Immutable Strings

- Objects of the `String` class are immutable.
- **String** class
  - every method in the class that appears to modify a `String` actually creates and returns a brand new `String` object containing the modification.
- The original `String` is left untouched.

```java
import static net.mindview.util.Print.*;
public class Immutable {
    public static String upcase(String s) {
        return s.toUpperCase();
    }
    public static void main(String[] args) {
        String q = "howdy";
        print(q); // howdy
        print(q); // howdy
        String qq = upcase(q);
        print(qq); // HOWDY
    }
}
```

>> howdy HOWDY howdy
Overloading ‘+’ vs. StringBuilder

- operator ‘+’ has been overloaded for String objects.
- Overloading
  - an operation has been given an extra meaning when used with a particular class.
  - (The ‘+’ and ‘+=’ for String are the only operators that are overloaded in Java, and Java does not allow the programmer to overload any others.)
  - The ‘+’ operator allows you to concatenate Strings

```java
public class Concatenation {
    public static void main(String[] args) {
        String mango = "mango";
        String s = "abc" + mango + "def" + 47;
        System.out.println(s);
    }
}
```

> abcmangodef47
Overloading ‘+’ vs. StringBuilder
Cont’d

public class UsingStringBuilder {
    public static Random rand = new Random(47);
    public String toString() {
        StringBuilder result = new StringBuilder("[");
        for(int i = 0; i < 25; i++) {
            result.append(rand.nextInt(100));
            result.append(", ");
        }
        result.delete(result.length()-2, result.length());
        result.append("]");
        return result.toString();
    }
    result.delete(result.length()-2, result.length());
    result.append("[");
    return result.toString();
}
public static void main(String[] args) {
    UsingStringBuilder usb = new UsingStringBuilder();
    System.out.println(usb);
}

>>
[58, 55, 93, 61, 61, 29, 68, 0, 22, 7, 88, 28, 51, 89, 9, 78, 98, 61, 20, 58, 16, 40, 11, 22, 4]
Overloading ‘+’ vs. StringBuilder

• `append(a + "": " + c)`
  – the compiler will jump in and start making more `StringBuilder` objects again.

• **StringBuilder** has a full complement of methods
  – `insert()`, `replace()`, `substring()` and even `reverse()`

• But the ones you will generally use are `append()` and `toString()`.

• Note the use of `delete()` to remove the last comma and space before adding the closing square bracket.

• **StringBuilder**
  – introduced in Java SE5
  – Prior: `StringBuffer`, which ensured thread safety
# Operations on Strings

<table>
<thead>
<tr>
<th>Method</th>
<th>Arguments, Overloading</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>length()</td>
<td></td>
<td>Number of characters in the String.</td>
</tr>
<tr>
<td>charAt()</td>
<td>int Index</td>
<td>The char at a location in the String.</td>
</tr>
<tr>
<td>getChars(), getBytes()</td>
<td>The beginning and end from which to copy, the array to copy into, an index into the destination array.</td>
<td>Copy chars or bytes into an external array.</td>
</tr>
<tr>
<td>toCharArray()</td>
<td></td>
<td>Produces a char[] containing the characters in the String.</td>
</tr>
<tr>
<td>equals(), equalsIgnoreCase()</td>
<td>A String to compare with.</td>
<td>An equality check on the contents of the two Strings.</td>
</tr>
<tr>
<td>compareTo()</td>
<td>A String to compare with.</td>
<td>Result is negative, zero, or positive depending on the lexicographical ordering of the String and the argument. Uppercase and lowercase are not equal!</td>
</tr>
<tr>
<td>contains()</td>
<td>A CharSequence to search for.</td>
<td>Result is true if the argument is contained in the String.</td>
</tr>
</tbody>
</table>
### Operations on Strings Cont’d

<table>
<thead>
<tr>
<th>Method</th>
<th>Arguments, Overloading</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>contentEquals()</code></td>
<td>A <code>CharSequence</code> or <code>StringBuffer</code> to compare to.</td>
<td>Result is true if there’s an exact match with the argument.</td>
</tr>
<tr>
<td><code>equalsIgnoreCase()</code></td>
<td>A <code>String</code> to compare with.</td>
<td>Result is true if the contents are equal, ignoring case.</td>
</tr>
<tr>
<td><code>regionMatches()</code></td>
<td>Offset into this <code>String</code>, the other <code>String</code> and its offset and length to compare. Overload adds “ignore case.”</td>
<td>boolean result indicates whether the region matches.</td>
</tr>
<tr>
<td><code>startsWith()</code></td>
<td>String that it might start with. Overload adds offset into argument.</td>
<td>boolean result indicates whether the <code>String</code> starts with the argument.</td>
</tr>
<tr>
<td><code>endsWith()</code></td>
<td>String that might be a suffix of this <code>String</code>.</td>
<td>boolean result indicates whether the argument is a suffix.</td>
</tr>
<tr>
<td><code>indexOf()</code>, <code>lastIndexOf()</code></td>
<td>Overloaded: <code>char</code>, <code>char</code> and starting index, <code>String</code>,</td>
<td>Returns -1 if the argument is not found within this <code>String</code>; otherwise, returns</td>
</tr>
<tr>
<td>Method</td>
<td>Arguments, Overloading</td>
<td>Use</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>substring() (also subSequence())</td>
<td>Overloaded: starting index; starting index + ending index.</td>
<td>Returns a new String object containing the specified character set.</td>
</tr>
<tr>
<td>concat()</td>
<td>The String to concatenate.</td>
<td>Returns a new String object containing the original String's characters followed by the characters in the argument.</td>
</tr>
<tr>
<td>replace()</td>
<td>The old character to search for, the new character to replace it with. Can also replace a CharSequence with a CharSequence.</td>
<td>Returns a new String object with the replacements made. Uses the old String if no match is found.</td>
</tr>
<tr>
<td>toLowerCase()</td>
<td></td>
<td>Returns a new String object with the case of all letters changed. Uses the old String if no changes need to be made.</td>
</tr>
<tr>
<td>toUpperCase()</td>
<td></td>
<td>Returns a new String object with the whitespace removed from each end. Uses the old String if no changes need to be made.</td>
</tr>
<tr>
<td>trim()</td>
<td></td>
<td>Returns a new String object containing a character representation of the argument.</td>
</tr>
<tr>
<td>valueOf()</td>
<td>Overloaded: Object, char[], char[], and offset and count, boolean, char, int, long, float, double.</td>
<td>Produces one and only one String reference per unique character sequence.</td>
</tr>
</tbody>
</table>
public class SimpleFormat {
    public static void main(String[] args) {
        int x = 5;
        double y = 5.332542;
        // The old way:
        System.out.println("Row 1: [" + x + " " + y + "]");
        // The new way:
        System.out.format("Row 1: [%d %f]\n", x, y);
        // or
        System.out.printf("Row 1: [%d %f]\n", x, y);
    }
}

Row 1: [5 5.332542]
Row 1: [5 5.332542]
Row 1: [5 5.332542]
The Formatter class

%[argument_index$][flags][width][.precision]conversion

• Need to control the minimum size of a field.
  – This can be accomplished by specifying a width.
  – The **Formatter** guarantees that a field is at least a certain number of characters wide by padding it with spaces
  – By default
    • the data is right justified
    • this can be overridden by including a ‘-’ in the flags section.
import java.util.*;
public class Receipt {
  private double total = 0;
  private Formatter f = new Formatter(System.out);
  public void printTitle() {
    f.format("%-15s %5s %10s\n", "Item", "Qty", "Price");
    f.format("%-15s %5s %10s\n", "----", "---", "-----");
  }
  public void print(String name, int qty, double price) {
    f.format("%-15.15s %5d %10.2f\n", name, qty, price);
    total += price;
  }
  public void printTotal() {
    f.format("%-15s %5s %10.2f\n", "Tax", "", total*0.06);
    f.format("%-15s %5s %10s\n", "----", "---", "-----");
    f.format("%-15s %5s %10.2f\n", "Total", "", total * 1.06);
  }
  public static void main(String[] args) {
    Receipt receipt = new Receipt();
    receipt.printTitle();
    receipt.print("Jack's Magic Beans", 4, 4.25);
    receipt.print("Princess Peas", 3, 5.1);
    receipt.print("Three Bears Porridge", 1, 14.29);
    receipt.printTotal();

    >>
    | Item          | Qty | Price |
    |---------------|-----|-------|
    | Jack's Magic  | 4   | 4.25  |
    | Beans         |     |       |
    | Princess Peas | 3   | 5.10  |
    | Three Bears   | 1   | 14.29 |
    | Porridge      |     |       |
    | Tax           |     | 1.42  |
    |               |     |       |
    | Total         |     | 25.06 |

All Rights Reserved.
## Formatter conversions

<table>
<thead>
<tr>
<th>Conversion Characters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>Integral (as decimal)</td>
</tr>
<tr>
<td>c</td>
<td>Unicode character</td>
</tr>
<tr>
<td>b</td>
<td>Boolean value</td>
</tr>
<tr>
<td>s</td>
<td>String</td>
</tr>
<tr>
<td>f</td>
<td>Floating point (as decimal)</td>
</tr>
<tr>
<td>e</td>
<td>Floating point (in scientific notation)</td>
</tr>
<tr>
<td>x</td>
<td>Integral (as hex)</td>
</tr>
<tr>
<td>h</td>
<td>Hash code (as hex)</td>
</tr>
<tr>
<td>%</td>
<td>Literal &quot;%&quot;</td>
</tr>
</tbody>
</table>
String.format()

- `String.format( )`
  - a static method which takes all the same arguments as `Formatter's format( )` but returns a `String`.
  - It can come in handy when you only need to call `format( )` once:

```java
public class DatabaseException extends Exception {
    public DatabaseException(int transactionID, int queryID, String message) {
        super(String.format("(t%d, q%d) %s", transactionID, queryID, message));
    }
}
```

```java
public static void main(String[] args) {
    try {
        throw new DatabaseException(3, 7, "Write failed");
    } catch(Exception e) {
        System.out.println(e);
    }
}
```

```
>>> DatabaseException: (t3, q7)
Write failed
```
# Creating regular expressions

<table>
<thead>
<tr>
<th>Characters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>The specific character B</td>
</tr>
<tr>
<td>\xhh</td>
<td>Character with hex value oxhh</td>
</tr>
<tr>
<td>\uhhhh</td>
<td>The Unicode character with hex representation oxhhhh</td>
</tr>
<tr>
<td>t</td>
<td>Tab</td>
</tr>
<tr>
<td>n</td>
<td>Newline</td>
</tr>
<tr>
<td>r</td>
<td>Carriage return</td>
</tr>
<tr>
<td>f</td>
<td>Form feed</td>
</tr>
<tr>
<td>e</td>
<td>Escape</td>
</tr>
</tbody>
</table>
### Character Classes

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>Any character</td>
</tr>
<tr>
<td>[abc]</td>
<td>Any of the characters a, b, or c (same as a</td>
</tr>
<tr>
<td>[^abc]</td>
<td>Any character except a, b, and c (negation)</td>
</tr>
<tr>
<td>[a-zA-Z]</td>
<td>Any character a through z or A through Z (range)</td>
</tr>
<tr>
<td>[abc</td>
<td>hij]</td>
</tr>
<tr>
<td>[a-z&amp;&amp;[hij]]</td>
<td>Either h, i, or j (intersection)</td>
</tr>
<tr>
<td>\s</td>
<td>A whitespace character (space, tab, newline, form feed, carriage return)</td>
</tr>
<tr>
<td>\S</td>
<td>A non-whitespace character ([^\s])</td>
</tr>
<tr>
<td>\d</td>
<td>A numeric digit [0-9]</td>
</tr>
<tr>
<td>\D</td>
<td>A non-digit [^0-9]</td>
</tr>
<tr>
<td>\w</td>
<td>A word character [a-zA-Z_0-9]</td>
</tr>
<tr>
<td>\W</td>
<td>A non-word character [^\w]</td>
</tr>
</tbody>
</table>
Creating regular expressions

<table>
<thead>
<tr>
<th>Logical Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>XY</td>
</tr>
<tr>
<td>X</td>
</tr>
<tr>
<td>(X)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Boundary Matchers</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
</tr>
<tr>
<td>$</td>
</tr>
<tr>
<td>\b</td>
</tr>
<tr>
<td>\B</td>
</tr>
<tr>
<td>\G</td>
</tr>
</tbody>
</table>
Scanner

• The **Scanner** class
  – added in Java SE5
  – relieves much of the burden of scanning input
• The **Scanner** constructor
  – It can take just about any kind of input object, including a **File** object (which will also be covered in the *I/O* chapter), an **InputStream**, a **String**, or in this case a **Readable**
• With **Scanner**, the input, tokenizing, and parsing are all ensconced in various different kinds of "next" methods.
• A plain **next( )**
  – returns the next **String** token
  – there are "next" methods for all the primitive types (except **char**) as well as for **BigDecimal** and **BigInteger**.
• All of the "next" methods **block**
  • they will return only after a complete data token is available for input.
import java.util.*;
public class BetterRead {
    public static void main(String[] args) {
        Scanner stdin = new Scanner(SimpleRead.input);
        System.out.println("What is your name?");
        String name = stdin.nextLine();
        System.out.println(name);
        System.out.println("How old are you? What is your favorite double?
(input: <age> <double>)");
        int age = stdin.nextInt();
        double favorite = stdin.nextDouble();
        System.out.println(age);
        System.out.println(favorite);
        System.out.format("Hi %s.
", name);
        System.out.format("In 5 years you will be %d.
", age + 5);
        System.out.format("My favorite double is %f.
", favorite / 2);
    }
}
import java.util.*;
public class ScannerDelimiter {
    public static void main(String[] args) {
        Scanner scanner = new Scanner("12, 42, 78, 99, 42");
        scanner.useDelimiter("\s*,\s*"******
        while(scanner.hasNextInt())
            System.out.println(scanner.nextInt());
    }
}

>>
12
42
78
99
42
StringTokenizer

• StringTokenizer
  – Before regular expressions (in J2SE1.4) or the Scanner class (in Java SE5)
  – the way to split a string into parts was to "tokenize"
  – But now it’s much easier and more succinct to do the same thing with regular expressions or the Scanner class.
import java.util.*;

public class ReplacingStringTokenizer {
    public static void main(String[] args) {
        String input = "But I’m not dead yet! I feel happy!";
        StringTokenizer stoke = new StringTokenizer(input);
        while(stoke.hasMoreElements())
            System.out.print(stoke.nextToken() + " ");
        System.out.println();
        System.out.println(Arrays.toString(input.split(" ")));
        Scanner scanner = new Scanner(input);
        while(scanner.hasNext())
            System.out.print(scanner.next() + " ");
    }
}

>>
But I’m not dead yet! I feel happy!
[But, I’m, not, dead, yet!, I, feel, happy!]
But I’m not dead yet! I feel happy!
Arrays

• Most efficient way to hold references to objects
• Limitation: size of an array is fixed
• Benefits
  – Array knows what type it holds, compile-time type checking
  – Knows its size, you can ask
Returning an Array

• Returning Java array == returning a reference
  – Reference knows the type of the array
  – Doesn’t matter where or how array is created
  – Array is around as long as needed, GC cleans up
Arrays of Primitives

- Arrays can hold primitive types directly
- Containers can only hold references
- Can use “wrapper” classes to put primitives into containers, but that’s read only
java.util.Arrays

• Algorithms for array processing:
  – binarySearch( )
  – equals( )
  – fill( )
    • The same object duplicated
  – sort( )
    • Unstable Quicksort for primitives
    • Stable merge sort for Objects

• Overloaded for Object and all primitives
Sorting

• No support for sorting in Java 1.0/1.1
  – Explain this one to me. They forgot??
• Your class must implement **Comparable**
• Single method, **compareTo(Object rv)**
• Negative value if the argument is less than the current object
• Zero if the argument is equal
• Positive if the argument is greater
Imposing a Different Order

• If a class doesn’t implement `Comparable` or you’d like a different order
• Create a `Comparator` class
• Two methods, `compare()` and `equals()`
  – Don't have to implement `equals()` except for special performance needs
  – Just use the default `Object equals()`
• The `compare()` method
  – must return a negative integer, zero, or a positive integer if the first argument is less than, equal to, or greater than the second, respectively
• Primitives can only sort in ascending order
Summary

• Array associates numerical indices to objects
  – Holds objects of a known type
  – Fixed size