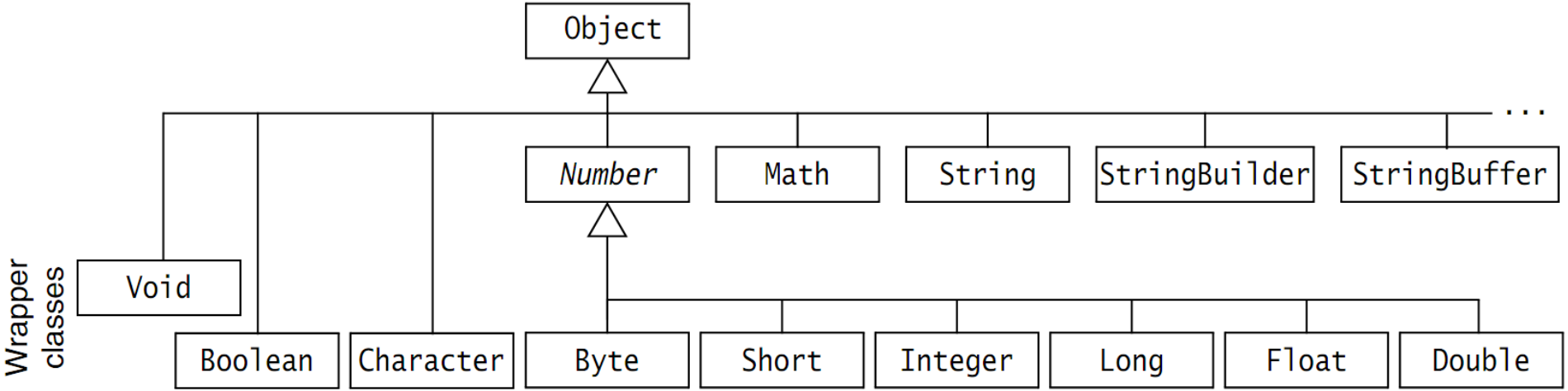
Fundamental Classes

Overview of the `java.lang` Package



The Object Class

The Object class provides the following general utility methods

- `int hashCode()`
- `boolean equals(Object obj)`
- `final Class<?> getClass()`
  - Returns the runtime class of the object, which is represented by an object of the class java.lang.Class at runtime.
- protected Object clone() throws CloneNotSupportedException
- `String toString()`
  - Returns by default "<name of the class>@<hash code value of object>“
- `protected void finalize() throws Throwable`

The Object Class

- In addition, the Object class provides support for thread communication in synchronized code, through the following methods, which are discussed in Section 13:

```
final void wait(long timeout) throws InterruptedException
final void wait(long timeout, int nanos) throws InterruptedException
final void wait() throws InterruptedException
final void notify()
final void notifyAll()
```

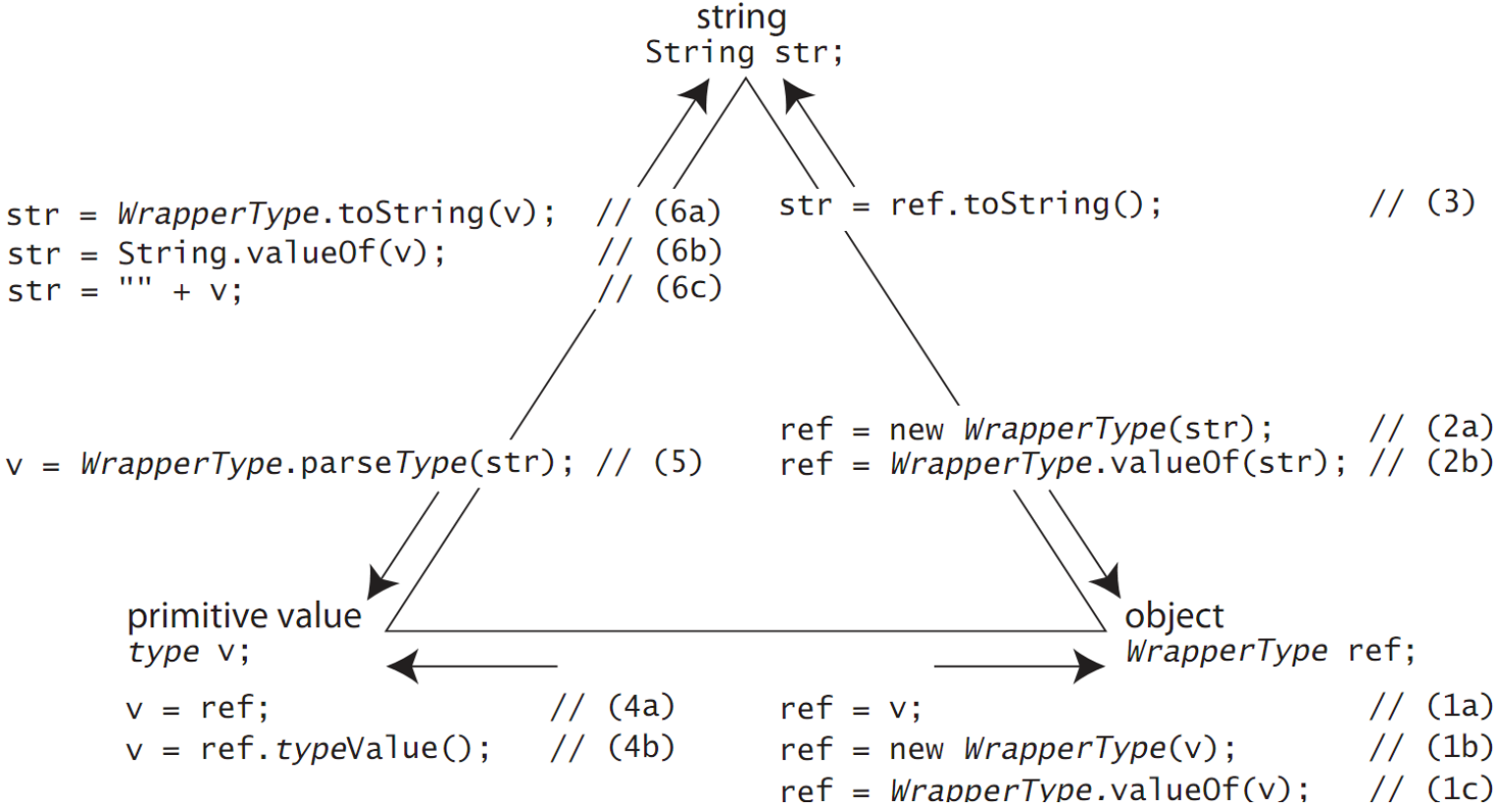
The Wrapper Classes

- Wrapper classes were introduced with the discussion of the primitive data types and also in connection with boxing and unboxing of primitive values.
- Primitive values in Java are not objects. In order to manipulate these values as objects, the java.lang package provides a wrapper class for each of the primitive data types.
- All wrapper classes are final.
- The objects of all wrapper classes that can be instantiated are immutable, i.e., the value in the wrapper object cannot be changed.

The Void class

- Although the Void class is considered 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. The Void class will not be discussed further

Converting Values Between Primitive, Wrapper, and String Types



Converting Values Between Primitive, Wrapper, and String Types

type is:	*WrapperType* is:	*Comments:*
`boolean`	`Boolean`	(1a) Boxing
`char`	`Character`	(2a) Not for `Character` type. Can throw `NumberFormatException`.
`byte`	`Byte`	(2b) Not for `Character` type. Can throw `NumberFormatException`.
`short`	`Short`	(4a) Unboxing
`int`	`Integer`	(5) For numeric wrapper types only. Can throw `NumberFormatException`.
`long`	`Long`	(6b) Not for `byte` and `short` primitive types.
`float`	`Float`	
`double`	`Double`	

Common Wrapper Class Constructors

- The Character class has only one public constructor, taking a char value as parameter.
- The other wrapper classes all have two public one-argument constructors:
  - one takes a primitive value

    and the
  - other takes a string.

Wrapping Primitive Values in Objects

- Boxing is a convenient way to wrap a primitive value in an object (see 1a)

```
Character charObj1    = '\n';
Boolean   boolObj1    = true;
Integer   intObj1     = 2008;
Double    doubleObj1 = 3.14;
```

Wrapping Primitive Values in Objects

- A constructor that takes a primitive value can be used to create wrapper objects (see 1b)

```
Character charObj1 = new Character('\n');
Boolean   boolObj1   = new Boolean(true);
Integer   intObj1    = new Integer(2008);
Double    doubleObj1 = new Double(3.14);
```

Wrapping Primitive Values in Objects

- We can also use the valueOf() method that takes the primitive value to wrap as an argument (see 1c)

```
Character charObj1 = Character.valueOf('\n');
Boolean   boolObj1   = Boolean.valueOf(true);
Integer   intObj1  = Integer.valueOf(2008);
Double    doubleObj1 = Double.valueOf(3.14);
```

Converting Strings to Wrapper Objects

```
Boolean boolObj2 = new Boolean("TrUe");
                                    // case ignored: true
Boolean boolObj3 = new Boolean("XX");      // false
Integer intObj2 = new Integer("2008");
Double  doubleObj2 = new Double("3.14");
Long    longObj1   = new Long("3.14");
                                  // NumberFormatException
```

Common Wrapper Class Utility Methods

- Converting Strings to Wrapper Objects

Each wrapper class (except Character) defines the static method valueOf(String str) that returns the wrapper object corresponding to the primitive value represented by the String object passed as argument (see 6b).

This method for the numeric wrapper types also throws a NumberFormatException if the String parameter is not a valid number.

Common Wrapper Class Utility Methods

- Converting Strings to Wrapper Objects

```
Boolean boolObj4 = Boolean.valueOf("false");
Integer intObj3 = Integer.valueOf("1949");
Double  doubleObj3 = Double.valueOf("-3.0");
```

Common Wrapper Class Utility Methods

In addition to the one-argument valueOf() method, the integer wrapper classes define an overloaded static valueOf() method that can take a second argument.

This argument specifies the base (or radix) in which to interpret the string representing the signed integer in the first argument:

Common Wrapper Class Utility Methods

```
Byte    byteObj1  = Byte.valueOf("1010", 2);
                              // Decimal value 10
Short   shortObj2 = Short.valueOf("012", 8);
                    // Not "\012". Decimal value 10.
Integer intObj4   = Integer.valueOf("-a", 16);
                   // Not "-0xa". Decimal value -10.
Long    longObj2  = Long.valueOf("-a", 16);
                  // Not "-0xa". Decimal value -10L.
```

Common Wrapper Class Utility Methods

- Converting Wrapper Objects to Strings

Each wrapper class overrides the toString() method from the Object class. The overriding method returns a String object containing the string representation of the primitive value in the wrapper object (see 3)

Common Wrapper Class Utility Methods

- String toString()

```
String charStr   = charObj1.toString();   // "\n"
String boolStr   = boolObj2.toString();   // "true"
String intStr    = intObj1.toString();    // "2008"
String doubleStr = doubleObj1.toString(); // "3.14"
```

Common Wrapper Class Utility Methods

- Converting Primitive Values to Strings

Each wrapper class defines a static method toString(type v) that returns the string corresponding to the primitive value of type, passed as argument (see 6a)

Common Wrapper Class Utility Methods

- static String toString(type v)

```
String charStr2   = Character.toString('\n');  // "\n"
String boolStr2   = Boolean.toString(true);    // "true"
String intStr2    = Integer.toString(2008);
                                        // Base 10. "2008"
String doubleStr2 = Double.toString(3.14);   // "3.14"
```

Common Wrapper Class Utility Methods

- Converting Wrapper Objects to Primitive Values
  - Unboxing is a convenient way to unwrap the primitive value in a wrapper object (see 4a)
  - Each wrapper class defines a typeValue() method which returns the primitive value in the wrapper object (see 4b)

Common Wrapper Class Utility Methods

- type typeValue()

```
char    c = charObj1.charValue();       // '\n'
boolean b = boolObj2.booleanValue();    // true
int     i = intObj1.intValue();         // 2008
double  d = doubleObj1.doubleValue();   // 3.14
```

Wrapper Comparison, Equality, and Hashcode

- Each wrapper class also implements the Comparable< > interface, which defines the following method:

```
int compareTo(Type  obj2)
```

This method returns a value which is less than, equal to, or greater than zero, depending on whether the primitive value in the current wrapper Type object is less than, equal to, or greater than the primitive value in the wrapper Type object denoted by argument obj2.

Wrapper Comparison, Equality, and Hashcode

```
// Comparisons based on objects created above
Character charObj2   = 'a';
int result1 = charObj1.compareTo(charObj2);
//  < 0
int result2 = intObj1.compareTo(intObj3);
//  > 0
int result3 = doubleObj1.compareTo(doubleObj2);
// == 0
int result4 = doubleObj1.compareTo(intObj1);
// ClassCastException
```

Wrapper Comparison, Equality, and Hashcode

- Each wrapper class overrides the equals() method from the Object class. The overriding method compares two wrapper objects for object value equality.

```
boolean equals(Object obj2)
```

Wrapper Comparison, Equality, and Hashcode

- `// Comparisons based on objects created above`
- `boolean charTest = charObj1.equals(charObj2);`
  `// false`
- `boolean boolTest = boolObj2.equals(Boolean.FALSE);`
  `// false`
- `boolean intTest    = intObj1.equals(intObj2);`
  `// true`
- `boolean doubleTest = doubleObj1.equals(doubleObj2);`
  `// true`

Wrapper Comparison, Equality, and Hashcode

- The following values are interned when they are wrapped during boxing, i.e., only one wrapper object exists in the program for these primitive values when boxing is applied:
  - `boolean values true or false,`
  - `a byte,`
  - `a char in the range \u0000 to \u007f,`
  - `an int or short value in the range -128...127`

Wrapper Comparison, Equality, and Hashcode

- If references w1 and w2 refer to two wrapper objects that box the same value which is among the ones mentioned above, then w1 == w2 is always true.
- In other words, for the values listed above, object equality and reference equality give the same result.

Wrapper Comparison, Equality, and Hashcode

```
// Reference and object equality
Byte bRef1 = (byte)10;
Byte bRef2 = (byte)10;
System.out.println(bRef1 == bRef2);                 // true
System.out.println(bRef1.equals(bRef2));            // true
Integer iRef1 = 1000;
Integer iRef2 = 1000;
System.out.println(iRef1 == iRef2);                 // false
System.out.println(iRef1.equals(iRef2));            // true
```

Wrapper Comparison, Equality, and Hashcode

- Each wrapper class also overrides the hashCode() method in the Object class. The overriding method returns a hash value based on the primitive value in the wrapper object.

```
int hashCode()
int index = charObj1.hashCode();
```

Numeric Wrapper Classes

- The numeric wrapper classes Byte, Short, Integer, Long, Float, and Double are all subclasses of the abstract class Number.
- Each numeric wrapper class defines an assortment of constants, including the minimum and maximum value of the corresponding primitive data type:

Numeric Wrapper Classes

NumericWrapperType.MIN_VALUE

NumericWrapperType.MAX_VALUE

The following code retrieves the minimum and maximum values of various numeric types:

```
byte minByte   = Byte.MIN_VALUE;      // -128
int  maxInt = Integer.MAX_VALUE;  // 2147483647
double maxDouble = Double.MAX_VALUE;
                        // 1.79769313486231570e+308
```

Converting Numeric Wrapper Objects to Numeric Primitive Types

- Each numeric wrapper class defines the following set of typeValue() methods for converting the primitive value in the wrapper object to a value of any numeric primitive type
  - `byte   byteValue()`
  - `short  shortValue()`
  - `int    intValue()`
  - `long   longValue()`
  - `float  floatValue()`
  - `double doubleValue()`

Converting Numeric Wrapper Objects to Numeric Primitive Types

- The following code shows conversion of values in numeric wrapper objects to any numeric primitive type.

```
Byte byteObj2   = new Byte((byte) 16);
                                        // Cast mandatory
Integer intObj5 = new Integer(42030);
Double doubleObj4 = new Double(Math.PI);
short  shortVal  = intObj5.shortValue(); // (1)
long   longVal = byteObj2.longValue();
int    intVal = doubleObj4.intValue();
                                        // (2) Truncation
double doubleVal = intObj5.doubleValue();
```

Converting Strings to Numeric Values

- Each numeric wrapper class defines a static method parseType(String str), which returns the primitive numeric value represented by the String object passed as argument.
- The Type in the method name parseType stands for the name of a numeric wrapper class, except for the name of the Integer class which is abbreviated to Int.
- These methods throw a NumberFormatException if the String parameter is not a valid argument (see 5)

Converting Strings to Numeric Values

```
byte   value1 = Byte.parseByte("16");
int    value2 = Integer.parseInt("2010");
                // parseInt, not parseInteger.
int    value3 = Integer.parseInt("7UP");
                // NumberFormatException
double value4 = Double.parseDouble("3.14");
```

Converting Strings to Numeric Values

- For the integer wrapper types, the overloaded static method parseType() can additionally take a second argument, which can specify the base in which to interpret the string representing the signed integer in the first argument:

```
type parseType (String str, int base)
```

Converting Strings to Numeric Values

```
byte  value6 = Byte.parseByte("1010", 2);
                                  // Decimal value 10
short value7 = Short.parseShort("012", 8);
            // Not "\012". Decimal value 10.
int   value8 = Integer.parseInt("-a", 16);
           // Not "-0xa". Decimal value -10.
long  value9 = Long.parseLong("-a", 16);
          // Not "-0xa". Decimal value -10L.
```

Converting Integer Values to Strings in Different Notations

- The wrapper classes Integer and Long provide static methods for converting integers to string representation in decimal, binary, octal, and hexadecimal notation.

```
static String toBinaryString(int i)
static String toHexString(int i)
static String toOctalString(int i)
```

These three methods return a string representation of the integer argument as an *unsigned* integer in base 2, 16, and 8, respectively, with no extra leading zeroes .

Converting Integer Values to Strings in Different Notations

static String toString(int i, int base)

static String toString(int i)

- The first method returns the minus sign '-' as the first character if the integer i is negative. In all cases, it returns the string representation of the magnitude of the integer i in the specified base.
- ➢ The last method is equivalent to the method toString(int i, int base), where the base has the value 10, that returns the string representation as a signed decimal (see also 6a)

The Character Class

- The Character class defines a myriad of constants, including the following which represent the minimum and the maximum value of the char type:
  - Character.MIN_VALUE
  - Character.MAX_VALUE

The Character Class

- The Character class also defines a plethora of static methods for handling various attributes of a character, and case issues relating to characters, as defined by the Unicode standard, version 4.0:

```
static int getNumericValue(char ch)
static boolean isLowerCase(char ch)
static boolean isUpperCase(char ch)
static boolean isTitleCase(char ch)
static boolean isDigit(char ch)
static boolean isLetter(char ch)
static boolean isLetterOrDigit(char ch)
static char    toUpperCase(char ch)
static char    toLowerCase(char ch)
static char    toTitleCase(char ch)
```

The Boolean Class

- The Boolean class defines the following wrapper objects to represent the primitive values true and false, respectively:
  - `Boolean.TRUE`
  - `Boolean.FALSE`

The String Class

Immutability

The **String** class implements immutable character strings, which are read-only once the string has been created and initialized, whereas the **StringBuilder** class implements dynamic character strings.

The **StringBuffer** class is a thread-safe version of the **StringBuilder** class.

Creating and Initializing Strings

- String Literals Revisited

The easiest way of creating a String object is using a string literal:

```
String str1 = "You cannot change me!";
```

A string literal can be used to invoke methods on its String object:

```
int strLength = "You cannot change me!".length();
```

Creating and Initializing Strings

- The compiler optimizes handling of string literals (and compile-time constant expressions that evaluate to strings): only one String object is shared by all stringvalued constant expressions with the same character sequence.
- Such strings are said to be interned, meaning that they share a unique String object if they have the same content. The String class maintains a private pool where such strings are interned.

Creating and Initializing Strings

- The String class maintains a private pool where such strings are interned.

```
String str2 = "You cannot change me!";
```

- Both String references str1 and str2 denote the same String object, initialized with the character string: "You cannot change me!".

Creating and Initializing Strings

- So does the reference str3 in the following code. The compile-time evaluation of the constant expression involving the two string literals, results in a string that is already interned:

```
String str3 = "You cannot" + " change me!";
// Compile-time constant expression
```

Creating and Initializing Strings

- In the following code, both the references can1 and can2 denote the same String object that contains the string "7Up":

```
String can1 = 7 + "Up";
// Value of compile-time constant expression: "7Up"
String can2 = "7Up";    // "7Up"
```

- However, in the code below, the reference can4 will denote a new String object that will have the value "7Up" at runtime:

```
String word = "Up";
String can4 = 7 + word;
// Not a compile-time constant expression.
```

String Constructors

- The String class has numerous constructors to create and initialize String objects based on various types of arguments. Here we present a few selected constructors:
- Note that using a constructor creates a brand new String object, i.e., using a constructor does not intern the string. A reference to an interned string can be obtained by calling the intern() method in the String class—in practice, there is usually no reason to do so.

```
String str4 = new String("You cannot change me!");
```

String Constructors

`String()`

This constructor creates a new String object, whose content is the empty string, "".

`String(String str)`

This constructor creates a new String object, whose contents are the same as those of the String object passed as argument.

`String(char[] value)`

`String(char[] value, int offset, int count)`

These constructors create a new String object whose contents are copied from a char array. The second constructor allows extraction of a certain number of characters (count) from a given offset in the array.

`String(StringBuilder builder)`

`String(StringBuffer buffer)`

These constructors allow interoperability with the StringBuilder and the StringBuffer class, respectively.

The CharSequence Interface

- This interface is implemented by all three classes: String, StringBuilder and StringBuffer. Many methods in these classes accept arguments of this interface type, and specify it as their return type.
- This facilitates interoperability between these classes. This interface defines the following methods:

The CharSequence Interface

- char charAt(int index)
- int length()
- CharSequence subSequence(int start, int end)
- String toString()

Reading Characters from a String

- char charAt(int index)
- void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)
- int length()
- boolean isEmpty()

Comparing Strings

- Characters are compared based on their Unicode values.

boolean test = 'a' < 'b'; // true since 0x61 < 0x62

- Two strings are compared lexicographically, as in a dictionary or telephone directory, by successively comparing their corresponding characters at each position in the two strings, starting with the characters in the first position.
- The string "abba" is less than "aha", since the second character 'b' in the string "abba" is less than the second character 'h' in the string "aha".

Comparing Strings

```
boolean equals(Object obj)
boolean equalsIgnoreCase(String str2)
int compareTo(String str2)
```

The String class implements the Comparable<String> interface.

The compareTo() method compares the two strings and returns a value based on the outcome of the comparison:

- the value 0, if this string is equal to the string argument
- a value less than 0, if this string is lexicographically less than the string argument
- a value greater than 0, if this string is lexicographically greater than the string argument

Character Case in a String

- `String toUpperCase()`
- `String toUpperCase(Locale locale)`
- `String toLowerCase()`
- `String toLowerCase(Locale locale)`

Concatenation of Strings

- Concatenation of two strings results in a string that consists of the characters of the first string followed by the characters of the second string. The overloaded operator + for string concatenation was discussed before.
- In addition, the following method can be used to concatenate two strings:

```
String concat(String str)
```

Searching for Characters and Substrings

```
int indexOf(int ch)
int indexOf(int ch, int fromIndex)
int indexOf(String str)
int indexOf(String str, int fromIndex)
int lastIndexOf(int ch)
int lastIndexOf(int ch, int fromIndex)
int lastIndexOf(String str)
int lastIndexOf(String str, int fromIndex)
String replace(char oldChar, char newChar)
String replace(CharSequence target,
               CharSequence replacement)
boolean contains(CharSequence cs)
```

Extracting Substrings

```
String trim()
String substring(int startIndex)
String substring(int startIndex,
                 int endIndex)
```

Converting Primitive Values and Objects to Strings

```
String toString()
static String valueOf(Object obj)
static String valueOf(char[] charArray)
static String valueOf(boolean b)
static String valueOf(char c)
static String valueOf(int i)
static String valueOf(long l)
static String valueOf(float f)
static String valueOf(double d)
```

Formatting Values

- The String class provides support for formatted text representation of primitive values and objects through its overloaded format() methods.

```
static String format(String format, Object... args)
static String format(Locale l, String format,
                                        Object... args)
```

Pattern Matching

- Methods for string pattern matching take an argument that specifies a regular expression

The following method attempts to match the current string against the specified regular expression. The call

```
str.matches(regexStr);
```

is equivalent to the call

```
Pattern.matches(regexStr, str);
//java.util.regex.Pattern class
```

Matching and Replacing

- The following methods can be used to replace substrings that match a given regular expression. The call

```
str.replaceFirst(regexStr, replacement);
```

- is equivalent to the call

```
Pattern.compile(regexStr).matcher(str).
                    replaceFirst(replacement);
```

Splitting

- The split() method can be called on a string to create an array by splitting the string according to a regular expression pattern. Given that the reference input is of type String, the call

```
input.split(regexStr,limit);
```

is equivalent to the call

```
Pattern.compile(regexStr).split(input,limit);
```

The StringBuilder and the StringBuffer Classes

Thread-Safety

- The classes StringBuilder and StringBuffer implement mutable sequences of characters. Both classes support the same operations. However, the StringBuffer class is the thread-safe analog of the StringBuilder class.
- Certain operations on a StringBuffer are synchronized, so that when used by multiple threads, these operationsare performed in an orderly way

Mutability

- In contrast to the String class, which implements immutable character sequences, the StringBuilder class implements mutable character sequences.
- Not only can the character sequences in a string builder be changed, but the capacity of the string builder can also change dynamically.
- The capacity of a string builder is the maximum number of characters that a string builder can accommodate before its size is automatically augmented.

Mutability

The StringBuilder class provides various facilities for manipulating string builders:

- constructing string builders
- changing, deleting, and reading characters in string builders
- constructing strings from string builders
- appending, inserting, and deleting in string builders
- controlling string builder capacity

Constructing String Builders

- `StringBuilder(String str)`
- `StringBuilder(CharSequence charSeq)`
- `StringBuilder(int length)`
- `StringBuilder()`

Reading and Changing Characters in String Builders

- int length()
- char charAt(int index)
- void setCharAt(int index, char ch)
- CharSequence subSequence(int start, int end)

Constructing Strings from String Builders

The StringBuilder class overrides the toString() method from the Object class. It returns the contents of a string builder in a String object.

String fromBuilder = strBuilder.toString(); // "Java"

Differences between the String and StringBuilder Classes

- Since the StringBuilder class does not override the equals() method from the Object class, nor does it implement the Comparable interface, the contents of string builders should be converted to String objects for string comparison.
- The StringBuilder class also does not override the hashCode() method from the Object class. Again, a string builder can be converted to a String object in order to obtain a hash value.

Appending, Inserting, and Deleting Characters in String Builders

```
StringBuilder append(Object obj)
StringBuilder append(String str)
StringBuilder append(CharSequence charSeq)
StringBuilder append(CharSequence charSeq, int start, int end)
StringBuilder append(char[] charArray)
StringBuilder append(char[] charArray, int offset, int length)
StringBuilder append(char c)
StringBuilder append(boolean b)
StringBuilder append(int i)
StringBuilder append(long l)
StringBuilder append(float f)
StringBuilder append(double d)
```

Inserting Characters in a String Builder

```
StringBuilder insert(int offset, Object obj)
StringBuilder insert(int dstOffset, CharSequence seq)
StringBuilder insert(int dstOffset, CharSequence seq,
                     int start, int end)
StringBuilder insert(int offset, String str)
StringBuilder insert(int offset, char[] charArray)
StringBuilder insert(int offset, char c)
StringBuilder insert(int offset, boolean b)
StringBuilder insert(int offset, int i)
StringBuilder insert(int offset, long l)
StringBuilder insert(int offset, float f)
StringBuilder insert(int offset, double d)
```

Deleting Characters in a String Builder

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

Among other miscellaneous methods included in the class StringBuilder is the following method, which reverses the contents of a string builder:

```
StringBuilder reverse()
```

Example of StringBuilder

- The compiler uses string builders to implement the string concatenation, +. The following example code of string concatenation

```
String str1 = 4 + "U" + "Only";          // (1) "4UOnly"
```

is equivalent to the following code using one string builder:

```
String str2 = new StringBuilder().
append(4).append("U").append("Only").toString(); // (2)
```

Controlling String Builder Capacity

```
int capacity()
void ensureCapacity(int minCapacity)
void trimToSize()
void setLength(int newLength)
```

That’s all!