

Object-Oriented Programming, Part 2

- Packages
 - The Java library unit ---“Think Big”
- Information hiding
- Access modifiers
 - Private, protected, public, and “friendly”
- The singleton design pattern
 - Making use of access modifiers
- Designing an email class
 - Conventional object-oriented design
- Designing and implementing a debug facility
 - Unconventional object-oriented design

Package Example

```
package com.mycompany.misc; // file Car.java
```

```
public class Car {  
    public Car(){  
        System.out.println("com.mycompany.misc.Car");  
    }  
}
```

```
package com.mycompany.misc; // file Truck.java
```

```
public class Truck {  
    public Truck(){  
        System.out.println("com.mycompany.misc.Truck");  
    }  
}
```

Packages in Java

- A package is a collection of classes (a *library*).
- The first line in a file must specify the package, e.g.,
 - `package mypackage;`
 - `package dk.aau.cs.torp.debug;`
- Characteristic of a package
 - Organized in a hierarchy
 - ◆ Uses the file system for implementing the hierarchy
 - ◆ A package corresponds to a directory (and typically subdirectories)
 - Every package is a name space
 - ◆ More than one class called **Test**
- By default, classes belong to the *unnamed package*.

Packages in Java, cont.

- Typical problems with packages
 - Tied to the local directory structure
 - Case sensitive
 - Default package in current directory.

- Good design
 - All classes should be in a explicit package, i.e., do not use the unnamed package
 - Hint: MIP exam

Accessing Classes in a Package

- A class **MyClass** in a package **mypackage** is accessed via
 - **mypackage.MyClass**
- This can be nested to any level
 - **mypackage1.mypackage2.mypackage3.MyOtherClass**
- Naming convention for package names: all lower case and words run together.
- To avoid too much dotting packages can be **imported**, e.g.,
 - In a file **import mypackage1.mypackage2.mypackage3.***, then, **MyOtherClass** does not have to be qualified.
- If name clashes, i.e., same class name in two imported packages, then use fully qualified name.
- The package **java.lang** is always imported.

Accessing Classes, Example One

```
import java.lang.*; // not needed always done implicit

public class Garage1 {
    com.mycompany.misc.Car car1;
    com.mycompany.misc.Truck truck1;
    public Garage1() {
        car1 = new com.mycompany.misc.Car();
        truck1 = new com.mycompany.misc.Truck();
    }

    public void toPrint() {
        System.out.println ("A garage: " + Math.PI);
        System.out.println ("A car: " + car1);
    }
}
```

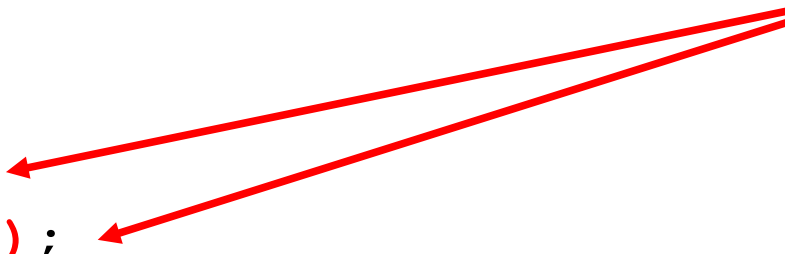
from java.lang



Accessing Classes, Example Two

```
import com.mycompany.misc.*;  
//import com.mycompany.*; // not possible  
//import com.*;           // not possible
```

```
public class Garage {                                     from com.mycompany.misc  
    Car car1;  
    Truck truck1;  
    public Garage() {  
        car1 = new Car();  
        truck1 = new Truck();  
    }  
  
    public void toPrint() {  
        System.out.println ("A garage: " + Math.PI);  
        System.out.println ("A car: " + car1);  
    }  
}
```



Static import

```
// without static import
```

```
public class WithoutStaticImport {  
    public static void main(String[] args) {  
        System.out.println("Value of PI is " + Math.PI);  
    }  
}
```

```
// with static import
```

```
import static java.lang.Math.PI;  
import static java.lang.System.out;
```

```
public class StaticImport {  
    public static void main(String[] args) {  
        out.println("Value of PI is " + PI);  
    }  
}
```

- Tip: Use with caution!

CLASSPATH

- Store **misc** package in /user/torp/java/com/mycompany/misc directory, i.e., the files **Car.class** and **Truck.class**.
- CLASSPATH = ./user/torp/java;./user/torp/something.jar
- CLASSPATH = c:\java:c:\user\torp\something.jar
 - Test **echo %CLASSPATH%** on Windows
 - Test **echo \$CLASSPATH** on *nix
 - Java ARchive (jar) files are used to store multiple files. Makes for example downloads over the Internet much faster.
- Compiler starts search at CLASSPATH

Information Hiding

- Separate interface from implementation!
 - Also hides the errors that you make in the implementation
- How much should a user of a class see?
- Rules of thumb
 - Make instance variables private
 - Make at least one constructor public
 - Make part of the methods public

Access Modifiers on Variables/Methods

- **private**
 - Variable/method is private to the class
 - *"Visible to my self"*
- **public**
 - Variable/method can be seen by all classes
 - *"Visible to all"*
- **protected**
 - public to other members of the same package
 - public to all subclasses
 - private to anyone else
 - *"Visible to the family" or "beware of dog"*

Access Modifiers on Variables/Methods, cont.

- “Friendly”
 - Default access modifier, has no keyword
 - public to other members of the same package
 - private to anyone outside the package
 - Also called *package access*
 - "*Visible in the neighborhood*"
- Is “friendly” more restrictive
 - than **private**?
 - than **protected**?

Public/"Friendly" Example

```
package com.mycompany.misc;

public class Car{
    public Car(){
        System.out.println("com.mycompany.misc.Car");
    }
    void foo () { // "friendly"
        System.out.println("foo");
    }
}
```

```
package mynewpackage; // in another package
import com.mycompany.misc.*;

public static void main(String[] args){
    Car car1 = new Car();
    car1.foo(); // compile error "private" in this package
}
```

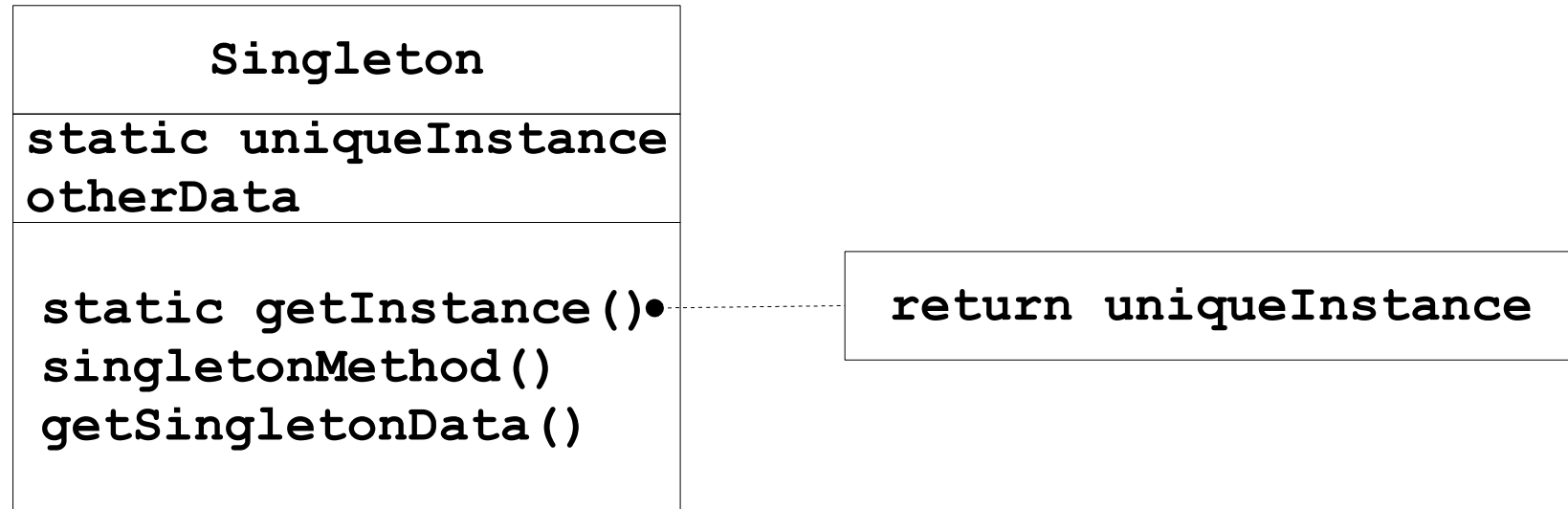
Singleton Design Pattern

```
public class Singleton{
    private int myData = 42;
    private static Singleton s = new Singleton();

    // make default constructor private
    private Singleton(){ }
    public static Singleton getSingleton(){
        return s;
    }
    public void singletonMethod() { /* do stuff */ };
    public int getSingletonData() { return myData; }
}
```

```
public class UseSingleton {
    public static void main (String[] args){
        Singleton s1 = new Singleton(); // compile error
        Singleton s2 = Singleton.getSingleton();
        s2.singletonMethod();
        // s2 and s3 are reference equal
        Singleton s3 = Singleton.getSingleton();
    }
}
```

Singleton Design Pattern, cont.



- Controlled access to one instance
- Reduced name space (not a “global” variable)
- Permits refinement of operations and representation
- Permits a variable number of instances
- More flexible than static methods
- Very nice object-oriented design

Design an **Email** Class

- Instance variables
 - **from** single email address, should provide a default
 - **to** multiple email addresses, mandatory
 - **cc** multiple email addresses, default empty
 - **bcc** multiple email addresses, default empty
 - **reply-to** single email address, default empty
 - **subject** string
 - **body** large string
- Open questions
 - What are the data types of the instance variables?
 - ◆ Should subject and body be of the same data type?
 - ◆ Should email address be a class or simply a string?
 - What should the access modifiers be for the instance variables?
 - How to store a list of multiple email addresses?
 - Should subject and body be mandatory?

Design an **Email** Class, cont.

- Methods
 - **setDefaultFrom(emailAddress)**, sets default from address
 - **setFrom(emailAddress)**, sets from this email
 - **getFrom() return emailAddress**, get the from for this email
 - **setTo(emailAddress)**, sets single to in to email address
 - **setTo(emailAddress[])**, sets more than one email address
 - **send()**, sends the email
 - **send(emailAddress)**, sends the email to the email address specified
 - **send(emailAddress[])**, sends the email to the list specified
 - **clean()**, clean all the instance variables
 - **show()**, shows what is currently stored in the instance variables in a nice human readable fashion
 - **setSubjectMandatory(boolean)**, should a subject be specified before the email can be send?

Design an **Email** Class, cont.

- Constructors
 - `Email()`
 - `Email(to, subject, body)`
 - `Email(to, cc, subject, body)`
 - `Email(to, cc, bcc, subject, body)`
 - `Email(to, cc, bcc, reply-to, subject, body)`
- Open questions
 - How many constructors are enough?
 - ◆ There can be too few, however there can also be too many!
 - ◆ Pick the most simple and the most complete and add some in between!
 - Should we automatically send the email when all mandatory instance variables are supplied to the constructor?

Design an **Email** Class, cont.

- Missing, then we must reiterate the design!
 - Must count the number of emails send
 - ◆ Simple let us do it, to make the customer satisfied.
 - Default at-address, e.g., torp means torp@cs.aau.dk
 - ◆ Semi complex, would be nice but not strictly needed! (postpone to the next release?)
 - Save draft of email that can be restored later!
 - ◆ Complicated and is not in the original requirement specification!
 - ◆ Postpone delivery deadline or add to the price of the product!
- List of good idea for next release of **Email** class
 - Adding attachments
 - Setup to mail server

Evaluation of Design of an **Email** Class

- Open questions
 - Does the class do *one and only one* thing well?
 - Do we have a coherent and general class?
 - ◆ Do we provide a good *abstraction* for clients?
 - Are the method names saying and easy to understand and use?
 - Are the internal data structures encapsulated (*information hiding*)?
 - ◆ The correct access modifiers applied?
 - Did we prepare for refinements of the class by other programmers?
 - ◆ Inheritance (covered in next lectures)
 - Do we have good documentation for the clients?
 - Is the source code stored in the right package?

Access Modifiers on Classes

- **private**
 - Not supported in Java! (however, works for *inner class*)
 - Default see the slide on friendly
- **protected**
 - Not supported in Java!
- **public**
 - Can be seen by all other classes in other packages
 - One public class per file
- “Friendly”
 - A class without an access modifier can be accessed from the classes within the same package.
- Packages have no access modifiers
 - What would it mean?

Design a Debug Message Facility

- Be able to produce output from method without having to recompile.

```
public class TestDebug{
    public void complicatedMethod() {
        int i = (int) (Math.PI + 89 * 62); // complicated stuff
        Debug.show("int debug", i);
        char c = 'x';
        Debug.show("char debug", c);
        String s = "build " + "a " + "string";
        Debug.show("Object debug", s);
    }
}
```

Design a Debug Message Facility, cont

- Methods
 - enable/disable debugging
 - show(int value)
 - show(String message, int value)
 - show(char value)
 - show(String message, char value)
 - show(Object value)
 - show(String message, Object value)
 - collect debug information and print all later, only when debugging
 - showCollect(), prints the collected debug information
 - clearCollect(), deletes all the collected debug information
- Open questions
 - All methods are static, is this okay?
 - Show method heavy overloaded okay?

Implementation of Debug Message Facility

```
public class Debug{
    /** Is debugging enabled, default it is off */
    private static boolean debugging = false;
    /** Is collecting debug information on, default is off */
    private static boolean collecting = false;
    /** Maximum no of String that can be collected */
    public static final int MAX = 99;
    /** Array to which messages are collected, fixed size */
    private static String[] coll = new String[MAX];
    /** index within coll array */
    private static int counter = 0;

    /** The method that actual displays the message */
    private static void realShow(String msg) {
        if (debugging) {
            if (collecting) {coll[counter++] = msg; }
            else { System.out.println(msg); }
        }
    }
}
```

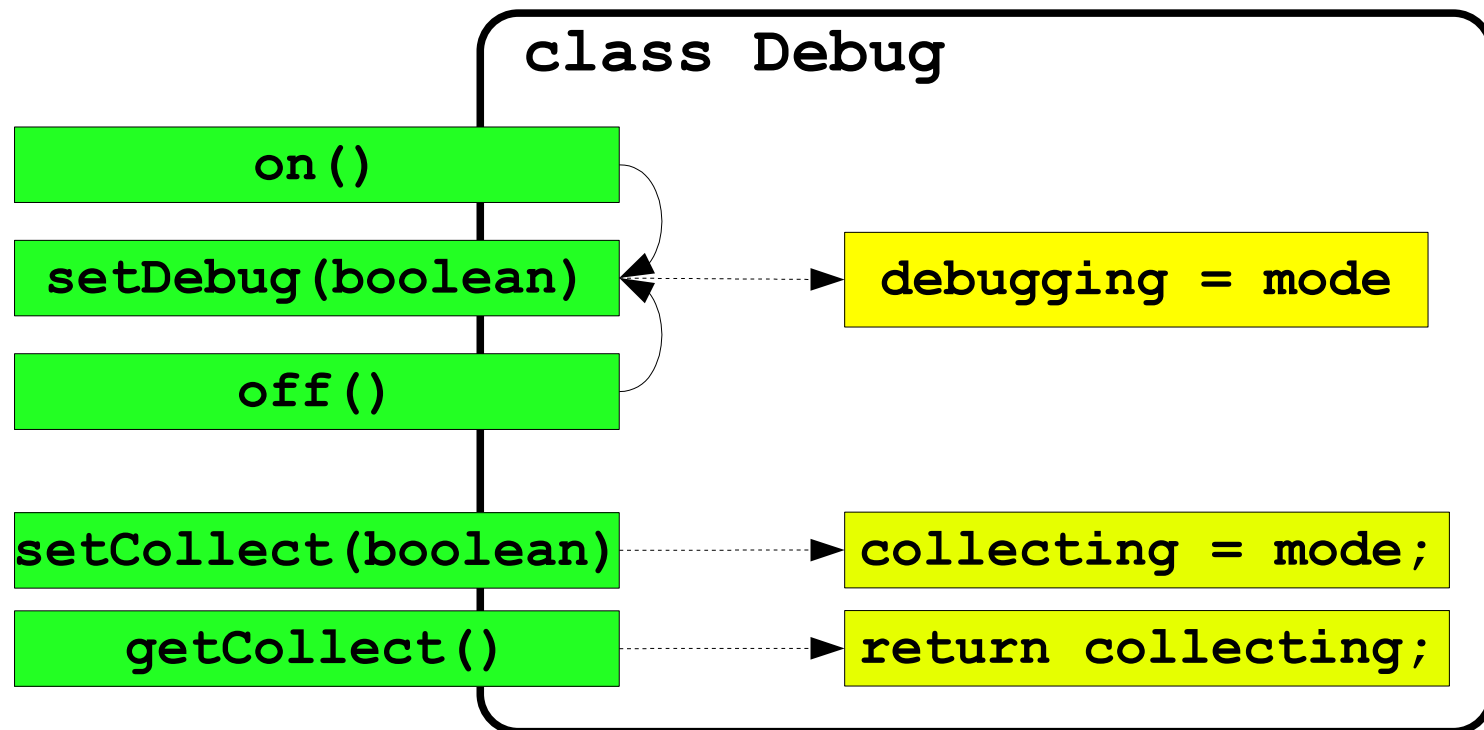

Implementation of Debug Message Facility, cont

```
public class Debug{
    // snip
    /** Sets the debugging on. */
    public static void on(){ setDebug(true); }
    /** Sets the debugging off. */
    public static void off(){ setDebug(false); }
    /** Sets the debugging mode. */
    public static void setDebug(boolean mode){
        debugging = mode;
    }
    /** Gets the debugging mode. */
    public static boolean getDebug(){ return debugging; }

    /** Sets the collect mode. */
    public static void setCollect(boolean mode){
        collecting = mode;
    }
    /** Gets the collecting mode. */
    public static boolean getCollect(){ return collecting; }
}
```

Evaluation of a Debug Message Facility

- All access to variables via methods.
- Only do one thing in one place, examples are
 - `setDebug()` `setCollect()`
- Provided both `on()/off()` and `setDebug()/getDebug()`
 - `on()/off()` method used a lot, and very saying method names
 - `setDebug()/getDebug()` typical way to access private data



Implementation of Debug Message Facility, cont

```
public class Debug{
    // snip
    /** Shows a debug message */
    public static void show(char value){show("", value); }

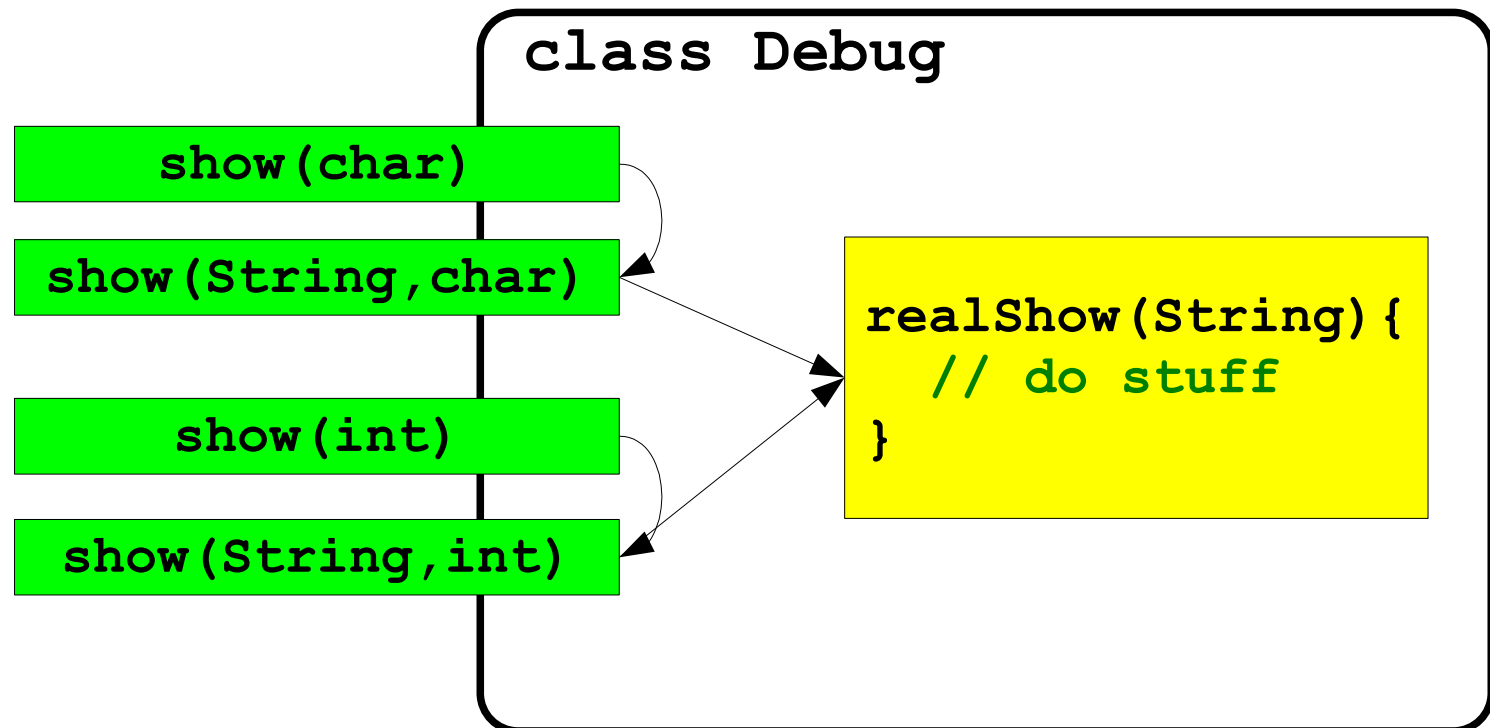
    /** Shows a debug message */
    public static void show(String message, char value){
        String msg = message + " " + value;
        realShow(msg);
    }

    /** Shows a debug message */
    public static void show(int value){show("", value); }

    /** Shows a debug message */
    public static void show(String message, int value){
        String msg = message + " " + value;
        realShow(msg);
    }
}
```

Evaluation of a Debug Message Facility, cont.

- Methods with few parameters add default parameters and call similar method that takes more parameters.
- Many similar public **show** methods map to a single private **realShow** method.



Evaluation of a Debug Message Facility, cont

- All variables and method are **static**, unusual but okay here
- The array that is collected to can easily be changed to a dynamic structure when we learn about collections
 - **MAX** should then be set to infinitive

- Open questions
 - Easy to add show methods for all basic type?
 - Can we write out to file or database instead of, must add functionality?

Summary

- Package, the library unit in Java.
- Access modifiers
 - Tells clients what they can and cannot see.
- Separation of interface from implementation.
 - Very important in design (and implementation).
- Guideline: Make elements as hidden as possible.
- Class properties
 - Use static methods with caution!
- Object-oriented design hard parts
 - Decomposing system into classes.
 - Defining the public interface of each class.
 - Finding out what is likely to change.
 - Finding out what is likely to stay the same.

Class Properties

- A *class variable* is a variable that is common to all instances of the same class.
- A *class method* is a method that operates on a class as it was an object.
- Classes are objects (*meta objects*)
 - Class variables are stored in meta objects
 - Java supports meta object via the class **Class**. Further, there is a set of classes in the package **java.lang.reflect**. See Chapter 12 “Run-Time Type Identification”.

Class Properties, cont.

- Variables marked with **static** are class variables.
 - `public static float tax = 22.75;`
- Methods marked with **static** are class methods
 - `public static void main (String[] args) {}`
- The **Math** class consists entirely of static methods and variables.
 - We never construct a **Math** object.
 - In general this is not a good object-oriented design.

Implementation of Debug Message Facility, cont

```
public class Debug{
    // snip
    /**
     * Show the debug information collected
     */
    public static void showCollect(){
        for(int i = 0; i <= counter - 1; i++){
            // do NOT call real show here!
            System.out.println(coll[i]);
        }
        clearCollect();
    }

    /**
     * Clean the collected debug information
     */
    public static void clearCollect(){
        counter = 0;
    }
}
```