

Introduction to web development in Java/JSP  
Dennis Olsson

Today

Composite types

Java

Common loops

Output to terminal

Example class

Example usage

Constructors

Polymorphism

Interfaces

Polymorphism

Packages

Dynamic class loading

Error Handling

Summary

# Introduction to web development in Java/JSP

Dennis Olsson

Tuesday 31 July

Introduction to web development in Java/JSP  
Dennis Olsson

Today

Composite types

Java

Common loops

Output to terminal

Example class

Example usage

Constructors

Polymorphism

Interfaces

Polymorphism

Packages

Dynamic class loading

Error Handling

Summary

- 1 Composite types
- 2 Java
- 3 Example class
- 4 Constructors
- 5 Polymorphism
- 6 Packages
- 7 Dynamic class loading
- 8 Error Handling
- 9 Summary

Introduction to web development in Java/JSP  
Dennis Olsson

Today

Composite types

Java

Common loops

Output to terminal

Example class

Example usage

Constructors

Polymorphism

Interfaces

Polymorphism

Packages

Dynamic class loading

Error Handling

Summary

# Composit types

Composite types are used in nearly every language. Some examples:

- **C** – struct
- **Pascal** – record
- **Object Oriented** - class

Introduction to web development in Java/JSP  
Dennis Olsson

Today

Composite types

Java

Common loops

Output to terminal

Example class

Example usage

Constructors

Polymorphism

Interfaces

Polymorphism

Packages

Dynamic class loading

Error Handling

Summary

# Accessing data

Composite types are containers for (one or) several other values.

This is a way to create a hierarchy for data in memory. A basic example of a set of variables that belong together is:

`person_age, person_fname, person_lname`

It is possible to join them under a common variable. This way `person` has three attributes, namely: `age, fname, lname`. In Java these can be accessed via punctuation, as:

```
person.fname + " is " + person.age + " old."
```

Introduction to web development in Java/JSP  
Dennis Olsson

Today

Composite types

Java

Common loops

Output to terminal

Example class

Example usage

Constructors

Polymorphism

Interfaces

Polymorphism

Packages

Dynamic class loading

Error Handling

Summary

# Objects

The advantage of this is structure. Instead of having `person1_age, person2_age, etc.`, only `person1` and `person2` is needed. the rest of the data is hidden inside them.

Since each variable can be seen as an object, as the person being an object owning it's age, which is a type of object of time etc., this is the base for Object Oriented languages.

Introduction to web development in Java/JSP  
Dennis Olsson

Today

Composite types

Java

Common loops

Output to terminal

Example class

Example usage

Constructors

Polymorphism

Interfaces

Polymorphism

Packages

Dynamic class loading

Error Handling

Summary

# Classes

An **object** is an instance of a class during run time. This is what the programmer interact with when the program is executed.

A **class** is a composite type with devoted functions (methods). This is the drawing the programmer constructs for the data handling.

A class is what states that there is an attribute named `age`. An object is the chunk of data accually having the attribute's value.

- Introduction to web development in Java/JSP
- Dennis Olsson
- Today
- Composite types
- Java
  - Common loops
  - Output to terminal
- Example class
  - Example usage
- Constructors
- Polymorphism
  - Interfaces
  - Polymorphism
- Packages
- Dynamic class loading
- Error Handling
- Summary

## More comparisons

An object could be compared with a box containing notes (attributes):

- A note with a primitive value.
- A note with the number of another box (reference).

Not in the scope of this course:

- Another box (inheritance)

- Introduction to web development in Java/JSP
- Dennis Olsson
- Today
- Composite types
- Java
  - Common loops
  - Output to terminal
- Example class
  - Example usage
- Constructors
- Polymorphism
  - Interfaces
  - Polymorphism
- Packages
- Dynamic class loading
- Error Handling
- Summary

## Hierarchy

Each type of class has a set of methods ( $\approx$ functions) it can perform.

If the method is unsupported by the class, the method will be called in the next class in the hierarchy. (More on this subject later during the course)

- Introduction to web development in Java/JSP
- Dennis Olsson
- Today
- Composite types
- Java
  - Common loops
  - Output to terminal
- Example class
  - Example usage
- Constructors
- Polymorphism
  - Interfaces
  - Polymorphism
- Packages
- Dynamic class loading
- Error Handling
- Summary

## Naming conventions

Naming Convention used by Sun:

- class
  - Noun
  - Composed words
  - All words have initial capital
  - E.g. CarEngine

- Introduction to web development in Java/JSP
- Dennis Olsson
- Today
- Composite types
- Java
  - Common loops
  - Output to terminal
- Example class
  - Example usage
- Constructors
- Polymorphism
  - Interfaces
  - Polymorphism
- Packages
- Dynamic class loading
- Error Handling
- Summary

## Naming conventions

Naming Convention used by Sun:

- method
  - Composed words
  - All words except the first have initial capital
  - E.g. doMagic()

- Introduction to web development in Java/JSP
- Dennis Olsson
- Today
- Composite types
- Java
  - Common loops
  - Output to terminal
- Example class
  - Example usage
- Constructors
- Polymorphism
  - Interfaces
  - Polymorphism
- Packages
- Dynamic class loading
- Error Handling
- Summary

## Naming conventions

Common prefixes in a method name:

- isXXX() - return boolean if true
- setXXX(value) - Set value of XXX
- getXXX() - Get value of XXX

Less common/more specific:

- addXXX() - Add a value to some kind of list/set
- removeXXX(value) - Remove an element from a list/set
- doXXX() - Execute a specific task

- Introduction to web development in Java/JSP
- Dennis Olsson
- Today
- Composite types
- Java
  - Common loops
  - Output to terminal
- Example class
  - Example usage
- Constructors
- Polymorphism
  - Interfaces
  - Polymorphism
- Packages
- Dynamic class loading
- Error Handling
- Summary

## Visibility

Visibility modifiers for classes:

- **private** – only reachable from the same class
- *no modifier* – also the same package
- **protected** – also through inheritance
- **public** – from any class

Only private and public of interest in this course.

The reason why anyone should use for instance private is to hide accessibility to other classes. This way error handling is made much easier, since a private method only will be called from the same class. This also eases re-writing of code. If all methods were public, the entire program would have to be checked.

Introduction to web development in Java/JSP  
Dennis Olsson

Today  
Composite types  
Java  
Common loops  
Output to terminal  
Example class  
Example usage  
Constructors  
Polymorphism  
Interfaces  
Polymorphism  
Packages  
Dynamic class loading  
Error Handling  
Summary

## Java syntax

Java looks and behaves a lot like C/C++. Therefore a lot of the following examples would be close to identical in the three languages.

◀ ▶ ⏪ ⏩ 🔍 ↺

Introduction to web development in Java/JSP  
Dennis Olsson

Today  
Composite types  
Java  
Common loops  
Output to terminal  
Example class  
Example usage  
Constructors  
Polymorphism  
Interfaces  
Polymorphism  
Packages  
Dynamic class loading  
Error Handling  
Summary

## while

```
while( condition ) {
    // code block
}
```

Execute the code block iff the condition is true, repeat until condition is false.

◀ ▶ ⏪ ⏩ 🔍 ↺

Introduction to web development in Java/JSP  
Dennis Olsson

Today  
Composite types  
Java  
Common loops  
Output to terminal  
Example class  
Example usage  
Constructors  
Polymorphism  
Interfaces  
Polymorphism  
Packages  
Dynamic class loading  
Error Handling  
Summary

## do ... while

```
do {
    // code block
} while( condition );
```

Execute the code block, then repeat iff the condition is true.

◀ ▶ ⏪ ⏩ 🔍 ↺

Introduction to web development in Java/JSP  
Dennis Olsson

Today  
Composite types  
Java  
Common loops  
Output to terminal  
Example class  
Example usage  
Constructors  
Polymorphism  
Interfaces  
Polymorphism  
Packages  
Dynamic class loading  
Error Handling  
Summary

## for

```
statement_A;
while( condition ) {
    // code block
    statement_B
}
```

Could be written as:

```
for( statement_A ; condition ; statement_B ) {
    // code block
}
```

Example:

```
for( int i = 0 ; i < 100 ; i++) {
    // code block
}
```

◀ ▶ ⏪ ⏩ 🔍 ↺

Introduction to web development in Java/JSP  
Dennis Olsson

Today  
Composite types  
Java  
Common loops  
Output to terminal  
Example class  
Example usage  
Constructors  
Polymorphism  
Interfaces  
Polymorphism  
Packages  
Dynamic class loading  
Error Handling  
Summary

## I/O

For the sake of example creation  
Standard output – send text to the console:

```
System.out.print("Print this");
System.out.println("Print this");
```

Standard error – send text unbuffered to the console:

```
System.err.print("Print this");
System.err.println("Print this");
```

Also, System.in exists, which can be used to read text from the console.

◀ ▶ ⏪ ⏩ 🔍 ↺

Introduction to web development in Java/JSP  
Dennis Olsson

Today  
Composite types  
Java  
Common loops  
Output to terminal  
Example class  
Example usage  
Constructors  
Polymorphism  
Interfaces  
Polymorphism  
Packages  
Dynamic class loading  
Error Handling  
Summary

## I/O in JSP

In JSP the variable out can be used to send text to the browser. For instance:

```
out.println("Hello World!");
```

◀ ▶ ⏪ ⏩ 🔍 ↺

## Example class - MyNumber

```
class MyNumber {  
    // Member variable, default value = 0  
    private int number = 0;  
  
    // Set the value of the number  
    public void setNumber(int number) {  
        this.number = number;  
    }  
  
    // Get the value of the number  
    public int getNumber() {  
        return number;  
    }  
}
```

## Explanation of MyNumber

```
private int number = 0;
```

A private variable, which means it's reachable from within the class itself, but not from the outside.

The value is set to 0 as soon as the object is created.

## Explanation of MyNumber

```
public void setNumber(int number) {  
    this.number = number;  
}
```

A public method, reachable from everywhere were this object is known.

Will set the number of the member variable (this is used since there are several variables with the same name) to the given value.

## Explanation of MyNumber

```
public int getNumber() {  
    return number;  
}
```

A public method, reachable from everywhere were this object is known.

Will get, and return, the number of the member variable. **number**.

## MyNumber

```
class MyNumber {  
    // Member variable, default value = 0  
    private int number = 0;  
  
    // Set the value of the number  
    public void setNumber(int number) {  
        this.number = number;  
    }  
  
    // Get the value of the number  
    public int getNumber() {  
        return number;  
    }  
}
```

## Using MyNumber(1)

```
class MyMax {  
    // Starting value  
    private MyNumber maxNum = new MyNumber();  
  
    // Update the value of the number  
    public boolean addNumber(MyNumber number) {  
        if (maxNum.getNumber() < number.getNumber()) {  
            maxNum.setNumber(number.getNumber());  
            return true;  
        }  
        return false;  
    }  
}
```

- Introduction to web development in Java/JSP
- Dennis Olsson
- Today
- Composite types
- Java
- Common loops
- Output to terminal
- Example class
- Example usage
- Constructors
- Polymorphism
- Interfaces
- Polymorphism
- Packages
- Dynamic class loading
- Error Handling
- Summary

## Using MyNumber(2)

```
// Get the value of the number
public int getMaxNumber() {
    return maxNum.getNumber();
}
}
```



- Introduction to web development in Java/JSP
- Dennis Olsson
- Today
- Composite types
- Java
- Common loops
- Output to terminal
- Example class
- Example usage
- Constructors
- Polymorphism
- Interfaces
- Polymorphism
- Packages
- Dynamic class loading
- Error Handling
- Summary

## Constructors

A constructor is the method first runned in an object. When using "new MyNumber()" there were no constructor defined, and therefore nothing was runned. We can create a constructor as:

```
public class Merlin {
    public Merlin() {
        System.out.println("No arguments");
    }
    public Merlin(int a) {
        System.out.println("Argument given: " + a);
    }
}
```

This enables "new Merlin()" as well as "new Merlin(4)";



- Introduction to web development in Java/JSP
- Dennis Olsson
- Today
- Composite types
- Java
- Common loops
- Output to terminal
- Example class
- Example usage
- Constructors
- Polymorphism
- Interfaces
- Polymorphism
- Packages
- Dynamic class loading
- Error Handling
- Summary

## Overloading

Several methods (for example constructors) can co-exist with the same name, as long as they have different arguments. This is called overloading.



- Introduction to web development in Java/JSP
- Dennis Olsson
- Today
- Composite types
- Java
- Common loops
- Output to terminal
- Example class
- Example usage
- Constructors
- Polymorphism
- Interfaces
- Polymorphism
- Packages
- Dynamic class loading
- Error Handling
- Summary

## Interfaces

An interface is a content list of required methods. This way you could know that a certain method is implemented if the class implements the interface.

Could be compared with a "content"-list on the outside of the box, guarenteeing some variabler/methods to exist.



- Introduction to web development in Java/JSP
- Dennis Olsson
- Today
- Composite types
- Java
- Common loops
- Output to terminal
- Example class
- Example usage
- Constructors
- Polymorphism
- Interfaces
- Polymorphism
- Packages
- Dynamic class loading
- Error Handling
- Summary

## Interfaces – example

```
public interface Magician {
    public void doMagic();
}

public class Merlin implements Magician {
    public void printName() {
        System.out.println("My name is Merlin");
    }

    public void doMagic() {
        System.out.println("Abracadabra");
    }
}
```



- Introduction to web development in Java/JSP
- Dennis Olsson
- Today
- Composite types
- Java
- Common loops
- Output to terminal
- Example class
- Example usage
- Constructors
- Polymorphism
- Interfaces
- Polymorphism
- Packages
- Dynamic class loading
- Error Handling
- Summary

## Polymorphism

A class is, as well as itself, a version of it's parents and all interfaces.

Merlin is both **Merlin** and a **Magician**.

Therefore we could use Merlin as a Magician only, if we want.

```
Magician m = new Merlin();
m.doMagic();
m.printName(); //<- Will result in a compile error
// A Magician is not Merlin
```



## Polymorphism

All objects are at least an Object.

Test any polymorphic match with **instanceof**.

```
Object m = new Merlin();

if(m instanceof Magician)
    System.out.println("We have a magician!");

if(m instanceof Merlin)
    System.out.println("The magician is Merlin.");
```

## Typecasting

If you are sure a class implements a certain interface you could typecast it. This will “force” the object to be handled as the given one.

Typecasting is done by giving the target type within parenthesis before the variable.

```
Object m = new Merlin();

if(m instanceof Merlin)
    ((Merlin)m).printName();

if(m instanceof Magician)
    ((Magician)m).doMagic();
```

## Structuring the code

Ten or more classes can lead to problems with colliding names and differences between classes becoming hard to remember.

There's a need for structure between classes.

## Packages

Packages is defined with the keyword **package**, for instance:

```
package fantasy;

public class Merlin implements Magician {
    // ...
}
```

## Importing

To use an external package or parts from it, use the keyword **import**. A trailing wild card will enable usage of all classes in the package. An alternative is to only define one class.

```
import fantasy.*;
import java.util.LinkedList;
...
Object m = new Merlin();
LinkedList l = new LinkedList();
```

This goes for classes as well as interfaces. By convention, a reversed domain name is used, for instance `se.umu.cs.doa.*`, which will enable usage of all classes in “se.umu.cs.doa” directly.

## Classpath

The packages are searched for in the same order as the **classpath** contains them. The Java API comes last.

Classes could also be used by calling them directly, as:

```
java.util.LinkedList l = new java.util.LinkedList();
```

If you have a hierarchy in your classpath, each “.” should mean “a sub directory of”. This means that “se.umu.cs.doa.AClass” would be placed in: `SOMEDIR/se/umu/cs/doa/AClass.java`, the package of AClass should be “se.umu.cs.doa” and SOMEDIR should be added to your **classpath**.

## Java ARchives

A set of directories in the base of a classpath (for instance SOMEDIR in the previous slide) can be packed into a JAR-file (Java ARchive).

A JAR-file is a special ZIP-file with a Manifest, an explanation of the archive. The manifest could also include user-specified fields readable by the application.

These archives can be added to the classpath.

## Dynamic class loading

To load a class in a dynamic way, we use code like:

```
try {
    Class c = getClass();
    Object m = c.forName("Merlin").newInstance();
} catch (Exception e) {
    e.printStackTrace();
}
```

Ignore the **try...catch** until the next slide.

This loads the class with the given name, and creates a new object (instance) of it.

## try ... catch

What about **try...catch**?

When something goes wrong, we can throw an **exception**. It is a small object used only to identify an error.

If an exception is created and thrown in a method, the parent (caller) must either handle it, or throw it to its parent. If a method could throw (or pass on) an exception, it must be specified in the declaration.

```
public void doMagic()
    throws OutOfMagicDustException {
    ...
}
```

## Using try ... catch

If you want to handle a thrown exception you must use **try...catch**. The **try**-block will be executed, and as soon as an exception is thrown, it's aborted, and the correct handler (prioritized from the top down) will be runned. The first polymorphic match will be used.

This means you can have several **catch**-blocks.

```
try {
    m.doMagic();
    anotherObject.setStuff(2);
} catch (OutOfMagicDustException oomde) {
    oomde.printStackTrace();
} catch (StuffReadOnlyException sroe) {
    sroe.printStackTrace();
}
```

## Runtime Exceptions

Some exceptions may be thrown without notice. For instance **NullPointerException**, which means you didn't initialize something properly before using it, or **ArithmeticException** when dividing something by zero.

These exceptions can be caught as well.

To catch all exceptions, use their most common polymorphic form, the interface **Throwable**.

```
} catch (Throwable t) {
```

## Summary

- What Classes and Objects are.
- Seen some examples of simple classes.
- Talked about constructors.
- How objects can be seen as more general ones with polymorphism.
- What an interface is, and how (and for what) it can be used, and its connection to polymorphism.
- Packages, classpath, dynamic class loading.
- Error handling.