

# Let us learn Core Java

**Presented By : Ramya Ramana**

**Email : [ramsubbu06@gmail.com](mailto:ramsubbu06@gmail.com)**

**Twitter : @ramsubbu06**

**GitHub : <https://github.com/ramsubbu06>**

# Why Java ?

- ▶ Portable
- ▶ Easy to learn
- ▶ Powerful Development tools
- ▶ Wonderful community support

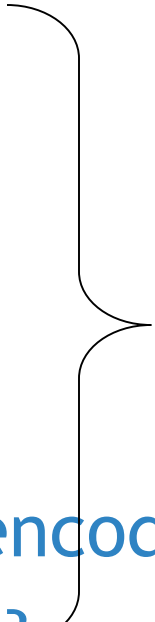
# Different Programming Paradigms

- ▶ Object-oriented programming
  - ▶ program is composed of a collection *objects that communicate with each other*

# Main Concepts of Core Java

- ▶ Object
- ▶ Class
- ▶ Inheritance
- ▶ Encapsulation
- ▶ Polymorphism

# Primitive types

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

**Note:**

*Primitive type  
always begin  
with lower-case*

# Wrappers

Java provides Objects which wrap primitive types and supply methods.

Wrappers classes are used to convert any data type into an object

Example:

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

[Read more about Integer in JDK Documentation](#)

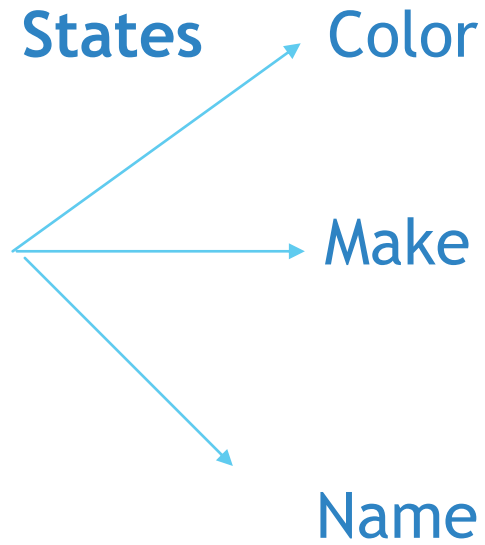
# Objects

- ▶ identity - unique identification of an object
- ▶ attributes - data/state
- ▶ services - methods/operations
  - ▶ supported by the object
  - ▶ It is 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



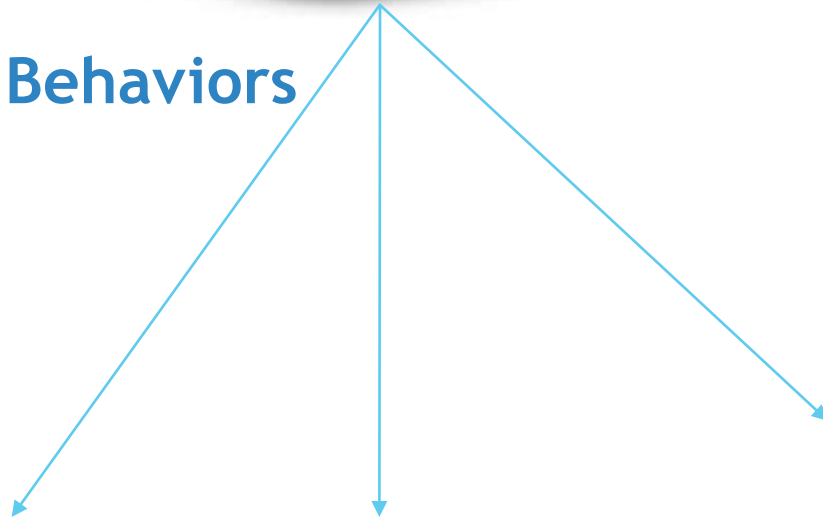


**Behaviors**

**Driving Backwards**

**Cruising**

**Driving Forward**



# Class in our example?

```
public class Car {}
```



```
Car c = new Car();
```

# Access Control

- ▶ ***public*** member (function/data)
  - ▶ Can be called from outside.
- ▶ ***protected***
  - ▶ Can be called from inherited classes.
- ▶ ***private***
  - ▶ Can be called only from the current class
- ▶ ***default ( if no access modifier stated )***
  - ▶ Can be called from the same package.

# Functions

- ▶ Function definitions must be inside classes. They are also called Methods.
- ▶ Function have return types and arguments.

## Example of a Function :

```
public void drivingForward(String s)
{
    System.out.println("whee..I am going forward");
}
```

# 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)

State

Behavior

Fuel

Purpose

Load Capacity

Automobile

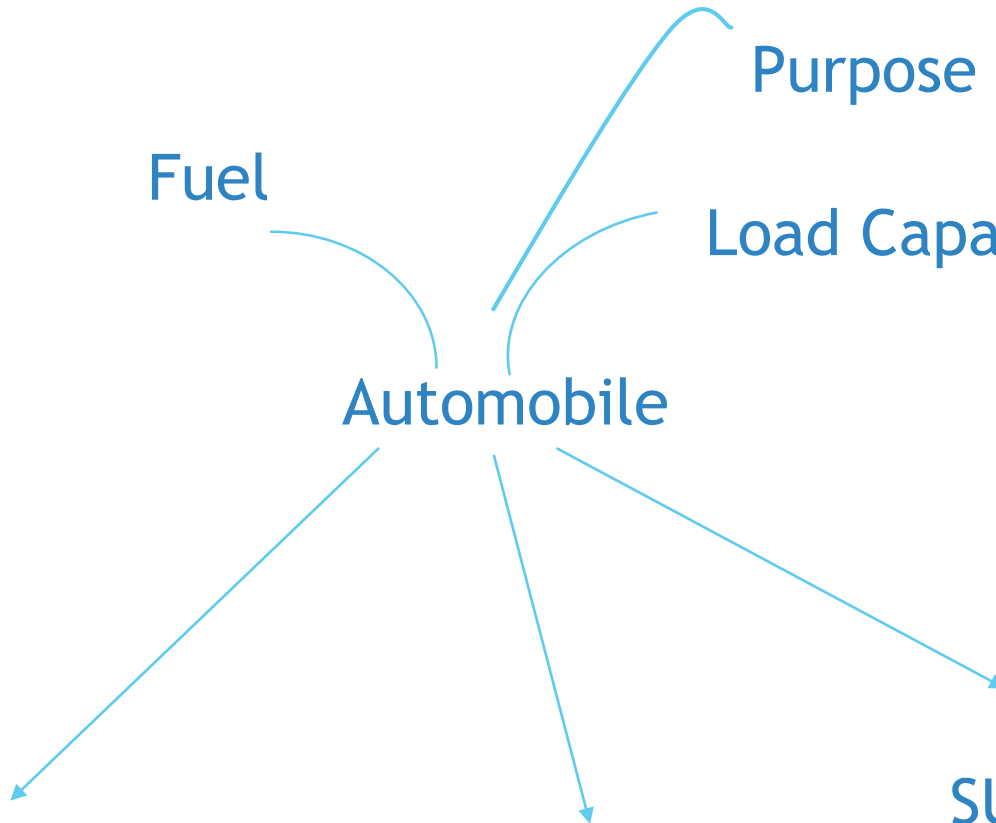
Subclasses

SUV

Full size Car

Truck

Example: InheritanceExample Class



# Encapsulation

- ▶ Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit.
- ▶ To achieve encapsulation in Java –
- ▶ Declare the variables of a class as private.
- ▶ Provide public setter and getter methods to modify and view the variables values.

Example: EncapsulationExample class

# 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 maintenance and reuse
- ▶ Information hiding
  - ▶ other objects can ignore implementation details
  - ▶ security (object has control over its internal state)
- ▶ but
  - ▶
  - ▶ performance overhead



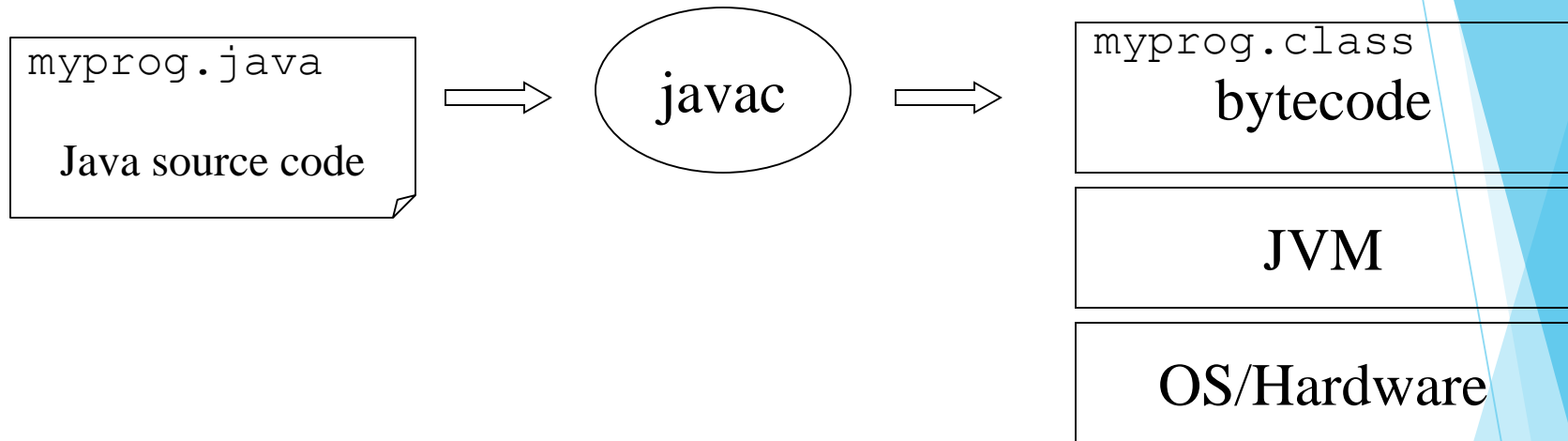
# JVM

- ▶ JVM stands for

## Java Virtual Machine

- ▶ Java was designed to allow application programs to be built that could be run on any platform without having to be rewritten or recompiled by the programmer for each separate platform.
- ▶ A Java virtual machine makes this possible because it is aware of the specific instruction lengths and other particularities of the platform.

# Platform Independent



# Hello World

Hello.java

```
public 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*)

# Arrays

- Array is an object
- Array size is fixed

```
double[] myList = new double[10];  
double temp=0.0;  
for(int i=0;i<myList.length;i++)  
{  
    myList[i]= temp+2.0001;  
    temp=myList[i];  
}
```

myList[0]	2.0001
myList[1]	4.0002
myList[2]	6.00030001
myList[3]	8.0004
myList[4]	10.0005
myList[5]	12.0006
myList[6]	14.0007
myList[7]	16.0008
myList[8]	18.0009
myList[9]	20.001

# Arrays - Multidimensional

- In Java

```
double dArray[][]={{1,1,1},  
                   {1,2,4},  
                   {1,3,9}};
```

A 3x3 grid representing a 2D array. The rows are indexed [0], [1], and [2] on the left, with a purple arrow labeled 'ROWS' pointing to the row indices. The columns are indexed [0], [1], and [2] at the bottom, with a purple arrow labeled 'COLUMNS' pointing to the column indices. The values in the grid are: Row 0: 1, 1, 1; Row 1: 1, 2, 4; Row 2: 1, 3, 9. All values are in green. A small 'I' is positioned to the left of the value '1' in the cell at row [1], column [0].

[0]	1	1	1
[1]	1	2	4
[2]	1	3	9
	[0]	[1]	[2]

# Static - [1/3]

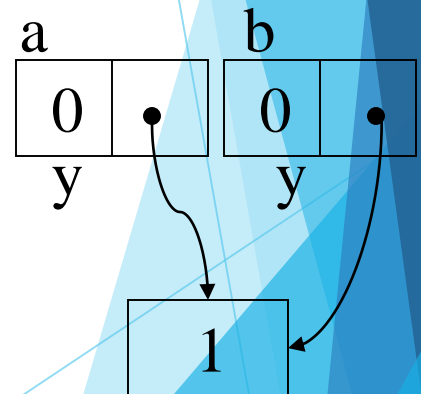
- Member data - Same data is used for all the instances (objects) of Class.

```
public class StaticApple {  
    public static int x = 1;  
    public static void main(String args[])  
    {  
        StaticApple a = new StaticApple();  
        StaticApple b = new StaticApple();  
        System.out.println("-----Static variable-----");  
        System.out.println(b.x);  
        a.x = 5;  
        System.out.println(b.x);  
        StaticApple.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



StaticApple.x

# Static - [2/3]

## ▶ Member function

- ▶ Static member function can access only static members
- ▶ Static member function can be called without an instance.
- ▶ Example : Fruit Class

# Static - [3/3]

## ▶ 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 s;  
  
    static {  
        System.out.println("I am static");  
    }  
}
```



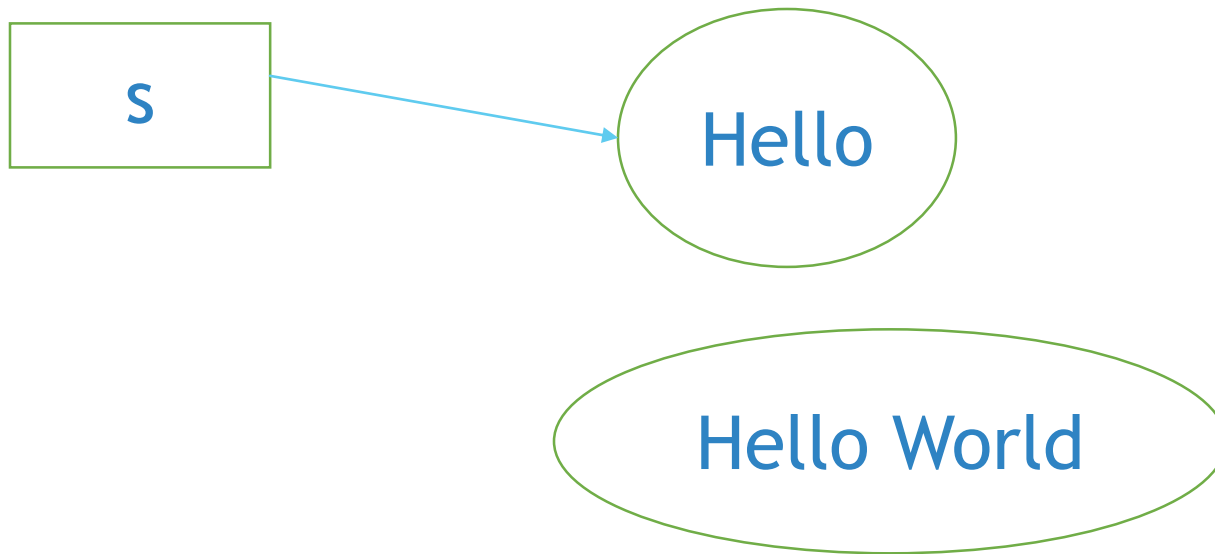
# String

- String are a sequence of characters. In Java strings are objects
- The most direct way to write a String

```
String greeting = "Hello World!";
```

- The String class is immutable, so that once it is created a String object cannot change. The string class has a number of methods that appear to modify the Strings. But what actually happens is methods create and return a new String that contains the result of the operation.

# String concatenation



```
String s = "Hello";  
System.out.println(s.concat(" World"));
```

# Flow control

## if/else

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

## do/while

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

## for

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

## switch

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

# Packages

- ▶ **Packages in Java** is a mechanism to encapsulate a group of classes, interfaces and sub packages.
- ▶ A hierarchical file system to manage source and class files. It is easy to organize class files into packages.
- ▶ Object full name contains the name full name of the package containing it.
- ▶ We use the “import” keyword to access packages.

# Constructors

- ▶ A *constructor* is a block of code similar to a method that's called when a class is instantiated. Here are the key differences between a constructor and a method:
- ▶ A constructor doesn't have a return type.
- ▶ The name of the constructor must be the same as the name of the class.
- ▶ A constructor is called automatically when a new instance of an object is created.

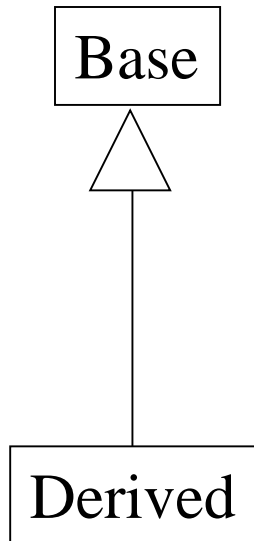
# What is Overriding

- ▶ If the subclass has the same method as its parent. Then it is called overriding.
- ▶ Rules:
  - ❖ Method must have same name as in the parent class
  - ❖ Method must have same parameter as in the parent class.
  - ❖ Must be IS-A relationship (inheritance).

# Overloading

- ▶ Method Overloading is a feature that allows a class to have two or more methods having same name, if their argument lists are different.
- ▶ **Argument lists could differ in -**
  1. Number of parameters.
  2. Data type of parameters.
  3. Sequence of Data type of parameters.

# 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();
    }
}
```

**Pros** avoids many potential problems and bugs.

**Cons** might cause code replication

Example :SuperConstructorExample



# Inheritance (2)

```
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  
    }  
}
```

# Polymorphism

- ▶ Polymorphism is the ability of an object to take many forms.
- ▶ Inheritance creates an “IS-A” relation:  
For example, if B inherits from A, then we say that “B is also an A”.

Any Object that passes a “IS-A” test is considered polymorphic.

```
public class Tree{}  
public class Oak extends Tree{}  
Oak IS-A tree  
Oak IS-A Object
```

# Abstract

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

```
new AbstractTest();  
      ^
```

```
1 err AbstractTest.java:6: class AbstractTest is an  
  abstract class.  
It can't be instantiated.
```

# Abstract

- ▶ An abstract class is not required to have an abstract method in it.
- ▶ An abstract method is declared without an implementation - without braces followed by a semi colon.
- ▶ Any class that has an abstract method in it must be marked abstract.
- ▶ Not provide an implementation for any abstract methods declared .
- ▶ A Subclass that implements the abstract class must implement all the abstract methods.

# Abstract - Example

```
public abstract class Shape {  
    public String BackGroundColor="black";  
    public String ForegroundColor="White";  
    public abstract void draw();  
    public void setColor(String color)  
    {System.out.println("The background color is  
    "+color);}  
    public void move(int x, int y) {  
        setColor(BackGroundColor);  
        draw();  
        setColor(ForegroundColor);  
        draw();}}}
```

```
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

- ▶ 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

# Interface

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

```
interface BabyHugger {  
    void Hug();  
}
```

```
interface Poet {  
    void writePoem();  
}
```

```
class Chef implements IChef, Poet{  
    public void WritePoem() { ... }  
    public void cook(Food f) { ... }  
}
```

\* access rights (Java forbids reducing of access rights)



# 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 Interface

## ▶ Basic Operations

- ▶ `int size();`
- ▶ `boolean isEmpty();`
- ▶ `boolean contains(Object element);`
- ▶ `boolean add(E element);`
- ▶ `boolean remove(Object element);`
- ▶ `Iterator iterator();`

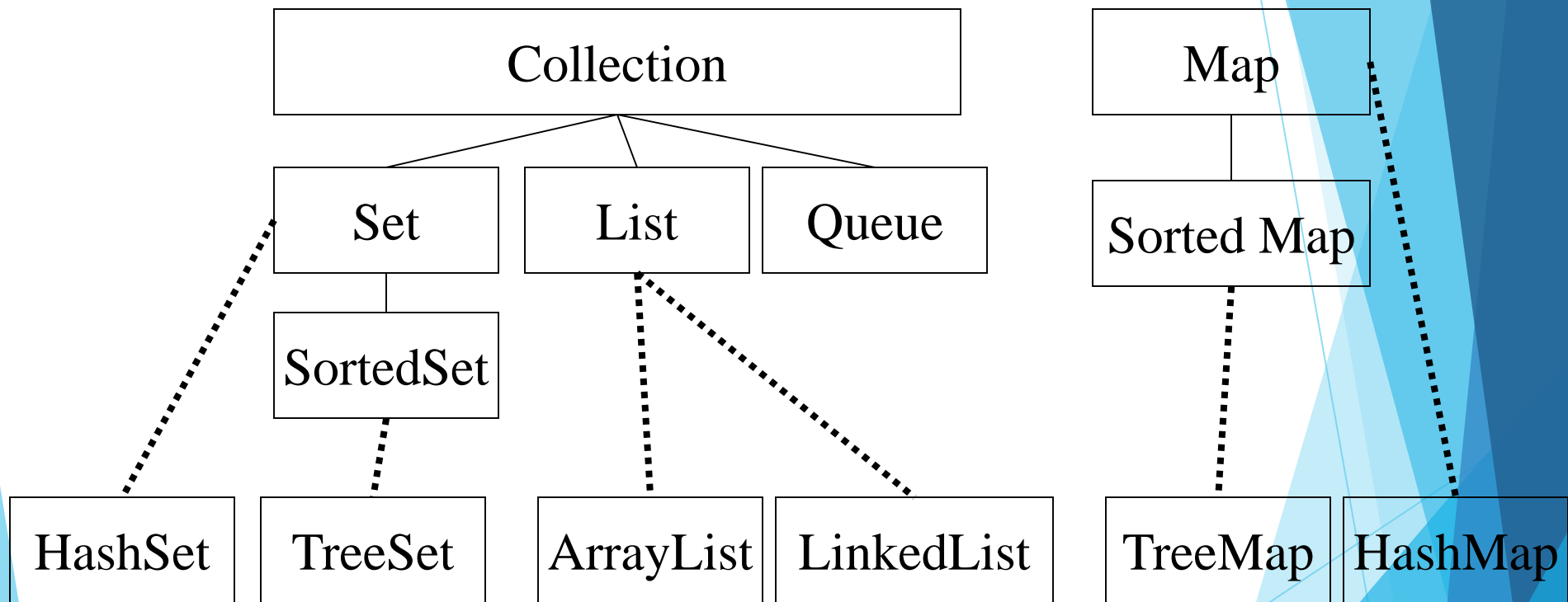
## ▶ Bulk Operations

- ▶ `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() {  
  
    }  
}
```

# 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

## 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;  
}
```

## Separating Error Handling Code from "Regular" Code (2)

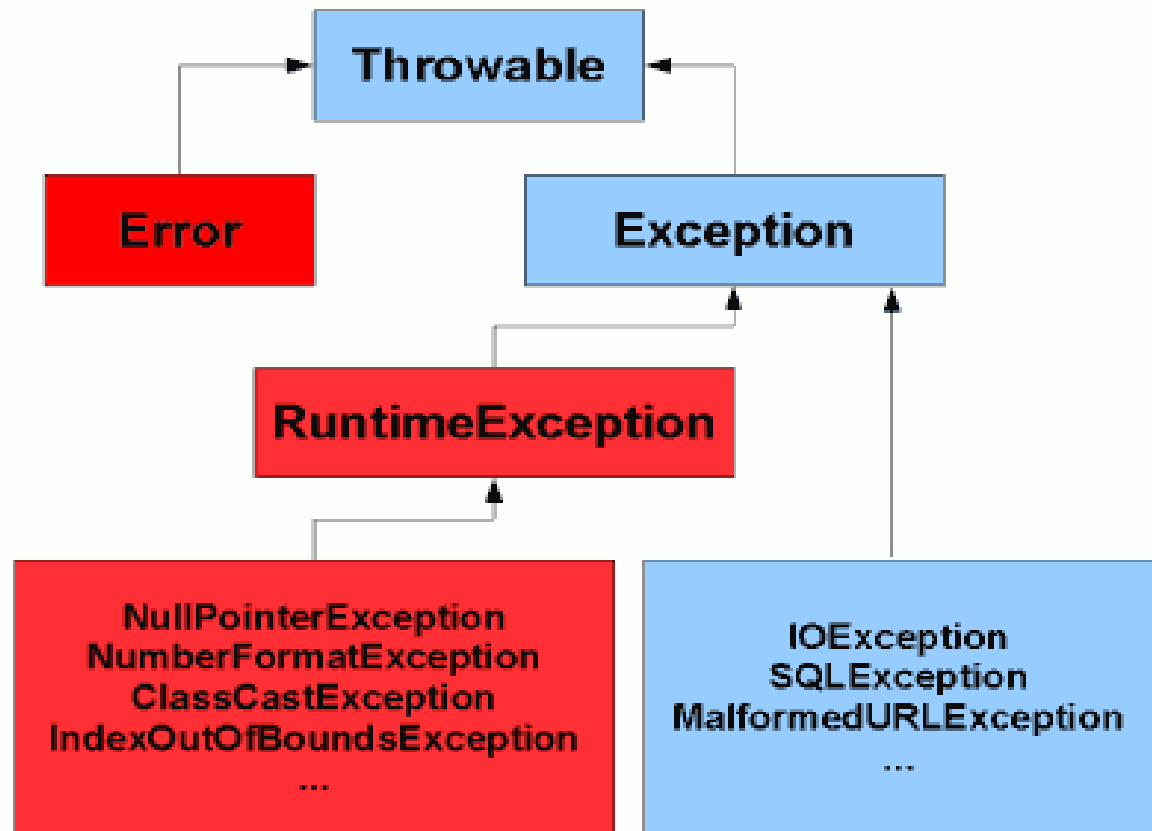
```
errorCodeType readFile {
    initialize errorCode = 0;
    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
      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

## 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;  
    }  
}
```

# Throwable and its Subclasses

Unchecked



# Checked Exceptions vs Unchecked Exceptions

Checked Exceptions	Unchecked Exceptions or Runtime Exceptions
All other Exceptions are called Unchecked Exceptions.	Runtime Exceptions ,Error and their subclasses are called Checked Exceptions.
Need to be handled.	Need not catch or specify Unchecked Exceptions.
Ex: IOException SQLException	Ex: NullPointerException NumberFormatException

Questions ?

Quiz !