EXCEPTIONS
It improves the code because the code does not have to include error handling code if it is not capable of handling it. It can propagate the exception up the chain and so that the exception can be handled somewhere at a more appropriate place.
? wer löst stack trace einer Exception aus?
? wann wird stack trace ausgelöst?
Ausgabe: 3 Ways to print an exception
- Java prints it out System.out.println(e);
- Print just the message System.out.println(e.getMessage());
- Print where the stack trace comes from e.printStackTrace();

- The main method of a program can declare that it throws checked exceptions
- A method declaring that it throws a certain exception class may throw instances of any subclass of that exception class.
- The overriding method may opt not to declare any throws clause even if the original method has a throws clause. An overriding method therefore cannot return a different type (except a subtype) or throw a wider spectrum of exceptions than the original method in the superclass. This, of course, applies only to checked exceptions because unchecked exceptions are not required to be declared at all.
A method that throws a 'checked' exception i.e. an exception that is not a subclass of Error or RuntimeException, either **must declare it in throws clause or put the code that throws the exception in a try/catch block**.

Which exceptions are not allowed if a method overrides a superclass or interface?
- When **a method overrides a method in a superclass or interface**, it **is not allowed to add checked exceptions**.
- It is allowed to declare fewer exceptions or declare a subclass of adeclared exception. **Methods declare exceptions with the keyword throws**.

STANDARD JAVA EXCEPTION CLASSES
- java.io.FileNotFoundException
- java.lang.SecurityException RT

POLYMORPHISM- A subclass may have a static method with the same signature as a static method in the base class but it is not called overriding. It is called **hiding** because **the concept of polymorphism doesn't apply to static members**.

PASSED BY VALUE
- **Primitives** are always passed by value
- **Object "references"** are passed by value. So it looks like the object is passed by reference but actually it is the value of the reference that is passed

THIS
... can only be called in a constructor and that too as a first statement.

VARIABLES -------
-- LOCAL
- You cannot specify visibilty of local variables

-- BOOLEAN
Boolean class has two constructors
- **Boolean(String):** true if string argument ist not null
and equal ignC to the strin "true"
- **Boolean(boolean**): false
new Boolean("True") produces a Boolean object that represents true.
new Boolean("yes") produces a Boolean object that represents false.

- When you use the **equality operator (==) with booleans**, if exactly one of the operands is a Boolean wrapper, it is first unboxed into a boolean primitive and then the two are compared (JLS 15.21.2). If both are Boolean wrappers, then their references are compared just like in the case of other objects. Thus, new Boolean("true") == new Boolean("true") is false, but new Boolean("true") == Boolean.parseBoolean("true") is true.

CONVERSION Primitives
- **Widening** byte < short < int < long < float < double

- Numeric **Promotion Rules**
1. If **two values have different data types**, Java will automatically promote one of the values **to the larger** of the two data types.
2. If one of the values is **integral** and the other is **floating-point**, Java will automatically promote the integral value to **the floating-point** value's data type.
3. Smaller data types, namely **byte**, **short**, and **char**, are **first promoted to int** any time they're used with a Java binary arithmetic operator, even if neither of the operands is int.
4. After all promotion has occurred and the operands have the same data type, **the resulting value will have the same data type as its promoted operands**.

- A **narrowing primitive conversion** may be used if all of the following conditions are satisfied:
1. The expression is a compile time constant expression of type byte, char, short, or int.
2. The type of the variable is byte, short, or char.
3. The value of the expression (which is known at compile time, because it is a constant expression) is representable in the type of the variable.
4. if you have a **final variable** and **its value fits into a smaller type**, then you can assign it **without a cast** because compiler already knows its value and realizes that it can fit into the smaller type. This is called **implicit narrowing** and is **allowed between byte, int, char, and, short** but not for long, float, and double

- Conversion form int to float needs a cast.

OPERATORS
valid one's
- instanceOf
hinweise:
- postinkrement k++; nur bei direkter Zuweisung nachrangig ausgeführt,
bsp: for (int i = 0; i <= 3; i++){ k++; sout: k } <> l = k++; //hier inkr n. Zuw

PRECEDENCE- . operator has more precedence than the cast operator
Bsp: int k = ((Integer) t).intValue()/9 (works); <>
int k = (Integer) t.intValue()/9 (fails);

FLOW ----------

-- SWITCH
switch Compile-Time Konstante:
Deklarierung und Zuweisung in Einem.
- Only **String, byte, char, short, int,** (and their wrapper classes Byte, Character, Short, and Integer), and **enums**
- An empty switch block is a valid construct

- LABELS
continue nur bei for and while
break, continue nicht bei if

STRING
Methods
- length() // is final
- charAt()
- indexOf()
- substring()
- toLowerCase()
- toUpperCase()
- equals() / equalsIgnoreCase()
method checks whether two String objects contain exactly the same characters
in the same order.
- startsWith() / endWith()
- contains()
- replace()
- trim()

STRINGS
- String is a final class and final classes cannot be extended

Strings <> References
1. Literal strings within the same class in the same package represent references to the same String object.
2. Literal strings within different classes in the same package represent references to the same String object.

3. Literal strings within different classes in different packages likewise represent references to the same String object.

4. Strings computed by constant expressions are computed at compile time and then treated as if they were literals.

5. Strings computed at run time are newly created and therefore are distinct. (So line 4 prints false.)

6. The result of explicitly interning a computed string is the same string as any pre-existing literal string with the same contents.

We advise you to read section 3.10.5 String Literals in Java Language Specification

TOSTRING
1. When one of the operands of the + operator is a String and other is an object (other than String), toString method is called on the other operand and then both the Strings are concatenated to produce the result of the operation.

2. Object class contains an implementation of toString that returns the name of the class (including the package name) and the hash code of the object in the format @. For example, System.out.println("Hello, "+new Object()); will print Hello, java.lang.Object@3cd1a2f1, where 3cd1a2f1 is the hash code of the object.

STRING BUFFER
Methods
- append()
- delete()
- insert()
- replace()
- reverse()
- keine trim() -Methode

STRING BUILDER

is final
Methods
- append()
- delete()
- deteteCharAt()
- insert() adds characters to the StringBuilder at the requested index and returns a reference to the current StringBuilder 3: StringBuilder sb = new StringBuilder("animals");

4: sb.insert(7, "-"); // sb = animals-
5: sb.insert(0, "-"); // sb = -animals-

- reverse()
- charAt()
- indexOf()
- length()
- substring()
- toString()

ARRAY
- In Java, arrays are just like regular Objects and arrays of different types have different **class names**. For example, the class name of an **int[]** is **[I** and the class name for **int[][]** is **[[I**.
- For array classes, the **isArray()** method of a Class **returns true**. For example, twoD.getClass().isArray() will return true.

ARRAY LIST
• Gehört zum Collection framework
• Kann seine Größe nachträglich verändern
• Implementiert das List Interface
• akzeptiert null
• Erlaubt Duplikate
• Typsicherheit durch Generics

Eigenschaften
• intern ein Array von java.lang.Object
• Reihenfolge der Elemente bleibt erhalten
• Kann mit addAll Elemente anderer Listen aufnehmen
• clone erzeugt eine „Schattenkopie“

Methods
- add()
- remove()
- set()
- isEmpty()
- size()
- clear()
- contains()
- equals()

ArrayList has a custom implementation of equals() so you can compare two lists to see if they **contain the same elements in the same order**.
boolean equals(Object object) s134

Sorting
- **Collection.sort**(numbers);

LIST

LIST <> ARRAYLIST
List is an Interface. Arraylist is a class.

List is Generic. Arraylist is Specific.
The two can be substituted, but it is not recommended. This is the most recommended syntax:
List list = new ArrayList();

EQUALS
- String (Pool)
- String (Object)
- ArrayList

CONVERTING array and List
ArrayList -> array
3: **List list = new ArrayList<>()**; //Liste erzeugen
4: list.add("hawk");
5: list.add("robin");

8: String[] stringArray = list.toArray(new String[0]);
9: System.out.println(stringArray.length); // 2

Array -> LIst
20: String[] array = { "hawk", "robin" }; // [hawk, robin]
21: **List list = Arrays.asList(array**); // returns fixed size list

SORTING

ACCESS MODIFIERS
Class -> protected in superclass
- Class inherits from superclass
ClassMethod
-- without reference (directly)
-> package access to superclass members
ClassMethod + new Object
-- with Class Reference
-> package access to superclass members
-- with superclass Reference
-> no package access to superclass members //No Compiles

HIDING- A final variable can be hidden in a subclass

CLASS
- The visibility of the class is not limited by the visibility of its members.
A class with all the members declared private can still be declared public or a class having all public members may be declared private.

CONSTRUCTOR
- It is provided by the compiler only if the class does not define any constructo
- It calls the no-args constructor of the super class.
- If a subclass does not have any declared constructors, the implicit default constructor of the subclass will have a call to **super( ).**

MAIN
static public void main(**String[] args**)
public static void main(**String args[]**)
public static void main(**String… args**)
- public static void main() throws IOException
- zeroth element of the string array contains: "java"
Most of the actual date related classes in the Date-Time API such as LocalDate, LocalTime, and LocalDateTime are immutable.
- you can create arrays of any type with length zero

IMPORT
- need to use the class's fully qualified name

exp: util.log4j.Logger logger = new util.log4j.Logger();

IMPORT STATIC
import static ..;

INTERFACES
- All Variables (member vars) are public static final and must be ASSIGNED

ABSTRACT CLASS
- A subclass can be declared abstract regardless of whether the superclass was declared abstract.
- A class cannot be declared abstract and final at the same time. This restriction makes sense because abstract classes need to be subclassed to be useful and final forbids subclasses.

INHERITANCE
- **Covariant** returns: .. an overriding method can chante the returnt type of a subclass of the return type declared in the overriden method. (not for primitives) enthu 1/21
PRINCIPLES OF OBJECT TYPES AND REFERENCING TYPES
1. The **type of the object** determines **which properties** exist within the object in memory.
2. The **type of the reference** to the object determines **which methods and variables** are **accessible** to the Java program.

CASTING OBJECTS
1. Casting an object from a **sub**class to a **süper**class **doesn't** require an **explicit cast**.
2. Casting an object from a **super**class to a **sub**lclass **requires** an **explicit cast**.
3. The **compiler will not allow casts to unrelated types**.
4. Even when the code compiles without issue, an **expression** may be **thrown at runtime** if the **object being cast ist not actually an instance of that cast**

ASSIGNMENT
A subclass can ALWAYS be assigned to a super class variable without any cast

INSTANCE

METHODS
- No access modifiers in methods (only in/with classes)

— OVERRIDING - When the **return type of the overridden method** (i.e. the method in the base/super class) is a **primitive**, the
return type of the **overriding method** (i.e. the method in the sub class) **must match** the **return type of the**
overridden method.
- In case of **Objects**, the **base class method** can have a **covariant return type**, which means, itcan return either
return **the same class** or a **sub class object**.
- RULES
- The **access modifier** must be the **same or more accessible**.
- The **return type** must be the **same or more restrictive**, also known as **covariant**
- If any **checked exception are thrown**, **only the same exceptions or subclasses of those exceptions are allowed** to be thrown.
- The methods must **not be static** (If they are, the method is hidden and not overridden).

STATIC-- STATIC METHOD - To call a static method of another class, you must use the name of the other class, for example OtherClass.staticMethod();
VIRTUAL METHODS

A **virtual method** is a method in which **the specific implementation is not determined until runtime**.

In fact, **all non-final, nonstatic**,and **non-private** Java methods are considered **virtual methods**, since **any of them can be overridden at runtime**

POLYMORHISM
The nature of the polymorphism is **that an object can take on many different forms**.
By **combining** your understanding of **polymorphism with method overriding**, you see that **objects may be interpreted in vastly different ways at runtime**, especially in methods defined in the superclasses of the objects.
Furthermore, a cast is not required if the object is being reassigned to a super type or interface of the object.

Java supports polymorphism,
the **property of an object to take on many different forms**.
To put this more precisely, **a Java object may be accessed using a reference**
- with the **same type** as the object,
- a reference that is a **superclass of the object**, or
- a reference that defines **an interface** the object implements, either directly or through a superclass.

LOCALDATE, LOCALTIME, LOCALDATETIME
- implement TemporalAccessor.
- DateTime API uses calendar system defined in ISO-8601
- Most of the actual date related classes in the Date-Time API such as LocalDate, LocalTime, and LocalDateTime are **immutable**.

DATE TIME FORMATTER
The **format() method** is **declared on** both the **formatter objects** and the **date/time objects**, allowing you to reference the objects in either order. The following statements print exactly the same thing as the previous code:

DateTimeFormatter shortDateTime =
DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);
System.out.println(**dateTime.format**(shortDateTime));
System.out.println(**date.format**(shortDateTime));
System.out.println(**time.format**(shortDateTime));

DATE
Represent just a date without any time or zone information
- java.util

LAMBDAs
- PREDICATE
- has only one abstract method (among other non-abstract methods)
- signature: public boolean test(T,t);
- **Predicate is typed to List** (not ArrayList) in the checkList method, therefore, the parameter type in the lambda expression must also be List. It cannot be ArrayList.

GARBAGE COLLECTION
1. An object can be made eligible for garbage collection by making sure there are no references pointing to that object.

2. You cannot directly invoke the garbage collector. You can suggest the JVM to perform garbage collection by calling System.gc();

DIGITS

▪ **octal** (digits **0–7**), which uses the number **0 as a prefix**—for example, 017
017 //15
▪ **hexadecima**l (digits **0–9** and letters **A–F**), which uses the number **0** followed by **x** or **X**
as a prefix—for example, 0xFF
0x1F //31
▪ **binary** (digits **0–1**), which uses the number **0** followed by **b** or **B as a prefix**—for example, 0b10
0b11 //3