Interfaces / Vererbung / Dynamisches Binden:

- In Interfaces dürfen **nur Methoden und Konstanten** (static und final Variablen) definiert werden, alle anderen Variablen sind verboten.
- **Polymorphismus:** ein Objekt kann durch Vererbung mehrere Typen haben
- Jeder Typ ist **Unter- und Obertyp** von sich selbst
- Jede Lösung mit **dynamischen Binden** ist auch durch if - und switch Anweisungen lösbar.
- Programme, die durch dynamisches Binden gelöst wurden sind einfacher wattbar als solche, die mittels switch und if Anweisungen erstellt wurden.
- **Prinzip der Ersetzbarkeit:** Typ U soll nur dann Untertyp von Typ T sein, wenn Objekte von U überall verwendbar sind, wo Objekte von Typ T erwartet werden.
- **Konstanten** werden meist **nur in Großbuchstaben** geschrieben und als **public static final deklariert** und müssen somit bei der **Deklarierung initialisiert** werden.
- **private Methoden** werden immer **statisch gebunden**
- In jeder .java Datei darf **nur eine public Klasse** enthalten sein.
- nicht public Klassen sind nur im Selben Paket verwendbar

Vererbung:

- Unterklassen können von Oberklassen durch das Schlüsselwort „extends“ erben, dadurch ergeben sich folgende Vorteile:
    - Alle in der Oberklasse definierten Methoden und Variablen sind **auch in der Unterklasse vorhanden**, insofern sie nicht in der Oberklasse als static deklariert wurden
    - Als **private deklariert Methoden** und Variablen in der Oberklasse sind in der Unterklasse nicht sichtbar, allerdings können vererbte Methoden von der Oberklasse auf diese Variablen/Methoden zugreifen.
    - Von der Oberklasse geerbte Methoden können **in der Unterklasse überschrieben werden** (final Methoden dürfen nicht überschrieben werden)
    - **Unterklasse kann die Oberklasse erweitern**, also neue eigenständige Methoden und Variablen implementieren.
    - Hat eine neu definierte Variable den gleichen Namen, wie eine Variable in der Oberklasse, **so wird die Variable in der Oberklasse verdeckt**. In der Unterklasse ist nur die Variable der Unterklasse zu sehen, obwohl beide Variablen existieren.
    - Beim Erzeugen einer Unterklasse wird zuerst immer der Konstruktor der Oberklasse ausgeführt
- In Java kann **eine Klasse immer nur von einer anderen Klasse abgeleitet** werden —> funktioniert nur, wenn die Typen eine Hierarchie bilden
- Jedes **Interface kann dagegen ein mehrere andere Interfaces erweitern** und jede Klasse kann mehrere Interfaces implementieren

- In **Interfaces können keine Variablen deklariert werden** und keine Methoden implementiert, sondern nur definiert werden
- Von **Interfaces können keine Instanzen** erzeugt werden
- Von einer **final deklarierten Klasse** darf keine andere Klasse abgeleitet werden, eine final definierte Methode darf in einer Unterklasse nicht überschrieben werden
- Von **„abstract“ definierten Klassen** darf keine Instanz erzeugt werden. Diese Klassen enthalten abstrakte Methoden, die ebenso wie in Interfaces nur definiert werden in der abstrakten Klasse. Diese können dann ähnlich wie Interfaces von Unterklasse vererbt werden und müssen dort implementiert werden. „abstract“ und „final“ sind somit fast gegensätzlich.
- **Die Klasse Object ist die Oberklasse aller Klassen**, die nicht von anderen selbst definierten Hierarchien erben. Die Klasse Object selbst erbt von keiner andern Klasse.
- Jede Klasse erbt **Methoden von Object**:
    - **<String> toString():** gibt eine lesbare Zeichenkette zurück, diese hat oft wenig Aussagekraft, deswegen muss diese Methode oft überschrieben werden.
    - **<boolean> equals(Object other):** Überprüft dieses mit other auf Identität, oft ist aber andere Überprüfung sinnvoll (zB: auf Gleichheit), deswegen muss auch equals oft überschrieben werden. Allerdings hat equals immer einen Typ von Object als Eingabeparameter.
        - Durch x.instanceof(y) kann festgestellt werden, ob der linke Operand vom selben Typ wie der rechte Operand ist.
        - Falls x.instanceof(y) == true, dann kann das zu vergleichend Objekt auf die gewünschte Instanz gecastet werden und somit dann auch auf die Variablen zugegriffen werden und diese dann wiederum vergleichen
        - anstatt x.instanceof(y), kann auch x.getClass() == y.getClass verwendet verwendet werden
    - **<int> hashCode():** Errechnet aus jedem Objekt einen **spezifischen Zahlenwert.**
        - Auf **gleiche Objekte wird auch der gleiche Zahlenwert ausgegeben** (aus diesem Grund muss falls equals überschrieben wurde auch hashCode überschrieben werden, damit diese Bedingung nicht verletzt wird),
        - **Die Umkehrung gilt nicht,** wenn zwei Objekte den gleichen hashCode haben, dann bedeutet das nicht, dass diese beiden Objekte gleich sind.
    - **clone:** Erzeugt eine Kopie des Objekts

Der Quellcode als Kommunikationsmedium:

- **Kommentare als Informationsträger** im Programm zentral
- Kommentare können als **Zusicherung** verstanden werden
    - **Vorbedingungen:** Bedingung die vor der Ausführung einer Methode oder eines Konstrukteurs erfüllt sein müssen
    - **Nachbedingungen:** Bedingungen, die nach der Ausführung von Methoden oder Konstruktiven erfüllt sein müssen
    - **Invarianten:** Bedingungen auf Variablen, Klassen und Interfaces, die stets eingehalten werden müssen, damit das Programm reibungslos funktioniert.

Qualitätssicherung:

- **Design by Contract:**
    - Anbieter wird **Server** genannt und Kunde **Client**, in der Softwareentwicklung können diese auch Objekte sein
    - **Server bietet Services** in Form von Methodenausführungen an
    - Client nimmt Services in Form von Methodenaufrufen in Anspruch
    - Details sind im **Software Vertrag** geregelt —> Man weiß was man sich von dem Aufruf einer Methode erwarten kann, ohne die genaue Implementierung zu kennen.
- **Klare Richtlinien** in Design by Contract:
    - **Vorbedingungen** in Untertypen dürfen schwächer aber nicht stärker sein, als entsprechende Bedingungen im Obertyp
    - **Nachbedingungen** im Untertyp dürfen stärker, jedoch nicht schwächer als die Bedingungen im Obertyp sein

    ——> Instanzen eines Untertyps verhalten sich so, wie man es sich von der Instanz eines Untertyps erwarten würde
- **Rechte/Pflichten des Servers:**
    - Darf davon ausgehen, das die Vorbedingung und die Invarianten vor dem Methodenaufruf erfüllt sind
    - Muss dafür sorgen, dass nach Methodenaufruf die Nachbedingungen eingehalten sind und die Invarianten des Objekts erfüllt sind.
- **Schleifeninvarianten:** Bedingen, die zu Beginn und am Ende jedes Schleifendurchlaufs erfüllt sein müssen

- **Testen und Softwarequalität:**
    - Testfälle decken nicht alle Möglichkeiten ab
    - gefunden Fehler lassen Rückschlüsse über vorhandene Fehler im Programm zu
    - Beseitigen von Fehlern führt meist zu neuen Fehlern —> Behebung der Ursache des Fehlers und nicht der Symptome
    - Fehler, die nur selten Auftreten stellen eine besondere Bedrohung dar
    - **Unittest:** Funktionaler Test eines klar abgegrenzten Programmteils
    - **Integrationstest:** Überprüft korrekte Zusammenarbeit zwischen den Units
    - **Systemtest:** prüft Erfüllung aller funktionalen und nicht funktionalen Eigenschaften
    - **Abnahmetest:** Test unter realen Bedingungen

- **Ausnahmebehandlung/Exceptionhandling:**
    - Exceptions werden als ganz **normale Objekte** dargestellt
    - All diese Objekte erweitern die Oberklasse Throwable, diese stellt bestimmte Methoden bereit. Klasse Error und Exceptions direkt davon abgeleitet
    - Exceptions **können abgefangen werden,** durch try {…} catch(Excetion ex){…} wird die im try-Block erzeugte Exception durch den catch - Block abgefangen, falls die Exceptions vom angegeben Typ ist
    - Die **Klasse Exception ist der Obertyp** von allen sinnvoll abfangbaren Exceptions
    - **Fehler vom Typ Error** sind nicht sinnvoll abzufangen
    - Analyse des Stack - Traces um Grund für den Exceptionwurf zu finden
    - finally {…} Blöcke folgen auf einen try {…} und beliebig viele catch {…} Blöcke und dienen zum Aufräumen

Algorithmen

- **Bubblesort:** Liste wird immer von rechts nach links durchlaufen, dabei werden immer 2 nebeneinander liegende Elemente betrachtet und diese, falls die Ordnung nicht stimmt vertauscht. Die durchschnittliche Laufzeit liegt bei n^2. Terminiert immer, die Laufzeit hängt allerdings stark von der Vorsortierung der Liste ab.
- **Quicksort: Wahl eines Pivotelemets** (meistens in der Mitte des Arrays), dann Teilen des Arrays in 2 Hälfte. Schnittstelle ist das Pivotelement. Dann **einlaufen in das Array von links und von rechts.** Dann jeweils so lange in das Array vorstoßen, bis auf der rechten Seite ein Element gefunden wurde, das größer ist als das Pivotelement und auf der linke Seite eines das kleiner ist als das Pivotelement. Anschließend werden **diese beiden Elemente vertauscht**. Dieser Vorgang geht so lange bis die beiden Schliefen am Pivotelement angekommen sind. Dann ist ein Zustand erreicht in dem l**inks neben dem Pivotelement nur kleinere Elemente stehen und**

rechts nur größere. Anschließend folgt ein **rekursiver Aufruf der gleichen Methode für die beiden Teillisten** —> **(Divide and Conquer)**

—> Durchschnittliche Laufzeit: **n*log(n)** oder der Worstecase: **n^2**