1. Exceptions

Introduction

Exception Hierarchy

Exception handling using : try-catch-finally
Methods to display error information.

Checked and Unchecked exceptions Multiple catch blocks.

Nested try blocks

User defined exceptions (Customized Exceptions) Throw and Throws

2. Java.lang Package

Hierarchy
Object
class String

StringBuffer & StringBuider (Jdk1.5) Wrapper classes

Autoboxing and Unboxing (Jdk1.5)

EXCEPTIONS

Introduction

Definition: An **unexpected problem** occurs while running our application at **runtime** is called as an exception.
For example: **ArithmeticExcetion**, **ArrayIndexOutOfBoundsException**, **FileNotFoundException**, etc.

Example:

```
public class ExceptionDemo {
        public static void main(String[] args) {
                int a = Integer.parseInt(args[0]);
                int b = Integer.parseInt(args[1]);
                int c = a/b;
                System.out.println("The result="+c);
                System.out.println("End of main method");
        }
}
```

o/p:

```
Exception in thread "main" java.lang.ArithmeticException: / by
        zero at ExceptionDemo.main(ExceptionDemo.java:5)
```

[Abnormal Termination]

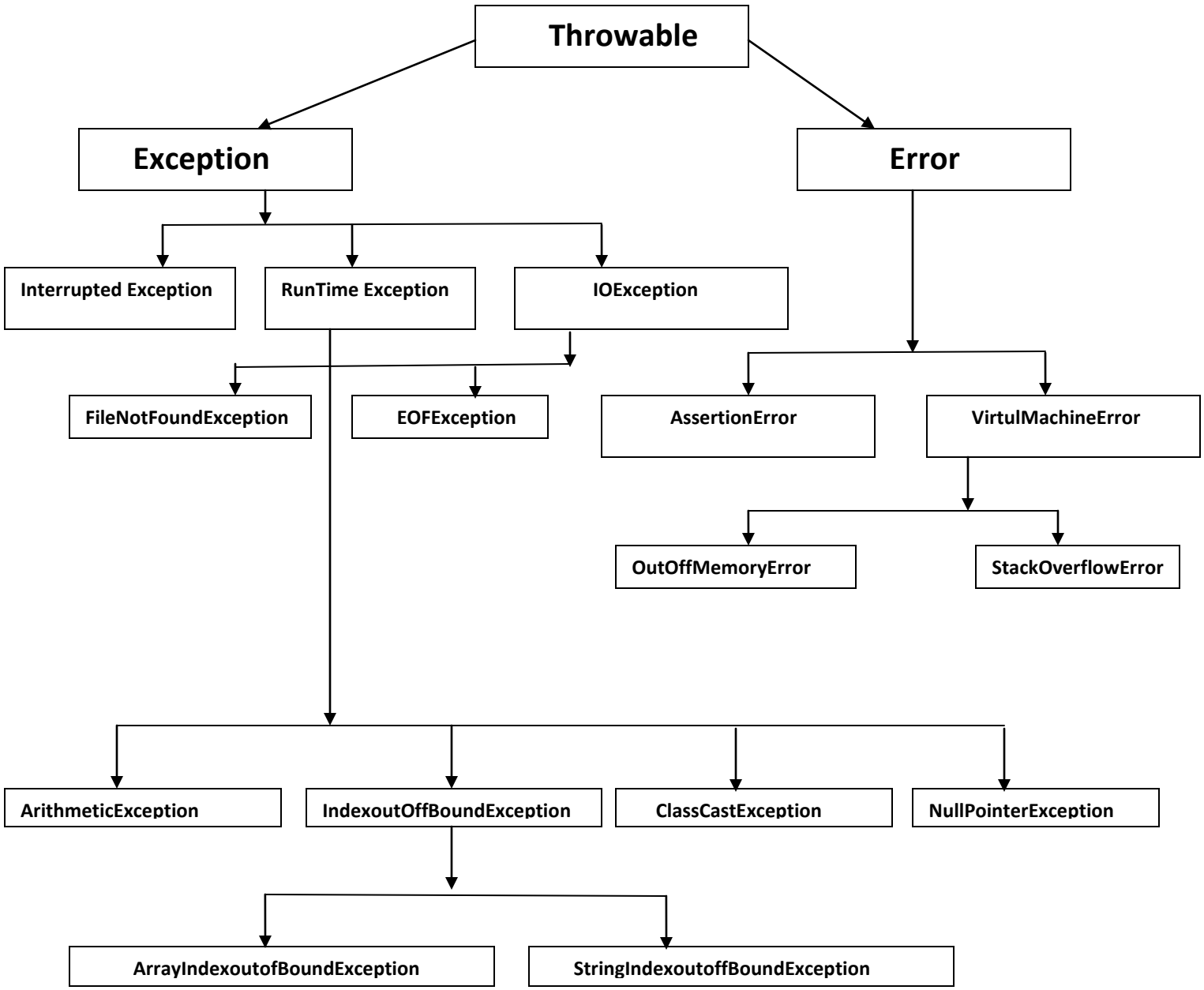
Exception Hierarchy

The root class for all exceptions is **java.lang.Throwable**.

The two main subclasses of Throwable are Exception and Error.

Difference between Exception and Error:

Exception	Error
Exceptions are **caused by our program** and hence **recovarable**. After recovering (handling) exception, the program will continue as if there is no exception.	Errors are not caused by our application and rather they are caused due to **lack of system resources** such as memory, etc. Hence, errors are **not recoverable**. The program will be terminated abnormally.


Throwable
Exception
Error
Interrupted Exception
RunTime Exception
IOException
FileNotFoundException
EOFException
AssertionError
VirtulMachineError
OutOffMemoryError
StackOverflowError
ArithmeticException
IndexoutOffBoundException
ClassCastException
NullPointerException
ArrayIndexoutofBoundException
StringIndexoutoffBoundException

Methods to display exception Information:

The **Java.lang.Throwable** root class contains the following 3 methods to print exception information:

1) **Public void printStackTrace()**

This method is used to print following error information:

a) Exception name
b) Exception Message
c) Line number which causes the exception
d) Method belongs to the above line number
e) Class belongs to the above method.
f) File name belongs to the above class.

Example:

Exception in thread "main" java.lang.ArithmeticException: / by zero
at ExceptionDemo.main(ExceptionDemo.java:5)

2) **Public String toString()**

This method is used to print following error information:

a) Exception name
b) Exception Message

Example:

java.lang.ArithmeticException: / by zero

3) **Public String getMessage()**

This method is used to print following error information. a) Exception Description.

Example:

/ by zero

Exception handling using try-catch-finally

It is always a good programming practice to handle an exception for graceful termination (Normal termination) of the program i.e., the purpose of handling an exception is to continue program execution as if there is no exception.

Try block

It contains the statements which likely to throw an exception.
If an exception is raised, the remaining statements in the try block are **skipped**.
Try block must be followed by either catch or **finally** or **both**.

Catch block

Catch block is used to handle an exception.
The catch block is executed only if try block throws an exception.
Catch block cannot be written without try block. For each try block there can be zero or more catch blocks.

Finally block

It is not recommended to write **clean up code** inside try block, because there is no guaranty for the execution of all statements inside try.

It is not recommended to write clean up code inside **catch** block because it won't executed if there is no Exception.

We required one place to maintain cleanup code which should be executed always irrespective of whether exception is raised or not and exception is handle or not. Such type of block is called as finally block. Hence the main objective of finally block is to maintain **cleanup code**.

Only one finally block can be associated with try block.

Example:

```
public class ExceptionHandlingDemo {
    public static void main(String[] args) {
        int a = Integer.parseInt(args[0]);
        int b = Integer.parseInt(args[1]);
        try{
            int c = a/b;
        }catch(ArithmeticException ae){
            System.out.println("Denominator should not be 0");
            Ae.printStackTrace();
        }finally{
            System.out.println(“finally balock”);
        }
        System.out.println("End of main method.");
    }
}
```

Java ExceptionHandlingDemo 10 0

O/P:

```
Denominator should not be 0
java.lang.ArithmeticException: / by zero
        at ExceptionHandlingDemo.main(ExceptionHandlingDemo.java:6)
finally balock
End of main method.
[Normal Termination]
```

Checked and Unchecked exceptions

If an exception(or error) is directly or indirectly inherited from RuntimeException(or Error) is called as Unchecked exception i.e., in the exception hierarchy, if **RuntimeException** (or Error) presents then it is a Unchecked exception, otherwise, it is a checked exception.

Example:

Unchecked exception	Checked exception
java.lang.Object java.lang.Throwable java.lang.Exception **java.lang.RuntimeException** **java.lang.ArithmeticExceptio n**	java.lang.Object java.lang.Throwable java.lang.Exception **java.lang.ClassNotFoundException**

Checked exception must be handled, otherwise the program will not be compiled.

Example:

```
Try{
        Thread.sleep(1000);
}catch(InterruptedException ie){}
```

Unchecked exception may or may not be handled. The compiler will ignore unchecked exceptions.

Example:

```
Void meth(){
      SOP(10/0);
}
```

The unchecked exceptions are caused due to invalid input data entered by end user at runtime, whose information never known by compiler in advance. Hence, unchecked exceptions are ignored by java compiler. The checked exceptions must be handled by the developer otherwise the compiler will not let the code to be compiled.

Try with Multiple Catch Blocks

The way of handling an exception is varied from exception to exception, hence it is recommended to place separate catch block for every exception.

If the statements in try block throws multiple exceptions, it is recommended to handle them individually.

In case of try with multiple catch blocks, the **order of catch blocks present is very important and it should be from subclass exception type to superclass exception type**.

The catch block for a super class exception should not shadow the catch block for the sub class exception i.e. the order of the catch blocks is **from sub class to the super class exception**, otherwise the program will not be compiled.

Example:

Try{ Int[] arr = new int[5]; SOP(arr[10]); String str = “hello”; SOP(str.charAt(10)); }catch(ArrayIndexOutOfBoundsException e){e.printStackTrace(); }catch(StringIndexOutOfBoundsException e){ Sop(e.toString()); } //OK	Try{ Int[] arr = new int[5]; SOP(arr[10]); String str = “hello”; SOP(str.charAt(10)); }catch(ArrayIndexOutOfBoundsException ae){ }catch(**IndexOutOfBoundsExceptio n** e){} //OK	Try{ Int[] arr = new int[5]; SOP(arr[10]); String str = “hello”; SOP(str.charAt(10)); }catch(IndexOutOfBoundsExceptio n ae){ }catch(ArrayIndexOutOfBoundsExceptione){ } **//C.E.**

Nested try blocks

Try block with in another try block is called as nested try block.

Example:

```
try{
      try{
      }catch(){}
}catch(){}
```

If an exception is raised inside inner try block, initially it will look for inner catch block, if it is not matched, then it will look for outer catch block.
If an exception is raised in outer try block, then directly it will look for outer catch block.

User defined (or Customized) exceptions

Like built-in exceptions, user can define application (project) specific exceptions is called as user defined exceptions.
Like built-in exceptions, there are two types of user defined exceptions as well:

Un-checked Exceptions
Checked Exceptions.

Un-checked Exceptions	**Checked Exceptions**
package edu.aspire; public class DivByOneException extends **RuntimeException**{ Public DivByOneException( ){super();} Public DivByOneException (String message){ super(message); } }	Public class DivByOneException extends **Exception**{ Public DivByOneException ( ){} Public DivByOneException (String message){ Super(message); } }

throw

The '**throw**' keyword is used to propagate (or delegate) an exception to its caller inorder to let the calling method to handle an exception rather than implementation method.

Syntax: throw <**throwable object**>;

Example:

```
throw new ArithmeticException("/ by zero");
```

The 'throw'keyword is always used at **block level** but not at method signature level.

All built-in exceptions are automatically thrown by the JVM. But, built-in exceptions can be programmatically thrown by the developer using throw keyword.
All user defined exceptions must be programmatically thrown using 'throw' keyword.

Example:

```
public class Division{
    public static void main(String[]
    args){ try{
        div(10,0);                  //method calling
    }catch(ArithmeticException ae)  {
        ae.printStackTrace();
    }
    }
    public static void div(int x, int y){ //method implementation If(y==0){

                    throw new ArithmeticException("/ by zero");
    }
```

```
                System.out.println(x/y);
        }
}
```

Built-in exceptions w.r.t throw keyword

Using throw keword with built-in exceptions is **optional**. Hence the result of the following two programs is exactly same.

```
class Test{
public static void main(String[] args){
      System.out.println(10/0);
 }
}
//R.E : A.E
```

```
Class Test{
    public static void main(String[] args){
        throw new ArithmeticException("/ by Zero");
    }
}
//R.E : A.E
```

User defined exceptions w.r.t throw keyword

User defined exceptions must be thrown explicitly using throw keyword. Hence the result of the the following two programs is not same.

```
class Test{
public static void main(String[] args){
      System.out.println(10/1);
 }
}
```

Compiles and Runs without exception.
O/P: 10

```
Class Test{
    public static void main(String[] args){
            throw new DivByOneException("/ by one");
    }
}
```

// R.E : edu.aspire.DivByOneException: / by one

throws

Throws keyword is used at **method signature level** to specify the type of exception a method throws.

Syntax:

```
<method modifiers> <return type> method_name(<formal parameter list>)
<throws> <ExceptionType1> … <,ExceptionTypen>{
        <statement>
}
```

Checked exceptions w.r.t throws keyword:

If the throw propagates checked exceptions inside method block, then such checked exceptions must be specified at method signature using '**throws**' keyword.

Example:

```
class Test{
   p.s.v main(String[] args)throws I.E{
     doStuff();
   }
  Public static void doStuff() throws IE{
      throw new InterruptedException();
   }
//R.E: InterruptedException
```

```
class Test{
   p.s.v main(String[] args){
     doStuff();
  }
  public static void doStuff(){
     throw new InterruptedException();
  }
//C.E: Unhandled exception type :
```

InterruptedException.

If an exception is propagated to JVM, it prints the stack trace followed by terminates the thread which causes the exception. This is called as **Default Handler**.

Unchecked exceptions w.r.t throws keyword:

The compiler does not verify the unchecked exceptions in throws clause i.e., if the throw propagates unchecked exceptions inside method block, then throws may or may not specify unchecked exceptions at method signature.

Example:

```
class Test{
p.s.v.main(String[] args)throws ArithmeticException{
      throw new ArithmeticException();
}
}
//R.E: A.E
```

```
class Test{
p.s.v.main(String[] args){
      throw new ArithmeticException();
}
}
//R.E: A.E
```

Throws w.r.t method overriding

The subclass method must not throws more checked exceptions than superclass method, i.e., the overriding method in subclass may throw **none, all, or sub-set of checked exceptions from overridden method from superclass, but not more checked exceptions.** This rule is not applicable for unchecked exceptions.

Example:

```
class One{
   public void meth()throws
IOException{
   }
}
class Two extends One{
  public void meth()throws
Exception{
  }
}
//C.E: Sub class method must not
throw supertype exception.
```

```
class One{
   public void meth()throws
IOException{
   }
}
class Two extends One{
  public void meth()throws
InterruptedException{
  }
}
//C.E: Sub class method must not
throw different checked exceptions.
```

```
class One{
public void meth()throws
IOException{
}
}
class Two extends One{
public void meth()throws
FileNotFoundException{
  }
}
//R.E: IOException
```

The overriding method in subclass may throw different unchecked exceptions than overridden method in superclass i.e., the overriding method in subclass may or may not specify unchecked exceptions in overridden method unchecked exceptions in superclass.

```
class One{
public void meth()throws
IndexOutOfBoundsException{
   }
}
class Two extends One{
public void meth()
  }
```

```
class One{
public void meth()throws
IndexOutOfBoundsExceptions{
}
}
class Two extends One{
public void meth()throws
RuntimeException
```

```
class One{
public void meth()throws
IndexOutOfBoundsExceptions{
}
}
class Two extends One{
public void meth()throws
ArrayIndexOutOfBoundsException
```

} **//No compilation error.**	} } **//No compilation error.**	} } **//No compilation error.**

Difference between Checked and Unchecked exceptions :

CheckedException	UnCheckedException
Except for RuntimeException, Error, and their subclasses, all other exceptions are called checked exceptions.	Exceptions inherited from Error or RuntimeException class and their subclasses are known as unchecked exceptions.
All checked exceptions must be handled, otherwise the program will not be compiled.	The compiler will not force unchecked exceptions to be handled i.e., unchecked exceptions may or may not be handled.
If a method throw checked exceptions, they must be specified in the throws clause. But, more checked exceptions must not be specified in the throws clause.	Specifying unchecked exceptions in throws clause is optional.

JAVA.LANG PACKAGE

Hierarchy
Object
class String
StringBuffer & StringBuider (Jdk1.5)
Wrapper classes
Autoboxing and Unboxing (Jdk1.5)

This package contains most common classes in java. Hence, this package is automatically imported into every source file at compile time.

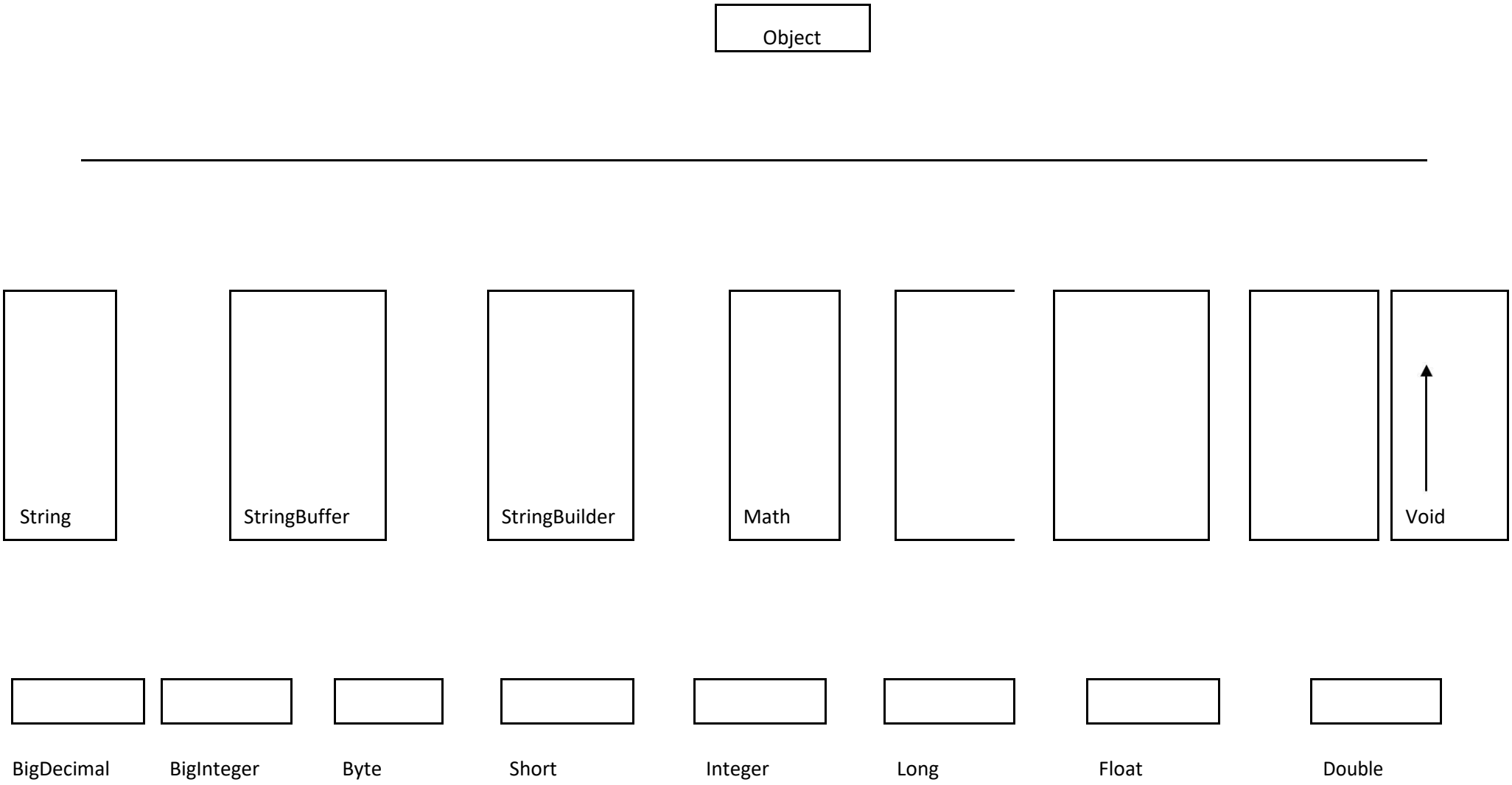


Figure: Partial Hierarchy Of java.lang Package classes

Note:

1) The six numeric wrapper classes are Byte, Short, Integer, Long, Float, Double.
2) The three non numeric wrapper classes are Character, Boolean, and Void.

3) All wrapper classes, String, StringBuffer, StringBuilder classes are final.
4) All wrapper classes and String classes are immutable.
5) All wrapper classes and String classes are Comparable and Serializable.

Object class

Every class in java, whether it is built-in or user defined, is implicitly inherited from Object class, i.e., the Object is a **root** class for all classes in java.

Object class defines following methods:

1. public String toString()
2. **public boolean equals(Object)**
3. public int hashCode() // During collection
4. **protected Object clone()**
5. public Class getClass()
6. **public void wait()**
7. **public void notify()**
8. **public void notifyAll()**
9. protected void finalize()

public String toString()

It is always recommended to override toString() method to provide **object state information**.

Example:

```
class Box{
        int  width;
        int height;
        int depth;
        Box(int w, int h, int
                d){ Width = w;
                Height = h;
                Depth = d;
        }

        Int volume(){
                return width * height * depth;
        }
        @Override
        public String toString(){
                return "Width="+width +"   Height="+height +" Depth="+depth;
        }
}
Box b1 = new Box(1, 2,3);
Box b2 = new Box(4,5,6);
System.out.println(b1.toString());
System.out.println(b2);//The toString() method will be automatically invoked.
```

O/P:

```
Width=1    Height = 2    Depth = 3
Width=4    Height = 5    Depth = 6
```

Public boolean equals()

Actually, equals() method is used to compare two object states (content comparision).

The **==** operator is used to compare reference values but not object states (Reference comparision).

Example:

```
Box b1 = new Box(1,2,3);
Box b2 = new Box(1,2,3);
Box b3 = b1;

SOP(b1.equals(b2));//true
SOP (b1 == b2);  //false
SOP(b1.equals(b3));//true
SOP (b1 == b3);  //true
SOP (b2 == b3);  //false
```

It is always recommended to override equals() method to compare two object states.

```
@Override
public boolean equals(Object o) {
        //Comparing with null reference always returns false.
        if (o == null)
                return false;
        //Comparing incompatible types always throws
        ClassCastException. if (!(o instanceof Box))
                throw new ClassCastException();
        //alias comparision.
        if (this == o) return
                true;

        Box b = (Box)o;
        //content comparison
        if ((this.width == b.width) && (this.height == b.height) && (this.depth == b.depth)) {
                return true;
        } else {
                return false;
        }
}
```

```
Box b1 = new Box(1,2,3);
Box b2 = new Box(1,2,3);
Box b3 = new Box(4,5,6);
Box b4 = b1;
Box b5 =null;
System.out.println(b1 == b2);//false
System.out.println(b1.equals(b2));//true
System.out.println(b1 == b3);//false
System.out.println(b1.equals(b3)); //false
System.out.println(b1==b4); //true
System.out.println(b1.equals(b4));//true
```

Difference between == operator and equals() method

== operator	equals() method
Used to compare reference values but not actual object states i.e., used for **Reference comparision.**	Used to compare actual object states (**Content comparision**).

public int hashCode()

The hashcode is associated with object. **More than one object may have same hashcode value**. The hashcode is different from reference value. The reference value is returned to reference variable, **but hashcode is associated with object.**

The hashing mechanism is used to **achieve better search results**.

The hashcode take us to the appropriate bucket. The equals() method is used to choose one of the object among many objects within the bucket.

```
@Override
public int hashCode() {
        final int prime =
        31; int result = 1;
        result = prime * result + depth;
        result = prime * result +
        height; result = prime * result
        + width; return result;
}

Box b1 = new Box(1,2,3);
Box b2 = new Box(4,5,6);

Box b3 = new Box(10,20,30);
Box b4 = new Box(40,50,60);
```

Contract between equals() and hashCode() methods:

1) If two objects are equal by equals() method, then their hashcodes must be same.
2) If two objects are not equal by equals() method, then their hashcodes may or may not be same.
3) If two objects hashcodes are equal by hashCode() method, then their equals() method may or may not return true.
4) If two objects hashcodes are not equal by hashCode() method, then their equals() method must return **false**.

Java.lang.String

In java, String is a sequence of characters **but not array of characters**. Once string object is created, we are not allowed to change existing object. If we are trying to perform any change, with those changes a new object is created. This behavior is called as **Immutability**.

Strings are created in 4 ways:

1) String Literals

Example:

String str = “aspire”;

2) String objects
   Example:
   String str = new String(“aspire”);
3) String constant expressions.
   Example:
   String str = “aspire ” + “ technologies”’;
4) String concatenation operation.
   Example:
   String str1 = “aspire”’; String
   str2 =” technologies”;
   String str = str1 + str2;

Example:

String str = new String(“aspire”);
str.concat(“technology”); //new string object is created.

Heap Memory

aspire

str

aspire technology

StringBuffer sb = new StringBuffer(“Aspire”);
Sb.append(“technology”);

Heap Memory

Aspire technology

sb

String class does not support **append()** method. Once we create StringBuffer object, we can perform changes. Hence it is a **mutable** object.

String Constructors

1) Public String()
   Creates an empty string whose length is zero.
2) Public String(String str)
   Example:
   String str = new String(“hello”); //string object
3) Public String(char[] value)
   Example:
   char[] values = {‘a’,’b’,’c’};
   String str = new String(values); //”abc”
4) Public String(StringBuffer sb)
   Example:
   StringBuffer sb = new StringBuffer(“hello”);
   String str = new String(sb);

5) Public String(StringBuilder sb)

Example:

```
StringBuilder builder = new StringBuilder("hello");
String str = new String(sb);
```

Methods

1. **Public char charAt(int index)**

   Returns the char value at the specified index.

   Example:

   ```
   String str = "aspire";
   SOP(str.charAt(2)); // p
   ```

2. **Public String concat(String str)**

   Appends at the end of the invoking string. Always returns new object.

   Example:

   ```
   String str1 = ''aspire ";//string literal
   String str2 = str1.concat("technologies"); //string
   object Sop(str1); // "aspire "
   Sop(str2);// "aspire technologies"
   ```

3. **Public String substring(int beginIndex, int endIndex)**

   Returns a new string that is a substring of this string. beginIndex is **inclusive**, but, endIndex is **exclusive**.

   Example:

   ```
   String str1 = "aspire technologies"
   String str2 =str1.substring(0, 6);//"aspire"
   ```

4. **public String substring(int beginIndex)**

   If the endIndex is not specified, it returns till end of the string.

   Example:

   ```
   String str1 = "aspire technologies";
   String str2 = str1.substring(7);//"technologies"
   ```

5. **The following lastIndexOf() methods are overloaded.**

   Public int lastIndexOf(char ch)
   Public int lastIndexOf(String str)

   Example:

   ```
   String name = "java.lang.Integer";
   String cName = name.substring(name.lastIndexOf('.')+1); //"Integer"
   String pName = name.substring(0,name.lastIndexOf('.')); // "java.lang"
   ```

6. **The following indexOf() methods are overloaded.**

   Public int indexOf(int ch)
   Public int indexOf(String str)

   Example:

   ```
   String pack = "java.lang.String";
   pack.substring(0,pack.indexOf('.')); //java
   ```

7 . public int length()

Returns the number of characters in the string object.

Example:

```
String str = “hello”;
SOP(str.length()); //5
```

8. public String toLowerCase()

Returns string in lowercase format.

Example:

```
String name=”RAMESH”;
String after = name.toLowerCase(); //ramesh
```

9. public String toUpperCase()

Returns string in uppercase format.

Example:

```
String name=”ramesh”;
String after = name.toUpperCase(); //RAMESH
```

10. public String trim()

Removes leading and trailing spaces, if any.

Example:

```
String str = “ ramesh ”
Sop(str.trim());//”ramesh”
```

11. Public String replace(char oldChar, char newChar)

Replaces all occurrences of old characters with new character.

Example:

```
String str = “abababb”;
String result =str.replace(‘b’,’a’); //aaaaaa
```

12. Public int compareTo(Object obj)

Returns –ve, if o1 < o2
Returns +ve, then o1 > o2;
Returns 0, then o1 == o2 are equal.

Example:

```
Sop(“A”.compareTo(“B”));//-1
Sop(“B”.compareTo(“A”));//+1
Sop(“A”.compareTo(“A”));//0
```

Java.lang.StringBuffer

For every change in string object, a new string object is created, because string is immutable object, which causes memory overhead. To avoid this, use StringBuffer, the changes are applied on the existing object rather than creating new object every time. Hence StringBuffer is **muttable** object.

Constructors

1) Public StringBuffer()
   Constructs an empty string buffer with default initial capacity is **16.**

2) Public StringBuffer(String str)

Constructs a string buffer initialized to the contents of the specified string.
The initial capacity of the string buffer is : **16 + str.length()**

Example:

```
StringBuffer sb = new StringBuffer("hello");
SOP(sb.capacity()); //21
SOP(sb.length());//5
```

3) Public StringBuffer(int initialCapacity)

Creates an empty string buffer with specified initial capacity.

Methods

1. Public int capacity() **//Not available in String class**

   Returns the current capacity of the StringBuffer. It means, the maximum number of characters in can hold. Once it reaches it capacity, it automatically increases.
2. Public int length()

   Returns the actual number of characters contained in the StringBuffer.
3. Public char charAt(int index)
4. The following are append() overloaded methods in StringBuffer class

   Public StringBuffer append(String s)

   Public StringBuffer append(Boolean

   b) Public StringBuffer append(char s)
   Public StringBuffer append(int s)
   Public StringBuffer append(long s)

   Public StringBuffer append(double

   s) Public StringBuffer append(float s)
   Public StringBuffer append(Object s)

   Note: Such methods are not available in String class.
5. The following are overloaded insert() methods in StringBuffer class

   Public StringBuffer insert(int pos, String s)

   Public StringBuffer insert (int pos, Boolean b)
   Public StringBuffer insert (int pos, char c)
   Public StringBuffer insert(int pos, int i) Public
   StringBuffer insert (int pos, long l) Public
   StringBuffer insert (int pos, float f) Public
   StringBuffer insert(int pos, double f)

   Note: String does not have this method.

   Example:

   ```
   StringBuffer sb = new StringBuffer("1221");
   sb.insert(2,"33"); //123321
   ```
6. Public StringBuffer delete(int start, int end) //Not in string class

   Start is **inclusive**, but end is **exclusive**.

   Example:

   ```
   StringBuffer sb = new StringBuffer("123321");
   Sb.delete(2,4);//1221
   ```
7. Public StringBuffer deleteCharAt(int index)
8. Public StringBuffer reverse() //not in string class

Example:

```
StringBuffer sb = new StringBuffer("satyam");
System.out.println(sb.reverse()); // O/P: maytas
```

9. Public void trimToSize()
   If the capacity is larger than length, then extra space will be removed.
   Example:

```
StringBuffer sb = new
StringBuffer("hello"); SOP(sb.capacity());
//21 SOP(sb.length()); //5
Sb.trimToSize();
SOP(sb.capacity());        //5
SOP(sb.length()); //5
```

Java.lang.StringBuilder (jdk1.5)

StringBuffer is a synchronized class. It cannot be accessed by more than one thread at a time. Hence performance is not good.

StringBuilder is non-synchronized version of StringBuffer class. Hence, it can be accessed by more than one thread at a time. So, Performance is good with StringBuilder.

Differences between StringBuffer and StringBuilder:

StringBuffer	StringBuilder
Synchronized class (i.e., thread-safe)	Non-Synchronized class (i.e., non thread-safe)
Performance is low	Performance is high
Legacy class	Introduced in Jdk1.5

Wrapper classes

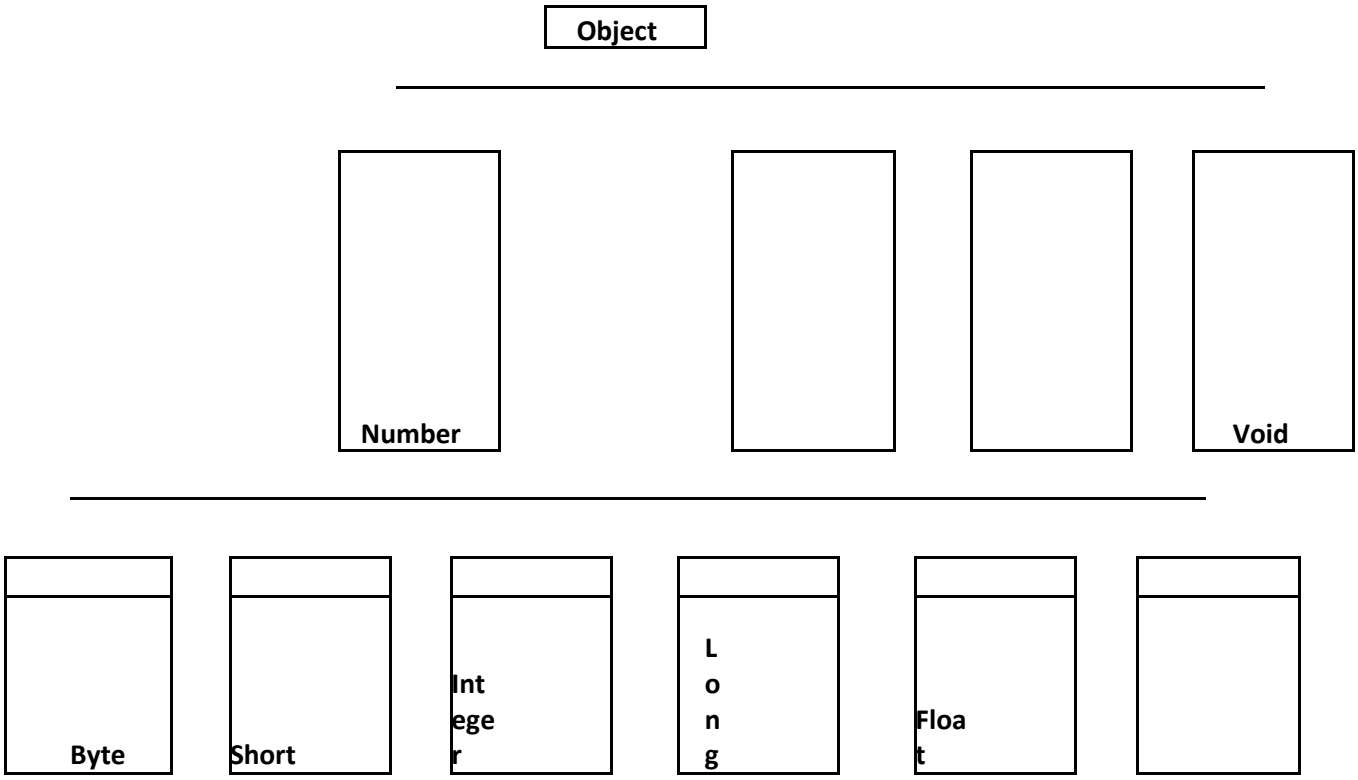


Fig: Wrapper classes

Wrapper classes are used to wrap (store) primitive data into an object.
Hence, primitive types can also be handled just like an object.
By default, all wrapper classes are final, hence cannot be inherited.
Also, the instance of all wrapper classes are immutable i.e., the value in the wrapper object cannot be changed.

```
Public final class Integer extends Number{
        private final int value;
        public Integer(int value){
                this.value = value;
        }
        Public Integer(String value){
                value = parseInt(value)
        } public byte byteValue()
        {
                return (byte)value;
        }
        public short shortValue() {
                return (short)value;
        }
        public int intValue() {
                return (int)value;
```

```
        }
        public long longValue() {
                return (long)value;
        }
        public float floatValue() {
                return (float)value;
        }
         public double doubleValue() {
                return (double)value;
        }
}
```

Constructors

All wrapper classes except Character have two overloaded constructors: Primitive type and String type constructors.

Java.lang.Integer

Public Integer(int value){}
Public Integer(String value){}
Example:

```
Integer iRef = new Integer(4);
Integer iRef = new Integer("4");
```

Java.lang.Long

Public Long(long value){}
Public Long(String value){}
Example:

```
Long lRef = new Long(10);
Long lRef = new Long(10L);
Long lRef = new Long("10");
```

Java.lang.Float

Public Float(float value){}
Public Float(double value){}
Public Float(String value){}
Example:

```
Float fRef = new Float(3.5F);
Float fRef = new Float(3.5);
Float fRef = new Float("3.5F");
```

Java.lang.Character (Contains only one constructor)

Public Character(char value)
Example:

```
Character cRef = new Character('a');
Character cRef = new Character("a"); //C.E
```

Java.lang.Boolean

Public Boolean(boolean value)
Public Boolean(String value)

Allocates a Boolean object representing the value true if the string argument is not null and is equal, ignoring case, to the string "true".

Example:

```
Boolean bRef = new Boolean("true");
Boolean bRef = new Boolean("True"); //true
Boolean bRef = new Boolean("TRUE");

Boolean bRef = new Boolean(null);
Boolean bRef = new Boolean("yes");   //false
Boolean bRef = new Boolean("no");
```

Boolean bRef = new Boolean("aspire");

Java.lang.Void

Although the Void class is considered as a wrapper class, it does not wrap any primitive value and is not instantiable i.e has no public constructors. It just denotes the Class object representing the keyword void.

Fields

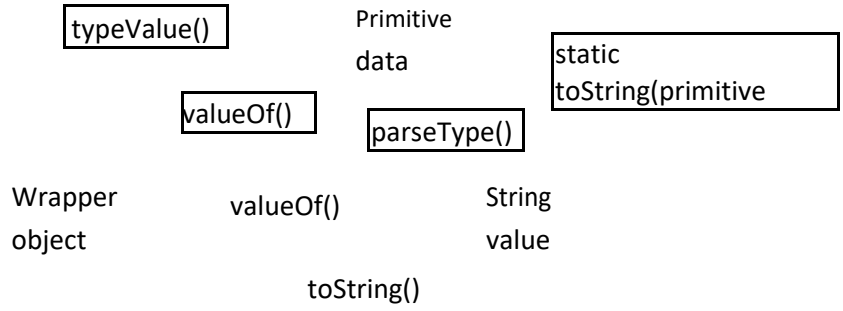
The following constants are declared for each wrapper class except Boolean:

Public static final primitive type MIN_VALUE
Public static final primitive type MAX_VALUE
Public static final int SIZE
Public static final Class TYPE

The following constants are declared only for Boolean class:

Public static final Boolean FALSE;
Public static final Boolean TRUE;
Public static final Class TYPE;

Methods



The methods are:

1. typeValue() methodsRetrieves primitive value from wrapper object.
2. parseType() methodsConverts numeric string into numeric value.
3. overloaded valueOf()Alternative to constructors.
4. overloaded toString()Returns string representation of primitive data.

1. <u>typeValue() methods</u>

Retrieving Primitive value from wrapper object.

a) Six Methods:

The following methods are applicable for all numeric wrapper classes:

public byte byteValue()
public short shortValue()
public int intValue()

All numeric wrapper classes [Byte, Short, Integer, Long, Float, Double] contains six methods.

public long longValue()
public float floatValue()
public double doubleValue()

b) booleanValue()

This method is applicable only for Boolean wrapper class.

Example:

```
Boolean bRef =new Boolean("yes");
SOP(bRef.booleanValue());// false
```

c) charValue()

This method is applicable only for Character wrapper class.

Example:

```
Character cRef = new Character('a');
SOP(cRef.charValue()); // 'a'
```

2. parseType(String) Methods

Every wrapper class except Character class contains following static parseType() method to convert String value to Primitive data.

Wrapper class	parseType(String)	Example
Java.lang.Byte	parseByte(String)	byte b = Byte.parseByte("1"); byte b = Byte.parseByte("one"); //NFE
Java.lang.Short	parseShort(String)	short s = Short.parseShort("1");
Java.lang.Integer	parseInt(String)	Int I = Integer.parseInt("1");
Java.lang.Long	parseLong(String)	Long l = Long.parseLong("1");
Java.lang.Float	parseFloat(String)	float f = Float.parseFloat("1.0F");
Java.lang.Double	parseDouble(String)	double d = Double.parseDouble("1.0");
Java.lang.Boolean	parseBoolean(String)	Boolean b = Boolean.parseBoolean("no");

3. valueOf() method

The overloaded static valueOf() methods are **alternative to constructors** to create wrapper object.

```
Public static Wrapper type valueOf(primitive type)
Public static Wrapper type valueOf(String)
```

Example:

```
Integer iRef = Integer.valueOf(10);
Integer iRef = Integer.valueOf("10");
Integer iRef = Integer.valueOf("ten");//throws NFE
```

4. toString()

All wrapper classes overrides toString() method to return wrapper object value in string format. Also, all wrapper classes have overloaded static toString(primitive type) method to convert primitive value into string format.

Example:

```
String str = Integer.toString(10);   // "10"
```

```
Integer iRef = new Integer(10);
String str = iRef.toString(); // “10”
```

Also, the Integer, Float, Double classes contains the following xxxString() methods for converting primitive to Binary, Octal, and Hexa format.

	toBinaryString(int)	toOctalString(int)	toHexString(int / float/ double)
Integer			
Float or Double			

Example:

```
String s = Integer.toBinaryString(10); //1010
String s = Integer.toHexString(10); //a
```

Autoboxing and Unboxing

The Auto & Un boxing first time introduced in **JDK1.5**.

Autoboxing (jdk1.5)

The java compiler automatically converts Primitive data to Wrapper object is called as autoboxing.

Example:

Before compilation:

```
Integer iRef = 10.
```

After compilation:

```
Integer iRef = new Integer(10); // Autoboxing
```

Unboxing

The java compiler automatically converts Wrapper object to primitive data is called as unboxing.

Example:

Before compilation:

```
Int I = new Integer(10);
```

After compilation:

```
Int I = new Integer(10).intValue();  //unboxing.
```

Reflection Mechanism

Reflection is the process of obtaining information about any java class i.e., getting information about fields, constructors, and methods of a java class.

Class Syntax:

<class modifiers> class <class-name> [extends <superclass name>] [implements interface$_1$, … interface$_n$] {

//Fields

<field modifiers> type name;

…

//Constructors

<constructor modifiers> name(<parameters list>){}

…

//Methods

<method modifiers> <return type> name(<parameters list>){}

}

To obtain information about classes, we have to use **java.lang.Class** and **java.lang.reflect** package.

Java.lang.Class

When the class is loaded into JVM, a class object is created in heap area. Such a class object contains complete information about specified class.

Methods:

1. public static Class **forName**(String fully qualified className) throws ClassNotFoundException
   Example:
   Class c = Class.forName("java.lang.String"); // Returns class object but not string type

2. Getting class modifiers
   Public int getModifiers();

3. Getting class name
   Public String getName() Returns fully qualified class name.
   Public String getSimpleName() Returns just class name without qualified with type.

4. Getting superclass information
   Public Class getSuperclass()

5. Getting interface(s) information
   Public Class[] getInterfaces()

6. Getting field(s) information
   Public **Field**[] getFields();

7. Getting constructor(s) information
   Public **Constructor**[] getConstructors()

8. Getting method(s) information

Public **Method**[] getMethods()

9. Creating new instance
   Public Object **newInstance()**

Java.lang.reflect.Field

Every field in a class must have the following format:
<**field modifiers**> <**field type**> <**field name**>;

Methods:

1) Getting field modifiers.
   Public int getModifiers()
2) Getting field type
   Public Class getType()
3) Getting field name
   Public String getName()

java.lang.reflect.Constructor

Every constructor in a class must have the following format:
<constructor modifier> <constructor name>(<parameters list>)

Methods:

1) Getting constructor modifier
   Public int getModifiers()

2) Getting constructor name
   Public String getName()

3) Getting constructor parameters
   Public Class[] getParameterTypes()

Java.lang.reflect.Method

Every method in a class must have the following format:
<method modifiers> <return type> <method name>(<parameters list>)

Methods:

1) Getting method modifiers
   Public int getModifiers()

2) Getting method return type
   Public Class getReturnType()

3) Getting method name
   Public String getName()

4) Getting method parameters list
   Public Class[] getParameterTypes()

Java Beans

[Reusable Software component]

Java bean is a reusable software component.

In generally, a Java Bean perform a specific task. Some Java beans are Visible and others are Invisible.

A Java bean may contains **Properties**, **Methods**, and **Events**.

Java beans Rules:

1) Java Bean class must be declared with **public** modifier.
2) Java Bean must support **Persistence** (**Serialization)** by extending java.io.Serializable interface.
3) Java Bean must support **Introspection.**

To support introspection, Beans must provide access methods to properties. 4) Java Bean must provides Zero-argument constructors.

Example:

```
package edu.aspire;  //Recommended

Public class Employee implements Serializable{

        Private int eno;

        Private String ename;              //Bean properties

        Private long mobile;

        Public Employee(){}

        Public void setEno(int eno){ this.eno = eno; }

        Public int getEno(){ return this.eno; }

        Public void setEname(String ename){ this.ename = ename; }
                                                    //Bean property methods.
        Public String getEname(){ return ename; }                    Public void
        setMobile(long mobile){ this.mobile = mobile; }

        Public String getMobile() { return this.mobile;}

}
```

Introspection Mechanism

Introspection is the process (or mechanism) of obtaining information about a Java Bean i.e., getting information about properties, events, and methods of a java bean class.

The Java bean class have following property types:

I. Simple properties
II. Boolean properties
III. Indexed properties

Naming Conventions

The following table detailing java bean class method naming conventions for every property type:

Property type	Naming patterns	Example
Simple	Public T getN() Public void setN(T value)	Public String getName(){} Public void setName(String name){}
Boolean	Public boolean isN() Public boolean getN() Public void setN(boolean value)	Public boolean isHoliday(){} Public boolean getHoliday(){} Public void setHoliday(boolean value){}
Indexed	Public T getN(int index) Public T[] getN() Public void setN(int index, T value) Public void setN(T[] value){}	Public boolean getInputs(int index){} Public boolean[] getInputs() Public void setInputs(int index, boolean value){} Public void setInputs(boolean[] values){}

Introspection mechanism uses **java.beans** package from JDK API to analyze Java Beans.

Java.beans.Introspector

It contains static methods that allow us to obtain information about properties, events, and methods of a bean. Public static BeanInfo getBeanInfo(Class beanClass);

The above static method returns BeanInfo object which contains complete Java Bean information.

Java.beans.BeanInfo<<interface>>

The following methods from BeanInfo object gives information about Properties, Events, and Methods of a Java Bean class.

1) Public PropertyDescriptor[] **getPropertyDescriptors**()
2) Public MethodDescriptor[] **getMethodDescriptors**()
3) Public EventSetDescriptor[] **getEventSetDescriptors**()

Example:

```
import java.awt.Button;
import java.beans.BeanInfo;
import java.beans.EventSetDescriptor;
import java.beans.IntrospectionException;
import java.beans.Introspector;
import  java.beans.MethodDescriptor;
import java.beans.PropertyDescriptor;
public class ItrospectionEx1 {
        public static void main(String[] args) {
                try {
                        BeanInfo beanInfo = Introspector.getBeanInfo(Button.class);
```

```
                    System.out.println("                    ----------");
                    System.out.println("Java Bean Properties:");
            System.out.println("----------          ");
            PropertyDescriptor[] props = beanInfo.getPropertyDescriptors();
            for (PropertyDescriptor prop : props) {
                System.out.println(prop.getPropertyType().getSimpleName() + " "
                        + prop.getName());
            }
            System.out.println("------------------          ");
            System.out.println("Java Bean Methods:");
            System.out.println("------------------          ");
            MethodDescriptor[] methods = beanInfo.getMethodDescriptors();
            for (MethodDescriptor method : methods) {
                System.out.println(method.getName());
            }
            System.out.println("-----------------          ");
            System.out.println("Java Bean Events:");
            System.out.println("-----------------          ");
            EventSetDescriptor[] events = beanInfo.getEventSetDescriptors();
            for(EventSetDescriptor event : events){
                System.out.println(event.getName());
            }
        } catch (IntrospectionException e) {
            e.printStackTrace();
        }
    }
}
```

Conclusion:

1) The Reflection mechanism work at a low level and deal only with fields, constructors, and methods of a class. The Introspection mechanism work at a higher level and deal with the properties, events, and methods of a Java Bean.
2) Introspection mechanism internally uses Reflection API to retrieve properties, events, or methods information.