Java Programming

Object Oriented concepts

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Abstraction

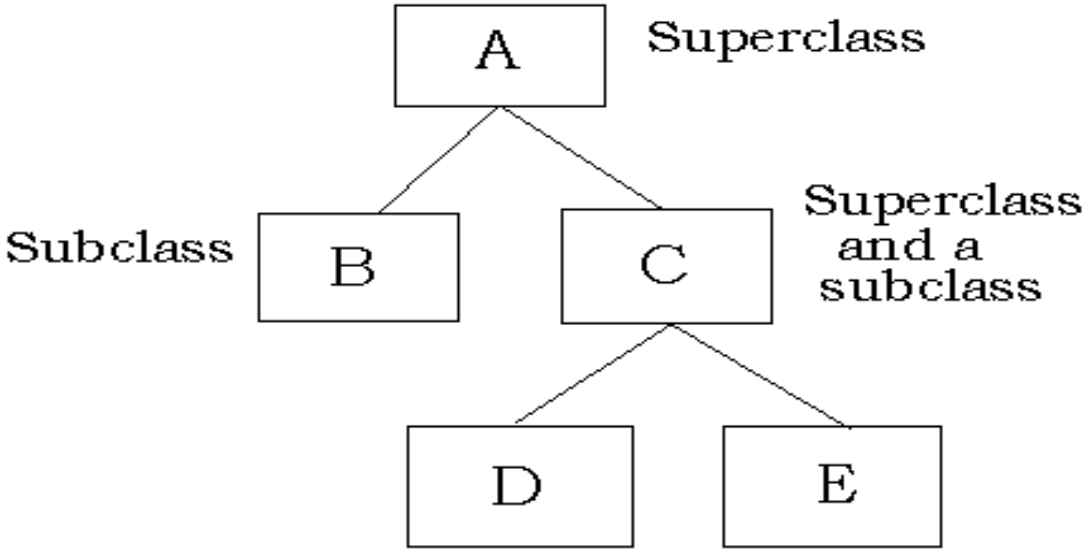
- Process of hiding the internal details and exposing only the essential details
- Also called as data hiding

Encapsulation

- Mechanism that binds code and data together
- Basis of encapsulation is a **class**
- Class- defines the structure and behaviour of a set of objects

Inheritance

- Allows common features to be defined as a "Superclass"
- Often referred to as "is a kind of" relationship
- Specialized subclasses can be defined to extend the superclass
- A subclass can
  - Use features of the superclass
  - Override the superclass behaviour
  - Add new attributes and behaviour


A
Superclass
Subclass
B
C
Superclass
and a
subclass
D
E

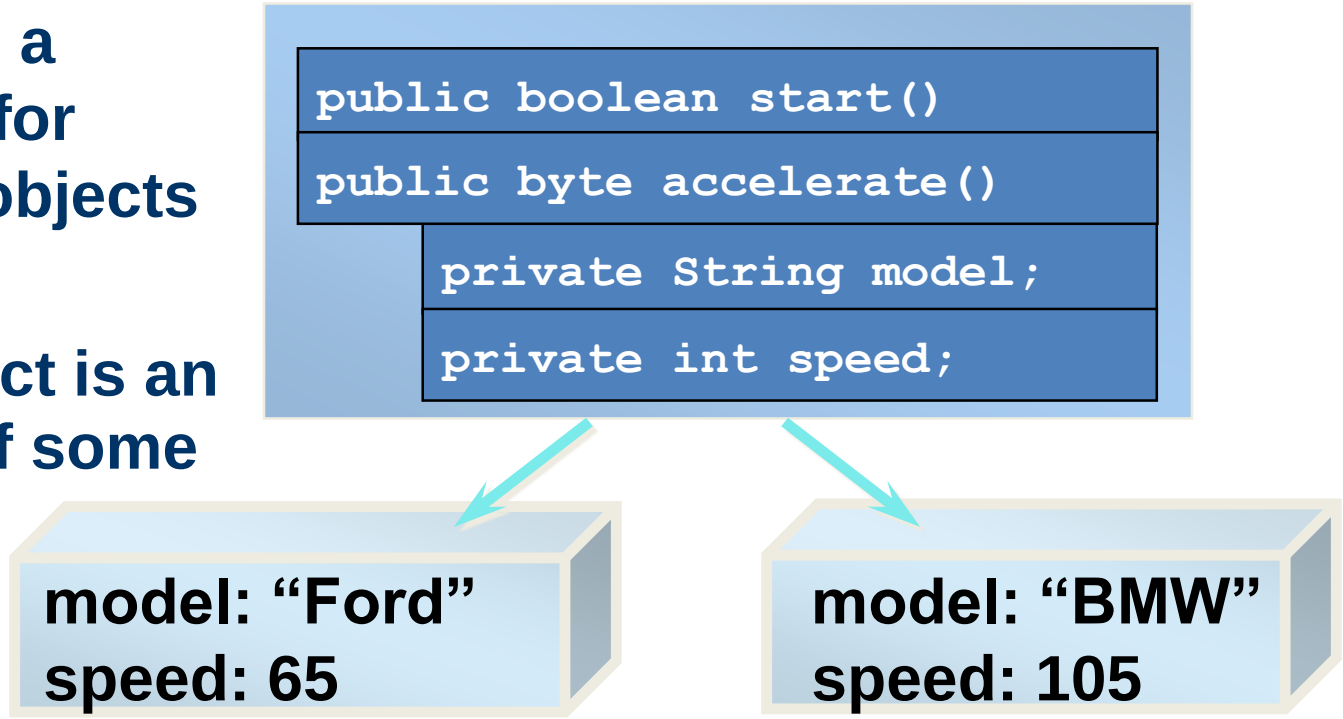
Polymorphism

- Capability of an action or method to do different things based on the object that is acting upon
- Types: overloading, overriding
- Overloading: methods with same name but different number or type of parameters
- Overriding: methods that are redefined within an inherited class

Classes and Objects

- **A class is a template for building objects**

Every object is an *instance* of some class



Creating Objects

- Objects are created using ***new*** operator
- General form:

  class-var = new classname();

  Example: Box mybox = new Box();

```
Car car1 = new Car();
Car car2 = new Car();
```

The **new** operator

- The `new` operator performs the following actions:
  - Allocates memory for the new object
  - Calls a special initialization method in the class, called a *constructor*
  - Returns a *reference* to the new object

Instance variables

- Variables declared inside a class are instance variables
- Each object has its own copy of instance variables

```
public class Car {
  public String model;
  public int speed;
  …
}
```

```
Car car1 = new Car();
car1.model = "BMW";
…
if (car1.speed > 55)
  car1.speed = 55;
```

Methods

- A *method* is equivalent to a function or subroutine in other languages
- A method can only be defined within a class definition

```
modifier returnType methodName (argumentList) {
    // method body
    …
}
```

‘this’ keyword

- Reference to the current object
- Example

```
public class Car {
  public void setSpeed(int s) {
    this.speed = s;
  }}
```

- Often it is omitted
- keyword *this* must be explicitly inserted whenever some local variables hold the same name of instance variables

Constructors

- Called automatically when an object is created
  - Usually declared public
  - Has the same name as the class
  - No specified return type
- Default: no-argument do-nothing constructor supplied by compiler

Defining constructors

```
public class Car {
   private String model;
   int speed = 0;

   public Car() {
      model = "Ford";
   }
   public Car(String m) {
      model = m;
   }
}
```

The Car class now provides two constructors

```
Car car1 = new Car();
Car car2 = new Car("Audi");
Car car3 = new Car("BMW");
```

Overloading methods

- Creating multiple versions of a method in the same class
- Overloading methods
  - Should have same names
  - But different signatures
  - Return type of method is never part of method signature.

Access control specifiers

- private-Can be accessed only within the class
- default-Can be accessed by any other class in the same package
- protected-Can be accessed by any other class in the same package or inherited classes from other package
- public-Can be accessed by any class inside or outside the package

Static Variables

- The static variable can be used to refer the common property of all objects.
- e.g. company name of employees, college name of students etc.
- The static variable gets memory only once at the time of class loading.

Static methods

- A method declared static can be called without creating an object

```
class t2 {
        static int triple (int n)       {
        return 3*n;}}
class t1 {
   public static void main(String[] arg) {
         System.out.println( t2.triple(4) );
          t2 x1 = new t2();
         System.out.println( x1.triple(5) );    }} // also allowed
```

final

- Similar to const in C/C++
- Coding convention: all uppercase
- Example

```
final int FILE-NEW = 1;
```

Inheritance

- Class that is inherited is called **super class.**
- Class that does the inheritance is called a subclass.
  - To incorporate the definition of one class into another "extends" clause is used

```
public class Car {
   …
}
```

```
public class Taxi extends Car {
   …
}
```

Inheritance

- Types of Inheritance
  - Single Inheritance
  - Hierarchical Inheritance
  - Multilevel Inheritance

‘super’ keyword

- To refer to the immediate superclass
- super(argument list)
- Should always be the first statement executed inside a subclass constructor

‘super’ keyword

- This form of super is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.
- This usage has the following general form:
  - super.member

When constructors are created?

- In a hierarchy, constructors complete their execution in order of derivation, from superclass to subclass.

Method overriding

- Occurs when the name and signatures of the subclass and superclass are identical
- When called from within a subclass, refers to the version of the method defined by the subclass
- If a method uses final keyword, that method cannot be overridden by the subclasses

Dynamic Method Dispatch

- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- Method overriding forms the basis for dynamic method dispatch.
- Java implements run-time polymorphism through Dynamic method dispatch.

Dynamic Method Dispatch

- When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time of the call occurs.
- If a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

Abstract class

- A class which is not used to construct objects, but is a basis for making subclasses
- Main purpose: to express the common properties of all the subclasses.
- It cannot be instantiated
- May or may not include abstract methods
- Abstract method: method without implementation

Abstract class

- If a class includes abstract methods, class itself should be declared abstract

```
public abstract class GraphicsObject
{
  //declare fields
 //declare non-abstract methods
abstract void draw();
}
```

Constructor Chaining

- Constructor chaining is the process of calling one constructor from another constructor.
- Constructor chaining can be done in 2 ways:
  - Within same class
    - It can be done using **this()** keyword for constructors in same class
  - From base class
    - by using **super()** keyword to call constructor from the base class.

Rules of Constructor Chaining

- An expression that uses **this** keyword must be the first line of the constructor.
- **Order** does not matter in constructor chaining.
- There must exist at least one constructor that does not use **this**

- Declaring a class as final, prevent the class from being inherited.
- Declaring a class as final implicitly declares all of its methods as final

Reference Datatypes

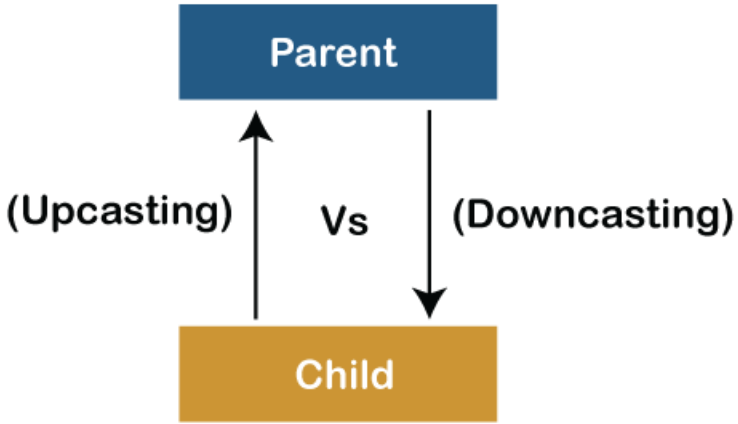
- Reference datatypes in java are **those which contains reference/address of dynamically created objects**.
- These are not predefined like primitive data types.
- Java Reference types are
  - Class
  - Array
  - Interface
  - Enumeration
  - Annotation

Difference between Reference Datatype and Primitive Datatype

Reference Datatype	Primitive Datatype
Unlimited number of reference types, as they are defined by the user.	Limited number of data types
Memory location stores a reference to the data.	Memory location stores actual data held by the primitive type.
When a reference type is assigned to another reference type, both will point to the same object.	When a value of a primitive is assigned to another variable of the same type, a copy is made.
When an object is passed into a method, the called method can change the contents of the object passed to it but not the address of the object.	When a primitive is passed into a method, only a copy of the primitive is passed. The called method does not have access to the original primitive value and therefore cannot change it.

Upcasting and Downcasting

- Typecasting a child object to a parent object is called upcasting
- Typecasting a parent object to a child object is called downcasting



Upcasting

- **Upcasting** is a type of object typecasting in which a **child object** is typecasted to a **parent class object**.
- **Upcasting** is also known s **Generalization** and **Widening**.

Downcasting

- **Downcasting** is another type of object typecasting.
- In Downcasting, we assign a parent class reference object to the child class.
- Downcasting is also called narrowing

Interfaces

- Similar to abstract classes, but all methods are abstract and all properties are static final
- Specifies what a class must do but not how
- Methods do not have any body.
- They can be implemented by classes or extended by other interfaces

Defining an interface

```
modifier interface interfacename{
 type final varname1=value;
type final-varname2=value;
…
return-type methodname1(parameter-list);
return-type methodname2(parameter-list);
}
```

Implementing interfaces

- class classname [extends superclass]

  [implements interface1[,interface2…]]

  {

  }

Nested interface

- Interface can be declared as member of class or another interface
- A nested interface can be declared as public, private or protected.
- When used outside the scope, must be qualified by the name of the class or interface of which it is a member

Example- nested interface

```
class a{
    public interface nestedif{
        boolean isnotnegative(int x);
 } }
class b implements a.nestedif{
   public boolean isnotnegative(int x){
      return x < 0 ? false : true
}}
```

Interfaces can be extended

- One interface can inherit another by use of the keyword extends.
- When a class implements an interface that inherits another interface, it must provide implementations for all methods required by the interface inheritance chain.

Default methods

- Default method is a new feature, which allows to add new methods to the interfaces without breaking the existing implementation of the interface.
- *default return-type methodname (parameter-list)*

```
{
    //body of method
}
```

Static methods in an interface

- A static method defined by an interface can be called independently of any object.
- A static method is called by specifying the interface name, followed by a period, followed by the method name.
- General form :
  - InterfaceName.staticMethodName

Private Interface Methods

- An interface can include a private method.
- A private interface method can be called only by a default method or another private method defined by the same interface.

Functional Interfaces

- A **functional interface** is an interface that contains only one abstract method.
- From Java 8 onwards, lambda expressions can be used to represent the instance of a functional interface.
- A functional interface can have any number of default or static methods.

@FunctionalInterface Annotation

- @FunctionalInterface annotation is used to ensure that the functional interface can't have more than one abstract method.
- In case more than one abstract methods are present, the compiler flags an 'Unexpected @FunctionalInterface annotation' message.

Functional Interfaces

- Functional Interface is additionally recognized as **Single Abstract Method Interfaces**.
- Functional interfaces are used and executed by representing the interface with an **annotation called *@FunctionalInterface***.
- Eg: Runnable, ActionListener, Comparable, Callable

Functional Interfaces

- There are 4 kinds of functional interfaces
  - Consumer
  - Predicate
  - Function
  - Supplier

Consumer

- The consumer interface of the functional interface is the one that accepts only one argument.
- It doesn't return any value.

Predicate

- A predicate functional interface of java is a type of function which accepts a single value or argument and does some sort of processing on it, and returns a boolean (True/ False) answer.

Function

- A function is a type of functional interface in Java that receives only a single argument and returns a value after the required processing.

Supplier

- The Supplier functional interface is also a type of functional interface that does not take any input or argument and yet returns a single output.

Packages

- Container for classes
- Contains groups of related types
- Use
  - To make types easier to find and use
  - To avoid naming conflicts
  - To control access

Creating a package

- Include package command as the first statement within a Java source program
- If no name is provided, classes are placed in the default package
- General form:

```
package pkg;
```

Naming conventions

- All lowercase to avoid conflicts with classes
- Packages in Java begin with java or javax
- To have a hierarchy of packages, separate each package name with a dot(.)

Importing packages

- Should appear immediately after the package statement and before any class definition
- General form

  import pkghierarchy;

Enumerations

- An enumeration is a list of named constants that define a new data type.
- An enumeration is defined using the enum keyword
- enum colors{Red, Blue, Green}
- Red, Blue, Green are called enumeration constants
- They are implicitly declared as public, static, final members of colors

Enumerations

- To declare a variable of colors enum,
  - colors c;
- All enumerations automatically contain 2 predefined methods
  - values() and valuesOf()
  - public static enum-type[] values()
  - public static enum-type valueOf(String str)

Enumerations

- Java Enumerations are class types
- Each enumeration constant is an object of its enumeration type.
- When you define a constructor for an enum, the constructor is called when each enumeration constant is created.

Enumerations

- All enumerations are inherited from java.lang.enum
- ordinal() returns an enumeration constant's position in the list of constants.
  - Ordinal values begin at zero
- To compare the ordinal values of 2 constants of the same enumeration compareTo() method is used

Stack and Heap memory

- JVM divides memory into stack and heap memory.
- Stack memory contains primitive values that are specific to a method and references to objects referred from the method.
- Access to this memory is in Last-In-First-Out (LIFO) order.

Stack and Heap memory

- Whenever we call a new method, a new block is created on top of the stack which contains values specific to that method, like primitive variables and references to objects.
- When the method finishes execution, its corresponding stack frame is flushed, the flow goes back to the calling method, and space becomes available for the next method.

Key features of Stack memory

- It grows and shrinks as new methods are called and returned, respectively.
- Variables inside the stack exist only as long as the method that created them is running.
- It's automatically allocated and deallocated when the method finishes execution.

Key features of Stack memory

- If this memory is full, Java throws *java.lang.StackOverFlowError*.
- Access to this memory is fast when compared to heap memory.
- This memory is threadsafe, as each thread operates in its own stack.

Heap memory

- Heap space is used for the dynamic memory allocation of Java objects and JRE classes at runtime.
- New objects are always created in heap space, and the references to these objects are stored in stack memory.

Key features of Heap memory

- If heap space is full, Java throws *java.lang.OutOfMemoryError*.
- Access to this memory is comparatively slower than stack memory
- It needs Garbage Collector to free up unused objects so as to keep the efficiency of the memory usage.
- A heap isn't threadsafe

Garbage Collection

- Garbage Collection is the process of reclaiming the runtime unused memory automatically.
- The main objective of Garbage Collector is to free heap memory by destroying unreachable objects.

Garbage Collection

- How can an object be unreferenced?
  - By nulling the reference
  - By assigning a reference to another
  - By anonymous object

Garbage Collection

- Ways for requesting JVM to run Garbage Collector
  - Using *System.gc()* method
  - Using *Runtime.getRuntime().gc()* method

Garbage Collection

- The finalize() method is invoked each time before the object is garbage collected.
- This method can be used to perform cleanup processing.
- This method is defined in Object class as:
  - protected void finalize() throws Throwable{}

Wrapper classes

Wrapper class

- Used to wrap primitive types in a class structure.
- All primitive types have an equivalent class.
- The class includes useful constants and static methods, including one to convert back to the primitive type.
- Wrapper class can be used in any situation where a primitive value will not suffice.

Wrapper classes

Primitive Data Type	Wrapper Class
double	*Double*
float	*Float*
long	*Long*
int	*Integer*
short	*Short*
byte	*Byte*
char	*Character*
boolean	*Boolean*

Wrapper Classes

Wrapper Class	**Primitive type**
variables contain the address of the object	variables contain a value
variable declaration example: ***Integer n***	variable declaration example: ***int n***
variable declaration & init ***Integer n = new Integer(0)***	variable declaration & init ***int n = 0***
assignment n = new Integer(5);	Assignment n=99

- **Autoboxing** is a process by which primitive type is automatically encapsulated(boxed) into its equivalent type wrapper.
- **Auto-Unboxing** is a process by which the value of object is automatically extracted from a type wrapper.

```
Integer  iOb;
iOb = 100;
++iOb;
```

- When we perform increment operation on Integer object, it is first unboxed, then incremented and then again reboxed into Integer type object.

Number Class

- Abstract Class
- Super Class of Integer, Long, Float, Double, Short, Byte
- Common Methods
  - int intValue() : Convert into int type
  - long longValue() : Convert into long type
  - float floatValue() : Convert into float type
  - double doubleValue() : Convert into double type
  - byte byteValue() : Convert into byte type
  - short shortValue() : Convert into short type

Integer Class

- Constant
  - public static final int MAX_VALUE = 2147483647
  - public static final int MIN_VALUE = -2147483648
- Method
  - static int parseInt(String s) :
    - Convert a Number in String into integer type
  - static int parseInt(String s, int radix) :
    - Convert a Number in String into integer type
    - A radix parameter specifies the number system to use

Double Class

- Constants
  - public static final double MAX_VALUE =1.7976931348623157e+308
  - public static final double MIN_VALUE = 4.9406564584124654e-308
  - public static final double NaN = 0.0 / 0.0
  - public static final double NEGATIVE_INFINITY = -1.0 / 0.0
  - public static final double POSITIVE_INFINITY = 1.0 / 0.0

Boolean Class

- Constant
  - public static final Boolean TRUE = new Boolean(true)
  - public static final Boolean FALSE = new Boolean(false)
- Constructors
  - Boolean(boolean b) :
    - Constructor to create boolean object receiving the initial value b
  - Boolean(String s) :
    - Constructor to receive the string value “true” or “false”
- Method
  - boolean booleanValue() :
    - Return the boolean value of object

Character Class

- Character(char value) : Constructor to initialize the object as value
- Methods
  - char charValue() : Convert into char type
  - static boolean isDigit(char ch) : Test whether is digit?
  - static boolean isLetter(char ch) : Test whether is letter?
  - static boolean isLetterOrDigit(char ch) : Returns true when it is letter or digit.
  - static boolean isJavaIdentifierStart(char ch): Returns true when it can be used as starting symbol of Java identifier.
  - static boolean isJavaIdentifierPart(char ch): Returns true when it can be used as part of Java identifier.

Constant Pool

- When we compile a *.java* file, we get a separate class file with a *.class* extension. The *.class* file consists of several sections and **constant pool** is one of them.
- It is a collection of entries similar to a symbol table.
- It contains the names of classes that are referenced, initial values of strings and numeric constants, and other miscellaneous data crucial to proper execution.
- To view the constant pool's content, we need to first compile the file and then run the command:
  - **javap -v name.class**

Types

- #n indicates the references to the constant pool.
- The constant pool table starts from index 1

Integer, Float:	32-bit constants
Double, Long:	64-bit constants
String:	16-bit string constant that points at another entry in the pool which contains the actual bytes
Class:	Contains the fully qualified class name
Utf8:	Stream of bytes
NameAndType:	Colon-separated pair of values, first entry represents the name while the second entry indicates the type
Fieldref, Methodref, InterfaceMethodref:	Dot-separated pair of values, first value points at Class entry whereas the second value points as NameAndType entry

String Handling

String Handling

- Strings are objects of type String
- A string object cannot be modified
  - For each altered version, a new String object is created

String constructors

- To create an empty string

  String s = new String();

- String initialised by an array of characters

  String(char chars[])

  Eg:

  ```
  char chars[] = {'a','b','c'};
  String s = new String(chars);
  ```

- Sub range of a character array
- Eg:

```
char chars[] = {'a','b','c','d','e','f'};
String s = new String(chars, 2, 3);
```

- String object with the same character sequence as another String

  String(String *strobj*)

  eg: String s2 = new String(s);

String length

- Returns the number of characters in the given string.
  - Call the **length()** method

```
char chars[] = {‘a’,’b’,’c’};
String s= new String(chars);
System.out.println(s.length());
```

String literals

- A string literal can be used to initialize a string object

  Eg:

```
char chars[]={'a','b','c'};
String s1 = new String(chars);
String s2 = "abc" ;
```

String concatenation

- Concatenation operator : +
- Longer string is broken into smaller pieces ,and then concatenated using '+' operator
- Other data types are automatically converted into string when one of the operand is string
- To overcome the precedence, use paranthesis
- Eg:

```
String s = "four" + (2 + 2);
```

toString()

- Converts a number to a string
- int i;

  double d;

  String s3 = Integer.toString(i);

  String s4 = Double.toString(d);

Character Extraction

- **charAt()**
  - To extract a single character from a string
  - General form

    char charAt(int *where*)
  - Example:

    char ch;

    ch = “abc”.charAt(1);

Character Extraction

- **getChars()**
  - To extract more than one character at a time
  - General form

```
void getChars(int sourcestart, int sourceEnd,
              char target[], int targetStart)
```

Character Extraction

- **getBytes()**
  - Stores characters in an array of bytes
  - General form

    byte[] getBytes()
- **toCharArray()**
  - To convert characters in a String object into a character array
  - General form

    char[] toCharArray()

String comparison

- **equals()**
  - Compares two strings for equality
  - General form: boolean equals(Object *str*)
  - Contains true if the strings contain the same characters in the same order
  - Comparison is case-sensitive
- **equalsIgnoreCase()**
  - Comparison ignoring case differences
  - General form: boolean equalsIgnoreCase(String str)

String comparison

- **regionMatches()**
  - Compares a specific region within a string with another specific region in another string
  - General form

```
boolean regionMatches(int startindex, String str2,
                      int str2StartIndex, int numChars)
boolean regionMatches(boolean ignorecase,
  int startindex, String str2, int str2StartIndex, int numChars)
```

String comparison

- **startsWith()**
  - Determines whether a given String begins with a specified string
- **endsWith()**
  - Whether string ends with a specified string
- General forms

  boolean startsWith(String str)

  boolean endsWith(String str)

String comparison

- **equals()**
  - Compares characters inside a String object
- **==**
  - Compares two object references to see whether they refer to the same instance

- String s1 = "abcde";
- String s2 = new String("abcde");
- String s3 = "abcde";

- (s1 == s2) is false
- (s1 == s3) is true
- (s1.equals(s2)) is true

String comparison

- **compareTo()**
  - Compares two strings(alphabetic order)
  - General form

    int compareTo(String *str*)
  - value < zero : invoking string is less than str

    value > zero : invoking string is greater than str

    value zero : two strings are equal
- **compareToIgnoreCase()**
  - To ignore case differences

Searching Strings

- To search a string for a specified character or substring
  - **indexOf()**
    - Searches for the first occurrence of a character or substring
  - **lastIndexOf()**
    - Searches for the last occurrence of a character or substring

Modifying a String

- **substring()**

  String substring(int startindex)

  String substring(int startindex, int endindex)

- **concat()**

  String concat(String str)

  same as ‘+’

Modifying a String

- **replace()**
  - String replace(char original, char replacement)
    - Replaces all occurences of one character in the invoking string
  - String replace(CharSequence original, CharSequence replace)
    - Replaces one character sequence with another
- **trim()**
  - Removes leading and trailing whitespaces from a string

    String trim()

    String s = “ Hello World “.trim()

Data conversion : valueOf

valueOf(boolean b): Returns the string representation of the boolean argument.

valueOf(char c): Returns the string representation of the char argument.

valueOf(double d): Returns the string representation of the double argument.

valueOf(int i): Returns the string representation of the int argument.

Changing case

- String **toLowerCase()**
- String **toUpperCase()**
- Example

```
String s = “Heloooooo”;
String u = s.toUpperCase()
```

Splitting a String

- **split()**
  - String[] split(String regexp)
    - Decomposes the invoking string into parts based on the regular expression
  - String[] split(String regexp, int max)
    - Decomposes the invoking string into parts based on the regular expression and returns an array that contains only ***max*** no: of pieces.

StringBuilder class

- Peer class of string
- Represents growable and writeable character sequences
- Internally, these objects are treated like variable-length arrays that contain a sequence of characters

StringBuilder Constructors

Constructor	Description
`StringBuilder()`	Creates an empty string builder with a capacity of 16 (16 empty elements).
`StringBuilder(CharSequence cs)`	Constructs a string builder containing the same characters as the specified `CharSequence`, plus an extra 16 empty elements trailing the `CharSequence`.
`StringBuilder(int initCapacity)`	Creates an empty string builder with the specified initial capacity.
`StringBuilder(String s)`	Creates a string builder whose value is initialized by the specified string, plus an extra 16 empty elements trailing the string.

length() and capacity()

- length()
  - int length() : length of string stored in the StringBuilder
- capacity()
  - int capacity() : no: of characters that can be stored in a StringBuilder

Length and Capacity Methods	
Method	**Description**
`void setLength(int newLength)`	Sets the length of the character sequence. If `newLength` is less than `length()`, the last characters in the character sequence are truncated. If `newLength` is greater than `length()`, null characters are added at the end of the character sequence.
`void ensureCapacity(int minCapacity)`	Ensures that the capacity is at least equal to the specified minimum.

Method	Description
StringBuilder append(boolean b) StringBuilder append(char c) StringBuilder append(char[] str) StringBuilder append(char[] str, int offset, int len) StringBuilder append(double d) StringBuilder append(float f) StringBuilder append(int i) StringBuilder append(long lng) StringBuilder append(Object obj) StringBuilder append(String s)	Appends the argument to this string builder. The data is converted to a string before the append operation takes place.

delete() and deleteCharAt()

- Deletes the specified character(s) in this StringBuilder.

  StringBuilder **delete**(int start, int end)
  StringBuilder **deleteCharAt**(int index)

insert()

StringBuilder **insert**(int offset, char c)

StringBuilder **insert**(int offset, String s)

- Inserts the second argument into the StringBuilder
- The first integer argument indicates the index at which the data is to be inserted

replace()

- Replaces the specified character(s) in this StringBuilder

StringBuilder **replace**(int start, int end, String s)

reverse()

- Reverses the sequence of characters in this StringBuilder

StringBuilder **reverse**()

StringBuffer

- Similar to StringBuilder
- Difference: thread-safe
- Slower performance
- For multithreading, StringBuffer is used rather than StringBuilder

Exception Handling

Exceptions

- Event that occurs during the execution of a program that disrupts the normal flow of instructions
- When an error occurs, an ***exception object*** is created and handed to the runtime system
- Creating an exception object and handing it to the runtime system is called ***throwing an exception***
- The block of code that can handle the exception is called the ***exception handler***
- The exception handler chosen is said to ***catch the exception***

Kinds of Exceptions

- Checked Exceptions
  - Eg: java.io.FileNotFoundException
- Unchecked Exceptions
  - Error
    - Errors that occur under abnormal conditions such as h/w or s/w malfunctions
  - Runtime Exceptions
    - Internal to the applications
    - Eg:divide by zero, invalid array indexing

Exception Handling

- Programs to be monitored for exceptions are contained within try block
- Immediately following the try block is the catch block which handles those errors
- General form

```
try{
        code
}catch(ExceptionType name){  }
```

- The exception type in catch is the type of exception that the handler can handle.
- It must be the name of a class that is inherited from the Throwable class.

Multiple catch clauses

- Try statement can have more than one catch clauses, each with a different type of exception
- The first catch statement which matches the exception type is executed.
- A single catch block can also handle more than one type of exception.
- It specifies the types of exceptions that the block can handle, and each exception type is separated with a vertical bar (|).

Nested try Statements

- A try statement inside the block of another try
- If an inner try doesn't have a catch handler for a particular exception, the next try statement's catch handlers are inspected for a match

Finally

- Creates a block of code that is executed after a try/catch block has completed and before the code following the try/catch block
- Executed whether or not an exception is thrown.

Throw clause

- Used to manually throw an exception of type Throwable class or a subclass of throwable
- Flow of execution stops immediately after the throw statement; subsequent statements are not executed
- General form

  throw throwableobject;

throws

- Throws clause lists the types of exceptions that a method might throw
- General form

type method-name(parameter-list) throws *exception-list*

```
{
  // body of loop
}
```

Exception list: comma-seperated list of exceptions that a method can throw

Checked Exceptions

- Types of exceptions that must be included in a method's throws list
  - Includes
    - ClassNotFoundException
    - CloneNotSupportedException
    - IllegalAccessException
    - InstantiationException
    - InterruptedException
    - NoSuchFieldException
    - NoSuchMethodException

Unchecked exceptions

- Includes
  - ArithmeticException
  - ArrayIndexOutOfBoundsException
  - ClassCastException
  - IndexOutOfBoundsException
  - IllegalStateException
  - NullPointerException
  - SecurityException

User defined Exceptions

- To create a Checked custom exception, it must extend **Exception** class.
- To create an Unchecked custom exception, it must extend **RuntimeException** class**.**
- To display the message
  - override the **toString()** method

    OR
  - call the superclass parameterized constructor by passing the message in String format.

java.io

java.io package

- java.io package provides a set of classes used for reading and writing data to files or other input and output sources.
- Java provides two streams of data
  - Character Streams are used for 16-bit Characters.

    Uses Reader & Writer classes
  - Byte Streams are used for 8-bit Bytes.

    Uses InputStream & OutputStream classes

File

- A **File** object represents a file or a directory on the host file system.
- Constructors
  - File(String dir)
  - File(String dir, String fname)
  - File(File f, String fname)

File(contd…)

- Methods
  - String getName()
  - String getPath()
  - boolean exists()
  - boolean isDirectory()
  - boolean isFile()
  - long lastModified()
  - long length()
  - boolean delete()
  - String[] list()

Byte Stream

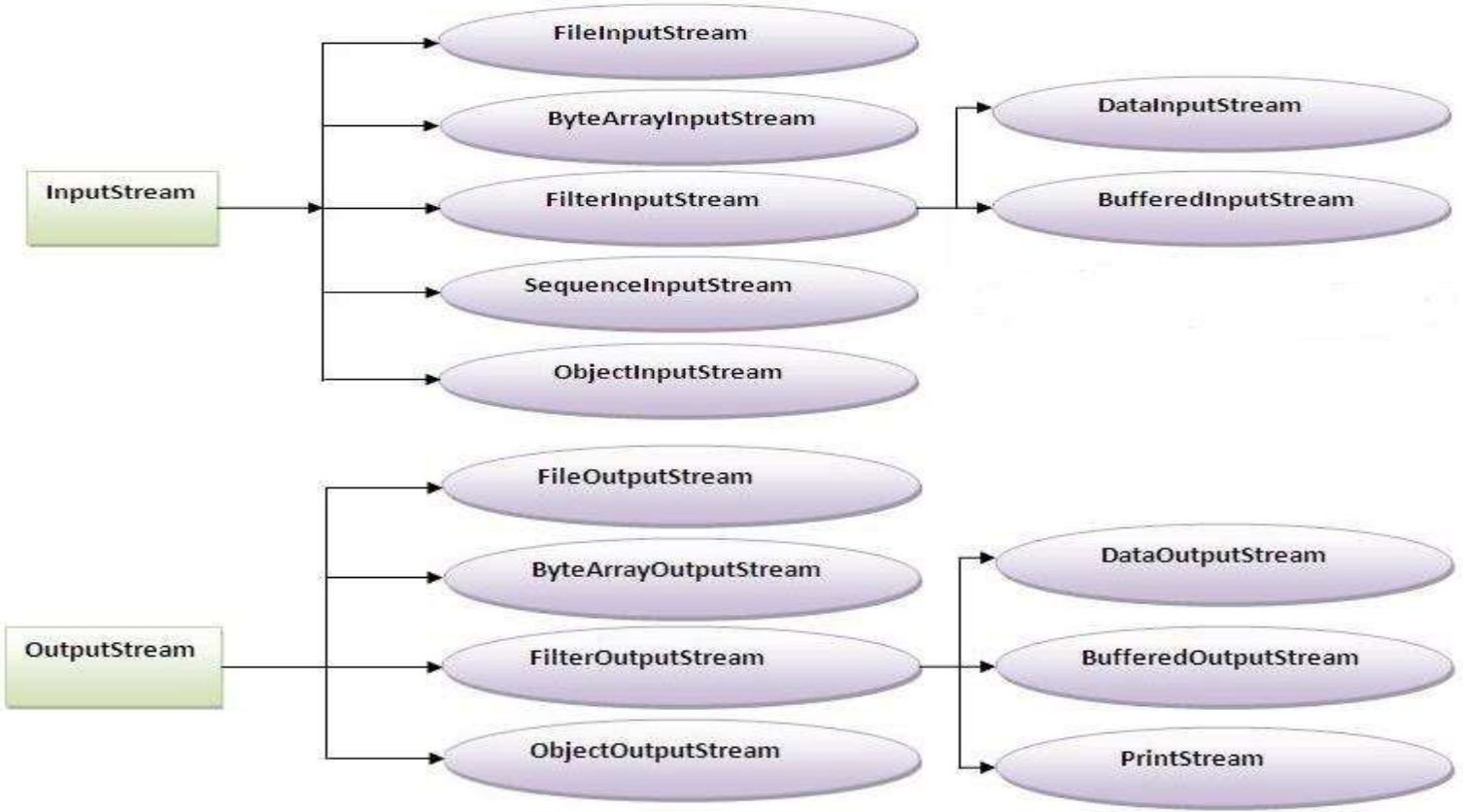
- Handles byte oriented i/o
- Can be used with any type of object, including binary data.
- Top of the byte stream hierarchies are the
  - InputStream class
  - OutputStream class

InputStream

- Abstract class that defines methods for reading data as bytes from an input source.
- Most of the methods throws I/O Exception
  - int read()
  - int read(byte buf[])
  - int read(byte buf[], int offset, int length)
  - int available()
  - void mark(int marklen)
  - void reset()
  - long skip(long m)
  - void close()

OutputStream

- Abstract class that defines methods to write data as bytes to an output source.
- Most of the methods throws an I/O Exception
  - void write(int c)
  - void write(byte buf[])
  - void write(byte buf[], int offset, int length)
  - void flush()
  - void close()


InputStream
FileInputStream
ByteArrayInputStream
FilterInputStream
SequenceInputStream
ObjectInputStream
DataInputStream
BufferedInputStream
OutputStream
FileOutputStream
ByteArrayOutputStream
FilterOutputStream
ObjectOutputStream
DataOutputStream
BufferedOutputStream
PrintStream

FileInputStream

- Creates an input stream that can be used to read bytes from a file.
- Constructors
  - FileInputStream(String filepath)
  - FileInputStream(File fileobj)

FileOutputStream

- Creates an output stream that can be used to write bytes to a file
- Constructors
  - FileOutputStream(String filepath)
  - FileOutputStream(File obj)
  - FileOutputStream(String Filepath,boolean append)
  - FileOutputStream(File obj,boolean append)

```
import java.io.*;
public class CopyBytes
{
 public static void main(String[] args) throws IOException
{ FileInputStream in = new FileInputStream("test.txt");
  FileOutputStream out = new FileOutputStream("new.txt");
    int c;
    while ((c = in.read()) != -1)
    {      out.write(c); }
     in.close();
     out.close();}}
```

ByteArrayInputStream

- Creates an input stream that uses a byte array as the source.
- Constructors
  - ByteArrayInputStream(byte array[])
  - ByteArrayInputStream(byte array[],int start,int numbytes)

ByteArrayOutputStream

- Implementation of OutputStream that uses a byte array as the destination.
- Constructors
  - ByteArrayOutputStream()
    //buffer of 32 bytes is created
  - ByteArrayOutputStream(int numbytes)
- Buffer size will increase automatically if needed.
- Method
  - byte[] toByteArray()

Filtered Streams

- A filter stream filters data as it's being read from or written to the stream
- To use a filter input or output stream, attach the filter stream to another input or output stream when it is created

DataInputStream

- Enables to read primitive data (int, float, long etc...) from an InputStream instead of only raw bytes.
- Constructor
  - DataInputStream(InputStream in)
- Methods throw IOException
  - final double readDouble()
  - final boolean readBoolean()
  - final int readInt()
  - final char readChar()

DataOutputStream

- Enables to write primitive data to OutputStream instead of only bytes.
- Constructors
  - DataOutputStream(OutputStream outputStream)
- Methods throw IOException
  - final void writeDouble(double value)
  - final void writeBoolean(boolean value)
  - final void writeInt(int value)
  - final void writeFloat(float value)

BufferedInputStream

- When the BufferedInputStream is created, an internal buffer array is created.
- As bytes from the stream are read, the internal buffer is refilled as necessary from the contained input stream, many bytes at a time.
- This is typically much faster than reading a single byte at a time from an InputStream, especially for disk access and larger data amounts.
- Constructor
  - BufferedInputStream (InputStream i)

BufferedOutputStream

- BufferedOutputStream uses an internal buffer to store data.
- It adds more efficiency than to write data directly into a stream.
- So, it makes the performance fast.
- Constructor
  - BufferedOutputStream (OutputStream i)

SequenceInputStream

- The SequenceInputStream creates a single input stream from multiple input sources.

- Constructor
  - SequenceInputStream (InputStream i1,InputStream i2)

ObjectInputStream

- ObjectInputStream is used to read objects from an inputstream of bytes.
- Deserialization is done in this class.
- Constructor
  - ObjectInputStream(Inputstream in)
- Method
  - Object readObject()

ObjectOutputStream

- ObjectOutputStream is used to write objects to an outputstream as series of bytes.
- Process of conversion of an object to byte is known as serialization.
- If an object is to be serialized, the class must implement Serializable interface that does not define any methods in it.
- Constructor
  - ObjectOutputStream(OutputStream ous)
- Method
  - void writeObject(Object ob)

PrintStream

- Provides all of the output capabilities that have been using from System.out.
- Constructors
  - PrintStream(OutputStream outputStream)
  - PrintStream(File fileobj)
- Method
  - PrintStream printf(String format, Object... args)
- printf() method allows to specify the precise format of the data to be written.

Character Stream

- Handles character oriented i/o

Top of the character stream hierarchies are the

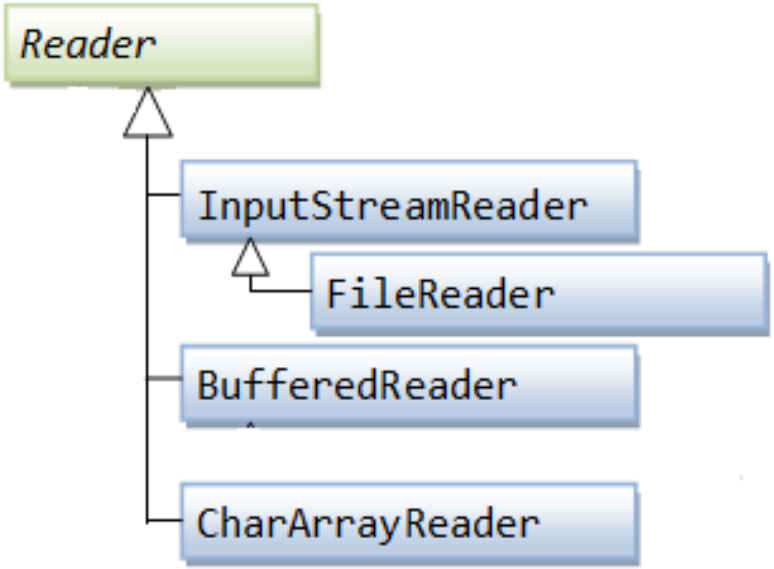
- Reader class
- Writer class

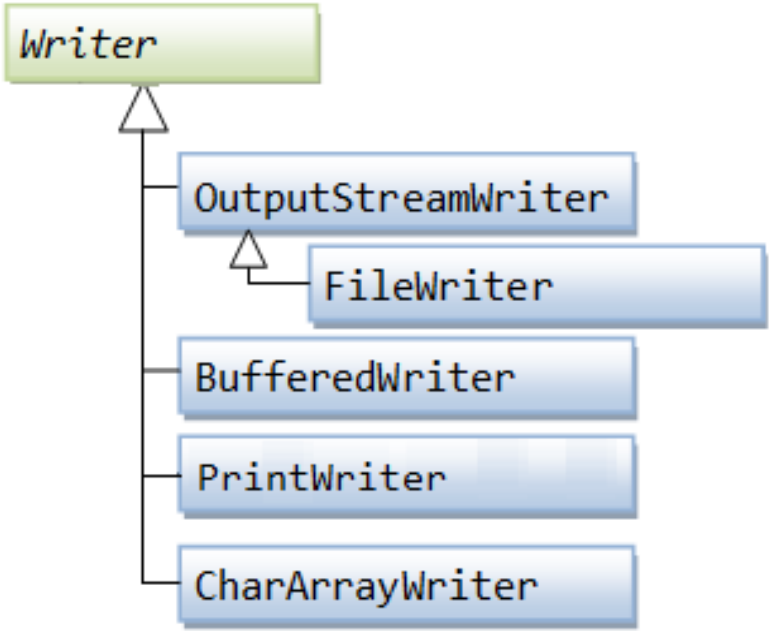
Reader

- Abstract class that defines methods for reading data as characters from an input source.
- All methods throws IOException
  - int read()
  - int read(char ch[])
  - int read(char ch[], int offset, int length)
  - void mark(int marklen)
  - void reset()
  - long skip(long m)
  - void close()

Writer

- Abstract class that defines methods to write data as characters to an output source.
- Most of the methods throws an I/O Exception
  - void write(int ch)
  - void write(char ch[])
  - void write(char ch[], int offset, int length)
  - void write(String s)
  - void write(String s, int offset, int length)
  - void flush()
  - void close()


Reader
InputStreamReader
FileReader
BufferedReader
CharArrayReader


Writer
OutputStreamWriter
FileWriter
BufferedWriter
PrintWriter
CharArrayWriter

CharArrayReader

- The CharArrayReader class is used to read the contents of a character array as a character stream.
- Constructors
  - CharArrayReader(char array[])
  - CharArrayReader(char array[],int start,int numchars)

CharArrayWriter

- Implementation of Writer that uses a character array as the destination.
- Constructors
  - CharArrayWriter()

    //buffer of 32 characters is created
  - CharArrayWriter(int numchars)
- Buffer size will increase automatically if needed.
- Method
  - char[] toCharArray()

InputStreamReader

- This class acts as a bridge from byte streams to character streams.
- The InputStreamReader reads bytes from the InputStream and translates them into characters according to the specified encoding.
- Constructors
  - InputstreamReader(InputStream in)
  - InputStreamReader(InputStream in, String enc)

OutputStreamWriter

- This class acts as a bridge from character streams to byte streams
- OutputStreamWriter will write bytes of data to the output stream after translating the characters according to the specified encoding.
- Constructors
  - OutputStreamWriter(OutputStream in)
  - OutputStreamWriter(OutputStream in, String enc)

FileReader

- Subclass of InputStreamReader.
- Used to read characters from a file.
- Constructors
  - FileReader(String filepath)
  - FileReader(File fileobj)

FileWriter

- Subclass of OutputStreamWriter.
- Used to write characters to a file
- Constructors
  - FileWriter(String filepath)
  - FileWriter(File obj)
  - FileWriter(String Filepath,boolean append)
  - FileWriter(File obj,boolean append)

BufferedReader

- Provides buffering to provide efficient reading of characters.
- Constructor
  - BufferedReader(Reader input)
- Method
  - String readLine()

BufferedWriter

- Provides buffering to provide efficient writing of characters.

- Constructor
  - BufferedWriter(Writer output)

- Method
  - void newLine()

PrintWriter

- Character oriented version of PrintStream.
- Constructors
  - PrintWriter(File F)
  - PrintWriter(Writer output)

Utility classes

Object class

- Super class of all classes
- All classes are to inherit the methods of Object class
- Methods
  - Object clone() :
    - To copy the object equally
  - boolean equals(Object obj) :
    - Compare the object as parameter with the object which calls this method
  - int hashCode() :
    - Calculate the hash code value of the object
  - String toString() :
    - Convert the object into string

Shallow Copy & Deep Copy

- In **Shallow copy**, if the object contains primitive and reference type variables, the cloned object will have an exact copy of all the fields of source object including the primitive type and object references but **the referred object is not copied.**

- In **Deep copy**, if the object contains primitive and reference type variables, the cloned object will have an exact copy of all the fields of source object including the primitive type and object references and **a new copy of the referred object is created**.

Date

- Encapsulates current date and time
- Constructors
  - Date()
  - Date(long date)
    - Arg: Number of milliseconds since Jan 1, 1970
- Does not allow individual date and time components

Date

- Methods
  - boolean after(Date date)
  - boolean before(Date date)
  - long getTime()
  - void setTime(long time)

Calendar

- Converts time in milliseconds to useful components like year, month, day, hour, minute, second

- No constructors
- Integer constants
  - DAY_OF_MONTH,JANUARY,…,SUNDAY.., AM, PM, HOUR, MINUTE,SECOND, YEAR, …

- Methods
  - int get(int calendarField)
  - final Date getTime()
  - final void setTime(Date d)
  - void set(int which,int val)
  - final void set(int year,int month,int day)
  - static Calendar getInstance()
  - abstract void add(int which, int val)

GregorianCalendar

- Concrete implementation of calendar
- Fields : AD and BC
- Constructors
  - GregorianCalendar()
  - GregorianCalendar(int year,int mon, int dayOfMonth)
  - GregorianCalendar(int year,int mon, int dayOfMonth, int hours, int minute,int seconds)
- Method
  - boolean isLeapYear(int year)

SimpleDateFormat

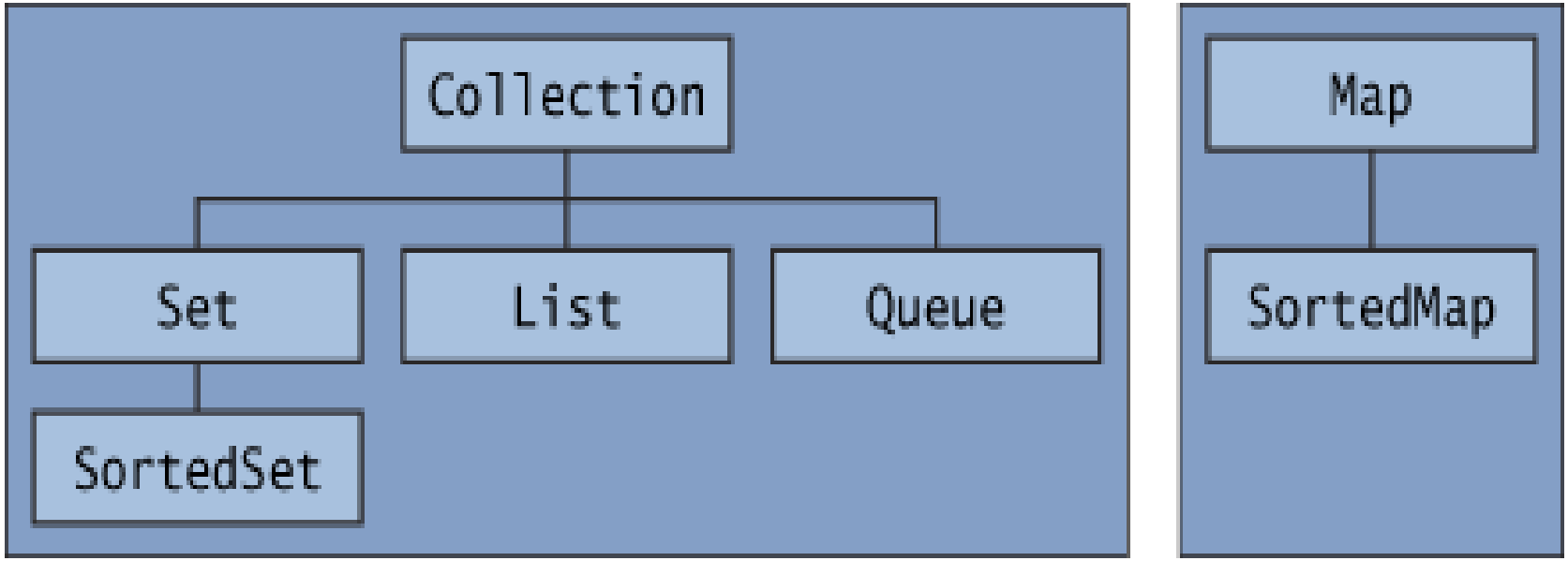
- Available in java.text package
- This class provides methods to format and parse date and time in java.
- Formatting means converting date to string and parsing means converting string to date.

Collections

Collections Framework

- A package that supports a wide range of functionalities
- Collections:
  - group of objects
- Collections framework
  - Represents a unified architecture for storing and manipulating group of objects
- Collection framework contains
  - Interfaces
  - Implementations
  - Algorithms

Interfaces



Collection interface

- Collection represents a group of objects known as its elements

- Declares the core methods that all the collections will have

- public interface Collection<E> {

```
// Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);
    boolean remove(Object element);
    Iterator<E> iterator();
```

```
// Bulk operations
   boolean containsAll(Collection<?> c);
   boolean addAll(Collection<?> c);
   boolean removeAll(Collection<?> c);
   boolean retainAll(Collection<?> c);
   void clear();
 // Array operations
   Object[] toArray();
}
```

List interface

- Ordered collection
- May contain duplicate elements
- In addition to operations inherited from collection, it has methods for
  - Positional access
  - Search
  - Iteration
  - Range-view

- public interface List<E> extends Collection<E> {

```
//Positional access
E get(int index);
E set(int index, E element);
boolean add(E element);
void add(int index, E element);
E remove(int index);
boolean addAll(int index, Collection<?> c);
```

```
// Search
        int indexOf(Object o);
        int lastIndexOf(Object o);
// Iteration
   ListIterator<E> listIterator();
    // Range-view
   List<E> subList(int from, int to); }
```

Set interface

- Collection that cannot contain duplicate elements
- Has only the methods inherited from Collection with the additional constraint that duplicate elements are prohibited
- Contains two general purpose set implementations:
  - HashSet
  - TreeSet

SortedSet interface

- The SortedSet interface extends Set and declares the behavior of a set sorted in ascending order.
  - —E first( );
  - —E last( );
  - —SortedSet headSet(E end);
  - —SortedSet subSet(E start, E end);
  - —SortedSet tailSet(E start);

NavigableSet interface

- A SortedSet extended with navigation methods reporting closest matches for given search targets.
  - —E ceiling(E obj);
  - —E floor(E obj);

Queue

- The Queue interface extends Collection interface and supports ordering on a FIFO (First In First Out) basis.

```
public interface Queue<E> extends Collection<E> {
   boolean offer(E e);
   E element();
   E peek();                    //returns null
   E poll();                    //returns null
   E remove(); }
```

Deque

- It represents a queue where we can insert and remove elements from both ends of the queue.
  - —void addFirst(E e);
  - —void addLast(E e);
  - —E getFirst();
  - —E getLast();
  - —E pollFirst();
  - —E pollLast();

ArrayList

- The ArrayList class implements List interface.
- ArrayList supports dynamic arrays that can grow as needed.
  - ArrayList( )
  - ArrayList(Collection c)
  - ArrayList(int capacity)
- Object[] toArray()
- Reason for converting a collection into an array
  - To obtain faster processing times for certain operations
  - To pass an array to a method that is not overloaded to accept a collection

LinkedList

- The LinkedList class implements List, Queue and Dequeue interfaces.
- The LinkedList stores its items in "containers.“
  - LinkedList()
  - LinkedList(Collection c)

HashSet

- HashSet class implements Set interface
- Stores elements in a Hash table
- Best-performing implementation
- No guarantee in the order of iteration
  - HashSet()
  - HashSet(Collection c)
  - HashSet(int capacity)

TreeSet

- TreeSet class implements NavigableSet interface
- Stores elements in tree structure
- Orders elements based on their values
- Slower than HashSet
  - TreeSet()
  - TreeSet(Collection c)
  - TreeSet(Comparator comp)

PriorityQueue

- Implements Queue interface
- Creates a queue based on queue's comparator

  - PriorityQueue()
  - PriorityQueue(Collection c)
  - PriorityQueue(Comparator comp)

Traversing collections

- There are three ways to traverse through a collection
  1) with the **for–each** construct
  2) by using **Iterator**
  3) by using **ListIterator**

Iterator interface

- Traverses a collection from start to finish and go through each element of the collection
  - public interface Iterator<E>

```
{
boolean hasNext();
 E next();
 void remove();
}
```

ListIterator interface

- Allows bidirectional traversal of a List
- public interface ListIterator<E> extends Iterator<E>

```
{   boolean hasNext();
    E next();
    boolean hasPrevious();
    E previous();
    int nextIndex();
    int previousIndex();
    void remove();
    void set(E e);
    void add(E e); }
```

Enumeration Interface

- Defines the methods by which you can enumerate(obtain one at a time) the elements.
- Methods in Enumeration
  - boolean hasMoreElements()
  - Object nextElement()

Vector

- Implements a dynamic array.
- Similar to ArrayList.
- Vector is synchronized.
- From java2 onwards Vector implements the List interface.

Constructors

- Vector()
- Vector(int capacity)
- Vector(int capacity,int incr)
- Vector(Collection c)

Vector defines three protected data members.

- int capacityincrement
- int elementCount
- Object[] elementData

Methods defined by Vector

—void addElement(Object element)

—boolean removeElement(Object element)

—Enumeration<E> elements()

—int capacity()

—boolean contains(Object element)

—Object elementAt(int index)

—Object firstElement()

—Object lastElement()

Maps

- Map is an object that stores associations between keys and values or key/value pairs
- Keys are unique
- Values may be duplicated
- Map implementation
  - HashMap
  - TreeMap

Map Interface

- Maps unique keys to values
- Key : object used to retrieve a value at a later date

```
public interface Map<K,V> {
boolean containsKey( Object k );
boolean containsValue(Object v);
int size();
boolean isEmpty();
V put(K key, V value);
void putAll(Map<K,V> m)
```

V get(Object key);

V remove(Object key);

default V replace(K k,V v); (Added by JDK8)

void clear();

Set<Map.Entry<K,V>> entrySet();

Set<K> keySet();

Collection<V> values();}

Map.Entry Interface

- Represents a single Map entry
- Useful Methods:
  - Object getKey()
  - Object getValue()
  - Object setValue(Object val)

SortedMap Interface

- Extends Map
- Maintains a sorted list of keys
- Useful Methods:
  - Comparator comparator();
  - Object firstKey();
  - Object lastKey();
  - SortedMap subMap(K start,K end);
  - SortedMap headMap(K end);
  - SortedMap tailMap(K start);

NavigableMap Interface

- Extends SortedMap
- Retrieves entries based on the closest match to a given key or keys
- Useful Methods:
  - Map.Entry ceilingEntry(K obj);
  - Map.Entry floorEntry(K obj));
  - K ceilingKey(K obj);
  - K floorKey(K obj);

HashMap class

- Implements Map interface
- Uses hash table to store the map
- Declaration
  - class HashMap<k,v>
- Constructors
  - HashMap()
  - HashMap(int capacity) //default 16
  - HashMap(Map<K,V> m)

TreeMap Class

- Implements NavigableMap interface
- Creates map stored in tree structure
- Elements will be sorted in ascending order
- Declaration
  - class TreeMap<K,V>
- Constructors
  - TreeMap()
  - TreeMap(Comparator comp)
  - TreeMap(Map< K, V> m)
  - TreeMap(SortedMap<K, V> sm)

Collection Algorithms

- Algorithms are defined as static methods within **Collections** class
- Common algorithms
  - copy(List list1,List list2), emptyList(), emptyMap(), frequency(Collection<?> c, object obj), reverse(List l), reverseOrder(Comparator com), sort(List l,Comparator comp), shuffle(List l),min(Collection c)
- Static variables
  - EMPTY_SET, EMPTY_LIST, EMPTY_MAP

Comparator

- Determines what the ‘sorted order ‘ is for TreeSet and TreeMap
- Methods
  - compare()
    - Compares two elements for order
    - int compare(T obj1, T obj2):Returns 0, +ive or –ive value
  - equals()
    - Tests whether the invoking comparator equals an object
  - reversed()
    - Returns a comparator that runs in reverse

Comparable

- **Comparable** in Java is an object to compare itself with another object.
- The java.lang.Comparable interface should be implemented by a class in order to compare its instances.
- Method
  - int compareTo(T obj)

Generics

- Prior to the JDK 5.0 release, when a Collection is created , any object can be put in it.

```
ArrayList myList = new ArrayList( );
myList.add(new Integer(10));
myList.add("Hello, World");
```

- Getting items out of the collection required to use a casting operation:

```
Integer myInt = (Integer)myList.iterator().next();
```

- If we accidentally cast the wrong type, the program would successfully compile, but an exception would be thrown at runtime.

What is Generics?

- Collections can store Objects of any Type.
- Generics restricts the Objects to be put in a collection.
- Generics ease identification of runtime errors at compile time.
- Generic programming = programming with classes and methods parameterized with types.

Generics

- Generics allows to specify, at compile-time, the types of objects to be stored in a Collection.
- When we add and get items from the list, the list already knows what types of objects are supposed to be acted on.
- No need to cast anything.
- The "<>" characters are used to designate what type is to be stored.
- If wrong type of data is provided, a compile-time exception is thrown.

Generics (cont'd)

- Example:

```
import java.util.*;
public class First {
    public static void main(String args[]) {
      ArrayList<Integer> myList = new ArrayList<Integer>( );
      myList.add(10);  // OK ???
      myList.add("Hello, World"); // OK ???
    }
}
```

Generic Class

```
class Pair <T>
{
        public T first;
        public T second;

        public Pair (T f, T s) { first = f; second = s; }
        public Pair ( ) { first = null; second = null; }
    }
```

- The generic class can be instantiated by substituting actual types for type variables, as: Pair <String>
- The result is a class with a constructor

  public Pair (String f, String s), etc . .
- The instantiated generic class can be used as it were a normal class:

  Pair <String> pair = new Pair <String> ("1","2");

Class with multiple type parameters

- Multiple type parameters are allowed

```
class Pair <T, U> {
    public T first;
    public U second;

    public Pair (T x, U y) { first = x; second = y; }
    public Pair ( ) { first = null; second = null; }
}
```

Generic Methods

- Generic methods can be defined both in ordinary classes and in generic classes

```
class Algorithms {
        public static <T> T getMiddle (T [ ] a) {
                return a[ a.length/2 ];
        }
}
```

- Calling generic method

```
String[] names={"Hello","Good"};
String s = Algorithms.<String>getMiddle (names);
```

- In most cases, the compiler infers the type:

```
String s = Algorithms. getMiddle (names);
```

Wildcards

- Wildcards help in allowing more than one type of class in the Collections.
- We can set an upperbound and lowerbound for the Types which can be allowed in the collection.
- The bounds are identified using a ? Operator which means ‘an unknown type’.

Upperbound

- Example:

If we want to write a method that works on List < Integer >, List < Double >, and List < Number > , we can achieve it using an upper bounded wildcard.

List<? extends Number> means the given list contains objects of some unknown type which extends the Number class

Lowerbound

- List<? super Number> means that the given list contains objects of some unknown type which is superclass of the Number class

Multithreading

Multithreading Introduction

- Multiprocessing: ability of an operating system to execute more than one process simultaneously on a multiprocessor machine.
- Multitasking : ability of an operating system to execute more than one task simultaneously on a single processor machine.
- Multithreading : ability of an operating system to execute the different parts of a program called threads at the same time.

Process Vs Threads

- **Process :**
  - Program in execution
  - Has its own memory space, variables etc.,
  - Heavy-weight process
- **Thread:**
  - A single process may have multiple threads
  - It is an execution path within a program
  - All threads share the same memory space, and variables, and can communicate with each other directly
  - Light-weight process

Creating a thread

- Two ways:
  - Create a thread by extending **Thread** class
    - i.e create a subclass of java.lang.Thread
  - Create a thread by implementing **Runnable** interface
    - Use the java.lang.Runnable interface

Thread class

- Primary methods:
  - **start ()** :

    prepares a thread to run
  - **run()** :

    actually performs the work of a thread

Thread Life cycle

- New State

  create an instance of Thread class

- Ready state

  ready for execution, waiting for CPU access

- Running state

  when being executed

- Waiting state

  waiting for some action to happen

- Dead state

  when thread has finished execution

Thread class: methods

- void setName(String s)
- String getName( )
- void setPriority(int p)
- int getPriority()
- boolean isAlive()
- static void sleep(long ms) throws InterruptedException
- static Thread currentThread()

Thread creation using Thread class

- Create a class by extending java.lang.Thread
- Override run() method from Thread class
- Keep thread task inside run method
- Create an instance to your thread class
- Call start() method from thread class instance

Contd..

```
class Mythread extends Thread {
    public void run() {
            ……..
            ……..
}   }
class ThreadManager{
    public static void main(String args[]) {
        MyThread t1 = new MyThread();
        t1.start();} }
```

Creating a thread by Runnable interface

- Write a class by implementing Runnable interface
- Implement run() method
- Keep thread task inside a run method
- Create an instance of your class
- Create an instance to a thread class by passing instance of your class
- Call start method by using thread class instance

```
class Mythread1 implements Runnable {
    public void run() {
            ……..
            }   }
class ThreadManager{
    public static void main(String args[]) {
         MyThread1 t1 = new MyThread1();
        Thread thread1 = new Thread(t1);
         thread1.start();
} }
```

join( )

- final void join( ) throws InterruptedException
- This method waits until the thread on which it is called terminates.
- Its name comes from the concept of the calling thread joining the invoked thread after completion.

Synchronization

- When two or more threads try to access the same resource at the same time, some way should be ensured that the resource will be used by only one thread at a time.
- Synchronization is the capability to control the access of multiple threads to any shared resource.
- Problem is prevented by using monitors
- Monitors are implemented by providing each object with a lock

- Achieved by simply adding the keyword **synchronized** to the declaration
  - Ensures that only one thread executes a protected section of the code at a time
  - Data changed by one thread is visible to other threads

```
synchronized(object)
{
// statements to be synchronized
}
```

Interthread Communication

- wait()
  - tells calling thread to give up the monitor and go to sleep
- notify()
  - wakes up a single thread that called wait() on the same object
- notifyAll()
  - wakes up all the threads that called wait() on the same object

Inner Classes

Nested Classes

- Java **Nested class** is a class that is declared inside another class.
- Nested classes enable to logically group classes and increase encapsulation.
- There are several types of Nested classes
  - Static nested classes
  - Non-static nested classes
  - Local classes
  - Anonymous classes

Static Nested Class

- **Static Nested class** can be accessed without creating an object of the outer class.

- Similar to static members, these belong to their enclosing class, and not to an instance of the class.

- They have direct access to static members in the enclosing class.

Non-Static Nested Class

- They are also called **Inner classes**.
- Similar to instance variables and methods, inner classes are associated with an instance of the enclosing class.
- They have access to all members of the enclosing class, regardless of whether they are static or non-static.
- They can only define non-static members.

Local Class

- It is a special type of Inner class in which the **class is defined inside a method**.
- The outer class method contains the inner class.
- They have access to both static and non-static members in the enclosing context.
- They can only define instance members.

Anonymous Class

- Inner classes have no name.
- They have access to both static and non-static members in the enclosing context.
- They can only define instance members.
- They are the only type of nested classes that cannot define constructors and are useful when we need only one object of the class.
- Anonymous classes is used to define an implementation of an interface or an abstract class without having to create a

Lambda Expression

- A **lambda expression** is an anonymous function, it's a method without a declaration, i.e., access modifier, return value and name.

- Provides a way to represent one method interface using an expression.

- Interface with only one abstract method is referred to as **Functional Interface**.

Lambda Syntax

- The basic syntax of a lambda is either:
  - *(parameters) ->expression* or
  - *(parameters) ->{ statements; }*
- A lambda can have zero or more parameters separated by commas and their type can be explicitly declared or inferred from the context.
- Parenthesis are not needed around a single parameter.
- ( ) is used to denote zero parameters.
- The body can contain zero or more statements.
- Curly braces are not needed around a single-statement body.

Lambda Examples

Eg:(n) -> n % 2 == 1

- n is the input parameter
- n % 2 == 1 is the expression
- **n -> n % 2 == 1** is read like: "input parameter named n goes to anonymous function which returns true if the input is odd".

Eg:(int x, int y) -> { return x + y; }

x -> x * x

( ) -> x