Object-Oriented Programming I

Wrappers, Overloading, Stack

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
- Method Overloading
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**

- **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**: `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**: `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`
Method Overloading

- **Overloading:**
  - Java allows methods to have the same name, even within the same class.
  - Where have we seen overloading before? Constructors!
  - For example, given a Date class we can add the following methods to change the current date. (implementations have been omitted)
    ```java
    public void setDate( int m, int d, int y ) { ... } // month given as integer
    public void setDate( String m, int d, int y ) { ... } // month given as string
    public void setDate( int m, int y ) { ... } // day defaults to 1
    ```
  - Sample calls:
    ```java
    Date dueDate = new Date( 10, 5, 2004 ); // set initial due date
    dueDate.setDate( 10, 7, 2004 ); // delay the due date
    dueDate.setDate( “Nov”, 12, 2004 ); // delay it further
    dueDate.setDate( 1, 2005 ); // delay until next year
    ```
  - **Question:** How does Java know which one to call?
  - **Answer:** It looks at the number and of types of arguments
Method Overloading and Signatures

- **Overloading**: using the same identifier name for different methods. Usually these methods perform very similar functions, but we want to provide different ways of accessing it (for convenience of the class user). Java determines which one to call based on the method’s **signature**

- **Signature**: of a method consists of the name of the method and the types of the parameters.

**Example:**

```java
public float doSomething( int x, double z, double w, String s )
```

**Corresponding Signature:**

```java
doSomething( int, double, double, String )
```

**Prototype**: of a method is the signature of the method with a return type and any additional modifiers
Note that the **return type** of a method is **not** part of the signature. You cannot overload two methods with the same parameter types but different return types.

**Example:** Two methods convert temperature in Fahrenheit to Celsius. One rounds to the nearest integer, the other doesn’t.

```java
public int toCelsius( double t ) { ... }
public double toCelsius( double t ) { ... }

... System.out.println( toCelsius( 98.6 ) );
```

**Which method should be called?** Unfortunately, Java cannot read your mind. (Solution: Use different names.)
**Parameter Type Promotion**

- **Automatic Casting:** We have seen that, in assignment statements, Java automatically promotes numeric types to the higher type:
  
  ```java
  int total = ...;
  double average = total;
  ```

  total's value is promoted to a `double` before doing the assignment.

- **Promotion of Parameters:** In the same way, Java automatically promotes each argument to match the type of its formal parameter. `Math.sqrt()` expects an argument of type `double`.
  
  ```java
  int area = 1024;
  double s = Math.sqrt(area);
  ```

  area's value is promoted to a `double` before passing it as a parameter.
Ambiguous Overloading

• Because of type promotion, there are times when Java cannot figure out which method to call.

```java
public void fooBar( int x, double y ) { ... }
public void fooBar( double u, int v ) { ... }
```

...  

```java
fooBar( 10, 23.0 ); // okay, use the first  
fooBar( 10.0, 23 ); // okay, use the second  
fooBar( 10, 23 ); // ???
```

• Do we promote 23 to 23.0 and call the first, or promote the 10 to 10.0 and call the second?
• Java issues a **compile-time error**, since it cannot resolve the ambiguity
• How can we solve the problem?
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 → 3  pop → 34  push(9) ...
```

- **Initial stack**
- **push(6)**
- **push(34)**
- **push(3)**
- **pop → 3**
- **pop → 34**
- **push(9) ...**
Stack Operations

- **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
- To create a stack you need to specify the type of elements. For example:
  ```java
  Stack<Integer> a = new Stack<Integer>();
  ```