Theory assignment 2

CLASS MODE	LECTURE
Notes	☐
Property	
SLIDES	
TOPIC	

▼ 12: Exceptions

- Class Exception has many subclasses
- Its superclass is the class Throwable
- Objects of type Throwable can be thrown

```
throw new Exeption("...");
```

- Because of polymorphism: every object of type Exception has type Throwable too
- Subclasses of Exception can be divided in

1. **Checked exceptions**
   - Throws

```
void m1() {
  ....
  r[i]
  ...
}

void m2 () throws Exception {
...
  if (...) {
  throw new Exception ("...");
  }
}
```

2. **Unchecked exceptions / Runtime exceptions**
   - Does not throw

```
void m1() {
  ....
  r[i]
  ...
}
```

- If you cannot recover from an error

```
throw new Error ("...")
```

▼ 13: Inheritance 4 (Overriding)

- Method from subclass overwrites method from superclass if they have same signature
- Java will always try to override if signatures are the same, but in some cases overrrides do not succeed

1) Return types are not the same or a mix of void and return type

- Example 1

```
Superclass:
void m (int i) --> signature: m(int)

Subclass:
int m (int x) --> signature: m(int)
```

Does not work, because the types (void and int) are not the same

- Example 2

```
class A {
  A m () {
    return null;
}

class B extends A {
  int m () {
    return 0;
  }
}
```

Does not work, because return types are not the same

- Example 3

```
class A {
  A m () {
    return null;
}

class B extends A {
  B m () {
    return null;
  }
}
```

Works, because m () returns object of type B, and every object of type B also has type A

2) The method in subclass has a more restrictive modifier than the method in the superclass

- A method "void m () {..}" cannot be overridden with a method "private void m () {..}"
- Example 1

```
class A {
  void m () {
  ...
}

class B extends A {
  private void m () {
    ...
  }
}
```

- Example 2

```
class A {
  public void m () {
  ...
}

class B extends A {
  void m () {
    ...
```

```
  }
}
```

3) If method in super class does not throw exception and method in subclass throws a checked exception

- Example 1

```
class A {
  void m () throws Exception {
  ...
}

class B extends A {
  void m () {
    ...
  }
}
```

Works, because it can work the other way around

- Example 2

```
class A {
  void m () {
  ...
}

class B extends A {
  void m () throws Exception {
    ...
  }
}
```

Does not work

- Example 3 with runtime exception

```
class A {
  void m () {
  ...
}

class B extends A {
  void m () throws ArrayIndexOutOfBoundsException{
    ...
  }
}
```

Does work

▼ 14: Interfaces 2 (Interfaces define a type)

- Interfaces, just as classes, define a type
- How to create a new class

```
Interface F {
...
}
```

The new type F has been created

- Consequences

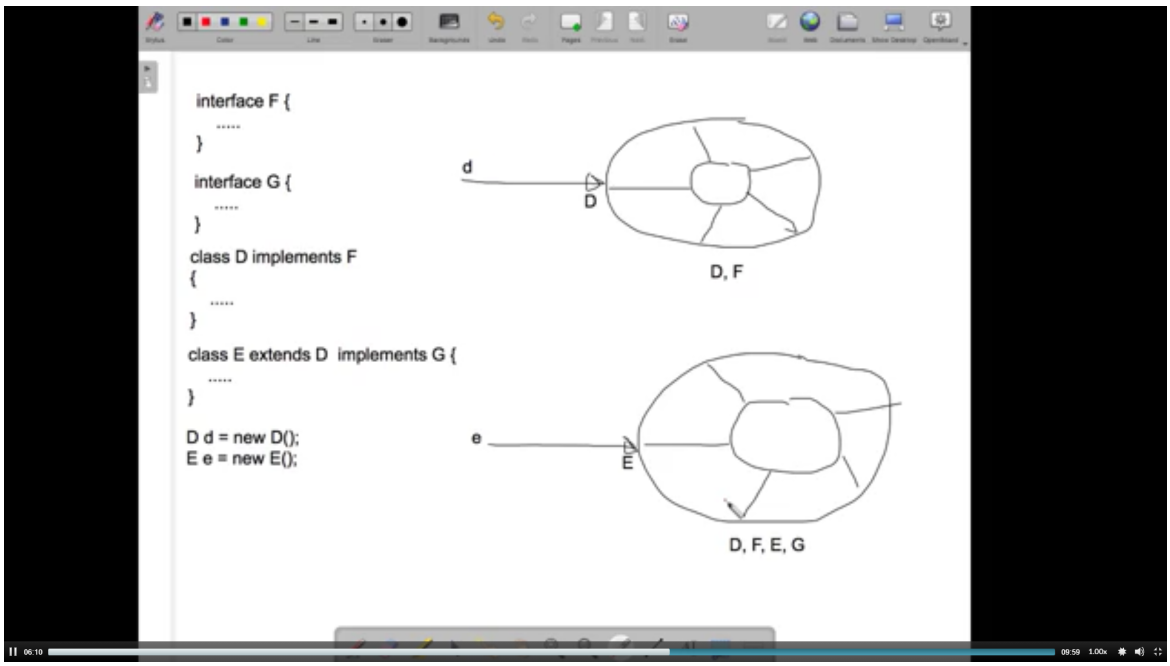
1. It is now possible to declare variables of an interface type
   F f''
2. If a class implements an interface than instances of that class have the type of that class and the type of the implemented interface (polymorphism)

```
class D implements F {
...
}

D d = new D() ;
```

Now, the object that the reference d is pointing to has both the types D and F

- Example 1

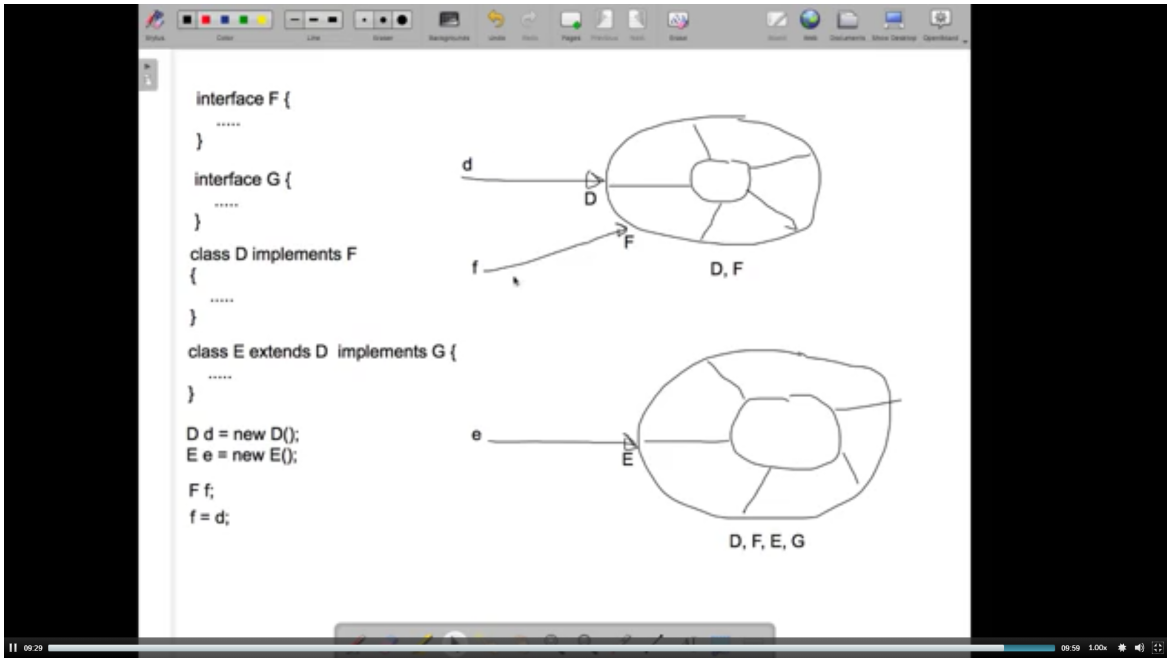


- After the declaration

```
f F;
```

How to assign a value to F?

```
f = new F(); --> error
f = d;
```



▼ 15: Wrapper classes - Autoboxing - Unboxing

Wrapper classes

- Classes that contain 1 value of a primitive type
- Used in cases where the value you want is of a primitive type, but only an object type can be used
- Examples

```
class Integer {...} --> 1 int value
class Double {...} ---> 1 double value
class Character {...} --> 1 char value
```

- Conversions primitive type → object type and object type → primitive type, done as follows:

```
Integer integer = new integer(7); // put 7 in an Integer-object
int i = integer.intValue(); // get the 7 out again
```

- Also contain constants +operations that are useful to have for values of the primitive type they wrap around
- Wrapper class Integer

```
Integer.MIN_VALUE; // most negative int value
Integer.MAX_VALUE; // most positive int value
```

- Wrapper class Character

```
Character.MIN_VALUE; // lowest char value
Character.MAX_VALU; // highest char value
Character.toUppercase("a");
Character.isWhiteSpace(" ");
Character.isLetter("a");
........
```

Autoboxing - Unboxing

- Conversion from primitive type → object that is instance of wrapper class = autoboxing
- Object that is instance of wrapper class → conversion from primitive type = unboxing
- Example 1

```
Integer number = 5; // autoboxing
int i = number; // unboxing
number = i + number; // number contains the int value 20
i = 3 * number; // unboxing, i = 15
```

- Example 2

```
class E {

  void start () {
    Integer number = 5;
    System.outprintf("number = %s\n", i); // every object contains method .toString()

    int i = number;
    System.out.printf("i = %d\n", i);

    i = 3 * number;
    System.out.printf("i = %d\n", i);

    number = i + number;
    System.out.printf("i = %s\n", number);
  }

  public static vid main (String[] argv) {
    New E().start(); // Java automatically writes default construcers
  }
}
```

```
Output:

number = 5;
i = 5;
i = 15;
number = 20;
```

▼ 16: Generic datastructures

- Datastuctures with fixed structure, but nonfixed elements. Thus, elements can be of any type.
- Make a generic row (a row in which we can put any value), as follows:

```
static final int MAX_NUMBER_OF_ELEMENTS = 4;
Object[] genericRow = new Object[MAX_NUMBER_OF_ELEMENTS];

Now, the following examples are all possible:

genericRow[0] = 3;
genericRow[0] = "abc";
genericRow[0] = new Scanner(System.in);
genericRow[0] = new StringBuffer("abc");
```

- Since all objects have, through polymorphism, the type Object, any object can be put in the array. Even values of primitive types can be put in the array into an object that is an instance of a wrapper class (autoboxing).
- Too generic: how do we know what type an element of the array is; genericRow[0] = ?
  → Using instanceOf does not work, because genericRow[i] can be anything
  → Using marker interfaces, we limit the possible amunt of tyoes that the elements of the generic datastructure can have
- Example

```
class Apple {...}
class Pear {...}
class Orange {...}
```

We only want the objects that are instances of the classes above → use a marker interface:

```
interface Fruit {
  //
}
```

As the type of the elements of the generic row:

```
static final int MAX_NUMBER_OF_PIECES_OF_FRUIT = 10;
Fruit[] fruitBasket = new Fruit [MAX_NUMBER_OF_PIECES_OF_FRUIT]
```

And then giving instances of the classes Apple, Pear and Orange the type fruit by letting these classes implement the marker interface

```
class Apple implements Fruit {...}
class Pear implements Fruit {...}
class Orange implements Fruit {...}
```

Now, only object with type fruit can be put in the array fruitBasket and we only gave this type to the Apple-, Pear- and Orange-objects