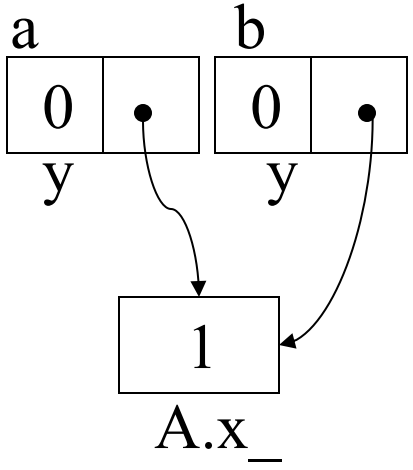
- **Member data** - Same data is used for all the instances (objects) of a given Class.

```
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 - [2/4]

- **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;
        }
        public static Color getColor() {
                return myColor_;
        }
}
```

Static - [2/4] cont.

Usage:

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

TeaPot tp2 = new TeaPot(Color.GREEN);

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

tp1 = new TeaPot(Color.RED);

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

Static - [3/4]

- **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 immutable.
- 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
}
```

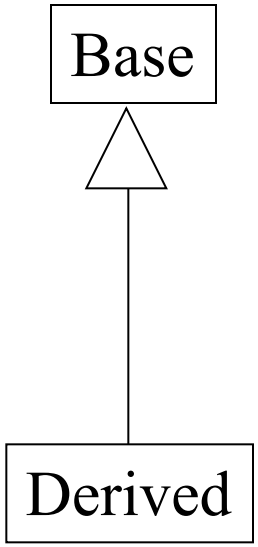
Packages

- 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 of the package containing it.

Access Control

- ***public*** member (function/data)
  - Can be called/modified from outside. Of?
- ***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.

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 (2)

- 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();
}
}
```

Inheritance (2)

- In Java, all methods are virtual :

```
class Base {
  void foo() {
    System.out.println("Base");
  }
}
class Derived extends Base {
   void foo() {
    System.out.println("Derived");
  }
}

class Derived1 extends Derived {
}

public class Test {
  public static void main(String[] args) {
    Derived b = new Derived1();
    b.foo();
       }
}
```

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.

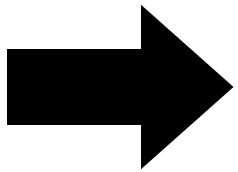
- Revealing an object's programming interface without revealing its class.

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.

- *The correct term is “to implement” an interface

Example



Interface

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

```
interface Kicker {
    void kick(Ball);
}
```

```
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

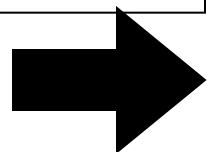
Abstract Class

- ***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***.

Example



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 ...
        }
}
```

Abstract

```
public class Test {
  public static void main(String[] args) {
    Shape b = new Circle();
    b.move(6,9);
  }
}
```

Abstract + Interface

```
Package sbm;
public abstract class Shape {
       private Color myColor = Color.WHITE;
       public abstract void draw();
       public void move(int x, int y) {
         setColor(BackGroundColor);
         draw();
         setCenter(x,y);
         setColor(ForeGroundColor);
         draw();
        }
       public Color getColor() {
           return myColor;
       }
}
```

```
public interface ToRoll {
  public void roll();
}
```

Abstract + Interface

```
Package sbm;
public class Circle extends Shape implements ToRoll{

        public void draw() {
          // draw the circle ...
        }
        public void roll() {
          // the circle roll...
        }

        public Color getColor() {
           if(super. getColor()== Color.WHITE
             || super. myColor == Color.WHITE )
                      return Color.GREY;

                return Color.BLACK;
        }
}
```

Abstract + Interface

```
package sbn;
public class Test {
  public static void main(String[] args) {
    Circle c = new Circle();
    ToRoll r = new Circle();
    Shape  s = new Circle();
    ToRoll r0 = (ToRoll) s;

    c.move(6,9);
    r.move(6,9);
    s.move(6,9);

    c.roll();
    r.roll();
    s.roll();

    Color clr = c.getColor();
    clr = r.getColor();
    clr = s.getColor();
    clr = s.myColor;
  }
}
```

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

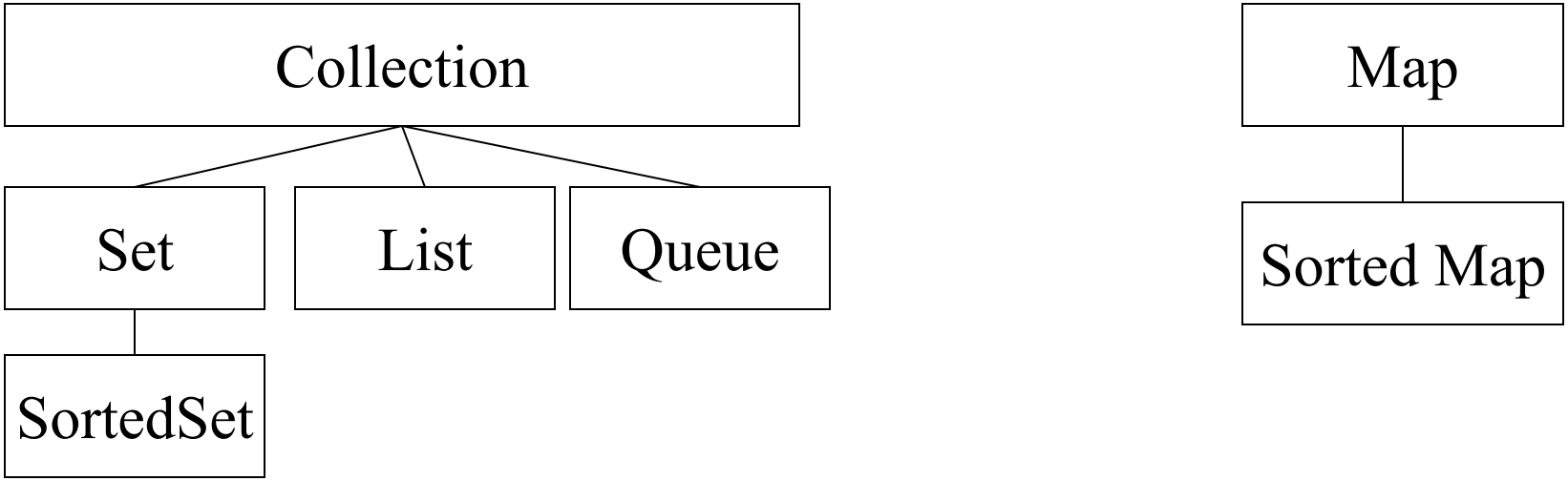
Java Collection Framework

- 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

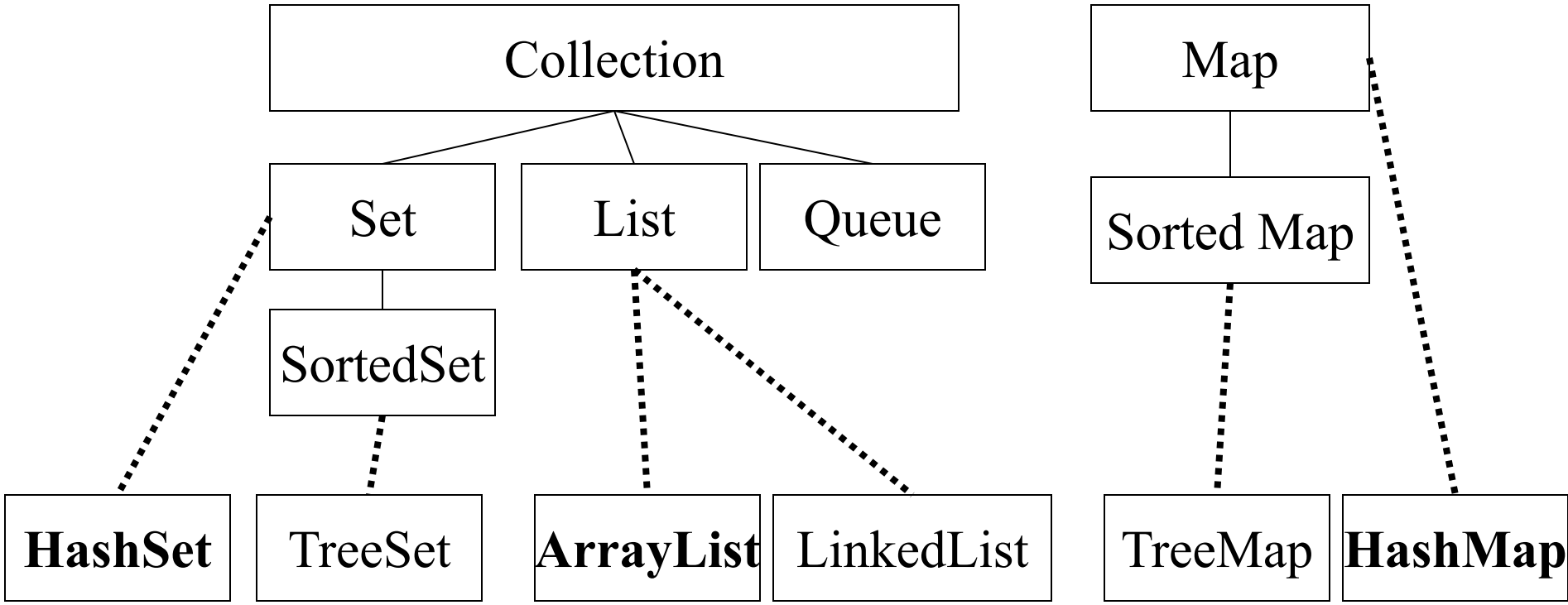
Collection Interfaces



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);
```

JDK ArrayList

```
import static java.lang.System.out;
import java.util.ArrayList;

class TestCollection {
    public void demonstrateJdkArrayListForDoubles() {
        ArrayList<Double> myCollection = new ArrayList<>();
        myCollection.add(15.5);
        myCollection.add(24.4);
        myCollection.add(36.3);
        myCollection.add(67.6);
        myCollection.add(10.0);
        out.println("JDK ArrayList<Double>:");
        myCollection.remove(36.3);
        out.println("\tDoubles List: " + myCollection.get(3));
    }

    public static void main(String[] args) {
        TestCollection test = new TestCollection();
        test.demonstrateJdkArrayListForDoubles();
    }
}
```

Trove ArrayList

```
class TroveTestCollection{
 public void demonstrateTroveArrayListForDoubles()
    {
       TDoubleArrayList doubles = new TDoubleArrayList();
       doubles.add(15.5);
       doubles.add(24.4);
       doubles.add(36.3);
       doubles.add(67.6);
       doubles.add(10.0);
       out.println("Trove TDoubleArrayList:”)
       out.println("\tDoubles List: " + doubles);
       out.println("\tMaximum double: " + doubles.max());
       out.println("\tMinimum double: " + doubles.min());
       out.println("\tSum of doubles: " + doubles.sum());
    }

public static void main(String[] args) {
        TroveTestCollectiontest = new TroveTestCollection ();
        test.demonstrateTroveArrayListForDoubles();
    }
}
```

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?(2)

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 (1)

```
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 (2)

```
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 (3)

```
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;
}
```

I/O - 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 "I/O" 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.)

Reading a File

```
FileInputStream fstream = new FileInputStream(file);

GZIPInputStream  gzStream = new GZIPInputStream(fstream);

InputStreamReader isr = new InputStreamReader(gzStream);

BufferedReader br = new BufferedReader(isr );

        String line;

        //Read File Line By Line
        while ((line = br.readLine()) != null) {
            // Print out the content on the console
            System.out.println(line);
        }

        //Close the input stream
        br.close();
```

Writing a File

```
FileOutputStream fstream = new FileOutputStream(file);
GZIPOutputStream  gzStream = new GZIPOutputStream(fstream);
OutputStreamWriter osw = new OutputStreamWriter(gzStream);
BufferedWriter bw = new BufferedWriter(osw);
PrintWriter pr = new PrintWriter(bw);

 for( int i=0 ; i<data.length ; i++ ) {
            for( int j=0 ; j<data[i].length ; j++ ) {
                pr.append( data[i][j]+"" );
                if( j == (data[i].length-1) )
                    pr.append("\n");
                else
                    pr.append(",");
     }
            pr.flush();
 }

  //Close the output stream
  pr.flush();
  pr.close();
```