```
class Node {
  Object value;
  Node next;
}
```

```
class List {
  Node head;
  void add(Object v) {
    ...
  }
  Object get();
}
```

Probleme

- Typumwandlung nötig

```
List.add(3);
int x = (Integer) List.get();   // Boxing kostet Zeit
                                // Typumwandlung und Unboxing kosten Zeit
```

- Homogenität kann nicht erzwungen werden

```
List.add(3); List.add(new Person);
Person p = (Person) List.get();   // kann zu Laufzeitfehler führen
```

- Spezielle Typen IntList, PersonList, ... führen zu Redundanz

beliebiger Buchstabe ⇒ Platzhalter

```
class Node <T>
  T value;
  Node<T> next;
}
```

```
class List <T> {
  Node <T> head;
  void add(T value) {
    ....
  }
  T get() {
    ....
  }
```

- geht auch für Interfaces

```
interface I<T>
```

- Platzhalter T kann wie normaler Typ verwendet werden

Funktion:

```
List<Integer> a = new List<Integer>(100);
a.add(3);          // nur Integer-Parameter erlaubt; hier Boxing
int i = a.remove(); // liefert Integer-Wert; kein Cast nötig
```

aktueller Typparameter muss ein Referenztyp sein (Integer, Rectangle, ...)

Generischer Typ

```
class List<T> {
   T[] data;
   List(int size) {...}
   void add(T x) {...}
   T remove() {...}
}
```

wird übersetzt zu →

Rohtyp

```
class List<T> {
   Object[] data;
   List(int size) {...}
   void add(Object x) {...}
   Object remove() {...}
}
```

Alle aus *List<T>* erzeugten Typen werden auf denselben Rohtyp abgebildet

```
List<String> stringList = new List<String>(10);
stringList.add("abc");
List<Integer> intList = new List<Integer>(100);
intList.add(15);
int x = intList.remove();
```

rufen beide *add* von *List* auf und übergeben einen *Object*-Parameter.

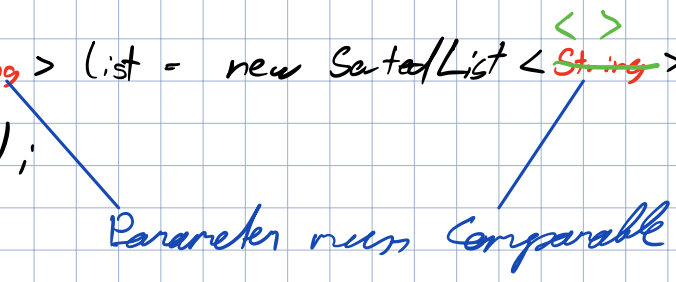
Compiler stellt aber sicher, dass *intList* nur mit *Integer*-Werten verwendet wird.

Compiler fügt Type Cast *(Integer)* ein

```
class SortedList<T extends Comparable> {
   T[] data;
   int nElements = 0;
   ....
   void add(T elem) {
      int i = nElements - 1;
      while (i >= 0 && elem.compareTo(data[i]) > 0) {
         data[i+1] = data[i];
         i--;
      }
      data[i+1] = elem;
      nElements++;
   }
}
```

Type-Constraint

```
SortedList<String> list = new SortedList<String>();
list.add("John");
```



Generische Arrays:

verboten!

```
class List<E, P extends Comparable> {
  E[] data = new E[10];
  P[] prio = new P[10];
  ....
}
```

Warum?

⇒ Java hat keine Ahnung was hinter E, P steckt.

Man darf weder ein Array von Typparametern erzeugen

```
new E[10]
new P[10]
```

noch ein Array aus konkretisierten Typen

```
new List<String, Integer>[10]
```

so funktioniert's

```
class List<E, P extends Comparable> {
  E[] data = (E[]) new Object[10];
  P[] prio = (P[]) new Comparable[10];
  ....
}
```

Compiler erzeugt trotzdem eine Warnung

Vererbung

- von einer gewöhnlichen Klasse — `class B<x> extends A {...}`
- von einer konkretisierten generischen Klasse — `class B<x> extends A<String> {...}`
- von einer generischen Klasse mit gleichem Platzhalter — `class B<x> extends A<x> {...}`

Wenn von konkretisierter Klasse geerbt

```
class MyList extends List<Integer> {
    ...
    void add(Integer x) {...}
}
```

T wird durch konkreten Typ Integer ersetzt

Wenn von generischer Klasse geerbt

```
class MyList<T> extends List<T> {
    ...
    void add(T x) {}
}
```

T bleibt als Platzhalter

Folgendes geht nicht (konkrete Klasse darf nicht von generischer Klasse erben)

~~`class MyList extends List<T> {`~~
`    void add(T x) {...}`
~~`}`~~