Programmiersprachen

Vorlesung 5

Zusammenfassung

Georg Ernst Moser

Object-oriented Programming

Object-oriented Languages

Abstrakter Datentyp: Ein Datentyp der durch durch die beiden Eigenschaften Kapselung (Zusammenfassen von Funktionen und Variablen zu einer Einheit) und Data hiding (das Verstecken von Informationen, Implementierungsdetails nach außen) beschrieben wird.
Objekt: Ein Objekt ist ein Abstrakter Datentyp
objekt basiert: Eine Objekt basierte Sprache bietet Objekte als Grundelemente an.

klassenbasiert: Eine Sprache ist klassenbasiert, wenn sie Klassen anbietet, deren Instanzen Objekte sind.

Nicht alle objektorientierten Sprachen sind auch klassenbasiert. Das bekannteste Gegenbeispiel ist hierbei self (Erweiterung von Objekten durch Vererbung, Decorator Pattern,..).

objekt-orientiert: Eine objekt basierte Sprache welche auch folgendes erlaubt:

- Definition abstrakter Datentypen (Objekte könnten auch nur predefiniert verfügbar sein)
- Vererbung
- Inclusion polymoprphismus (subtyping): Der Name inclusion polymorphismus kann anhand folgenden Beispiels erklärt werden: Ein Student ist Untertyp von Person, die Menge aller Personen enthält auch alle Studenten.
- Dynamisches Binden: Zur sinnvollen Verwendung von Subtyping

in der Objektorientierten Programmierung spricht man nicht vom Aufrufen einzelner Methoden, sondern vom Senden von Nachrichten. Dies hat seinen Ursprung in den Anfängen von Smalltalk als jedes Objekt noch als eigener Prozess gedacht war.

Objekte enthalten Variablen/Objektvariablen/Instanzvariablen und bieten Methoden (diese sind nicht in den einzelnen Objekten gespeichert) an.

In vielen Sprachen sind Klassen selbst auch Objekte. Die Variablen dieser Klassenobjekte werden als Klassenvariablen bezeichnet.

Derived Class in C++

```
class stack {
      public:
      void push(int) {
            elements[top++] = i;
      }
      int pop() {
            return elements[--top];
      }
      private:
            int elements[100];
            int top = 0;
};

class counting_stack : public stack {    //derived from stack
      public:
            int size();            //return number of elements on the stack
};
```

Im Unterschied zu Java gibt es in C++ neben Konstruktoren auch Destruktoren. Diese enthalten den Code der Ausgeführt werden soll, wenn ein Objekt zerstört wird(Ressourcen freigeben,...).

Um eine Klasse als Unterklasse einer Oberklasse zu markieren schreibt man in C++ nach dem Klassennamen einen Doppelpunkt und anschließend den Namen der Oberklasse. Per Default würde jedoch *private Inheritance* verwendet werden (alle aus der Oberklasse übernommenen Werte, Routinen sind in der Unterklasse *private*). Aus diesem Grund wird zusätzlich public vor den Namen der Oberklasse geschrieben (um zu erreichen dass in diesem Beispiel *push* und *pop* auch in *counting_stack public* sind).

Polymorphism and Dynamic Binding

```
stack s;                        // not polymorph
counting_stack cs;
stack* sp = new stack();        //polymorph
counting_stack* csp = new counting_stack();
```

s und *cs* sind sogenannte Autovariablen. Die Zeile *stack s;* erzeugt bereits ein neues *stack* Objekt auf dem Stack. Mit dem Schlüsselwort *new* wird ausgedrückt, dass das Objekt am Heap erzeugt werden soll.

```
sp = csp;                       // supported
csp = sp;                       // not supported; statically unknown
```

Hier wird jeweils lediglich die Neuzuweisung eines Zeigers durchgeführt

```
s = cs;                         // supported; converted to stack
cs = s;                         // not supported; cs.size() undefined
```

Bei der Zuweisung *s = cs* wird *cs* implizit in einen stack umgewandelt, also alle zusätzlichen Eigenschaften das Untertyps counting_stack gehen verloren.

```
sp->push(...);                  //stack::push
csp->push(...);                 //counting_stack::push
sp = csp;
sp->push(...);                  //which push?
```

Subclass Versus Subtype

Subtyping ist durch das Prinzip der Ersetzbarkeit (Liskovsches Substitutionsprinzip) definiert:

„*Eine Instanz eines Untertyps U kann überall dort verwendet werden wo eine Instanz seines Obertyps T erwartet wird.*“

Eine Unterklasse einer Oberklasse ist nur genau dann auch ein Untertyp wenn (stark vereinfacht):

- Objektvariablen und Methoden dürfen nur hinzugefügt, nicht entfernt werden.
- Methoden dürfen nur auf kompatible Art überschrieben werden:
  - müssen über die gleiche oder über eine kompatible Signatur verfügen. Dies ist von Compiler statisch prüfbar.
  - Methode muss sich äquivalent verhalten. Hierbei weiß nur der Programmierer ob dieses Verhalten gegeben ist. Vom Compiler ist dies nicht prüfbar.

Strong Types and Polymorphism

```
class base {...};
class derived: public base {...};
...
base* b;
derived *d;
```

Kann *b* auf *d* zugewiesen werden, ohne dass sich dadurch Typfehler ergeben? Ist dies möglich, so ist *derived* ein Untertyp von *base*.

Method Overriding

```
class polygon {
      public:
            polygon(...) {...}                        // constructor
            virtual float perimeter() {...};          //virtual
            ...
}

class square: public polygon {
      public:
            square(...) {...}                         // constructor
            float perimeter() {...};                  // virtual, overridden
            ...
}
```

Das *virtual* (kann mit VFT – Virtual Function Tables umgesetzt werden) Statement aktiviert in C++ dynamisches Binden auf der Methode.

Contra-variant Parameter Types

```
class base {
      public:
      virtual void fnc(s1 par) {...}
      };

class derived: public base {
      public:
      void fnc(s2 par) {...}
};
...

      base* b;
      derived* d;
      s1 v1;
      s2 v2;
      if(...) b = d;
      b->fnc(v1);            // what if b is of type derived?
```

Die überschriebene Methode in *derived* nimmt einen anderen Parameter entgegen.
Beim Aufruf der jeweiligen *fnc* kann der Compiler überprüfen, dass der Übergebene Typ ein Untertyp des virtuellen Parameters ist.
Beim Aufruf von *b->fnc(v1)* ist nicht bekannt ob *fnc* in *base* oder *derived* aufgerufen wird. Damit es zu keinen Problemen kommt muss in diesem Fall muss *s2* ein Obertyp von *s1* sein.
Eingangsparameter müssen um dem Ersetzbarkeitsprinzip gerecht zu werden kontravariant („die Typhierarchien laufen gegeneinander“) sein.

Co-variant Result Types

```
class base {
      public:
            virtual t1 fnc(...) {...}
};

class derived: public base {
      public:
            t2 fnc(...) {...}
};
...
      base* b;
      derived* d;
      t0 v0;
      if(...) b = d;
      v0 = b->fnc(...); // what if b is of type derived?
```

In diesem Fall muss *t1* ein Obertyp von *t2* sein (ein Untertyp verfügt mindestens über die nötige Information um seinen Obertyp zu füllen).
Ergebnistypen müssen *kovariant* sein (variieren in dieselbe Richtung wie die Klassenhirachie).

Co-variant Problem

```
class point {
      public:
            float x, y;
            virtual bool equal (point p)
                  { return x == p.x && y == p.y; }
};
class colorPoint : public point {
      public:
            int color;
            bool equal (colorPoint p) // error, co-variant
                  { return x == p.x && y == p.y
                                          && color == p.color; }
};
```

Characteristics of C++

Konstruktoren initialisieren Objekte: Basisklassen werden immer vor Unterklassen initialisiert (in der Typhirachie von oben[1] nach unten)...
Destruktoren (optisch ähnlich den Konstruktoren nur mit vorangestellter Wellenlinie): Basisklassen werden zuletzt zerstört (in der Typhirachie von unten nach oben)

Automatische Objekte: Werden automatisch am Stack initialisiert. Diese leben nur solange wie die Funktion in der sie angelegt wurden. Ein Zurückgeben deren Adresse ist deswegen unklug.

Objekte am Heap müssen explizit allokiert und deallokiert werden. Die Funktionen new und delete sind selbst programmierbar.

Statische Variablen gehören allen Objekten einer Klasse gemeinsam. Dies ist im eigentlichen Objektkonzept allerdings nicht vorgesehen. Bei Templates, wenn die Klasse zu der ein Objekt gehört nicht bekannt ist führt dies zu Problemen.

Zuweisungen und **Vergleichsoperatoren** können durch Überladen selbst programmiert werden.

Virtual Funtions in C++

```
class student {
      public:
            virtual void print() {...}
};
class college_student: public student {
      void print() {...}
};
...
      student* s;
      college_student* cs;
      ...
      s->print();       // calls student::print()
      s = sc;           // okay
      s->print();       // calls college_student::print()
```

1 Unter der Annahme, dass Basistypen ganz oben in der Hierarchie stehen.

Pure Virtual Funtions in C++

Entsprechen den abstrakten Methoden in Java. Dadurch dass sich in der Klasse *shape pure virtual functions* befinden ist diese selbst auch abstrakt. Es können folglich keine direkten Instanzen von *shape* erzeugt werden. Ein Konstruktor ist trotzdem notwendig.

```
class shape {
      public:
            virtual void draw() = 0;      // pure virtual function
            virtual void move(...) = 0;
            virtual void hide() = 0;
            point center;

};
class rectangle : public shape {
      private:
            float length, width;
      public:
            void draw() { ... };          // impl. of derived function
            void move(...) { ... };
            void hide() { ... };
};
```

Inheritance and Visibility in C++

```
class stack {
      public:
            stack() { top = 0 }; // constructor
            void push(int i) { s[top++] = i; };
            int pop() { return s[--top]; };
      protected:
            int top;
      private:
            int s[100];
};
class counting_stack : public stack {
      public:
            int size() { return top; };
};
```

Der Sichtbarkeitsmodifikator *protected* sagt in C++ dass das Feld/die Funktion in der eigenen und allen Unterklassen sichtbar sein soll.
Der Erfinder von C++ Bjarne Stroustrup äußerte sich später zum *protected* Modifikator, dass dessen Einführung sein größter Fehler war.

Beispiel:
Derjenige der *counting_stack* implementiert muss nicht der gleiche sein, der auch *stack* implementiert hat und muss sich deswegen nicht über die Bedeutung von *top* im Klaren sein.
Hat der Entwickler von *stack* auch keine Ahnung von der Existenz von *counting_stack* und verschiebt die Bedeutung von *top* in eine ihm sinnvoll/angenehm erscheinende Richtung, so wird *counting_stack* nicht mehr funktionieren.
Man glaubt durch *protected* auf der sicheren Seite zu sein (man verändert ja nur die eigenen Variablen), ist sich aber potentieller Auswirkungen nicht bewusst.

Eine Lösung wäre statt *protected public* zu verwenden und in eine Kommentar anzugeben wer die Variable wie verwenden kann (also sich gleich im Klaren zu sein, dass diese nicht rein für den internen Gebrauch ist).

Polymorphism in C++

Generizität (parametrischer Polymorphismus) wird in C++ durch Templates umgesetzt. Die Templates an sich sind so mächtig, dass sie selbst eine turingvollständige Sprache bilden. Mächtig ist allerdings nicht gleichbedeutend mit einfach zu verwenden. Im Gegenteil, diese beiden Eigenschaften stehen meist in direktem Gegensatz.

In Haskell wird eine ganz ähnliche Form der Generizität verwendet wie in C++.

Jede Form der Generizität wird statisch aufgelöst, Generizität ist ein statisches Konzept.
C++ verwendet heterogene Generizität: Aus einer generischen Klasse werden ggf. mehrere (so viele wie benötigt) übersetzte Klassen erstellt.
Java verwendet homogene Übersetzung: Aus Jeder generischen Klasse wird genau eine übersetzte Klasse erstellt (in allgemeinster Form mittels *Object*)

Weiters unterstützt C++ Mehrfachvererbung.
Funktionen und Operatoren können beliebig überladen werden. Überladung wird immer statisch aufgelöst.

Es gibt implizite (zum Beispiel von *short* auf *int*, wenn der Parameter vom Typ *int* ist) und explizite (explizit hingeschrieben) Casts. Diese können sowohl unüberprüft, als auch überprüft sein.

Typinformation ist zur Laufzeit verfügbar (run-time type information – rtti).

Package and Tagged Record in Ada

Tagged Records wurden erstmals in Ada95 eingeführt. s Schlüsselwort „*is tagged*“ gibt an, dass irgendwo in dem Record ein Tag versteckt ist, der Aufschluss über den Typ gibt. Dies ist notwendig um zur Laufzeit den Typ abfragen zu können.

```
package Planar_Objects is
      type Planar_Object is tagged
            record
                  X, Y: Float := 0.0;
            end record;
      function Distance (O: Planar_Object) return Float;
      procedure Move (O: in out Planar_Object; DX, DY: Float);
      procedure Draw (O: Planar_Object);
end Planar_Objects;
```

Eine *Function* ist eine ganz gewöhnliche Funktion die einen Parameter entgegen nimmt. Sie operiert nicht auf einem Objekt (ein *this* ist nicht verfügbar). Deswegen kann die Funktion *Distance* auch nur den Abstand vom 0 Punkt berechnen.

In Ada wird zwischen folgenden Arten von Parametern unterschieden:

- **in**: Kann von der aufgerufenen Routine nur gelesen werden
- **out**: Ist dafür da von der aufgerufenen Routine mit dem Ergebnis beschrieben zu werden. Er ist zu Beginn der Routine nicht initialisiert.
- **in out**: Ähnlich dem out Parameter. Allerdings wird in einem in out Parameter ein Wert mit übergeben, welcher für die aufgerufenen Routine relevant ist, von dieser aber geändert werden kann.

Extending Tagged Record in Ada

Nur von getaggten Typen kann man andere Typen *ableiten*. Dabei können zusätzliche Komponenten und primitive Operationen definiert werden. Der abgeleitete Typ wird *Erweiterung* des Vorgängertyps oder einfach *Typerweiterung* genannt. Jede Typerweiterung ist wieder ein getaggter Typ. Der Typ, von dem abgeleitet wird, heißt *Vatertyp*. Es gibt also nur *einfache Vererbung* und keine *mehrfache Vererbung*.

```
with Planar_Objects; use Planar_Objects;
package Special_Planar_Shapes is
      type Point in new Planar_Object with null record;
      procedure Draw (P: Point);
      type Circle is new Planar_Object with
            record
                  Radius: Float;
            end record;
      procedure Draw (C: Circle);

      type Rectangle is new Planar_Object with
            record
                  Length, Width: Float;
            end record;
      procedure Draw (T: Rectangle);
end Special_Planar_Shapes;
```

„*with null record*“ gibt an, dass der Record aus *Planar_Object* um nichts erweitert wird. Die Funktion *Draw* wird überschrieben. Wird keine neue Implementierung für *Draw* angegeben, wird die aus *Planar_Object* verwendet.
Point ist ein Untertyp von *Planar_Object* .
Circle erweitert *Planar_Object* um das Feld *Radius*.

Use of Tagged Records in Ada

Als erstes werden Objekte der vorher beschriebenen Records erzeugt. Dies geschieht ähnlich den Auto-Objecten in C++.

```
O1: Planar_Object;
O2: Planar_Object := (1.0, 1.0);
C: Circle := (3.0, 4.5, 6.7);
```

In Ada ist (oder war es zumindest) möglich, unter der Angabe der fehlenden Information, von einem Obertyp auf einen Untertyp zu casten.

```
O1 := Planar_Object(C);                         // type cast
C := (O2 with 6.7)
```

Planar_Object'Class ist ein klassenweiter Typ (können nur aus tagged records gebildet werden). Dieser umfasst den angegeben und alle davon abgeleiteten Typen. Der konkrete Typ von *O* ist demnach nicht bekannt (kann *Planar_Object*, *Point*, *Rectangle*, *Circle* sein)

```
procedure Process_Shapes (O: Planar_Object'Class) is
begin
      ... Draw(O); ...                          // dynamic binding
end Process_Shapes;
```

Draw kann in Process_Shapes aufgerufen werden, da genau bestimmt ist, dass für jeden von *Planar_Object* abgeleiteten Typen die Methode *Draw* definiert ist. Es wird dynamisch festgestellt, welche Implementierung von *Draw* aufgerufen werden muss.
Dynamisch gebunden wird dann, wenn der Typ des Arguments ein (Operation darf nicht von mehr als einem klassenweiten Typ abhängig sein) klassenweiter Typ ist und der Typ des Parameters kein klassenweiter Typ ist.
Ohne dynamisches Binden (also bei Verwendung eines nicht klassenweiten Typs als Parameter) würde die passende überladene Funktion (statisch) ausgewählt werden.

```
type Planar_Object_Ptr is access Planar_Object'Class;
```

Abstract Type in Ada

```
package Planan_Objects is
      type Planar_Object is abstract tagged null record;
      function Distance (O: Planar_Object) return Float
            is abstract;
      procedure Move (O: in out Planar_Object; DX, DY: Float)
            is abstract;
      procedure Draw (O: Planar_Object) is abstract;
end Planar_Objects;
```

Planar_Object ist hier als abstract markiert. Es können keine Instanzen davon erzeugt werden. Auch Funktionen können abstract sein.

Polymorphism in Ada

- Ada unterstützt Generizität (wurde praktisch von Ada erfunden) – Es kommen Typparameter zum Einsatz.
- Unterstützt inclusion polymorphismus, aber nur mit Einfachvererbung.
  Statisch und dynamsich aufgelöst (klassenweiter Typ, nicht klassenweiter Typ)
- Überladen von Funktionen als auch Operatoren
- In Ada gibt es nur explizite Typecasts (ein Mitgrund für die längliche Syntax).
- Typvergleiche können zur Laufzeit gemacht werden

Classes and Inheritance in Eiffel

Eiffel ist ebenfalls ein Urvater der Objekt orientierten Programmierung. Besonders Java hat viel Anleihe an Eiffel genommen.

```
class A feature
fnc(t: TI): TO is ... do ... end   -- fnc
end -- class A

class B inherit A redefine fnc feature
fnc(s: SI): SO is ... do ... end   -- fnc
-- supports co-variant parameter types, not type-safe
end -- class B

...
a: A;                              –- Zeiger auf Instanzen
b: B;
...
create b;                          -- old syntax: !!b;
a := b;                            -- dynamic binding
... a.fnc(x) ...
```

Eiffel unterstützt kovariante Eingangsparameter. Das Typprüfsystem kann allerdings nicht garantieren, dass dies nicht zu einem Laufzeitfehler führt. Zum Zeitpunkt der Entwicklung war diese Theorie allerdings noch nicht bekannt. Selbst als die Notwendigkeit von kontravarianten Eingangsparametern erkannt wurde, verzichtete Bertram Meyer bewusst auf diese umzusetzen (Funktionalität auf Kosten von Sicherheit).

Polymorphism in Eiffel

- Eiffel unterstützt Generizität (erste Sprache die Generizität im Zusammenhang mit Objekt orientierten Prinzipien unterstützt). Die Verwendung von kovarianten Eingangsparametern erleichtert dies erheblich.
- Inclusion Polymorphismus wird unterstützt und dynamisch aufgelöst, Expanded Objects (Ein Objekt welches in einem anderen inkludiert ist) statisch.
- Überladen von Funktionen, Prozeduren und Operatoren
- Statt eines Casts kommt in Eiffel der Zuweisungsversuch zum Einsatz:
  x ?= y (ACHTUNG! Alte Syntax) → Die Zuweisung erfolgt nur wenn der dynamische Typ von y ein Untertyp des deklarierten Typs von x ist.

Characteristics of Smalltalk

Smalltalk, entwickelt in den späten 70er Jahren, ist eine dynamisch typisierte Sprache und gleichzeitig ein weiterer Urvater der objektorientierten Programmierung. Klassen sind hier ebenfalls normale Objekte (mit Namen und Variablen, Methoden). Smalltalk unterscheidet sich von anderen Programmiersprachen insofern, dass der Programmierer keine Quellcode Files compiliert, sondern den bestehenden Smalltalk Code in einem Browser erweitert, oder verändert.
Zusammen mit Smalltalk erblickten auch Fenster und der Mauszeiger das Licht der Welt.

Variablen (immer nur Referenzen, auch bei primitiven Typen) haben in Smalltalk keinen deklarierten Typ. In einer Variable kann eine Referenz auf beliebiges gespeichert werden. Aus dieser Tatsache ergibt sich zugleich auch eine Form des Polymorphismus (es kann einer Variable alles zugewiesen werden → keine Notwendigkeit für Polymorphismus oder Generizität).

Smalltalk unterstützt aus Gründen der Einfachheit (Erklärung folgt) nur Einfachvererbung. Subtyping verliert in Smalltalk ebenfalls seine Relevanz, da jede Nachricht an jedes Objekt gesendet werden kann. Zuerst wird im adressierten Objekt nachgesehen ob es die Nachricht bearbeiten kann, wenn nicht wird in der Oberklasse nachgesehen. Wird in der Hierarchie niemand gefunden, der die Nachricht bearbeiten könnte, wird vom obersten Object die Nachricht „message not understood“ zurückgeschickt. Diese Suche erfolgt dynamisch, zur Laufzeit, da auch zur Laufzeit der Code geändert werden kann (neuer Obertyp zur Laufzeit). Bei Mehrfachvererbung müssten immer mehrere Pfade (nach oben) berücksichtigt werden und das Auffinden von Methoden wäre wesentlich komplizierter.
Diese dynamische Suche läuft durch mehrere Optimierungen performant ab. So merkt sich das System zum Beispiel einen erfolgreichen Aufruf zusammen mit der Information dass sich nichts geändert hat kann dieser schnell erneut erfolgen (dynamic caching vom Methoden).

Es gibt 3 Arten eine Methode aufzurufen:

Nr.	Deklaration	Aufruf	
1	`value`	`setZero value`	`Unary`
2	`> param`	`x > y`	`Binary`
3	`from: f to: t`	`s from: x to: y`	`keyword`

Bei der unären Form wird beim Methodenaufruf einfach die Nachricht *value* an *setZero* geschickt. Links steht immer der Empfänger der Nachricht, rechts die Nachricht an sich.
Bei binären Nachrichten steht zuerst immer ein Operatorsymbol und danach ein Operand. Der Aufruf ereignet sich in diesem Beispiel folgendermaßen: An das Objekt *x* wird die Nachricht > mit dem Argument *y* geschickt (deswegen binäre Nachrichten).
In Smalltalk gibt es keine Operatorprioritäten oder ähnliches, es wird immer von links nach rechts aufgelöst (dem kann natürlich durch Klammersetzung entgegengewirkt werden).
Zuletzt gibt es noch die sogenannten Keyword Nachrichten. In diesem Beispiel heißt die Methode

„from: f to: t“, was als ein Name zu verstehen ist. Nach jedem Doppelpunkt in diesem Namen steht ein Parameter. *f* und *t* sind hierbei die beiden formalen Parameter.
Objekte werden in Smalltalk explizit erzeugt (myPoint <- point new) wobei „<-“ der Zuweisungsoperator ist. An das Objekt *point* (welches eine Klasse sein sollte, aber auf jeden Fall die Nachricht *new* verstehen muss) wird die Nachricht *new* gesendet. Das Ergebnis dieser Auswertung ist (wahrscheinlich) ein neues Objekt, welches an *myPoint* zugewiesen wird.
Um Speicherverwaltung muss sich der Programmierer nicht kümmern, Speicherfreigabe erledigt der Garbage Collector (kam erstmals in Lisp zum Einsatz, also eine eher alte Technik).

Beispiele in Smalltalk

```
[x <- 0. y <- 0] value
```
Der Teil in eckigen Klammern wird als Block bezeichnet. Ein Block ist selbst ein Objekt. In diesem Beispiel wird an das Objekt *Block* die Nachricht *value* (von evaluate = auswerten) geschickt.
Der obige Block enthält 2 Anweisungen. Der Punkt hat in Smalltalk dieselbe Bedeutung wie der Strichpunkt in z.B. C. Er trennt Statements.
Die Zuweisungen im Block werden erst ausgeführt, wenn die Nachricht *value* empfangen wurde.

```
setZero <- [x <- 0. y <- 0].
```
An die Variable *setZero* wird der obige Block zugewiesen.
```
setZero value
```
Danach wird an *setZero value* gesendet, was zur Zuweisung von 0 an *x* und *y* führt.

```
x > y
ifTrue: [max <- x]
ifFalse: [max <- y]
```
An x wird die Nachricht > mit dem Argument y geschickt. Diese Auswertung (erfolgt im Objekt *x*) gibt einen Boolean zurück.
An den retournierten Boolean wird die Nachricht *ifTrue: ifFalse:* (Nachricht mit 2 Argumenten, wobei diese jeweils Blöcke sind) geschickt. Der Boolean schickt je nach seinem Wahrheitswert entweder an den Block aus *ifTrue* oder an den Block aus *ifFlase* die Nachricht *value*.

```
10 timesRepeat: [x <- x + a]
```
An *10* wird die Nachricht timesRepeat mit dem Block *[x <- x + a]* als Argument geschickt.10 wird daraufhin *value* an den Block schicken. Daraufhin wird das Objekt welches sich aus *10* ergibt, nachdem es sich um 1 dekrementiert erneut die Nachricht *value* an den Block schicken. Dies wird fortgeführt, bis 0 erreicht wird.

```
[x < b] whileTrue: [x <- x + a]
```
An den Block *[x < b]* wird die Nachricht *whileTrue* mit dem Argument *[x <- x + a]* geschickt. Der Block wird sich hier selbst auswerten. Falls das Ergebnis *true* ist an das Argument die Nachricht *value* schicken. Dies wird wiederholt, bis der Block zu *false* evaluiert.

Polymorphism in Smalltalk

Generizität: Nachdem jede Variable alles enthalten kann wird Generizität zu einem sinnlosen Begriff. Es existiert zwar keine explizite Generizität, doch durch die dynamische Typisierung ergibt sich alles was man von Generizität erwartet von selbst.

Inclusion Polymorphismus: Ist vorhanden, man kann sich jede nur denkbare Hierarchie aufbauen (wird nicht überprüft).
Beispielsweise kann die Untertypbeziehung *Person* ist Obertyp von *Student* problemlos erstellt werden. Wenn eine Instanz von *Student* vorliegt kann sicher gesagt werden, dass diese auch alle Nachrichten versteht, welche von *Person* verstanden werden

Überladen kann in Smalltalk nicht existieren: Überladen entscheidet anhand des deklarierten Typs welche Routine aufgerufen wird. In Smalltalk gibt es jedoch keine deklarierten Typen.

Casts auf Referenztypen sind auch nicht explizit möglich. Bei jeder Zuweisung wird implizit gecastet.

Typvergleiche zur Laufzeit sind natürlich möglich.

Characteristics of Java

In Java werden alle Objekte auf dem Heap allokiert. Zur Deallokierung wird ein Garbage Collector verwendet.

Programme werden auf portablen JVM (Java Virtual Machine) Code übersetzt. Dies kommt von der ursprünglichen Idee von Java als Netzwerk-Sprache. Übersetzter Code sollte übertragen werden, obwohl die Zielmaschine (und deren OS) nicht bekannt sind.

Java unterstützt Einfachvererbung auf Klassen- und Mehrfachvererbung auf Interface-Basis. Die Einführung von Interfaces war damals eine große Innovation. Auf konventionelle Mehrfachvererbung (wie in C++) wurde aufgrund von Performance-Vorteilen verzichtet.

Variablen und Methoden gehören zu Objekten oder Klassen.

In Java gibt es Packages. In C++ Namespaces.

Unterstützung von final Klassen und final Methoden.

In Java werden viele Methoden (equals, clone,..) bereits von *Object* angeboten, welches dadurch bereits relativ „groß“ ist. Dies war ursprünglich einer der Unterscheidungspunkte zu C#.

Classes and Inheritance in Java

```
abstract class PlanarObject {
      protected float x, y;
      public float distance() { ... }
      public void move(float x, y) { ... }
      public abstract void draw();
}

class Circle extends PlanarObject {
      private float radius;
      public void draw() { ... }
}

class Rectangle extends PlanarObject {
      private float length, width;
      public void draw() { ... }
}
```

Interfaces in Java

```
interface Dictionary {
      void insert(String c; int i);
      int lookup(String c);
}

class ArrayDict implements Dictionary {
      private String[] names;
      private int[] values;
      private int size = 0;
      public void insert(String c; int i) { ... }
      public int lookup(String c) { ... }
}

class ListDict extends SomeList implements Dictionary { ... }
```

Polymorphism in Java

Genericity including bounded genericity since version 1.5
— statically resolved

inclusion polymorphism supported by single/multiple inheritance
— dynamically resolved (statically when private, final or static)

overloading of routines, but no operators
— statically resolved

checked casts, type comparison at run-time
— semantic actions at run-time