CMSC 131
Object-Oriented Programming I

Wrappers, Stack, ArrayList, Switch

Dept of Computer Science
University of Maryland College Park

This material is based on material provided by Ben Bederson, Bonnie Dorr, Fawzi Emad, David Mount, Jan Plane
Overview

- Wrappers
- Stack
- ArrayList
- Switch
Wrappers

- Java variables are either:
  
  **Primitive types** (int, float, double, ...):
  - do **not** need to be created using "new"
  - do **not** support class **methods**

- **Class Objects** (String, Date, Rational, ...):
  - must be created using "new"
  - support class **methods**

- Wouldn’t it be nice if we could associate **methods** with **primitive types**? To do this, the Java library defines special classes, called **wrappers**, each of which contains a single primitive type as its instance data.
Wrappers: Each class "wraps" a class around a primitive type.

<table>
<thead>
<tr>
<th>Primitive type</th>
<th>Wrapper</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>Byte</td>
</tr>
<tr>
<td>short</td>
<td>Short</td>
</tr>
<tr>
<td>int</td>
<td>Integer</td>
</tr>
<tr>
<td>long</td>
<td>Long</td>
</tr>
<tr>
<td>double</td>
<td>Double</td>
</tr>
<tr>
<td>char</td>
<td>Character</td>
</tr>
<tr>
<td>boolean</td>
<td>Boolean</td>
</tr>
</tbody>
</table>

Note that names differ from primitive type.
Each Wrapper provides a number of useful methods.

**Integer Wrapper:** (other numeric wrappers are similar)

- **Constructor:**
  ```java
  Integer x = new Integer(324);
  ```

  (no default constructor is provided)

- **Max and min:**
  - `Integer.MAX_VALUE` largest positive int
  - `Integer.MIN_VALUE` smallest negative int

- **Conversions:**
  - `byte b = x.byteValue();` cast x to byte
  - `double d = x.doubleValue();` cast x to double
  - `int i = x.intValue();` return integer value

- **Convert string to int:**
  ```java
  int k = Integer.parseInt("123");
  ```

- **Convert int to string in various bases:**
  - `String s1 = Integer.toBinaryString(21);` base 2
  - `String s2 = Integer.toHexString(21);` base 16
  - `String s3 = Integer.toOctalString(21);` base 8

**Example:** WrapperExample.java
Stacks

- **Stack**: A stack is an abstract data type for storing a collection of items. Items can be **inserted** into the stack and **removed** from the stack, but the rule is the most recent item inserted is the first item to be removed. (**Last in, first out**)

- **Intuition**: Think of it like a stack of plates in a restaurant. Items:
  - can be inserted (or **pushed**) onto the top of the stack.
  - can be removed (or **popped**) off of the top of the stack.
  - insertions/removals from other positions are **not allowed**.

```
Initial stack
```

```
push(6)
push(34)
push(3)
pop \to 3
pop \to 34
push(9) ...
```

```
6
34
3
9
6
```
**Stack Operations**: An abstract (mathematical) stack supports:

- **push(x)**: inserts item x at the top of the stack
- **pop()**: removes the item at the top of the stack (if one exists) and returns its value
- **top()**: returns the value of the item at the top of the stack, without removing it
- **empty()**: returns true if the stack is empty
Java’s Stack Class

- **Java’s Stack class**: (in `java.util`) Java provides a Stack, with the following corresponding operations
  - `Stack()`: creates an empty stack
  - `push( Object x )`: pushes `x` on the stack
  - `pop()`: pops the stack and returns its value (Exception if empty)
  - `peek()`: returns (without removal) the top value of the stack (Exception if empty)
  - `empty()`: returns true if the Stack is empty.
**ArrayList**

- **The Problem with Arrays:**
  - **Resizing:** Arrays are not suitable for situations where the size of the array changes frequently.
  - **Appending to an Array:** if we reach the maximum capacity of an array and we need to add an element, we have to create a new array, copy over elements, and add the desire element.

- **ArrayList:**
  - A class in the Java class library that implements a *resizable array*.
  - It is part of the `java.util` package, and therefore an appropriate `import` statement is required.
  - An ArrayList holds references to objects. We need to specify the kind of object the ArrayList will store. If we are storing any *primitive type* then we need to use the appropriate *wrapper* (e.g., Integer).
ArrayList Methods

- **ArrayList Default Constructor** Initializes an array list of size 0
- **add** ➔ adds object to the end of the array. (Automatically expands the array if needed.)
- **remove**(int i): Removes the element at index i. (Shifts the remaining elements to close the gap.)
- **get**(int i): Returns a reference to the element at index i
- **toArray()**: Returns a (standard) array with all the elements.
- **clear()**: removes all the elements from ArrayList
- **size()**: returns the number of elements in ArrayList

Java API Entry
- [http://download.oracle.com/javase/6/docs/api/index.html](http://download.oracle.com/javase/6/docs/api/index.html)

**Example:** ArrayListExample.java
The Switch Statement

Switch Statement: is a convenient (and often more efficient) way to perform a multi-way conditional based on a single control value.

Example:

```java
switch ( option ) {
    case 1:
        System.out.println("Read image");
        break;
    case 2:
        System.out.println("Double");
        break;
    case 9:
        System.out.println("Quit");
        break;
    default:
        System.out.println("Sorry, invalid");
        break;
}
```

The case that is chosen depends on the value of “option”

if ( option == 1 )
    System.out.println(“Read image”);
else if (option == 2 )
    System.out.println(“Double”);
else if ( option == 9 )
    System.out.println(“Quit”);
else
    System.out.println(“Sorry, invalid”);

The “default” case is chosen if none of the cases match
The Switch Statement

- General form:

  ```java
  switch (control-expression) {
  case case-label-1:
    statement-sequence-1
    break;
  case case-label-2:
    statement-sequence-2
    break;
  ...
  case case-label-n:
    statement-sequence-n
    break;
  default:
    default-statement-sequence
    break;
  }
  ```

  - The control-expression is one of the following types: char, int, short, byte
  - Each case label must be of a type that is compatible with the control expression.
  - You may have any number of statements, including nesting of if-else and loops.
  - The “break” statement jumps out of the switch statement.

  - The “default” case is optional, and is executed if none of the other cases match.
The Switch Statement

- The **control expression** can be of one of the following types: `char, int, short, byte`.
  - **not** float or double,
  - **not** boolean or long
  - **not** an object (Too bad! Strings would have been nice.)
- The “**break**” statement jumps out of the switch statement. Otherwise control flow just “**falls through**” into the next case.

```java
int option = 2;
switch ( option ) {
    case 1:
        System.out.println( "Read image" );
    case 2:
        System.out.println( "Double" );
    case 9:
        System.out.println( "Quit" );
    default:
        System.out.println( "Sorry, invalid" );
}
```

**Output:**
```
Double
Quit
Sorry, invalid
```

This is probably not what you intended.

This is not correct!
The **falling though** behavior is handy, because it allows you to combine cases.

**Example:** Allowing either upper-case or lower-case for characters:

```java
char command = 'D';
switch ( command ) {
    case 'i':
    case 'I':
        MyUtility.insert();
        numberOfItems++;
        break;
    case 'd':
    case 'D':
        MyUtility.delete();
        numberOfItems--;
        break;
    ...
}
```

Note: This is a char, not a String.

This is performed for either ‘I’ or ‘i’