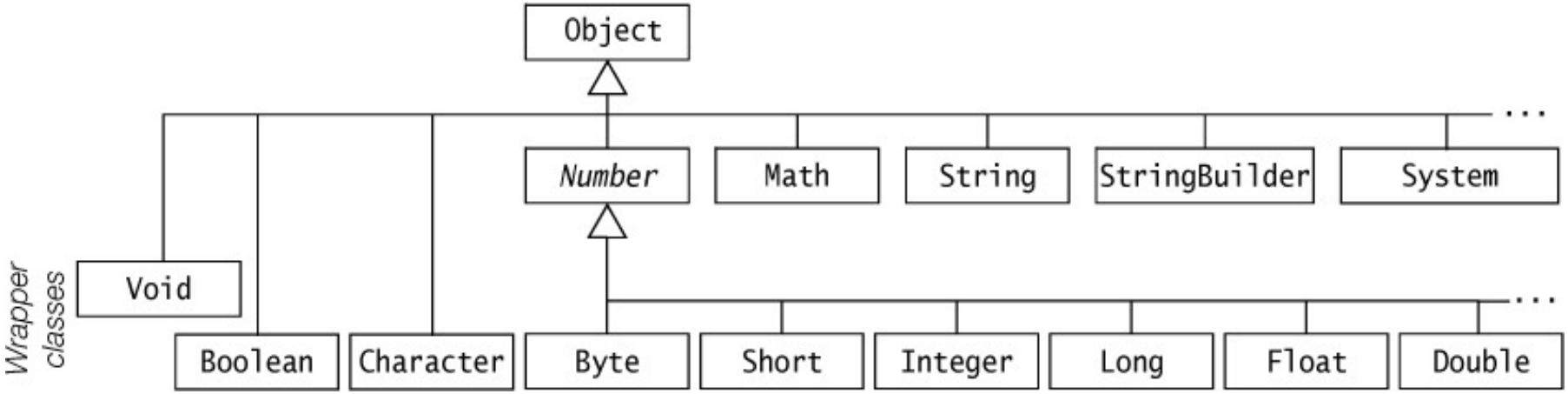
Java APIs

Java API

- Java.lang package
  - Indispensable when programming in Java
  - Automatically imported into every source file at compile time.
  - Contains
    - Object class
    - Type wrapper classes
    - classes essential for interacting with the JVM
    - Threads
    - Exceptions

Java API

- Java.lang package
  - Class Heirarchy



Java API

- java.lang package
  - Object class

`Object clone()`	Create and return a copy of the current object.
`boolean equals(Object obj)`	Determine if the current object is equal to the object identified by `obj`.
`void finalize()`	Finalize the current object.
`Class<?> getClass()`	Return the current object's `Class` object.
`int hashCode()`	Return the current object's hash code.
`void notify()`	Wake up one of the threads that are waiting on the current object's monitor.
`void notifyAll()`	Wake up all threads that are waiting on the current object's monitor.
`String toString()`	Return a string representation of the current object.

Java API

- java.lang
  - Wrapper classes
    - Primitive values in Java are not objects.
    - To manipulate these values as objects, the java.lang package provides a wrapper class for each of the primitive data types

Primitive Data Types	Wrapper Classes
int	Integer
float	Float
double	Double
boolean	Boolean (Added from 1.5)
short	Short
byte	Byte
char	Character
long	Long

Java API

- java.lang
  - Wrapper Classes
    - Creating Wrapper Objects from primitives

```
Character charObj1   = Character.valueOf('\n');
Boolean   boolObj1   = Boolean.valueOf(true);
Integer   intObj1    = Integer.valueOf(2020);
Double    doubleObj1 = Double.valueOf(3.14);
```

Java API

- Java.lang
  - Converting Numeric Wrapper Objects to Numeric Primitive Types

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

Java API

- java.lang
  - Wrapper Classes
    - Converting Primitive Values to Strings

```
String charStr2   = Character.toString('\n');  // "\n"
String boolStr2   = Boolean.toString(true);    // "true"
String intStr2    = Integer.toString(2020);    // "2020"
String doubleStr2 = Double.toString(3.14);     // "3.14"
```

Java API

- Java.lang
  - Wrapper Classes
    - Converting Strings to Numeric Values

```
byte   value1 = Byte.parseByte("16");
int    value2 = Integer.parseInt("2020");
// NumberFormatException.
int    value3 = Integer.parseInt("7UP");
double value4 = Double.parseDouble("3.14");
double value5 = Double.parseDouble("Infinity");
```

Java API

- Java.lang
  - Wrapper Classes
    - Converting Strings to Boolean Values

```
boolean b1 = Boolean.parseBoolean("TRUE");      // true.
boolean b2 = Boolean.parseBoolean("true");      // true.
boolean b3 = Boolean.parseBoolean("false");     // false.
boolean b4 = Boolean.parseBoolean("FALSE");     // false.
```

Java Generics

Amit Gulati
amit.gulati@gmail.com

Introduction

- Generics
  - Java version 5 introduced generics into the programming language.
  - Declare and use type-agnostic classes and interfaces.
  - Similar to templates classes in C++.
  - Generics promote type safety in the context of the collections classes
    - Better collection classes

Introduction

- Collections prior to Java 5
  - Prior to java 5, Collection classes in Java were not generic.
  - Collection allowed programmer to store any type of object in a collection.

```
public static void main(String[] args)
{
    List employees = new ArrayList();
    employees.add(new Employee("John Doe"));
    employees.add(new Employee("Jane Doe"));
    employees.add("No Doe");
    Iterator iter = employees.iterator();
    while (iter.hasNext())
    {
        Employee emp = (Employee) iter.next();
        System.out.println(emp.getName());
    }
}
```

Programmer error

```
java.lang.ClassCastException: java.lang.String cannot be cast to Employee
        at GenericsExample.main(GenericsExample.java:28)
```

Introduction

- Collections post java 5 (with generics)
  - Generics add type safety to collections.

```
public static void main(String[] args)
{
    List<Employee> employees = new ArrayList<Employee>();
    employees.add(new Employee("John Doe"));
    employees.add(new Employee("Jane Doe"));
    employees.add("No Doe");
    Iterator<Employee> iter = employees.iterator();
    while (iter.hasNext())
    {
        Employee emp = iter.next();
        System.out.println(emp.getName());
    }
}
```

Compiler catches programmer error

Generic Types

- A generic type is a class or interface that introduces a <u>family of parameterized types</u> by declaring a formal type parameter list.

```
class identifier<formal_type_parameter_list> {}
interface identifier<formal_type_parameter_list> {}
```

- Example
  - List<E> is a generic type
    - List is an interface
    - E is **type parameter** that identifies the list's element type.
  - Map<K, V> is a generic type
    - Map is an interface
    - K and V are **type parameters** that identify the map's key and value types.

Generic Types

- Parameterized Types
  - Parameterized types instantiate generic types.
  - Each parameterized type replaces the generic type's **type parameters** with **type names**.
  - From a Generic Type you can create a family of parameterized types
    - List<String>, List<Employee>, etc.
    - Map<String, String> etc.

E, **type parameter** has been replaced with **type names** String & Employee

Generic Types

- Type Parameter Bounds
  - Unbound Type Parameters
    - List<E> - E type parameter
    - Map<K, V> - K and V type parameters
    - You can pass any actual type argument to an unbounded type parameter.
  - It is sometimes necessary to restrict the kinds of actual type arguments that can be passed to a type parameter.
  - For example,
    - You might want to declare a class whose instances can only store instances of classes that subclass an abstract *Shape* class.

Generic Types

- Type Parameter Bounds
  - Setting an Upper Bound
    - To restrict actual type arguments, you can specify an upper bound .
    - A type that serves as an upper limit on the types that can be chosen as actual type arguments.
    - The upper bound is specified via reserved word **extends** followed by a type name.
    - For example
      - ShapesList<E **extends** Shape> *identifies Shape as an upper bound*.
        - You can specify ShapesList<Circle>, ShapesList<Rectangle>, but not ShapesList<String> because String is not a subclass of Shape .

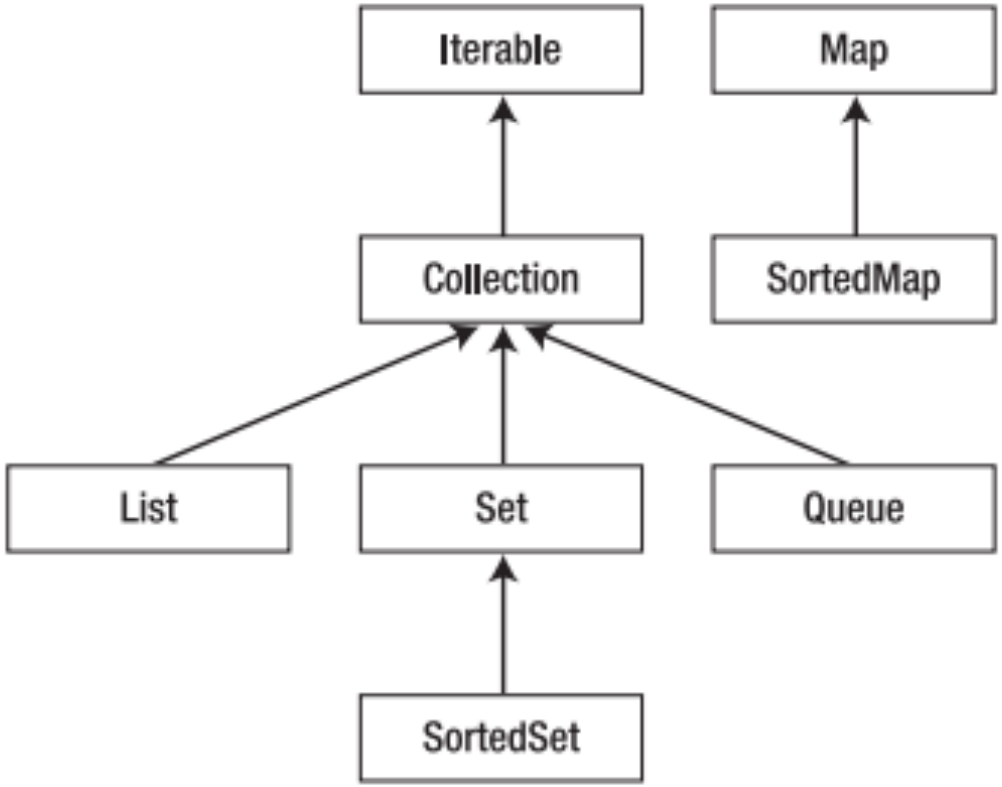
Collections

Introduction

- Collection
  - A collection is a data structure—actually, an object—that can hold references to other objects.
- Collections Framework
  - Contains *prebuilt* generic data structures
  - Consists of three components
    - Core interfaces
      - Interfaces used for working with group of objects (independent of implementation)
    - Implementation classes
      - Concrete classes that implement Core Interfaces
    - Utility classes
      - Helper classes

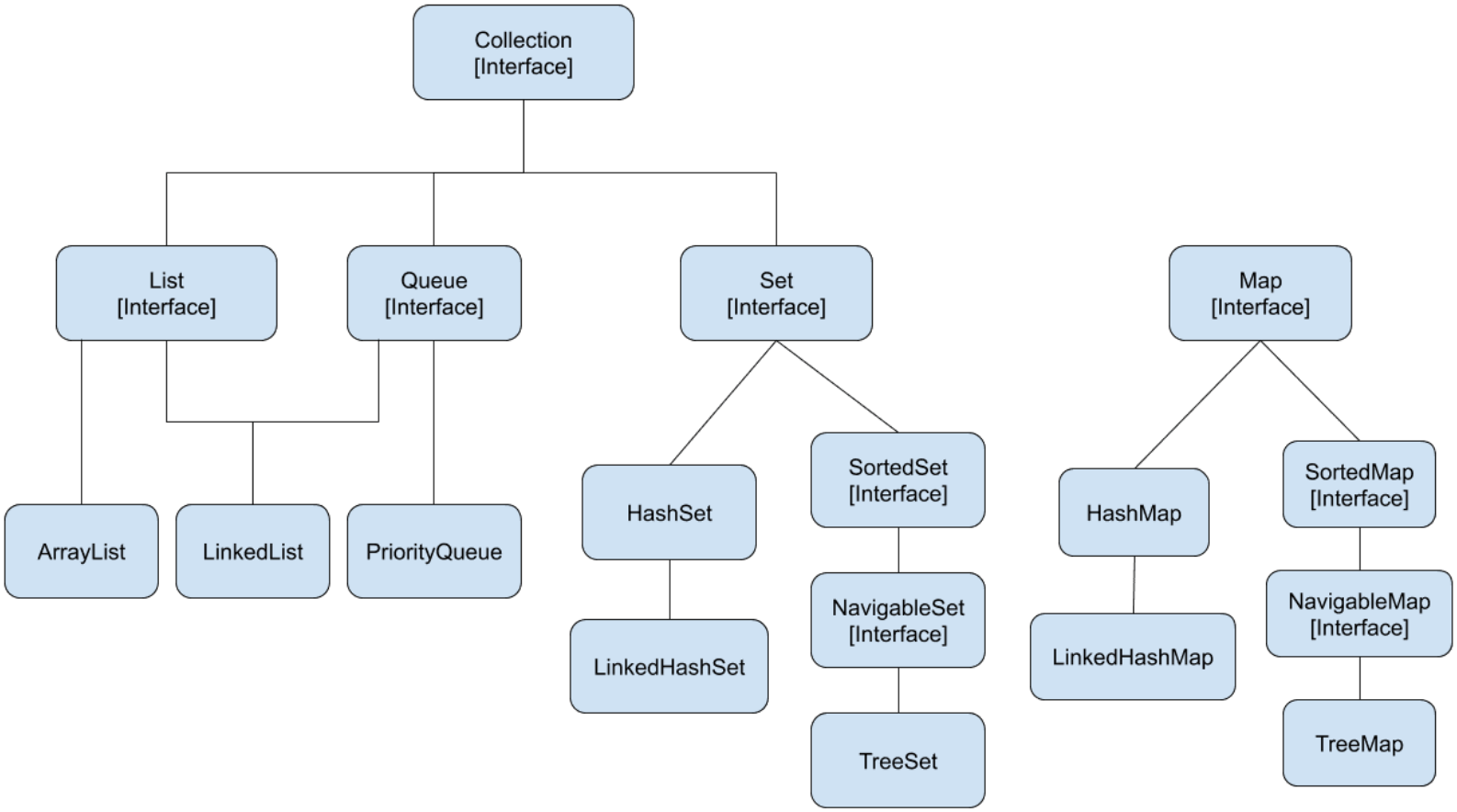
Collection Framework

- Core Interfaces
  - Interfaces defined in Collection Framework



Collection Framework

- Implementation classes
  - Names of the classes ends in core interface name.



Collection Framework

- Core Interfaces
  - **Collection** interface

```
public interface Collection<E> extends Iterable<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object o);

    Iterator<E> iterator();
    boolean add(E e);
    boolean remove(Object o);
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    void clear();
    boolean equals(Object o);
    int hashCode();
}
```

Collection Framework

- Core Interfaces
  - **Iterator** interface

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
}
```

Collection Framework

- **Iterator** Interface

```
ArrayList<String> workingDays = new ArrayList<String>();
workingDays.add("Monday");    workingDays.add("Tuesday");
workingDays.add("Wednesday"); workingDays.add("Thursday");
workingDays.add("Friday");

Iterator<String> iter = workingDays.iterator();
for(int i = 0; iter.hasNext(); ++i)
{
    String str = iter.next();
    System.out.println(str);
}
```

Collection Framework

- Iterator Interface
  - Enhanced For Loop (Syntactic Sugar)

```
ArrayList<String> workingDays = new ArrayList<String>();
workingDays.add("Monday");     workingDays.add("Tuesday");
workingDays.add("Wednesday"); workingDays.add("Thursday");
workingDays.add("Friday");

for(String str : workingDays)
{
    System.out.println(str);
}
```

Collection Framework

- Boxing and Unboxing
  - In Java, collections can only contain Java types.
    - Primitive types cannot directly be stored in collection classes.
  - Primitive Types must be boxed inside Java types and then placed in collection objects.
  - Each primitive type has a corresponding type-wrapper class (in package java.lang).
    - Boolean, Byte, Character, Double, Float, Integer, Long and Short.

Collection Framework

- Boxing and Unboxing
  - A boxing conversion converts a value of a primitive type to an object of the corresponding type-wrapper class.
  - An unboxing conversion converts an object of a type-wrapper class to a value of the corresponding primitive type. `
  - These conversions are performed automatically—called autoboxing and auto-unboxing.

```
// create integerArray
Integer[] integerArray = new Integer[5];
integerArray[0] = 10;
// get int value of Integer
int value = integerArray[0];
```

Collection Framework

- **List** interface
  - A list is an ordered collection, which is also known as a sequence. Elements can be stored in and accessed from specific locations via integer indexes.

void **add**(int *index*, E e)

E **remove**(int *index*)

boolean **addAll**(int index, Collection<? extends E> *c*)

E **get**(int *index*)

E **set**(int index, E e)

int **indexOf**(Object o)

ListIterator<E> **listIterator**()

Collection Framework

- **ListIterator** interface
  - Methods of ListIterator
    - Move to Next Item in List and return the Object

      E **next**()

    - Check if there is another object in the list (moving forward)

      boolean **hasNext**()

    - Check if there is another object in the list (moving backward)

      boolean **hasPrevious**()

    - Return the previous item in the list

      E **previous**()

Collection Framework

- **ArrayList** class
  - Provides a list implementation that is based on an internal array.
  - Access to the list's elements is fast.
  - Insertions and deletions of elements is slow.
  - Creating ArrayList

ArrayList()

ArrayList(Collection<? extends E> *collection*)

collection, Collection of objects that inherit from E to be inserted into the newly created ArrayList.

ArrayList(int capacity)

capacity, Initial capacity.

Collection Framework

- **LinkList** class
  - Provides a list implementation that is based on linked nodes.
  - Access to the list's elements is slow.
  - Insertions and deletions of elements are fast.
  - Creating LinkList

LinkedList()

LinkedList(Collection<? extends E> *collection*)

Collection Framework

- Map
  - A map stores key/value pairs.
    - You can find a value if you provide the key.
  - Quickly find an existing element
  - Cannot contain duplicate keys.
  - Each key can map to at most one value.

Collection Framework

- **Map**<K,V> Interface
  - Methods of Map<K, V> Interface
    - Check if a key is present in the map

```
boolean containsKey(Object key)
```

key, Key to check.

    - Check if a value is present in the map

```
boolean containsValue(Object value)
```

key, Value to check.

    - Get Value for a key

```
V get(Object key)
```

V, Object associated with a key.
key, Key for which value is required

Collection Framework

- Map<K,V> Interface
  - Methods of Map<K, V> Interface
    - Insert a key-value pair into the map

```
V put(K key,V value)
```

key, Key to insert.

value, Value to insert.

    - Remove a value for a given key

```
V remove(Object key)
```

key, key for the key-value pair to remove.

Collection Framework

- **HashMap** class
  - Hashes the keys
  - Hashing is usually a bit faster
  - Preferred choice if you don't need to visit the keys in sorted order.

Collection Framework

- **Set** interface

```
public interface Set<E> extends Collection<E> {

}
```

  - The Set interface is identical to the Collection interface
  - **add** method of a set only allows unique values to be added.
  - **equals** method returns true if two sets have the same elements, but not necessarily in the same order.

Collection Framework

- **TreeSet** class
  - The TreeSet class provides a set implementation that is based on a tree data structure.
  - Elements are stored in sorted order.
  - Accessing these elements is somewhat slower than with the other Set implementations (which are not sorted) because links must be traversed.

Collection Framework

- TreeSet
  - Creating TreeSet

TreeSet()

TreeSet(Collection<? extends E> *collection*)

collection, Objects in the collection are added to the Set.

TreeSet(Comparator<? super E> *comparator*)

comparator, Comparator Object to use for sorting.

Collection Framework

- HashSet
  - Provides a set implementation that is backed by a hashtable data structure.
  - HashSet is much faster than TreeSet
  - Creating HashSet

HashSet()

HashSet(Collection<? extends E> *collection*)

HashSet(int *initialCapacity*)

Java Packages

Java Package

- Introduction
  - Java supports the partitioning of top-level types into multiple namespaces
    - Better organize types
    - Prevent name conflicts.
  - Java uses packages to accomplish these tasks.
- Java Package
  - A package is a unique namespace.
    - Container for top-level classes, other top-level types, and sub-packages.
  - Used for logically grouping types

Java Package

- Package type access
  - Only types that are declared public can be accessed from outside the package
  - Constants, constructors, methods, and nested types that are described in a class's interface must be declared public to be accessible from beyond the package
- Package Naming
  - Every package has a name, which must be a non-reserved identifier.
  - Package name must be unique.
  - The convention in choosing package name is to use the reverse domain name notation.

Java Package

- Adding a java type to package
  - **package** statement
    - Identifies the package in which a source file's types are located.
    - Consists of reserved word **package** , followed by a member access operator–separated list of package and sub-package names
    - Only one package statement can appear in a source file.
    - When it is present, nothing apart from comments must precede this statement.

Static Import

Introduction

- Static method and variable access

```
package org.training;

public class StaticClass {
    public static int staticVariable = 100;

    public static void staticMethod()
    {
        System.out.println(staticVariable);
    }
}
```

- Another class that wants to access “StaticClass” static members.

```
import org.training.StaticClass;

public class StaticCaller {

    public StaticCaller(){
        int val = StaticClass.staticVariable;
        StaticClass.staticMethod();
    }
}
```

Always have to use the class name to access static method or variable

Introduction

- Static Import

```
import static org.training.StaticClass.*;

public class StaticCaller {

    public StaticCaller(){
        int val = staticVariable;
        staticMethod();
    }
}
```

Working with File System

File System

- **java.io** package contains classes that allow programs to work with the File System.
  - Primary classes for File System access
    - File
    - FilePermission
- **File** class
  - Java offers access to the underlying platform's available file system via **File** class
  - Abstract representation of file and directory pathnames.
  - Instances of the File class are immutable.

File System

- Path Names
  - An *abstract pathname* has two components:
    - An optional system-dependent *prefix* string, such as a disk-drive specifier.

      Example:

      "/" for the UNIX root directory, or

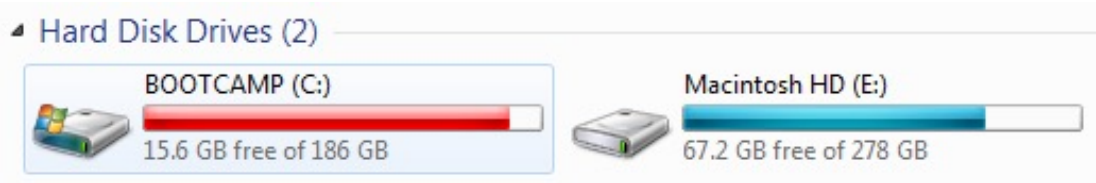
      Drive specifier i.e “c:\” or “d:\” for Microsoft Windows.

    - A sequence of zero or more string *names* separated by path separator.

Windows Path:		c:\Users\guest\hello.txt
Unix Path	:	/usr/home/guest/hello.txt

File System

- Linux/Unix File System
  - Root based file system.
  - Single directory "/" is the root directory.
  - All other file systems are mounted under root.
- Windows File System
  - Associates separate file system with each active disk drive partition.
  - Example:



File System

- **File** class
  - Method used to get root directories of available filesystems

public **static** File[] **listRoots**()

File[], An array of File objects representing root directories.

```
public class DumpRoots
{
   public static void main(String[] args)
   {
      File[] roots = File.listRoots();
      for (File root: roots)
         System.out.println(root);
   }
}
```

File System

- **File** class
  - Creating an Instance of File

public **File**(String *pathname*)

pathname, A pathname string.

public **File**(String *parent*, String *child*)

parent, If parent is not null, parent string is taken to denote a directory.

child, child pathname string is taken to denote either a directory or a f file.

public **File**(File *parent*, String *child*)

parent, If parent is not null, parent File object is taken to denote a directory.

child, child pathname string is taken to denote either a directory or a f file.

File System

- **File** class
  - Some Methods for working with File path name
    - Method to get absolute form of this File object's pathname.

```
public String getAbsolutePath()
```

    - Method to get name of the file or directory denoted by this File object's pathname

```
public String getName()
```

    - Method to get parent pathname string of this File object's pathname

```
public String getParent()
```

File System

- **File** class
  - Example

```
File file = new File(args[0]);
System.out.println("Absolute path = " + file.getAbsolutePath());
System.out.println("Canonical path = " + file.getCanonicalPath());
System.out.println("Name = " + file.getName());
System.out.println("Parent = " + file.getParent());
System.out.println("Path = " + file.getPath());
System.out.println("Is absolute = " + file.isAbsolute());
```

File System

- **File** class
  - Methods in File class to get basic information about the File
    - Method to check is physical file represented by File object exists

```
public boolean exists()
```

    - Methods to check if the file represented by File object has read, write, execute permissions

```
public boolean canRead()
```

```
public boolean canWrite()
```

```
public boolean canExecute()
```

File System

- **File** class
  - Methods in File class to get basic information about the File
    - Methods to check if File object represents a file or directory

```
public boolean isDirectory()
```

```
public boolean isFile()
```

    - Method to return the size of file in bytes

```
public long length()
```

File System

- **File** class
  - Method that returns list of file and directory names in a directory

public String[] **list**()

String[], An array of strings naming the files and directories in the directory denoted by this abstract pathname.

The array will be empty if the directory is empty.

The array will be null if this abstract pathname does not denote a directory.

```
File file = new File("c:\\Users\\gulati\\Desktop\\shared");
String[] filesstr = file.list();
for(String f : filesstr )
{
    System.out.println(f);
}
```

```
flags-32
jdk-7u4-windows-i586.exe
jdk-7u4-windows-x64.exe
```

File System

- **File** class
  - Method that returns a list of files in directory that satisfy a specified file name filter

public String[] **list**(FilenameFilter *filter*)

```
File file = new File("c:\\Users\\gulati\\Desktop\\shared");
String[] names = file.list(new FilenameFilter() {

    @Override
    public boolean accept(File arg0, String arg1) {
        boolean retval = false;
        if(arg1.endsWith(".exe"))
            retval = true;
        return retval;
    }
});
```

File System

- **File** class
  - Method that returns a list of File objects representing contents of directory

public File[] **listFiles**()

```
File[] files = file.listFiles();
for(File f : files )
{
    System.out.println(f);
}
```

```
c:\Users\gulati\Desktop\shared\flags-32
c:\Users\gulati\Desktop\shared\jdk-7u4-windows-i586.exe
c:\Users\gulati\Desktop\shared\jdk-7u4-windows-x64.exe
```

File System

- **File** class
  - Method that creates a new file

```
public boolean createNewFile()
```

boolean, true if file created successfully.

Atomically create a new, empty file named by this File object's pathname if and only if a file with this name does not yet exist.

  - Method that creates a new directory

```
public boolean mkdir()
```

File System

- **File** class
  - Method that creates a new directory and any necessary intermediate directories named by this File object's pathname

```
public boolean mkdirs()
```

  - Method used for deleting files and directories

```
public boolean delete()
```

If file path specifies a directory, it should be empty before it can be deleted.

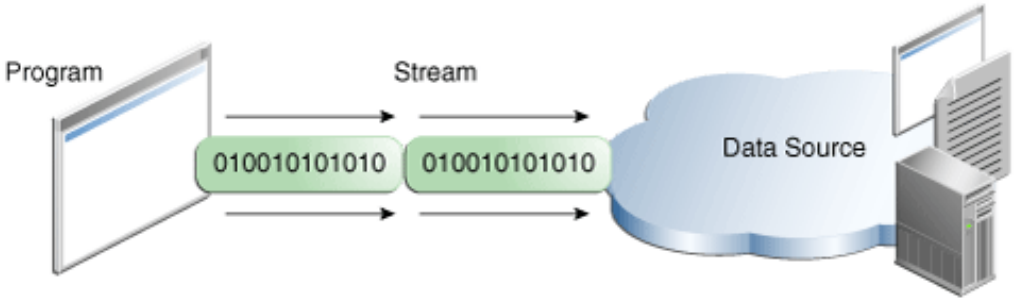
Java I/O

Introduction

- I/O Streams
  - Represents a stream of data flowing into a program or out from a program.
  - Source and destination for IO stream includes
    - Disk files
    - Memory Arrays
    - Networks
    - Etc.
  - Input Stream
    - A program uses an *input stream* to read data from a source.

Introduction

- I/O Streams
  - Output Stream
    - A program uses an *output stream* to write data to a destination.



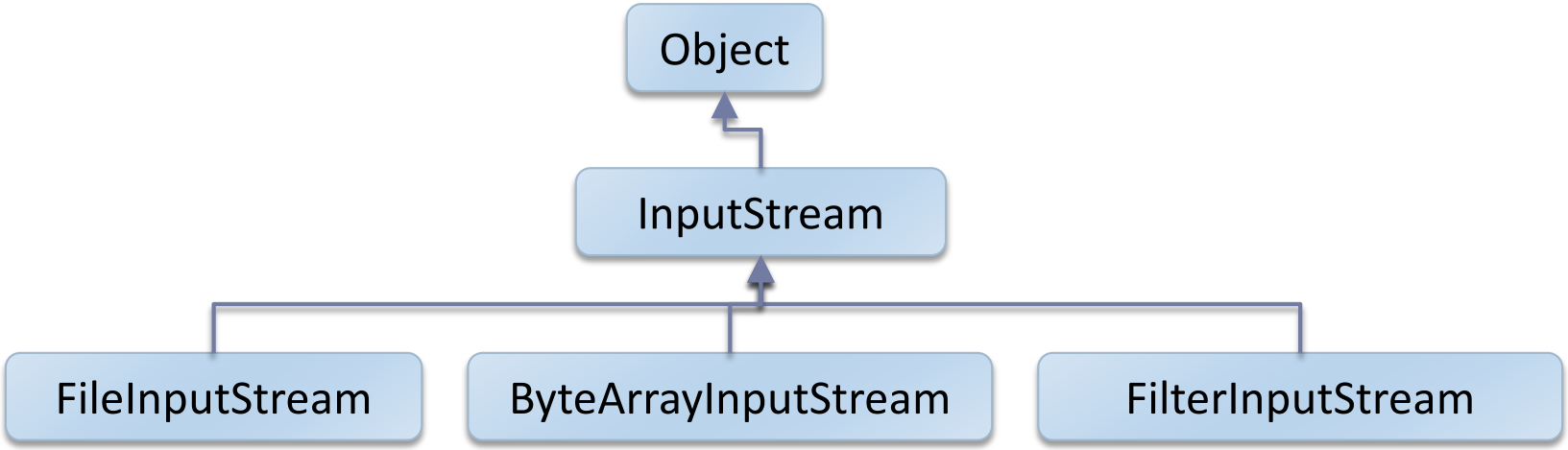
  - Java Stream classes can handle all kinds of data, from primitive values to advanced objects.

Introduction

- **java.io** package
  - Contains classes that are used
    - Working with data streams
    - Working with the File System
    - Object Serialization
  - Streams supported by java.io package
    - Byte Streams
    - Character Streams
    - Buffered Streams

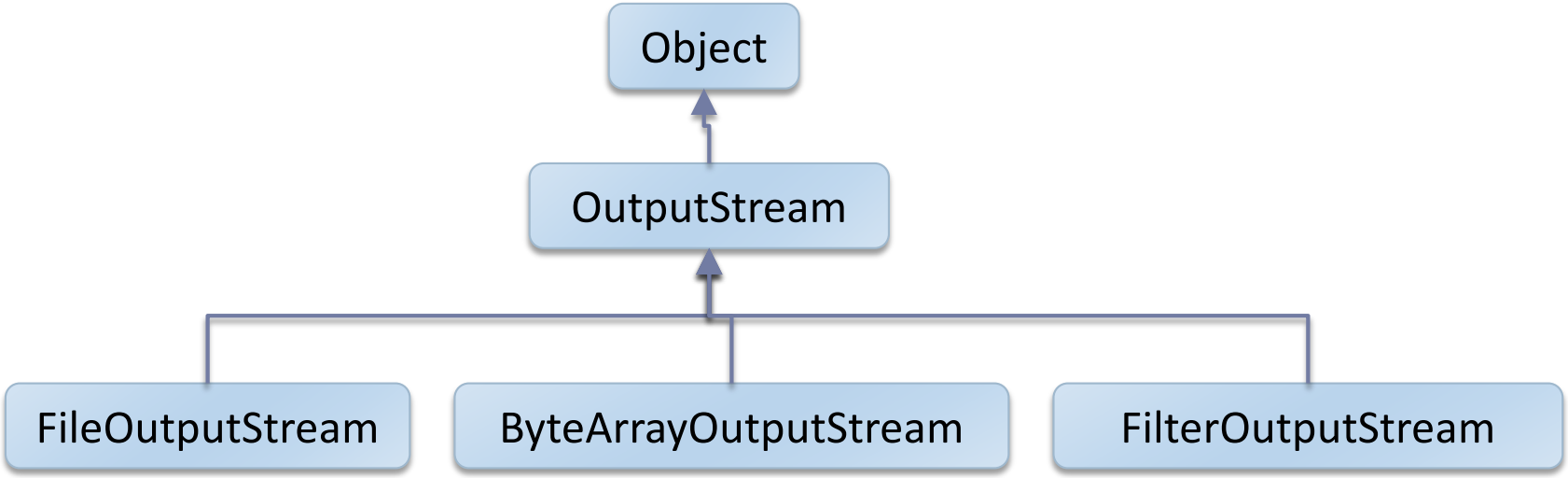
Byte Stream

- Byte Stream
  - A byte stream represents a stream of bytes for input or output.
  - **InputStream** and **OutputStream** classes represent the top level classes for reading/writing bytes in java.
  - **InputStream** class hierarchy (common classes)



Byte Stream

- Byte Stream
  - **OutputStream** class hierarchy (common classes)



  - InputStream and OutputStream are abstract base classes and hence cannot be directly instantiated.

Input Stream

- **InputStream** class
  - Methods provided by the InputStream class

int **read**()

Read and return (as an int in the range 0 to 255) the next byte from this input stream, or return -1 when the end of the stream is reached. This method blocks until input is available.

int **read**(byte[] b)

Read some number of bytes from this input stream and store them in byte array b.
Return the number of bytes actually read (which might be less than b's length but is never more than this length), or return -1 when the end of the stream is reached (no byte is available to read). This method blocks until input is available.

Input Stream

- **InputStream** class
  - Methods provided by the InputStream

int **read**(byte[] b, int off, int len)

Read no more than len bytes from this input stream and store them in byte array b, starting at the offset specified by off.
Return the number of bytes actually read (which might be less than len but is never more than len), or return -1 when the end of the stream is reached (no byte is available to read). This method blocks until input is available.

long **skip**(long n)

Skip over and discard n bytes of data from this input stream.
The actual number of bytes skipped is returned.
If n is negative, no bytes are skipped.

Input Stream

- **InputStream** class
  - Methods provided by the OutputStream class

void **close**()

Closes the output stream, and release any system resources associated with the stream

Output Stream

- **OutputStream** class
  - Methods provided by the OutputStream class

void **write**(byte[] b)

Write b.length bytes from byte array b to this output stream..

void **write**(byte[] b, int off, int len)

Write len bytes from byte array b starting at offset off to this output stream.

void **write**(int b)

Write byte b to this output stream. Only the 8 low-order bits are written; the 24 high-order bits are ignored.

Output Stream

- **OutputStream** class
  - Methods provided by the OutputStream class

int **flush**()

Flush this output stream by writing any buffered output bytes to the destination..

void **close**()

Closes the output stream, and release any system resources associated with the stream

Concrete Stream classes

Byte Array Stream

- **ByteArrayInputStream** class
  - Contains an internal buffer that contains bytes that may be read from the stream.
  - An internal counter keeps track of the next byte to be supplied by the read method.
  - Creating object of ByteArrayInputStream

```
public ByteArrayInputStream(byte[] buf)
```

buf, byte array buffer for the stream.

```
public ByteArrayInputStream(byte[] ba, int offset, int count)
```

Byte Array Stream

- ByteArrayInputStream class

```
// Creates an array of byte
byte[] array = {1, 2, 3, 4};

try {
    ByteArrayInputStream input = new ByteArrayInputStream(array);
    for(int i= 0; i < array.length; i++) {
        // Reads the bytes
        int data = input.read();
        System.out.print(data + ", ");
    }
    input.close();
}
catch(Exception e) {
    e.getStackTrace();
}
```

Byte Array Stream

- **ByteArrayOutputStream** class
  - Write a stream of bytes to a byte array.
  - The buffer automatically grows as data is written to it.
  - Creating object of ByteArrayOutputStream

public **ByteArrayOutputStream**()

Creates a byte array output stream with an internal byte array whose initial size is 32 bytes. This array grows as necessary.

public **ByteArrayOutputStream**(int size)

Creates a new byte array output stream, with a buffer capacity of the specified size, in bytes.

File Stream

- **FileInputStream** class
  - Obtains input bytes from a file in a file system.
  - Meant for reading streams of raw bytes such as image data.
  - Creating an object of FileInputStream

```
public FileInputStream(String name) throws FileNotFoundException
```

name, Name of the file to open.

```
public FileInputStream(File name) throws FileNotFoundException
```

name, File object to be used for opening the file.

File Stream

- **FileInputStream** class
  - Reading bytes

    Read methods block if no input is yet available.
    - Reading a single byte from the Input Stream

```
public int read() throws IOException
```

int, Single byte read from the file.
-1 if the end of the file is reached.

- Reading and array of bytes

```
public int read(byte[] b) throws IOException
```

int, Number of bytes read from the file.
-1 if the end of the file is reached.
b, Array into which bytes will be read.

File Stream

- **FileInputStream** class
  - Closing Stream

```
public void close() throws IOException
```

File Stream

- **FileOutputStream** class
  - Output stream for writing data to a File.
  - Meant for writing streams of raw bytes.
  - Creating an object of FileOutputStream

```
public FileOutputStream(String name) throws FileNotFoundException
```

name, Name of the file to create.

```
public FileOutputStream(String name, boolean append)
                                        throws FileNotFoundException
```

name, Name of the file to create.

append, If true, then bytes will be written to the end of the file rather than the beginning

File Stream

- **FileOutputStream** class
  - Writing file
    - Writing a single byte to the output Stream

```
public void write(int b) throws IOException
```

int, Single byte to write to the file.

    - Writing an array of bytes to the output stream

```
public void write(byte[] b) throws IOException
```

b, Array to write to the file.

File Stream

- **FileOutputStream** class
  - Writing file
    - Writing an array of bytes to the Output Stream

```
public void write(byte[] b, int off, int len) throws IOException
```

b	Buffer into which the data is read.
off	The start offset in the array
len	The maximum number of bytes to write.

  - Closing File

```
public void close() throws IOException
```

Buffered Stream

- Buffered Stream
  - **FileOutputStream** and **FileInputStream** have a performance problem.
  - Each file output stream write() method call and file input stream read() method call results in a call to one of the underlying platform's native methods, and these native calls slow down I/O.
  - **BufferedOutputStream** and **BufferedInputStream** filter stream classes improve performance by minimizing underlying output stream write() and underlying input stream read() method calls.

Buffered Stream

- **BufferedInputStream** class
  - Creating an object of BufferedInputStream class

BufferedInputStream(InputStream in)

Creates a buffered input stream that uses the InputStream in as the input stream to buffer.

BufferedInputStream(InputStream in, int size)

Creates a buffered input stream that uses the InputStream in as the input stream to buffer. Buffer size is set to size argument.

Buffered Stream

- **BufferedOutputStream** class
  - Creating an object of BufferedOutputStream class

BufferedOutputStream(OutStream out)

Creates a new buffered output stream to write data to the specified underlying output stream with a default 512-byte buffer size..

.

BufferedOutputStream(OutStream out, int size)

Creates a new buffered output stream to write data to the specified underlying output stream with the specified buffer size..

Piped Stream

- Piped Stream
  - Threads often need to communicate. One communication approach involves using shared variables.
  - Another approach involves using piped streams provided by Java's **PipedOutputStream** and **PipedInputStream** classes.
  - **PipedOutputStream** class lets a sending thread write stream of bytes to an instance of the PipedInputStream class, which a receiving thread uses to subsequently read those bytes.

Piped Stream

- **PipedInputStream** class
  - Creating an object of PipedInputStream class

PipedInputStream()

Creates a piped input stream that is not yet connected to a piped output stream. It must be connected to a piped output stream before being used.

PipedInputStream(int pipeSize)

Creates a piped input stream that is not yet connected to a piped output stream and uses pipeSize to size the piped input stream's buffer. It must be connected to a piped output stream before being used.

Piped Stream

- **PipedInputStream** class
  - Creating an object of PipedInputStream class

PipedInputStream(PipedOutputStream src)

Creates a piped input stream that is connected to piped output stream src . Bytes written to src can be read from this piped input stream.

PipedInputStream(PipedOutputStream src, int pipeSize)

creates a piped input stream that is connected to piped output stream src and uses pipeSize to size the piped input stream's buffer. Bytes written to src can be read from this piped input stream..

Piped Stream

- **PipedOutputStream** class
  - Creating an object of PipedOutputStream class

PipedOutputStream()

Creates a piped output stream that is not yet connected to a piped input stream. It must be connected to a piped input stream before being used.

.

PipedOutputStream(PipedInputStream dest)

Creates a piped output stream that is connected to piped input stream dest. Bytes written to the piped output stream can be read from dest.

Data Stream

- Data Stream
  - FileOutputStream and FileInputStream are useful for writing and reading bytes and arrays of bytes. However, they provide no support for writing and reading primitive type values (such as integers) and strings.
  - Java provides the concrete **DataOutputStream** and **DataInputStream** filter stream classes.
  - Overcomes this limitation by providing methods to write or read primitive type values and strings in a platform-independent way

Data Stream

- **DataInputStream** class
  - Creating an object of DataInputStream class

    DataInputStream(InputStream in)

    Creates a DataInputStream that uses the specified underlying InputStream.

  - Methods to read java types

    public final boolean **readBoolean**()

    public final byte **readByte**()

    public final int **readUnsignedByte**()

    public final short **readShort**()

Data Stream

- **DataInputStream** class
  - Methods to read java types

```
public final int readUnsignedShort()
```

```
public final char readChar()
```

```
public final int readInt()
```

```
public final long readLong()
```

```
public final float readFloat()
```

```
public final double readDouble()
```

Data Stream

- **DataOutputStream** class
  - Creating an object of DataOutputStream class

    DataOutputStream(OutStream out)

    Creates a new data output stream to write data to the specified underlying output stream.
  - Methods to write java types

    public final void **writeBoolean**(boolean v)

    public final void **writeByte**(int v)

    public final void **writeShort**(int v)

    public final void **writeChar**(int v)

Data Stream

- **DataOutputStream** class
  - Methods to write java types

```
public final void writeInt(int v)
```

```
public final void writeLong(long v)
```

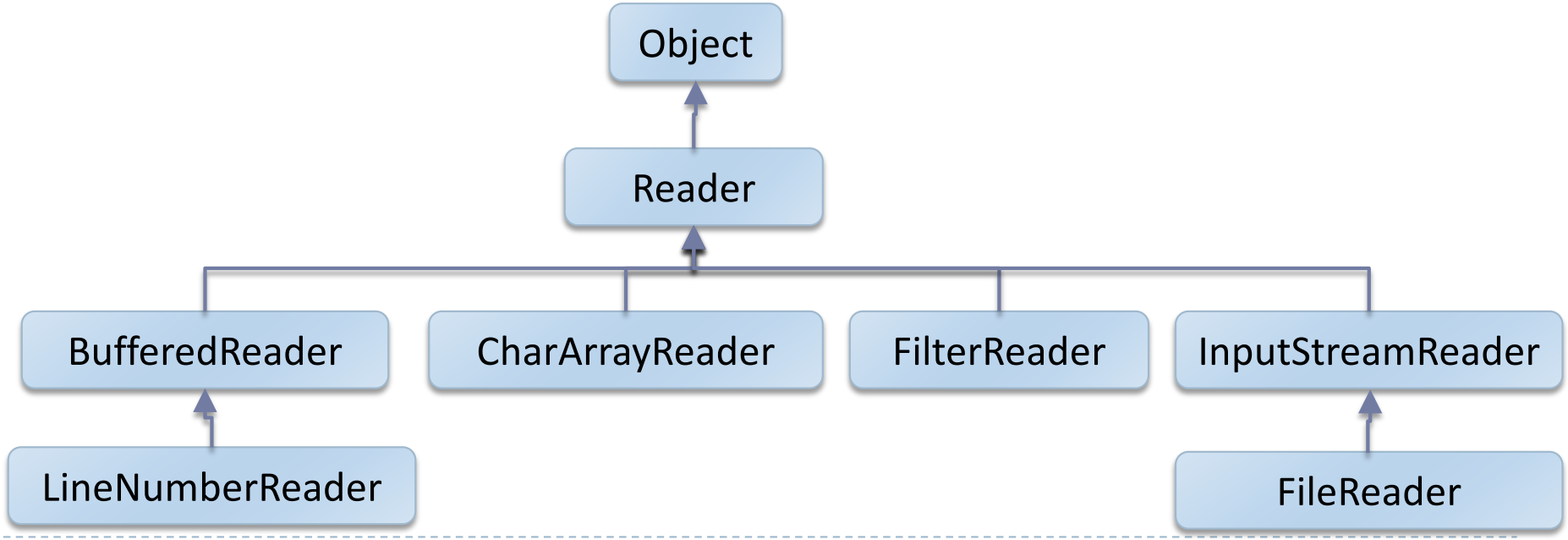
```
public final void writeFloat(float v)
```

```
public final void writeDouble(double v)
```

```
public final void writeUTF(String str)
```

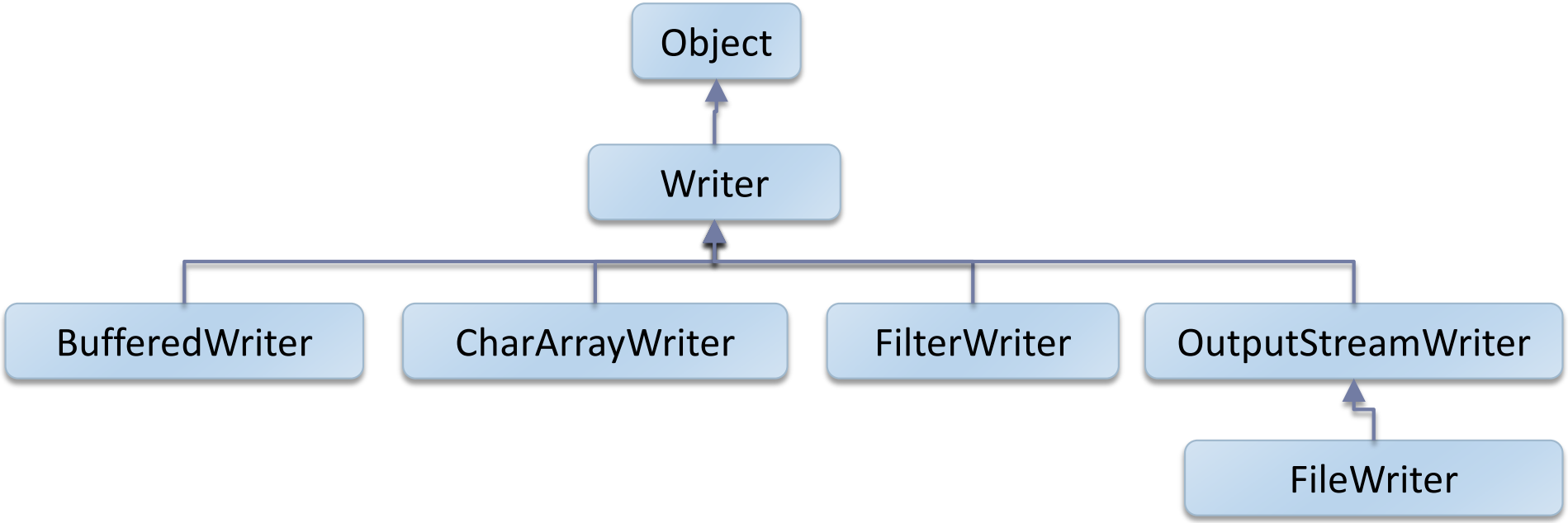
Character Stream

- Character Stream
  - A character stream represents a stream of characters for input or output.
  - **Reader** and **Writer** classes represent the top level classes for reading/writing characters in java.
  - **Reader** class hierarchy (common classes)



Character Stream

- Character Stream
  - **Writer** class hierarchy (common classes)



  - Both **Reader** and **Writer** are abstract base classes and hence cannot be directly instantiated.

Bridging Streams

- Bridging Byte streams and Character Streams
  - **InputStreamReader** & **OutputStreamWriter** allow us to bridge byte streams and character streams.
  - **OutputStreamWriter** class is a bridge between an incoming sequence of characters and an outgoing stream of bytes.
  - **InputStreamReader** class is a bridge between an incoming stream of bytes and outgoing sequence of characters.

Bridging Streams

- **OutputStreamWriter** class
  - Characters written to this writer are encoded into bytes according to the default or specified character encoding.
  - Creating OutputStreamWriter

```
public OutputStreamWriter(OutputStream out)
```

Bridging Streams

- **InputStreamReader** class
  - Bridge from byte streams to character streams.
  - Reads bytes and decodes them into characters using a specified character set encoding.
  - Creating InputStreamReader

```
public InputStreamReader(InputStream in)
```

File Reader

- **FileReader** class
  - Convenience class for reading character files.
  - Meant for reading streams of characters from files.
  - Creating an object of FileReader

```
public FileReader(String name) throws FileNotFoundException
```

name, Name of the file to open.

```
public FileReader(File name) throws FileNotFoundException
```

name, File object to be used for opening the file.

File Reader

- **FileReader** class
  - Reading file

    Read methods block if no input is yet available.
    - Reading a single character

```
public int read() throws IOException
```

int, Single character read from the file.
-1 if the end of the file is reached.

- Reading and array of characters

```
public int read(char[] b) throws IOException
```

int, Number of characters read from the file.
-1 if the end of the file is reached.

b, Array into which characters will be read.

File Reader

- **FileReader** class
  - Reading file

    Read methods block if no input is yet available.
    - Reading an array of characters

```
public int read(char[] b, int off, int len) throws IOException
```

int,	Number of characters read from the file.
	-1 if the end of the file is reached.
b	Buffer into which the data is read.
off	The start offset in the destination array
len	The maximum number of bytes to read.

    - Reading an array of characters

```
public int read(CharBuffer buf) throws IOException
```

int,	Number of characters read from the file.
	-1 if the end of the file is reached.
buf	Buffer into which the data is read.

File Reader

- **FileReader** class
  - Skipping characters

```
public long skip(long n) throws IOException
```

  - Closing file

```
public void close() throws IOException
```

File Writer

- **FileWriter** class
  - Output stream for writing characters to a File.
  - Meant for writing streams of characters.
  - Creating an object of FileWriter

```
public FileWriter(String name) throws IOException
```

name, Name of the file to create.

```
public FileWriter(String name, boolean append)
                                    throws IOFoundException
```

name, Name of the file to create.

append, If true, then bytes will be written to the end of the file rather than the beginning

File Writer

- **FileWriter** class
  - Creating an object of FileWriter

public **FileWriter**(File *file*) throws IOException

file, File Object to be used for creating file output stream.

public **FileWriter**(File *name*, boolean *append*) throws IOException

file, File Object to be used for creating file output stream.

append, If true, then bytes will be written to the end of the file rather than the beginning

File Writer

- **FileWriter** class
  - Writing file
    - Writing a single character

```
public void write(int c) throws IOException
```

int,	Single character to write to the file.

    - Writing a portion of string

```
public void write(String str, int off, int len) throws IOException
```

str	String to write to the file
off	The start offset in the string
len	The maximum number of characters to write.

File Writer

- **FileWriter** class
  - Writing file
    - Writing an array of characters

```
public void write(char[] b, int off, int len) throws IOException
```

b	Buffer to write to the file.
off	The start offset in the array.
len	The maximum number of characters to write.

    - Writing a String

```
public void write(String str) throws IOException
```

str	String to write to file.

File Writer

- **FileWriter** class
  - Close File

```
public void close() throws IOException
```

Buffered Reader

- **BufferedReader** class
  - Reads text from a character-input stream.
  - Buffers characters so as to provide for the efficient reading of characters, arrays, and lines.
  - The buffer size may be specified, or the default size may be used.
    - The default is large enough for most purposes.
  - Creating a BufferedReader

```
public BufferedReader(Reader in)
```

in, An Object of class that is a Reader.

  - Method used for reading a single line from stream

```
public String readLine() throws IOException
```