# 02 A Object-oriented Programming in Java

## CS1102S: Data Structures and Algorithms

Martin Henz

January 20, 2010

Generated on Tuesday 19th January, 2010, 19:08

1. Classes and Objects
   - Methods
   - Parameter Passing
   - Identifiers
2. Data Types

# Classes

A class is a data type that specifies the components of and methods available for its *instances*.

Java is object-oriented. Classes can extend other classes and thus *inherit* components and methods from other classes.

Classes can *implement* interfaces.

# Example Class

```
final public class MyClass
    extends otherPackage.YourClass
    implements someInterface {
    ...
}
```

# Components of Classes

| Component | Syntax | Description |
|---|---|---|
| Subclassing | abstract/ final | must/cannot be extended |
| Access | public (or none) | available outside of package (or not) |
| Class name | class name | Class name, same as file name |
| Extends | extends name | Name of super-class |
| Implements | implments list | Implemented interfaces |
| Body | enclosed in braces | Data fields and methods for the class |

# Example Data Fields

```
public    static  final      int UPPER_LIMIT;
private                      int internal_counter;
protected         transient  int volume;
          static  volatile   int global_counter;
```

# Data Access Modifiers

public : available wherever the class is available

private : only available within the class

protected : available in subclasses and within same package

none : available within the same package

# Data Use Modifiers

static : only one data field available for all instances

final : value cannot be modified (constant)

transient : value not persistent when storing object (not used in this module)

volatile : value can be accessed by multiple threads concurrently (not used in this module)

# Method Example

```
public static int max(int x, int y) {
    if (x > y) {
      return x;
      } else {
      return y;
      }
}
```

Which is the access modifier and which is the use modifier?

# Method Use Modifiers

static : invoked on the class, not on the instances; can only refer to static data fields

final : method cannot be overridden in a subclass

abstract : method must be overridden in every subclass

native : method body not written in Java
(how can that be?)

synchronized : method can be run by only one thread of control at a time
(what are threads?)

# Invoking a Method

```
class IntUtil {
   ...
   public void printLargest(int a, int b, int c) {
      int largerAB = max(a, b);
      // int largerAB = IntUtil.max(a, b);
      int largest  = max(largerAB,c);
      System.out.println(largest +
                         " is the largest.");
   }
   ...
}
```

# Parameter Passing

Java uses pass-by-value parameter passing.

```
public static void tryChanging(int a) {
    a = 1;
    return;
}
...
int b = 2;
tryChanging(b);
System.out.println(b);
```

# Parameter Passing with Objects

```
public static void tryChanging(SomeObject obj) {
    obj.someField = 1;
    obj = new SomeObject();
    obj.someField = 2;
    return;
}
...
SomeObject someObj = new SomeObject();
tryChanging(someObj);
System.out.println(someObj.someField);
```

# Identifiers

Java is a typed language.
All identifiers must be declared with a type.

```
float radius;
SomeObject obj;
```

1 Classes and Objects

2 Data Types
- Primitive Data Types
- Arrays
- Useful Java Classes
- Exceptions
- Text I/O

# Wrapper Types

Every primitive data type comes with a corresponding wrapper type, representing objects that hold values of the primitive type.

```
int x = 9;
Integer intObject = new Integer(x);
System.out.println("Value is " +
                   intObject.intValue());
```

# Primitive Data Types

| Category | Data Type | Wrapper Class |
|---|---|---|
| Boolean | boolean | Boolean |
| Character | char | Character |
| Integer | byte | Byte |
| | short | Short |
| | int | Integer |
| | int | Integer |
| Floating point | float | Float |
| | double | Double |

# Autoboxing and Auto-unboxing

```
Integer intObject = 9;

int x = intObject + 1;
```

# References

All identifiers that do not represent primitive data types are references to objects.

A reference value `null` indicates that the identifier currently has no object to refer to.

# Overloading of Operators and Methods

Operators such as +, ∗ etc are “overloaded”; they can work on multiple types and return corresponding values:

5 + 5 returns the integer 10.

5.0 + 5.0 returns the float 10.0.

The method println is also overloaded to accept int, String, float, etc.

# Type Promotion

If arithmetic operators are applied to numerical values of different type, promotion happens according to

$\text{int} \rightarrow \text{long} \rightarrow \text{float} \rightarrow \text{double}$

Example: 10 + 5.0 results in float 15.0.

# String Conversion

The operator + (and only +) is compiled such that one argument is converted to a string using `toString()` as soon as the other argument is of type `String`.

# Arrays

- Array declaration

```
final int DAYS_PER_WEEK = 7;
double [] maxTemps = new double[DAYS_PER_WEEK];
```

- Array access

```
System.out.println("Monday value: " +
                   maxTemps[1]);
```

- Declaration with initializer list

```
double [] weekDayTemps
        = {2.0, 71.5, 1.8, 75.0, 88.3};
```

# Multidimensional Arrays

- Declaration

```
final int DAYS_PER_WEEK = 7;
final int WEEKS_PER_YEAR = 52;
double [] minTemps
    = new double[DAYS_PER_WEEK][WEEKS_PER_YEAR];
```

- Access: minTemps[3][3]
- Multidimensional initializer list

```
int[][] x = {{1,2,3},{4,5,6}};
```

# The Object Class

A class *A* cannot extend another class *B* that directly or indirectly extends *A*.

Classes that do not have extends implicitly extend Object.

Theorem

*The methods of Object are available for all objects.*

Can you prove this theorem?

# Methods of Class Object

```
public boolean equals(Object obj) // equality of re

protected void finalize() // for garbage collection
```

What is garbage collection?

```
public int hashCode() // generate a code for hashing
```

What is hashing?

```
public String toString() // string representation
```

# Class String

Objects of class String are immutable sequences of characters. Literal strings are instances of String.

```
public int compareTo(String s)
    // negative if the string comes after s
    // 0 if equal and positive if before s
public String substring(int x,int y)
    // take substring starting at position x
    // until but excluding position y
public int indexOf(String s, int x)
    // return index of first occurrence of s
    // start looking at position x
```

# Exceptions

- Exceptional situations abandon the current execution context, for example division by zero.

What is “execution context”?

- An Exception object is associated with the situation.
- A “*catcher*” of exceptions can be installed so that execution can be resumed.

# Example

```
static int divide(int x, int y) {
    return x / y;
}

...
int x = computeSomething(...);
int y = computeSomethingElse(...);
showToUser(divide(x,y));
```

# Example

```
int x = computeSomething(...);
int y = computeSomethingElse(...);
try {
    showToUser(divide(x,y));
} catch (ArithmeticException e) {
    showToUser("The␣duration␣must␣be␣" +
               "at␣least␣one␣day");
}
```

# Throwing Exceptions

- The programmer can define his/her own exceptions by extending the class Exception.
- The keyword throw can be used to generate such user-defined exceptions.

```
throw new PercentageException(
                "percentage exceeds 100");
```

# Input using java.util.Scanner

```
import java.util.Scanner;
...
int sum=0;
Scanner kbInput = new Scanner(System.in);
int nextValue = kbInput.nextInt();
while (nextValue > 0) {
    sum += nextValue;
    nextValue = kbInput.nextInt();
}
kbInput.close();
```

# Output using System.out

System.out provides println and printf. Examples:

```
System.out.println("The answer is " + answer);

String name = "Jamie";
int x = 5, y = 6;
int sum = x + y;
System.out.printf("%s, %d + %d = %d",
                  name, x, y, sum);
```