Chapter 6: Data Abstraction

- In Java, there are three types of data values
  - primitives
  - arrays
  - objects
    - actually, arrays are a special type of object

Class

- In Java, objects are used to represent data values
- Just as 3 is a primitive value of type int, every object must have a type
  - These types are called classes

Class

- A class defines a type of object, including
  - its data: the information that is used to represent the object
  - the operations (methods) that can be performed on the data
- After the type is defined, objects of that type can be defined and used
- An individual value of a class is called an object, or an instance of the class

Example

- Suppose we want to represent planets in Java
- We can define a class Planet, with
  - data: diameter, mass, distance from the sun, orbital period, rotational period, location, ...
  - methods:
    - setDiameter, setMass, etc.
    - getDiameter, getMass, etc.
    - move
  - Instances of Planet might include earth, mars, venus, saturn
Objects and Methods

• Different objects have different methods for manipulating their data
• The specific methods are determined based on what makes sense for that type of object
• For example, with strings:
  —length, concatenation, comparison, substring

String: A standard class

• String is a standard Java class
• Remember, values from a class are called objects, so
  —“hello” is an object from the class String, or an instance of the class
• The class String has instance methods that operate on an instance of the class
• For example:
  —length(), concat(), compareTo(), charAt()

Methods

• Each type of object supports a specified set of methods
• There are two types of methods in a class
  —instance methods
  —class methods

Instance Methods

• Instance methods are called for a specific object and have direct access to that object’s data without having to pass the object as a parameter

```java
String s = "abc";
s.length();
```

• The object is implicitly passed as a parameter to the method.
• We use the object's name followed by a '.', followed by the method name
String Instance Methods

• boolean equals( Object anObject )
  – compares this String with another object
  – not the same as ‘==’
  – See StringCompare.java
• int length()
  – Number of characters in this String
• char charAt( int index )
  – The character at position index in this String
    – index ranges from 0 to length() - 1
    – IndexOutOfBoundsException

String Instance Methods 2

• int compareTo(String str)
  – Returns an integer value, based on lexicographic order
• int indexOf(int ch)
  – Index of where the first occurrence of ch occurs in this string or -1 if not present
• int indexOf(String str)
  – Index of the first character of a matching substring str or -1 if not present

String Instance Methods 3

• String concat(String str)
  – Concatenates this string instance with str and returns the result
• String toLowerCase()
  – Returns a copy of this string, but converted to all lower case
• String toUpperCase()
  – Returns a copy of this string, but converted to all upper case

Class Methods

• These methods are part of the class definition, but don't operate on a specific instance of the class
• Class methods are indicated by the keyword static
  – We have written class methods up to now
• Class methods provide some type of operation related to the objects provided by the class
  – Math.random()
  – Math.sqrt()
String.valueOf()

• static String valueOf( type prim )
  —returns the String representation of the value of prim
  —valueOf is an overloaded function
    • valueOf( char c )
    • valueOf( char[] data )
    • valueOf( int i )
    • valueOf( double d )

StringTest.java

```java
// StringTest.java - demo some String methods
public class StringTest {
    public static void main(String[] args) {
        String str1 = "aBcD", str2 = "abcd", str3;
        System.out.println(str1.equals(str2));
        System.out.println(str1.length());
        System.out.println(str1.charAt(1));
        System.out.println(str1.compareTo("aBcE"));
        System.out.println(str1.compareTo("aBcC"));
        System.out.println(str1.compareTo("aBcD"));
        System.out.println(str1.indexOf('D'));
        System.out.println(str1.indexOf("Bc"));
        System.out.println(str1.indexOf("zz"));
        // parts cut out here
    }
}
```

StringTest.java

```java
// StringTest.java - demo some String methods
public class StringTest {
    public static void main(String[] args) {
        String str1 = "aBcD", str2 = "abcd", str3;
        // parts cut out here
        System.out.println(str1.concat("efg"));
        str3 = str1.toLowerCase();
        System.out.println(str3);
        str3 = str1.toUpperCase();
        System.out.println(str3);
        System.out.println(str1);
        str3 = String.valueOf(123);
        System.out.println(str3.equals("123"));
    }
}
```
String Literals

- String literals are instances of String
- Therefore, we can invoke instance methods on them
  
  
  "abc".length()
  "hello".charAt(1);

String is unusual

- Because strings are so common, Java provides some special syntax for the class String
  - special support for concatenation
  - special support for String literals
- String is the only class that is treated specially

String Concatenation

- The '+' operator is overloaded to support String concatenation
  
  "hello" + "world"
  
  is equivalent to
  
  "hello".concat("world");

- Instance methods are operations on objects

String Literals

- String literals are supported
  
  String s = "hello";

- is equivalent to
  
  char[] temp={'h','e','l','l','o'};
  String s = new String(temp);
Strings are immutable

• Instances of class String are **immutable**
  – Once its value is assigned, it cannot be changed
  • We can create a new string, but we can’t change the one
    we have.
  – One implication of this is that when we pass a
    String to a method
      ```java
      String s = “hello”;
      someMethod(s);
      ```
    – We know that its value will be unchanged when the
      method returns
    • s will have the value “hello” when someMethod returns

StringBuffer

• StringBuffer is another standard Java class for
  representing strings
  – It has instance methods that allow you to change the
    value of a StringBuffer object
  • These are called mutator methods
    – insert(), reverse(), replace(), setCharAt(),
      deleteCharAt(), ...

```java
// StringBufferTest.java - demo StringBuffer
methods
public class StringBufferTest {
    public static void main(String[] args) {
        StringBuffer sbuf1 = new StringBuffer();
        StringBuffer sbuf2 = new StringBuffer("abcd");
        StringBuffer sbuf3 = new StringBuffer(30);
        System.out.println(sbuf1.length());
        System.out.println(sbuf2.length());
        System.out.println(sbuf3.length());
        System.out.println(sbuf1.capacity());
        System.out.println(sbuf2.capacity());
        System.out.println(sbuf3.capacity());
        System.out.println(sbuf2.charAt(1));
        sbuf2.setCharAt(2,'Z');
        System.out.println(sbuf2);
        sbuf2.append("xyz");
        System.out.println(sbuf2);
        sbuf2.append(’?’);
        sbuf2.insert(4, "---");
        sbuf2.insert(2, ’+’);
        System.out.println(sbuf2);
        sbuf2.reverse();
        System.out.println(sbuf2);
        System.out.println("sbuf2 capacity " +
          sbuf2.capacity());
        System.out.println("sbuf2 length " +
          sbuf2.length());
    }
}
```
Object Declarations

- When you declare an object, you are really creating a reference
  - Similar to array declaration and allocation

```java
StringBuffer sbuf1 = new StringBuffer();
StringBuffer sbuf2 = new StringBuffer("abcd");
StringBuffer sbuf3 = new StringBuffer(30);
```

- Can also do this with separate statements

```java
StringBuffer sbuf1;
sbuf1 = new StringBuffer();
```

Elements of a Simple Class

- As discussed earlier, a class describes the data values that make up an object along with the operations that can be applied to the object
- The data values are stored in instance variables, which are also known as fields or data members.
- The operations are described by instance methods, which are sometimes called procedure members.

Example: Counter

- We often want to count things, why not create an abstraction for doing it?
  - Advantage: you can reuse it in different places in the program, or even in other programs
- **Data:**
  - Current value of the counter (initially zero)
- **Operations:**
  - Reset, Increment, Decrement, Get the current value

CounterTest.java

```java
class CounterTest {
    public static void main(String[] args) {
        Counter countThis = new Counter();
        Counter countThat = new Counter();
        countThis.increment();
        countThis.increment();
        countThat.increment();
        System.out.println("countThis: " + countThis.get());
        System.out.println("countThat: " + countThat.get());
    }
}
```
Counter.java

class Counter {
    int value;
    void reset() { value = 0; }
    void increment() { value++; }
    void decrement() { value--; }
    int get() { return value; }
}

• No main() method
• value is an instance variable
• Instance methods
  – no static keyword

Important Details

• Each Counter object has its own copy of the member variables
  – In this case, the integer variable called value
• When the methods are called, the call is of the form <objectname>,<methodname>()
• The object itself is an implicit parameter to the method, so that any references to the data access that object’s copy of the instance variables

Objects in Memory

Abstract Data Type

• Counter is an example of an Abstract Data Type (ADT) – an abstraction representing a particular type of data
• Classes allow us to implement ADTs
  – The data and methods combine to implement the functionality we desire or expect for this type of data
  – The implementation details are hidden from the user
  – The implementation is all in one place
  – The type can be used in many different places in the program or in many programs
**Package**

- In general, each class is in a separate file
  - The name of the file should match the class name (with .java at the end)
- All classes in the same directory are part of the same package
- Whether or not a method is in the same class or package as the data or method it is accessing affects what it can see and do.

**Data Hiding**

- It is desirable to hide the inner details of a class (ADT) from the users of the class
- We want to be able to determine the correctness of our class without having to examine the entire program that it is used in
- For example, with our Counter class, we want to ensure that the value doesn’t change by more than 1.

**Data Hiding**

- Accessing instance variables from outside the class violates the data hiding principle

```java
class CounterTest2 {
    public static void main( String[] args ) {
        Counter c = new Counter();
        c.value = 150;
        System.out.println( "c.value = " + c.get() );
    }
}
```

**A Better Counter Class**

- Use the `private` and `public` access specifiers to control access to instance variables and methods

```java
class Counter {
    private int value;

    public void reset() { value = 0; }
    public void increment() { value++; }
    public void decrement() { value--; }
    public int get() { return value; }
}
```
**Public/Private/Default**

- private methods/fields cannot be accessed from outside of the class
- public methods/fields can be accessed from anywhere
- default (no specifier) methods/fields have package access – they can be accessed from other classes in the same package
  - if you don't specify a package (see section 12.11), all classes in the same directory are part of the same default, un-named package