Mit **Information Hiding** ist das Verstecken der konkreten Implementierung von Daten gemeint. Der Zugriff wird nur über Methoden („getter“ und „setter“) gestattet. (Keine Abhängigkeit von Implementationsdetails durch Konsumenten.)

Die **Bindung** einer Methode m bestimmt welche Implementierung von m bei einem Aufruf von m verwendet wird. Statische Bindung passiert zur Compiletime, dynamische Bindung zur Runtime.

Bei dem Ausdruck „$X\ a$ = new $Y()$“ ist X der statische, und Y der dynamische Typ von a. Klarerweise kann sich Y (sofern kompatibel) ändern, nicht aber X. Implikation: X bestimmt welche Methoden sichtbar sind, Y bestimmt welche ausgeführt werden (dynamische Bindung).

Felder, statische Methoden und Overloads (?) werden statisch gebunden.

Die **Sichtbarkeitsattribute** sind

private: in der deklarierenden Klasse

(default): im eigenen Paket sichtbar

protected: im eigenen Paket und in allen Unterklassen

public: überall sichtbar (in anderen Paketen nur sofern importiert)

Überschreibende Methoden können in ihrer Deklaration die Sichtbarkeit erweitern.

Ein Objekt wird als **unveränderlich** bezeichnet wenn alle seine Felder **final** sind. Wird eine final Variable mit einem Objekt belegt kann zwar der Wert der Variable (Referenz) nicht geändert werden, wohl aber der Inhalt des Objekts (sofern es kein unveränderliches ist).

Eine final Methode kann von Subklassen nicht Überschrieben werden, von einer finalen Klasse kann nicht geerbt werden.

Eine Methode ist **abstrakt** wenn sie keine Implementierung hat. Beinhaltet eine Klasse solche Methoden muss sie auch abstrakt sein. Abstrakte Klassen können nicht instanziert werden.

Ein **Interface** definiert eine Schnittstelle. Beinhaltete Methoden können entweder public abstract (default), default, static oder private sein. Statische Konstanten (final) sind ebenso erlaubt. Aber keine Objektfelder, keine Konstruktoren und ergo keine Instanzierung.

Können von Klassen in beliebiger Anzahl (im Gegensatz zu extends) implementiert werden („implements“) und als statischer Typ verwendet werden.

Innere Klassen sind in einer anderen Klasse deklariert. Statische innere Klasse verhalten sich wie gewohnt, nur eben im Namespace einer anderen Klasse und mit Zugriff auf die statischen Felder (jeder Sichtbarkeit) der deklarierenden Klasse. Instanzen von Member-Klassen können nur innerhalb des deklarierenden Objekts leben, und nur mit dem äußeren Objekt instanziiert werden.

Im Kontext von Generics ermöglichen **Constraints** bounded type parameters („T extends B“) wobei B ein Interface oder eine Klasse ist, die T zumindest erfüllt (also gleich oder spezieller).

Kovariante Zuweisungen sind bei Generics (und sonst auch oft?) nicht typsicher und somit nicht möglich.

Mit **Wildcards** kann bei der Variablendeklaration mit generischem Typ ein lower-bound („? super A“) oder ein upper bound („? extends A“) auf den tatsächlichen Typ gesetzt werden. Der tatsächliche Typ ist dann entweder eine Subklasse oder eine Superklasse von A.

Beim lower-bound (super) sind nur schreibende, beim upper-bound (extends) nur lesende Zugriffe möglich.

Type Erasure meint, dass alle generischen Typen auf Rohtypen übersetzt werden, die mit Object arbeiten. ($A < T >$ wird zu Object, A ist der Rohtyp.) Generics als Konzept gibt es nur zur Compilezeit, nicht zur Laufzeit.

Konsequenz: Basisdatentypen können nicht verwendet werden (boxed reference types müssen verwendet werden) und exceptions können nicht generisch sein (VM kann sie nicht unterscheiden).

Der Typ eines **Lambdas** ist ein Single Abstract Method (SAM) Type, ein Interface mit genau einer abstrakten Methode (auch funktionales Interface). Annotation „@FunctionalInterface“ garantiert SAM Status. Ein SAM Typ kann default Methoden beinhalten. Der genaue Typ eines Lambdas ergibt sich aus dem Kontext (welches funktionale Interface, welche generischen Typparameter, zu implementierende abstrake Methode).