

Java



Java

- Java is object oriented.
- Java works on most platforms. While C/C++ programs are platform specific.
- Java is network enabled. It is trivially simple to write code in Java that works across networks.

What is JVM?

- It is a hypothetical processor which executes java programs.
- The java compilers produces binary byte code designed to execute on JVM rather than on a PC or Sun Workstation.
- It is abbreviated as Java Virtual Machine.

Declaring Variables

- Java requires that you *declare* every variable before you can use it.
- You can declare the variables in several ways. Often, you declare several at the same time.

```
int y, m, x //all at once
```

or one at a time.

```
int y; //one at a time
```

```
int m;
```

```
int x;
```

Declaring Variables (cont...)

- You can also declare the variables as you use them.

```
int x = 4;  
int m = 8;  
int b = -2;
```

- However, you **MUST** declare variables by the time you first refer to them.

Constants

- Constants can be defined by using the **final** modifier.
- It is a good practice to CAPITALIZE symbols referring to constants.

- Examples

```
final float PI = 3.1416;
```

```
final int NUMBER_OF_DAYS_IN_MONTH = 30;
```

Data Types

Type	Contents
boolean	true or false
byte	Signed 8-bit value
short	16-bit integer
int	32-bit integer
long	64-bit integer
float	32-bit floating point
double	64-bit floating point
char	16-bit unicode character

Data Type Conversions

- Any “wider” data type can have a lower data type assigned directly to it and the promotion to the new type will occur automatically.
- For example, If y is of type **float** and j is of type **int** then you can write:
 - `float y; //y is of type float`
 - `int j; //j is of type int`
 - `y = j; //convert int to float`to promote an integer to a float.

Data Type Conversions (cont...)

- You can reduce a wider type to a narrower type using by *casting it*. You do this by putting the data type name in parentheses and putting this name in front of the value you wish to convert:

- For example,

```
j = (int)y; //convert float to integer
```

Boolean Data Type

- Boolean variables can only take on the values represented by the reserved words **true** and **false**.



Unlike C, you cannot assign numeric values to a boolean variable and you cannot convert between **boolean** and any other type.

Numeric Data Types

- Any number you type into your program is automatically of type **int** if it has no fractional part or of type **double** if it does.
- For example in a program if a number, say 5.5 is used then it will be of double data type by default.

Numeric Data Types (cont...)

- If you want to indicate that it is a different type, you can use various suffix and prefix characters to indicate what you had in mind.

- For example,

```
float loan = 1.23f; //float
```

```
long pig = 45L; //long
```

```
long color = 0x12345; //hexadecimal
```

Simple Java Program

```
import java.io.*  
public class Add2 {  
  
    public static void main(String argv[ ]) {  
        double a, b, c;  
        a = 1.75;  
        b = 3.46;  
        c = a + b;  
  
        // print out sum  
        System.out.println("sum = " + c);  
    }  
}
```

Output : sum = 5.21

import statement

- You must use the **import** statement to define libraries or *packages* of Java code that you want to use in your program
- This is similar to the C and C++ **#include** directive.

“main” Method

- The program starts from a function called **main** and it must have *exactly* the form shown here:

```
public static void main(String argv[])
```

or

```
public static void main(String[] argv)
```

Comments

- Single line comments start with “//” (double forward slash)
- Multiple lines can be commented by enclosing the required code block between /* and */

Class Definition

- Every program module must contain one or more classes.
- The class and each function within the class is surrounded by *braces* ({ }).
- Like most other languages the equals sign is used to represent assignment of data.

String concatenation

- You can use the “+” sign to combine two strings. The string “sum =” is concatenated with the string representation of the double precision variable `c`.

Arithmetic Operators

+	Addition
-	Subtraction, Unary Minus
*	Multiplication
/	Division
%	Modulus

Assignment Operators

=	Assignment
+=	Compound assignment
-=	
*=	
/=	
etc	

Increment & Decrement Operators

- Java allows you to express incrementing and decrementing of integer variables using the “++” and “--” operators.

- For example

```
i = 5; j = 10;
```

```
x= i++ //x = 5, then i = 6
```

```
y = --j; //y = 9 , j = 9;
```

Control structures



Making decisions in Java

- The familiar if-then-else of Visual Basic has its analog in Java. Note that in Java, however, we do not use the “then” keyword. For Example,

```
if ( y > 0 )  
    z = x / y;
```

- Parentheses around the condition are **REQUIRED** in Java.

Making decisions in java (cont...)

- If you want to have several statements as part of the condition, you must enclose them in braces:

```
if (y > 0) {  
    z = x / y;  
    System.out.println("z = " + z);  
}
```


Comparison Operators

Java	Meaning
>	Greater than
<	Less than
==	Is equal to
!=	Is not equal to
>=	Greater than or equal to
<=	Less than or equal to
!	Not
Note :- All of these operators return boolean results	

Logical operators

Operator	Meaning
&	Logical AND
	Logical OR
&&	Shortcut Logical AND
	Shortcut Logical OR

switch statement

```
switch(expression) {  
    case constant : statements  
    case constant : statements  
    ----  
    ----  
    default : statements  
}
```

switch statement - example

```
switch(number) {  
    case 1: System.out.println("One");  
            break;  
    case 2: System.out.println("Two");  
            break;  
    case 3: System.out.println("Three");  
            break;  
    default : System.out.println("Invalid");  
}
```

Loops

<pre>while(boolean-expression) statement</pre>
<pre>do statement while(boolean-expression);</pre>
<pre>for(initialization; boolean-expression; step) statement</pre>

break and continue

- You can also control the flow of the loop inside the body of any of the iteration statements by using **break** and **continue**
- **break** quits the loop without executing the rest of the statements in the loop
- **continue** stops the execution of the current iteration and goes back to the beginning of the loop to begin the next iteration
- For nested loops, labels can be used along with these statements to specify the loop

Classes and Objects



Class

- It is a basic unit in java programming language.
- It provides a structure for objects.
- Contract – A combination of methods, data and semantics.

Java Class Structure

```
package <package_name>;

import <other_packages>;

public class ClassName {
    <variables(also known as fields)>;

    <constructor(s)>;

    <other methods>;
}
```

Simple class

```
class BankAccount {  
    private double balance = 0.0;  
}
```

Class Members

1. Fields
2. Methods
3. Classes – Nested classes
4. Interfaces – Nested interfaces

Fields

- Class variables are called fields.
- Fields are data variables associated with a class and its objects.
- Instance variables – associated with objects
- Static variables – associated with class

Field Initialization

- When a field is declared it can be initialized by assigning it a value of the corresponding type
 - `double zero = 0.0; // constant`
 - `double sum = 4.5 + 3.7; // constant expression`
 - `double zeroCopy = zero; // field`
 - `double rootTwo = Math.sqrt(2); // method invocation`
 - `double someVal = sum + 2 * Math.sqrt(rootTwo)`

Field Initialization *contd..*

- If a field is not initialized a default initial value is assigned to it depending on its type.

Type	Default value
boolean	false
char	'\u0000'
int types	0
float	0.0f
double	0.0
Object ref	null

Final fields

- A final variable is one whose value cannot be changed after it has been initialized.
- Ex:

```
final double PI = 3.141592;
```

Methods

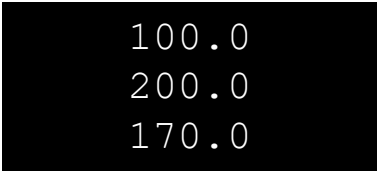
- Methods also are members of a class
- Method should have a type
- Type of a method is the type of data it returns
- If the method does not return a value its type is void
- 'this' can be used to refer instance variable explicitly

Method Overloading

- **Rules for method overloading :**
 1. Overloaded methods must change the argument list.
 2. Overloaded methods can change the return type.
 3. Overloaded methods can change the access modifier.

Method Overloading

```
public static void main(String[] args) {  
    Sales s = new Sales();  
  
    System.out.println(s.computeSales(100));  
    System.out.println(s.computeSales(100,2));  
    System.out.println(s.computeSales(100,2,30));  
}  
  
class Sales {  
    double computeSales(double price) {  
        double sales;  
        sales = price;  
        return sales;  
    }  
    double computeSales(double price, int qty) {  
        double sales;  
        sales = price * qty;  
        return sales;  
    }  
    double computeSales(double price, int qty, double discount) {  
        double sales;  
        sales = (price * qty) - discount;  
        return sales;  
    }  
}
```



100.0
200.0
170.0

Static members

- Variables shared by all objects of a class are called static fields or class variables.

```
static int nextID;
```

- But, when accessed externally it must be accessed via class name.

Ex: System.out

- Methods also can be static

Access control

- It provides a way to who has access to what members of a class.
 - **private** – accessible only in the class itself.
 - **package** – accessible in
 - classes in the same package
 - class itself
 - **protected** – accessible in
 - subclasses of any package
 - classes in same package,
 - the class itself.
 - **public** – accessible
 - anywhere the class is accessible.

Creating Objects

- Object of a class is created using the keyword – new

- Ex:-

```
BankAccount anAccount = new BankAccount()  
anAccount.balance = 1000.00;
```

- You never delete objects. JVM manages memory for you using *garbage collection*.

Constructors

- Constructors are blocks of statements that can be used to initialize an object.
- Constructors have the same name as the class they initialize.
- Constructors take zero or more arguments.

Constructors (contd)

EX:

```
class BankAccount {  
    double balance = 0.0  
    BankAccount(double initialBalance) {  
        balance = initialBalance  
    }  
}
```

Using constructor to create objects:

```
BankAccount anAccount = new BankAccount(1000.00);
```

Constructors (contd)

- All Java classes have constructors that are used to initialize a new object of that type
- A constructor has the **SAME NAME** as the class
- A constructor **DOESNOT** have return type.
- For example, a no argument constructor for `Stack` class can be

```
public Stack() {  
    items = new Vector(10);  
}
```


Constructors -Overloading

- Java supports name overloading for constructors so that a class can have any number of constructors, all of which have the same name
- For example, constructors for the stack classes can be
 - `public Stack()` // no argument ctor
 - `public Stack(int initialSize)` // 1 argument ctor

Default Constructor

- When writing your own class, you **DON'T HAVE** to provide constructors for it
- The default no argument constructor is automatically provided by the runtime system for any class that contains no constructors
- If a constructor with arguments is provided, default constructor is not automatically created by runtime system

Create Object

- A class provides the blueprint for objects
- Variable of class type is object reference
- Unless assigned with object reference, variable value is null

```
Point p1 = new Point(23, 94);  
Rectangle r1 = new Rectangle(origin_one, 100, 200);  
Point p2;    // value of p2 is null  
P1 = new Point(23,34); // now p1 refers a new object
```

- **Referencing an Object's Variables (instance variables)**
objectReference.variableName
- **Calling an Object's Methods (instance methods)**
objectReference.methodName(argumentList);

Life Cycle Of an Object

Cleaning Up Unused Objects

- The Java runtime environment deletes objects when it determines that they are no longer used. This process is called *garbage collection*.
- An object is eligible for garbage collection when there are no more references to that object.
- We can explicitly drop an object reference by setting the variable to the special value *null*

Garbage Collector

- JRE has a garbage collector that periodically frees the memory used by objects that are no longer referenced.

Encapsulation

- Encapsulation is one of the four fundamental object-oriented programming concepts.
- The term *encapsulation* means to enclose in a capsule, or to wrap something around an object to cover it.
- Encapsulation covers, or wraps, the internal workings of a Java object.
 - Data variables, or fields, are hidden from the user of the object.
 - Methods, the functions in Java, provide an explicit service to the user of the object but hide the implementation.
 - As long as the services do not change, the implementation can be modified without impacting the user.

Public and Private Access Modifiers

- The `public` keyword, applied to fields and methods, allows any class in any package to access the field or method.
- The `private` keyword, applied to fields and methods, allows access only to other methods within the class itself.
- One way to hide implementation details is to declare all of the fields `private` and methods as `public`

String objects



String

- The String class represents character strings
- All string literals in Java programs, such as "abc", are implemented as instances of this class.
- Strings are constant; their values cannot be changed after they are created
- The class String includes methods for examining individual characters of the sequence, for comparing strings, for searching strings, for extracting substrings, and for creating a copy of a string with all characters translated to uppercase or to lowercase
- The Java language provides special support for the string concatenation operator (+), and for conversion of other objects to strings.

String (contd)

- Ways of creating String objects :

```
String s = "Hello"
```

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

```
char[ ] ch = { 'a','b','c'};
```

```
String s = new String(ch);
```

String methods

Some of the methods

char charAt(int index)

String concat(String)

boolean equals(Object)

boolean equalsIgnoreCase(String)

int indexOf(String str)

int length()

String replace(char old, char new)

String substring(int begin, int end) // begin to end – 1

String toLowerCase()

String toUpperCase()

String trim()

static String valueOf(alltypes)

StringBuffer & StringBuilder

Both classes represent mutable string objects

Both have same methods

StringBuffer is threadsafe, StringBuilder is not

Constructors

`StringBuilder()` // initial capacity 16

`StringBuilder(int capacity)`

`StringBuilder(String st)` //create with capacity 16 + length of st

StringBuffer & StringBuilder

Some methods

StringBuilder append(alltypes)

int capacity()

char charAt(int index)

int capacity()

StringBuilder delete(int start, int end) //start to end – 1

StringBuilder deleteCharAt(int index)

int indexOf(String)

StringBuilder insert(int offset, alltypes)

int length()

StringBuilder replace(int start, int end, String new)

StringBuffer reverse()

Arrays

- Arrays provide ordered collections of elements.
- Components of array can be primitive types or references to objects, including references to other arrays.
- Arrays themselves are objects and extend the class `Object`
- Examples:

```
int[] x = new int[3];  
int y[] = new int[3];
```

Arrays (cont...)

- An `ArrayIndexOutOfBoundsException` is thrown if the index is out of bounds.
- The index expression must be of type `int`
- Implicit length variable used to know the size of the array
- An array with length zero is said to be an empty array.

Arrays - example

```
public class ArrayTest {  
    public static void main(String[] args) {  
        int a1[] = {10,34,56,23,67,87};  
        int a2[]; // value is null  
        int a3[] = new int[5];  
        a2 = a1; // a1 and a2 hold same array  
        /* use length to know the size of the array */  
        for(int i=0;i<a1.length;i++)  
            System.out.println(a1[i]);  
    }  
}
```

Array of objects

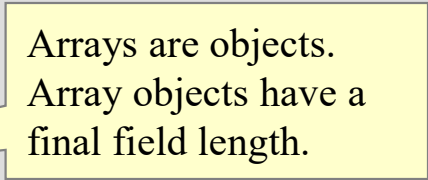
- Array of objects is array of references to the objects as shown in this example

```
public class ArrayOfStringsDemo {
    public static void main(String[] args) {
        Test b[] = new Test[5];
        Test a[] = { new Test("Ramana"), new Test("Surender"),
                     new Test("Hires"), new Test("Haritha") };
        for (int i = 0; i < a.length; i++) a[i].show();
    }
}

public class Test {
    private String name;
    public Test(String s) {
        name = s;
    }
    public void show() {
        System.out.println(name);
    }
}
```


Arrays and for-each Loop

```
1  public class ArrayOperations {
2      public static void main(String args[]){
3
4          String[] names = new String[3];
5
6          names[0] = "Blue Shirt";
7          names[1] = "Red Shirt";
8          names[2] = "Black Shirt";
9
10         int[] numbers = {100, 200, 300};
11
12         for (String name:names){
13             System.out.println("Name: " + name);
14         }
15
16         for (int number:numbers){
17             System.out.println("Number: " + number);
18         }
19     }
20 }
```



Arrays are objects.
Array objects have a
final field length.

Inheritance



INHERITANCE

- Inheritance denotes Specialization.
- A *subclass* is a class that extends another class. A subclass inherits state and behavior from all of its ancestors (a.k.a super class(es)).
- A subclass inherits variables and methods from its superclass and all of its ancestors. The subclass can use these members as is, or it **can hide** the member variables or **override** the methods.

Constructors in inheritance

- While creating subclass objects super class default constructor is automatically invoked
- To invoke argumented constructor of super class use `super()` with arguments
- `super()` should be first statement in subclass constructor

Overriding

- Overriding a method means replacing the superclass's implementation of a method with one of your own. The signature must be identical
- When a method is overridden it means both the signature and return type are **SAME** as in the superclass.
- If two methods differ only in return type it is an ERROR and the compiler will reject the class.

Overriding (cont...)

- Overriding methods can have their own access specifiers. A subclass can change the access of a superclass's methods, but **ONLY** to provide **MORE** access.

- For example

```
class Base {  
    Protected void show() {  
    }  
}  
Class Derived extends Base{  
    Public void show() { //this is valid  
    }  
}
```

Overriding (cont...)

- The Overriding method can be made final but not the method being overridden.
- Overriding method's throws clause CAN BE different from that of the superclass method's as long as every exception type listed in the overriding method is the same or a subtype of the exceptions listed in the superclass's method.
- An overriding method can have **NO** throws clause though the method in superclass has.
- Static method **CANNOT** be overridden.

Polymorphism

- Super class reference variable can hold sub class object
- When a overridden method is invoked on super class reference variables, the method of the object held by it is invoked

Polymorphism

Example :

```
class Person {  
    public void display(){ system.out.println("Person"); }  
}
```

```
class Employee extends Person {  
    public void display(){ system.out.println("Employee Data"); }  
}
```

```
class Student extends Person {  
    public void display(){ system.out.println("Student Data"); }  
}
```

Polymorphism

Example :

```
Person p ;
```

```
p = new Person();
```

```
p.display();           // output : Person
```

```
p = new employee();
```

```
p.display();           // output : Employee Data
```

```
p = new Student();
```

```
p.display();           // output : Student Data
```

Abstract methods & classes

- Methods whose design is not complete are called abstract methods
- Ex: `public abstract void printIt(int x, String y);`
- Class having abstract methods should be declared as abstract class
- Abstract class cannot be instantiated (cannot create objects)
- Abstract classes are for subclassing
- A class declared as abstract need not have abstract methods

final class

- Class declared as final cannot be extended
- final class cannot have abstract methods
- Ex:

```
public final class LastOne {
```

```
-----
```

```
-----
```

```
}
```

Interfaces



Introduction

- The fundamental unit of OO design is the *type*.
- *Interfaces* define types in an **abstract** form as a collection of methods.
- *Interfaces* contain **no implementation** and you cannot create instances of an interface.
- Classes can expand their own types by implementing interfaces.

Introduction (contd...)

- Classes can implement more than one interface.
- In a given class, the classes that are extended and the interfaces that are implemented are collectively called the *supertypes*, the new class is a *subtype*.

Interface example

- An example of a simple interface :

```
public interface Comparable {  
    int compareTo(Object o);  
}
```


Declarations

- An *interface* is declared using the keyword **interface**, giving the interface a name and listing the interface members between braces.
- An interface can have
 1. Constants
 2. Methods (only signature)
- Interface members are implicitly *public*.

Interface constants

- An interface can declare named constants.
- These constants are *implicitly* *public*, *static*, and *final*.
- *interface* Verbose {
 int SILENT = 0;
 int NORMAL = 1;
 int VERBOSE = 3;
}

Interface methods

- The methods declared in an interface are implicitly **abstract** and **public**, no other modifiers are permitted
- Methods **cannot be** *static* – because static methods cannot be abstract.

Extending interfaces

- Interfaces can be extended using the extends keyword.

```
public interface SerializableRunnable  
    extends Serializable, Runnable {  
    //.....  
}
```

- The interfaces that are extended are the *superinterfaces* and the new interface is a *subinterface*

Implementing interfaces

- A class can implement one or more interfaces using *implements* keyword
- ```
public class XXXX implements
Comparable {
 public int compareTo(Object o)
 // Implementation details
}
```

# Marker interfaces

- Some interfaces do not declare any methods. These are marker interfaces.
- Marker interfaces simply mark a class as having some general property.
- Examples of marker interfaces:
  - `Serializable`
  - `Cloneable`
  - `java.rmi.Remote`

# Packages

- Packages are convenient ways of grouping related classes according to their functionality, usability as well as category they should belong to.
- Classes under different packages **CAN HAVE** same names.
- Packaging help us to avoid class name collision when we use the same class name as that of others

# How to create Packages?

1. Suppose we have a file called **HelloWorld.java**, and we want to put this file in a package **world** then add the package definition in the top of the file as shown below...

```
// only comment are allowed before this definition
package world;
public class HelloWorld {
 //...
}
```

2. Create subdirectories to represent package hierarchy of the class. In our case, we have the **world** package, which requires only one directory. So, we create a directory **world** and put our HelloWorld.java into it.



# How to use Packages?

- There are 2 ways in order to use the public classes stored in package.
1. Declare the fully-qualified class name. For example,

```
world.HelloWorld hw = new world.HelloWorld();
world.moon.HelloMoon hm = new world.moon.HelloMoon();
String holeName = helloMoon.getHoleName();
```
  2. Use an "import" keyword:

```
import world.*;
import world.moon.*;
HelloWorld helloWorld = new HelloWorld();
HelloMoon helloMoon = new HelloMoon();
```

# CLASSPATH

- Environment variable CLASSPATH should be set to search for packages
- Specify a series of folders in classpath
- Ex:  
**Set CLASSPATH=c:\;d:\javaprg;c:\test\classes**
- Packages / classes are searched for in c:\ , d:\javaprg and c:\test\classes in that order
- Classes with no package statement belong to default package

# Using Access Control

- There are four access levels that can be applied to data fields and methods. The following table illustrates access to a field or a method marked with the access modifier in the left column.

| Modifier<br>(keyword)  | Same Class | Same<br>Package | Subclass in<br>Another<br>Package | Universe |
|------------------------|------------|-----------------|-----------------------------------|----------|
| <code>private</code>   | Yes        |                 |                                   |          |
| <code>default</code>   | Yes        | Yes             |                                   |          |
| <code>protected</code> | Yes        | Yes             | Yes *                             |          |
| <code>public</code>    | Yes        | Yes             | Yes                               | Yes      |

# Points to Note

- Classpath not required for JDK classes
- While using JDK classes, package to which the class belongs should be identified for import statement
- java.lang package classes need not be imported
- Object, String, StringBuffer, Runtime, System classes and wrapper classes belong to java.lang package

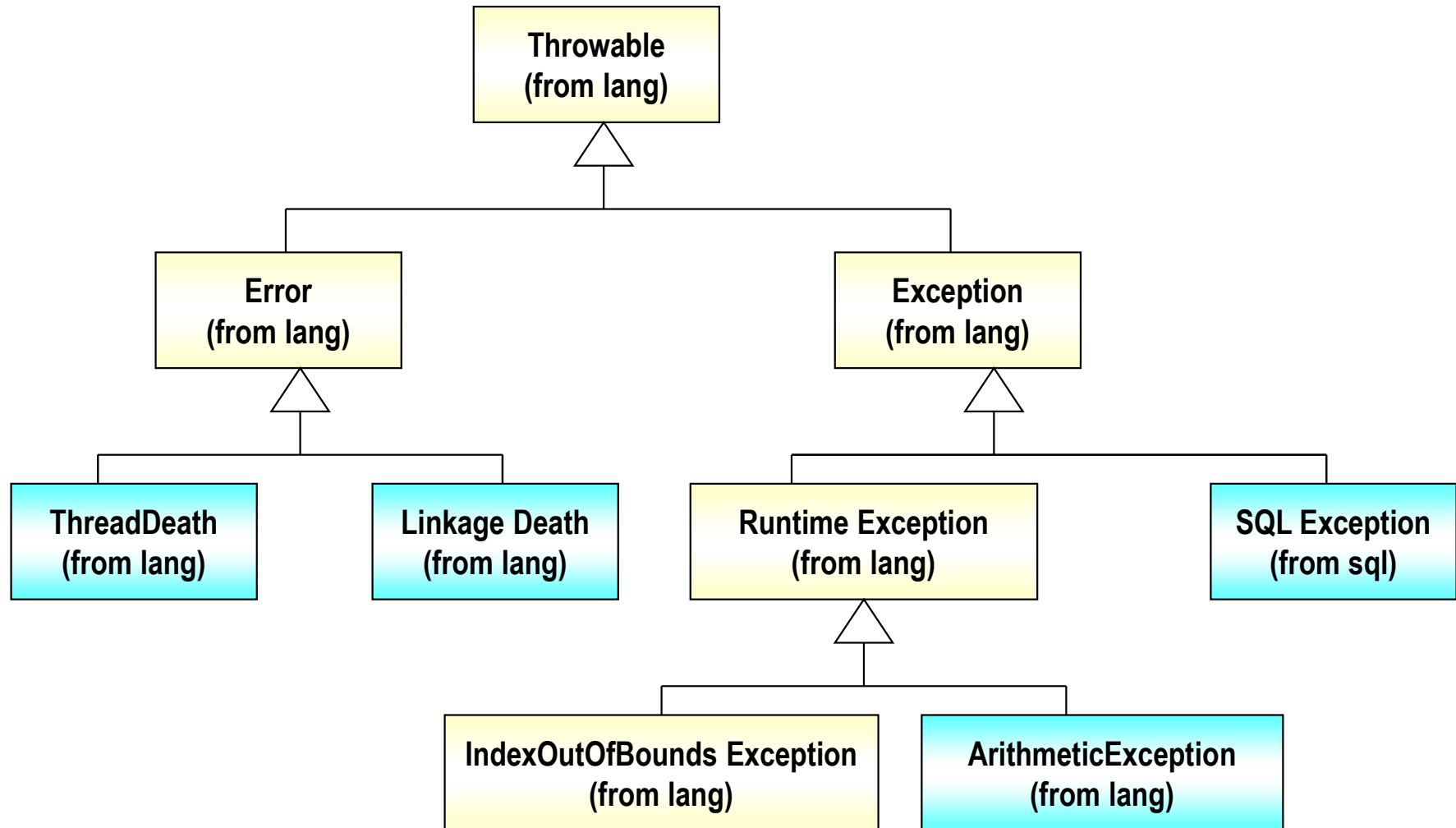
# Exceptions



# Exception Handling

- Exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions
- Exceptions can occur when
  - The file you try to open does not exist
  - The network connection is disrupted
  - Operands being manipulated are out of prescribed ranges
  - The class files you are interested in loading are missing

# Exception class hierarchy



# Exception Handling

- Checked Exceptions
  - Extends the `java.lang.Exception` class.
  - Needs to be caught or specified.
- Unchecked Exceptions
  - Extends the `java.lang.RuntimeException` class
  - Need not be caught or specified.



# Exception Handling

- Methods should either catch or specify all checked exceptions that can be thrown within the scope of that method
- A method can catch an exception by providing an exception handler for that type of exception
- If a method chooses not to catch an exception, the method must specify that it can throw that exception
- Callers of a method must know about the exceptions that a method can throw

# Dealing with Exceptions

- Three components of an exception handler  
**try, catch, and finally blocks**

## **try Block**

- Enclose the statements that might throw an exception within a try block
- Defines the scope of any exception handlers

## **catch Block**

- Associate exception handlers with a try block by providing one or more catch blocks directly after the try block

## **finally Block**

- Allows the method to clean up after itself regardless of what happens within the try block

# Exceptions - example

```
public void writeList() {
 PrintWriter out = null;
 try {
 System.out.println("Entering try statement");
 out = new PrintWriter(
 new FileWriter("OutFile.txt"));

 for (int i = 0; i < size; i++)
 out.println("Value at: " + i + " = " +
 victor.elementAt(i));
 } catch (ArrayIndexOutOfBoundsException e) {
 System.out.println("Array refererror " + e.getMessage());
 } catch (IOException e) {
 System.out.println("IOException: " + e.getMessage());
 }
 finally {
 if (out != null)
 out.close();
 }
}
```

# Specifying the Exceptions Thrown by a Method

- To let a method transfer the exception to the caller, specify throws
- Caller is responsible to handle this exception

Ex:

```
public void writeList() throws IOException {
 PrintWriter pw =
 new PrintWriter(new FileWriter("File.txt"));
 for (int i = 0; i < size; i++)
 pw.println(b[i]);
 pw.close();
}
```

# Defining Exceptions

## **throw statement**

**Before you can catch an exception, some Java code somewhere must throw one**

```
throw someThrowableObject;
```

**Throwable objects should be subclass of the Throwable class**

# Defining Exceptions - example

```
public class ApplicationException extends Exception{
 public ApplicationException(String msg) {
 super(msg);
 }
}

public void readFile(String fileName) throws
 ApplicationException {
 try {
 FileInputStream fis = new FileInputStream(fileName);

 }
 catch (FileNotFoundException e) {
 throw new ApplicationException("File Not Found",
 e);
 }
}
```

# Classes of java.lang package



# Object class

- Every java class extends Object class either directly or indirectly
- Object class provides useful methods required in every class
- Some of the methods need to be overwritten
- If a class does not extend any class, it automatically extends Object
- Every java object is instanceof Object



# Object class

## Some Object class methods

### `boolean equals (Object obj)`

Decides whether two objects are meaningfully equivalent.

### `void finalize( )`

Called by garbage collector when the garbage collector sees that the object cannot be referenced

### `int hashCode( )`

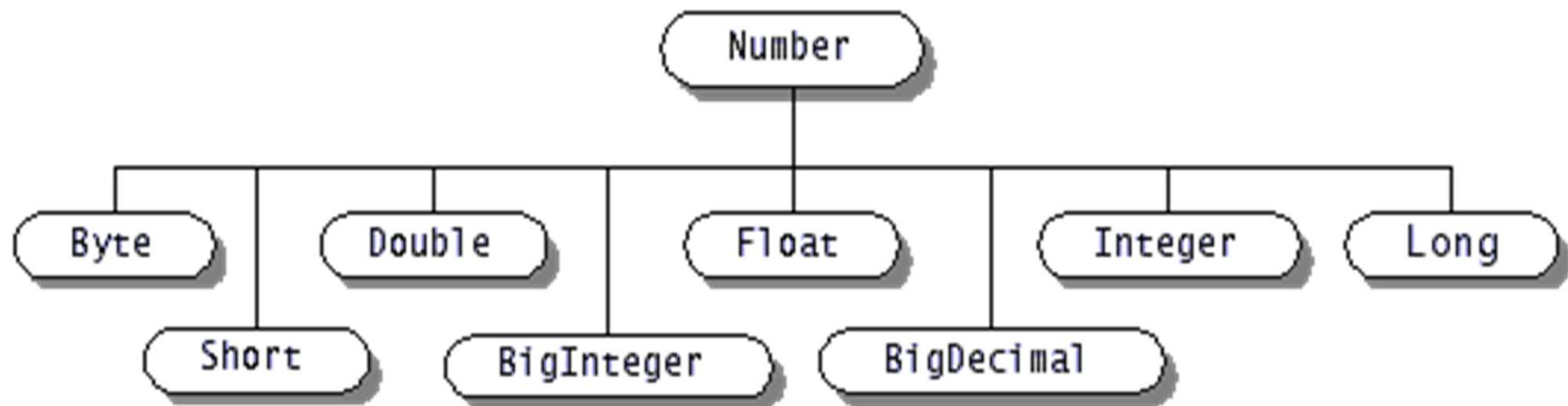
Returns a hashcode int value for an object, so that the object can be used in Collection classes that use hashing, including Hashtable, HashMap, and HashSet

### `String toString( )`

Returns a “text representation” of the object

# Wrapper classes

- Wrapper classes available to create objects for all primitive types
- These classes provide methods to convert wrap primitive types and also convert back to primitives



- Also available, Boolean and Character classes

# Wrapper classes

- Designed to convert primitives into objects
- Wrapper objects are immutable
- provide methods for conversion of primitives to/from String objects to different bases
- All wrapper class names map to primitives they represent except Integer and Character
- Byte, Short, Integer, Long, Float, Double are sub classes of Number
- constructors overloaded to take primitives as well as their String representation

# Wrapper classes

- Common methods of wrapper classes

## Methods of Number

byte    byteValue( )  
short   shortValue( )  
int     intValue( )  
long    longValue( )  
float   floatValue( )  
double doubleValue( )

## Character

char charValue( )

## Boolean

boolean booleanValue( )

# Auto Boxing Unboxing

- ☞ Before Java 5, wrapping and unwrapping was done explicitly
- ☞ For example to perform arithmetic on a wrapped value involved unwrapping and re-wrapping after operation

```
Integer x = new Integer(45);
```

```
int y = x.intValue();
```

```
y = y + 10 ;
```

```
x = new Integer(y) ;
```

- ☞ java 5 provides auto boxing / unboxing
- ☞ In that wrapping and unwrapping done automatically based on the operation

```
Integer x = 45;
```

```
x = x + 10 ;
```

```
x++; // and so on
```

# Auto Boxing Unboxing

All these are possible now !!!

```
Integer x = 36; // wrap it
x++; // unwrap and re-wrap
```

```
List l = new ArrayList();
l.add(0,36); //wrap and add
```

```
Integer a = 30; //wrap
Integer b = 20; //wrap
Integer c = a + b; //unwrap, add, wrap the sum
```

```
if (a.equals(30)) //unwrap and compare
```

# Math class

- Math class provides many arithmetic, trigonometric and logarithmic methods
- All these methods are static
- It is not possible to create Math class object
- Example methods:

static double sqrt(double)

static double sin(double)

static double random( )

# Date and time handling

- An instance of java.util.Date represents a specific instant in time with millisecond precision
- java.util.Calendar is an abstract base class for extracting detailed information such as year, month, date, hour, minute and second from a Date object
- Subclasses of Calendar can implement specific calendar systems such as Gregorian calendar, Lunar Calendar and Jewish calendar
- Currently, java.util.GregorianCalendar for the Gregorian calendar is supported in the Java API
- You can use new GregorianCalendar() to construct a default GregorianCalendar with the current time
- use new GregorianCalendar(year, month, date) to construct a GregorianCalendar with the specified year, month, and date. The month parameter is 0-based, i.e., 0 is for January



# Date and time handling

- **Calendar.getInstance()** returns current date
- **getTime()** of Calendar object returns Date object
- **get(field)** method of calendar object returns individual elements of the date&time
- **add(field, number)** method of calendar object can be used to perform date calculations
- **set(y,m,d)** of calendar object can be used to change field values
- **set(field,value)** of calendar object used to change any field value

# Calendar - Examples

```
SimpleDateFormat sdf = new SimpleDateFormat("dd/MMM/yyyy");
Calendar cal = new GregorianCalendar(2013,0,31);
System.out.println(sdf.format(cal.getTime()));
```

```
cal.set(2010, 6, 21);
System.out.println(sdf.format(cal.getTime()));
```

```
Calendar calendar = Calendar.getInstance();
```

```
int year = calendar.get(Calendar.YEAR);
int month = calendar.get(Calendar.MONTH); // Jan = 0, dec = 11
int dayOfMonth = calendar.get(Calendar.DAY_OF_MONTH);
int dayOfWeek = calendar.get(Calendar.DAY_OF_WEEK);
int weekOfYear = calendar.get(Calendar.WEEK_OF_YEAR);
int weekOfMonth = calendar.get(Calendar.WEEK_OF_MONTH);
```

```
calendar.add(Calendar.MONTH, 1);
calendar.add(Calendar.DAY_OF_MONTH, -10);
```

# Working with Local Date and Time

- The `java.time` API defines two classes for working with local dates and times (without a time zone):
  - `LocalDate`:
    - Does not include time
    - A year-month-day representation
    - `toString` – ISO 8601 format (YYYY-MM-DD)
  - `LocalTime`:
    - Does not include date
    - Stores hours:minutes:seconds.nanoseconds
    - `toString` – (HH:mm:ss.SSSS)

# LocalDate: Example

```
import java.time.LocalDate;
import static java.time.temporal.TemporalAdjusters.*;
import static java.time.DayOfWeek.*;
import static java.lang.System.out;

public class LocalDateExample {

 public static void main(String[] args) {
 LocalDate now, bDate, nowPlusMonth, nextTues;
 now = LocalDate.now();
 out.println("Now: " + now);
 bDate = LocalDate.of(1995, 5, 23); // Java's Birthday
 out.println("Java's Bday: " + bDate);
 out.println("Is Java's Bday in the past? " + bDate.isBefore(now));
 out.println("Is Java's Bday in a leap year? " + bDate.isLeapYear());
 out.println("Java's Bday day of the week: " + bDate.getDayOfWeek());
 nowPlusMonth = now.plusMonths(1);
 out.println("The date a month from now: " + nowPlusMonth);
 nextTues = now.with(next(TUESDAY));
 out.println("Next Tuesday's date: " + nextTues);
 }
}
```

next method

TUESDAY

LocalDate objects are  
immutable – methods  
return a new instance.

# Working with `LocalTime`

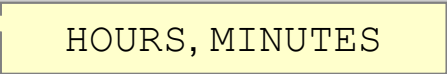
`LocalTime` stores the time within a day.

- Measured from midnight
- Based on a 24-hour clock (13:30 is 1:30 PM.)
- Questions you can answer about time with `LocalTime`
  - When is my lunch time?
  - Is lunch time in the future or past?
  - What is the time 1 hour 15 minutes from now?
  - How many minutes until lunch time?
  - How many hours until bedtime?
  - How do I keep track of just the hours and minutes?

# LocalTime: Example

```
import java.time.LocalTime;
import static java.time.temporal.ChronoUnit.*;
import static java.lang.System.out;

public class LocalTimeExample {
 public static void main(String[] args) {
 LocalTime now, nowPlus, nowHrsMins, lunch, bedtime;
 now = LocalTime.now();
 out.println("The time now is: " + now);
 nowPlus = now.plusHours(1).plusMinutes(15);
 out.println("What time is it 1 hour 15 minutes from now? " + nowPlus);
 nowHrsMins = now.truncatedTo(MINUTES);
 out.println("Truncate the current time to minutes: " + nowHrsMins);
 out.println("It is the " + now.toSecondOfDay()/60 + "th minute");
 lunch = LocalTime.of(12, 30);
 out.println("Is lunch in my future? " + lunch.isAfter(now));
 long minsToLunch = now.until(lunch, MINUTES);
 out.println("Minutes til lunch: " + minsToLunch);
 bedtime = LocalTime.of(21, 0);
 long hrsToBedtime = now.until(bedtime, HOURS);
 out.println("How many hours until bedtime? " + hrsToBedtime);
 }
}
```



HOURS, MINUTES