Wrapper Classes

Use of Wrapper Classes

- In primitive Data types, user can not pass primitive type by reference on a method but they are passed by value to methods.
- That means user can not use any object as a reference on a method.
- This primitive data types are not part of the object hierarchy.
- To handle these situation, Java provides type wrappers, which are classes that encapsulate a primitive type within an object.

- Each of Java's eight primitive data types has a class dedicated to it. These are known as wrapper classes, because they "wrap" the primitive data type into an object of that class.
- So, there is an **Integer** class that holds an **int** variable, there is a **Double** class that holds a **double** variable, and so on.
- The **wrapper classes are part of the java.lang package**, which is imported by default into all Java programs.

- The following two statements illustrate the difference between a primitive data type and an object of a wrapper class:

```
int  x = 25;
Integer y = new Integer(33);
```

- The first statement declares an int variable named x and initializes it with the value 25.
- The second statement instantiates an Integer object. The object is initialized with the value 33 and a reference to the object is assigned to the object variable y.
- The memory assignments from these two statements are visualized in Figure 1.

Figure – 1 Variables vs. objects
(a) declaration and initialization of an int variable
(b) instantiation of an Integer object

- Clearly x and y differ by more than their values: x is a variable that holds a value; y is an object variable that holds a reference to an object.

- As noted earlier, data fields in objects are not, in general, directly accessible. So, the following statement using x and y as declared above is not allowed:

int z = x + y; // wrong!

- The data field in an Integer object is only accessible using the methods of the Integer class.
- One such method — the intValue() method — returns an int equal to the value of the object, effectively "unwrapping" the Integer object:

int z = x + y.intValue(); // OK!

Wrapper Classes

- Wrapper classes do exactly what they say.
- That mean sometime you want to treat a primitive type as an object at that time wrapper classes used.
- They wrap primitive (like int, float etc) into an object .
- Note that wrapper class object start with capital letters and primitives start with small letters.

- An instance of a wrapper contains, or *wraps*, a primitive value of the corresponding type.
- The wrapper classes also provide various tools such as constants and static methods.
- You will often use wrapper methods to convert a number type value to a string or a string to a number type.

This table lists the primitive types and the corresponding wrapper classes:

Primitive	Wrapper
boolean	java.lang.Boolean
byte	java.lang.Byte
char	java.lang.Character
double	java.lang.Double
float	java.lang.Float
int	java.lang.Integer
short	java.lang.Short
void	java.lang.Void
long	java.lang.Long

- The wrappers are normal classes that extend the **Object** superclass like all Java classes.

- **Wrapper class can be created by calling wrapper class constructor with the appropriate primitive value as a parameter to their constructor.**

- All the classes have two constructor forms

1. A constructor that takes the primitive type and creates an object,

   e.g. **Character (char), Integer (int)**

2. A constructor that converts a String into an object

   e.g. **Integer("1").**

- **The Character class does not have a constructor that takes a String argument .**

Converting primitive numbers to object numbers using constructor methods.

- **Primitive integer to Integer object**
  - Integer IntVal = new Integer (10)

- **Primitive float to Float object**
  - Float FloatVal = new Float (10.50)

- **Primitive double to Double object**
  - Double DoubleVal = new Double(10.123654)

Converting String object to Numeric object using static method valueOf()

- **Integer from String:**
  Integer i = Integer.valueOf(”125″);

- **Double from String:**
  Double d = Double.valueOf(”5.829754097″);

- **Float from String:**
  Float f = Float.valueOf(”8.43543″);

Converting Numeric to String object using method toString()

- **Primitive integer to String object**
  - Str = Integer.toString (i);

- **Primitive float to Float object**
  - Str= Float.toString (f);

- **Primitive double to Double object**
  - Str = Double.toString (d);

How to convert a variable to wrapper classes and back to a primitive.

- **Integer:**
  ```
  int i = 5;
  Integer I = Integer.valueOf(i);   // wrapper
  int i2 = I.intValue(); //back to primitive
  ```

- **Float:**
  ```
  float f = 5.5f;
  Float F = Float.valueOf(f); //wrapper
  float f2 = F.floatValue(); //back to primitive
  ```

Converting Numeric String to Primitive Numbers Using Parsing Methods.

- **Integer:**
  int i = Integer.parseInt(”234”);

- **Double**
  double d =
  Double.parseDouble(”234.6576533254”);

- **Float**
  float f = Float.parseFloat(”234.78”);

- **Long**
  long l = Long.parseLong(”23454654”);

Autoboxing and Auto-unboxing

- Autoboxing is the process by which a primitive type is automatically encapsulated into its equivalent wrapper type whenever an object of that type is needed.
- There is no need to explicitly construct an object.

- Example: To autobox an object

Construct an Integer object that has value 100

Integer iob = 100; // autobox an int

- Notice that no object is explicitly created using **new** keyword.

- **Auto-unboxing** is the process by which the value of a encapsulated object is automatically extracted from a wrapper type when its value is needed.

- You need only assign the value to a wrapper-type reference.

- Java automatically constructs the object.

- Example: To auto-unbox an object

```
int i = iob; // auto-unbox
```

Enumeration

Enumerations

- Enumeration is a list of named constants.
- In languages such as C++, enumerations are simply list of named integer constants.
- In Java, an enumerations defines a class type.
- Enumeration can have constructors, methods, and instance variables.

Enumeration Fundamentals

- An enumeration is created using the new **enum** keywords**.**

- Example : A simple enumeration that lists various Color types.

```
// An enumeration of color varieties.
Enum Color {
 RED, BLACK, BLUE, GREEN, PINK
}
```

- The identifiers Red, Blue, and so on, are called **enumeration constants**.

- Each is implicitly declares as a public, static member of **Color**.

- Their type is the type of the enumeration in which they are declares, which is **Color** in our case.

- Once you have defined an enumeration, you can create a variable of that type.

- Enumeration define a class type, you do not instantiate an **enum** using **new**.

- Instead, you declare and use an enumeration variable in much the same way as you do one of the primitive types.

- Example :

Declare **cp** as a variable of enumeration type **Color :**

Color cp;

- The only values that it can be assigned to Color variable type ‘cp’ are those defined by the enumerations.

- Example :

  cp = Color. Red;

- Notice that the symbol Red is preceded by Color.

- Two enumeration constants can be compared for equality by using == relational operator.

- Example :

  If (cp ==Color. Red) …

- An enumeration value can be used to control a **switch** statement.

- Example :

```
switch (cp) {
        case Red:
        case Blue:
                }
```

The values() and valueOf() methods

- All enumerations automatically contain two predefined methods :
  - **values()**
  - **valueof()**
- Their general forms are shown here:

public static enum-type[] values()
public static enum-type valueOf() (String str)

- The **values()** method returns an array that contains a list of the enumeration constants.

- The **valueOf()** method returns the enumeration constant whose value corresponds to the string passed in str.

- In both cases, enum-type is the type of the enumerations.

Assertions

- The ***assert*** keyword is used during program development to create an **assertion**, which is ***a condition that should be true during the execution of the program.***

- For example, you might have a method that should always return a positive integer value.
- You might test this by asserting that the return value is greater than zero using an assert statement.
- At run time, if the **condition actually is true, no other action** takes place.
- However, if the **condition is false**, then an **AssertionError** is thrown.

- Assertions are often **used during testing** to verify that some expected condition is actually met.
- They are not usually used for released code.

- The assert keyword has two forms.

- The first is shown here.

  assert *condition;*

- Here, *condition* is an expression that must evaluate to a Boolean result.
- If the result is true, then the assertion is true and no other action takes place.
- If the condition is false, then the assertion fails and a default **AssertionError object is thrown.**

- The second form of assert is shown here.

  assert *condition : expr;*

- In this version, *expr* is a value that is passed to the ***AssertionError constructor.***
- This value is **converted to its string format** and displayed if an assertion fails.
- Typically, you will specify a string for *expr,* but **any non-void *expression is allowed*** as long as it defines a reasonable string conversion.

Regular Expression in Java

Overview

- The RE can be processed with the use of **java.util.regex** package which supports regular expression processing.
- A ***regular expression*** *(RE) is* a string of characters that describes a character sequence.
- This general description, called a ***pattern***, can then be used to find matches in other character sequences.
- Regular expressions can specify wildcard characters, sets of characters, and various quantifiers.
- Thus, you can specify a regular expression that represents a general form that can match several different specific character sequences.

- There are two classes that support regular expression processing: **Pattern and Matcher.**
- These classes work together.

- Use **Pattern** to define a regular expression.
- Match the pattern against another sequence using **Matcher.**

Pattern

- The Pattern class defines **no constructors**.
- Instead, a pattern is created by calling the **compile( )** factory method.
- One of its forms is shown here:

  static Pattern compile(String *pattern)*

- Here, pattern is the regular expression that you want to use.
- The **compile( )** method **transforms the string in pattern into a pattern** that can be used for pattern matching by the Matcher class.

- It **returns a Pattern object** that contains the pattern.

- Once you have created a Pattern object, you will use it to create a Matcher.
- This is done by calling the **matcher( )** factory method defined by Pattern.
- It is shown here:

  Matcher matcher(CharSequence str)
- Here str is the character sequence that the pattern will be matched against.
- This is called the *input sequence*.
- CharSequence is an interface that defines a read-only set of characters.
- It is implemented by the String class. Thus, you can pass a string to matcher( ).

Matcher

- The Matcher class has **no constructors**.
- Instead, you create a Matcher by calling the **matcher( )** factory method defined by Pattern.
- Once you have created a Matcher, you will use its methods to perform various pattern matching operations.

- The simplest pattern matching method is **matches( )**, which simply **determines whether the character sequence matches the pattern**.
- It is shown here:

  boolean matches( )
- It returns true if the sequence and the pattern match, and false otherwise.

- To determine if a subsequence of the input sequence matches the pattern, use **find( ).**
- One version is shown here:

  boolean find( )

- It returns true if there is a matching subsequence and false otherwise.
- This method **can be called repeatedly**, allowing it to find all matching subsequences.

- Each call to find( ) begins where the previous one left off.

- You can obtain a string containing the last matching sequence by calling **group( ).**
- One of its forms is shown here:

  String group( )
- The matching string is returned.
- If no match exists, then an **IllegalStateException** is thrown.

- You can obtain the index within the input sequence of the current match by calling **start( ).**
- The index one past the end of the current match is obtained by calling end( ).
- These methods are shown here:

  int start( )

  int end( )

- You can replace all occurrences of a matching sequence with another sequence by calling **replaceAll( )**, shown here:

  String replaceAll(String *newStr*)

- Here, *newStr* specifies the new character sequence that will replace the ones that match the pattern.
- The updated input sequence is returned as a string.

Regular Expression Syntax

- In general, a regular expression is comprised of normal characters, character classes (sets of characters), wildcard characters, and quantifiers.

- Characters such as newline and tab are specified using the standard escape sequences, which begin with a **\\.**

- In the language of regular expressions, a normal character is also called **a *literal.***

- A character class is a set of characters.
- A character class is specified by putting the characters in the class between brackets.

Character Classes	
[abc]	a, b, or c (simple class)
[^abc]	Any character except a, b, or c (negation)
[a-zA-Z]	a through z or A through Z, inclusive (range)
[a-z-[bc]]	a through z, except for b and c: [ad-z] (subtraction)
[a-z-[m-p]]	a through z, except for m through p: [a-lq-z]
[a-z-[^def]]	d, e, or f

Predefine Character Classes	
.	Any character (may or may not match line terminators)
\d	A digit: [0-9]
\D	A non-digit: [^0-9]
\s	A whitespace character: [\t\n\x0B\f\r]
\S	A non-whitespace character: [^\s]
\w	A word character: [a-zA-Z_0-9]
\W	A non-word character: [^\w]

- The **wildcard** character is the . (dot) and it matches any character.
- Thus, a pattern that consists of "." will match these (and other) input seqeunces: "A", "a", "x", and so on.

- A **quantifier** determines how many times an expression is matched.
- The quantifiers are shown here:

  + → Match one or more.

  * → Match zero or more.

  ? → Match zero or one.