

# **CSC 347 - Concepts of Programming Languages**

## **Parametric Polymorphism**

Instructor: Stefan Mitsch



## Learning Objectives

- ❓ Inheritance for container classes?
  - Compare inheritance and type parameters



# Monomorphic Linked Lists (C)

- Implement a linked list

```
1 typedef struct Node Node;
2 struct Node {
3     int* item;
4     Node* next;
5 };
6
7 int* get_last (Node* xs) {
8     while (xs->next != NULL) {
9         xs = xs->next;
10    }
11    return xs->item;
12 }
```

- ❗ Code duplication

```
1 typedef struct FloatNode FloatNode;
2 struct FloatNode {
3     float* item;
4     FloatNode* next;
5 };
6
7 float* get_last (FloatNode* xs) {
8     while (xs->next != NULL) {
9         xs = xs->next;
10    }
11    return xs->item;
12 }
```

- ❓ Benefits? Downsides?



# Generic Linked Lists (C)

- Use type `(void*)` in C

```
1 typedef struct Node Node;
2 struct Node {
3     void* item;
4     Node* next;
5 };
6
7 void* get_last (Node* xs) {
8     while (xs->next != NULL) {
9         xs = xs->next;
10    }
11    return xs->item;
12 }
```

- ! No static protection against casts

```
1 int main () {
2     int p      = (int*) malloc (sizeof(int));
3     *p         = 2123456789;
4     Node* xs   = (Node*) malloc (sizeof(Node));
5     xs->next    = NULL;
6     xs->item    = p;           // store pointer
7     double* q  = get_last(xs); // alias of p
8     printf ("q=%f\n", *q);    // unsafe access
9 }
```

```
1 $ clang -m32 parametric-03.c && ./a.out
2 q=96621069057346178268049192388430659584.000000
```

- ? Benefits? Downsides?



# Generic Linked Lists (Java)

- Generic list using subtype polymorphism

```
1 static class Node {
2     Object item;
3     Node next;
4 }
5
6 static Object getLast (Node xs) {
7     while (xs.next != null) {
8         xs = xs.next;
9     }
10    return xs.item;
11 }
```

`ClassCastException` at runtime, but no unsafe memory access

```
1 public static void main (String[] args) {
2     Integer p = Integer.valueOf(2123456789);
3     Node xs = new Node();
4     xs.next = null;
5     xs.item = p; // store Integer
6     Double q = (Double) getLast(xs); // ClassCastException
7     System.out.printf ("d=%f\n", q); // unsafe access
8 }
```

```
1 $ javac Parametric2.java
2 $ java Parametric2
3 java.lang.ClassCastException: Integer cannot be cast to Double
```

❓ Benefits? Downsides?



# Can Inheritance Solve This?

- Container base class

```
1 abstract class Node {
2     Object item;
3     Node next;
4     Object getLast() {
5         Node xs = this;
6         while (xs.next != null) xs = xs.next;
7         return xs.item;
8     }
9 }
```

- Implementations reuse `getLast`

```
1 class IntNode extends Node {
2     int getLastItem() {
3         return (int)getLast(); // cast
4     }
5 }
```

❗ static casts shifted into library, but problem not solved

❓ How to adapt a class?

💡 Need type parameters

```
1 class Node {
2     ??? item;
3     Node next;
4     ??? getLast() {
5         Node xs = this;
6         while (xs.next != null) xs = xs.next;
7         return xs.item;
8     }
9 }
```

❓ Benefits? Downsides?



# Generic Linked Lists (Java)

- Generic list using parametric polymorphism

```
1 class Node<X> {
2     X      item;
3     Node<X> next;
4     X getLast () {
5         Node<X> xs = this;
6         while (xs.next != null) xs = xs.next;
7         return xs.item;
8     }
9 }
```

- Create a list of Integer

```
1 public static void main (String[] args) {
2     Node<Integer> xs = new Node<>();
3     xs.next      = null;
4     xs.item      = Integer.valueOf(37);
5 }
```

- Incompatible read: compile error

```
1 Double q = (Double) getLast(xs); // compiler error
2 System.out.printf ("d=%f\n", q); // unsafe access
3 }
```

```
1 $ javac Parametric1.java
2 error: incompatible types: Integer cannot be converted to Double
3     Double q = (Double) getLast(xs);    // compiler error
4                                ^
```



# Generic Linked Lists (Scala)

## Java

```
1 class Node<X> {
2     X item;
3     Node<X> next;
4     X getLast () {
5         Node<X> xs = this;
6         while (xs.next != null)
7             xs = xs.next;
8         return xs.item;
9     }
10 }
```

## Scala (imperative)

```
1 class Node[X](val item: X, val next: Node[X]):
2     def getLast () : X =
3         var xs = this
4         while xs.next != null do
5             xs = xs.next
6         xs.item
7     end getLast
8 end Node
```

## Scala (ADT and recursive)

```
1 enum List[+X]:
2     case Nil
3     case Node(item: X, next: List[X])
4
5     def lastOption : Option[X] = this match
6         case Nil => None
7         case Node(item, Nil) => Some(item)
8         case Node(_, next) => next.lastOption
9     end lastOption
10
11     def last : X = lastOption.getOrElse(throw ...)
12 end List
```



# Java Type Parameter Erasure

- Java type parameters are not part of runtime type information

```
1 static class ArrayList<X> {  
2     X[] a;  
3  
4     ArrayList(int n) {  
5         a = new X[n];  
6     }  
7  
8     void put (int i, X item) { a[i] = item; }  
9     X       get (int i)      { return a[i]; }  
10 }
```

```
1 $ javac Parametric3.java  
2 error: generic array creation  
3     a = new X[n];  
4             ^
```



# Java Type Parameter Erasure

- Cast is not checked at runtime

```
1 static class ArrayList<X> {  
2     X[] a;  
3  
4     ArrayList(int n) {  
5         a = (X[]) new Object[n];  
6     }  
7  
8     void put (int i, X item) { a[i] = item; }  
9     X      get (int i)      { return a[i]; }  
10 }
```

```
1 $ javac -Xlint:unchecked Parametric4.java  
2 warning: [unchecked] unchecked cast  
3     a = (X[]) new Object[n];  
4           ^
```

❓ Why is cast not checked at runtime?



# Java Type Parameter Erasure

- ? Why is `@SuppressWarnings` ok?
- ? What are the entries of `a` before any item is placed in it?

```
1 static class ArrayList<X> {  
2     X[] a;  
3  
4     @SuppressWarnings("unchecked")  
5     ArrayList(int n) {  
6         a = (X[]) new Object[n];  
7     }  
8  
9     void put (int i, X item) { a[i] = item; }  
10    X      get (int i)      { return a[i]; }  
11 }
```

```
1 $ javac -Xlint:unchecked Parametric4.java
```

- Okay to ignore, since all references in array are `null` (can be assigned to any reference type)



# Java Type Parameter Erasure

- Not possible to check casts when writing to unparameterized list

```
1 ArrayList<String> ss = new ArrayList<>(10);
2 ArrayList os = ss;
3 os.put (1, 2123456789);
4
5 // ClassCastException when reading
6 String s = ss.get (1);
```

```
1 $ javac Parametric6.java
2 Note: Parametric6.java uses unchecked or unsafe operations.
3 Note: Recompile with -Xlint:unchecked for details.
```

```
1 $ java Parametric6
2 ClassCastException: Integer cannot be cast to class String
3   at Parametric.main(Parametric6.java:6)
```



## Arrays Checked When Assigned

- Java stores types with arrays

```
1 String[] ss = new String[10];
2 Object[] os = ss;
3
4 // ArrayStoreException when writing
5 os[1] = 2123456789;
6 String s = ss[1];
```

```
1 $ javac Parametric7.java
2 // no warnings
```

```
1 $ java Parametric7
2 ArrayStoreException: Integer
3     at Parametric.main(Parametric7.java:5)
```



# C++ Templates

- C++ class templates are instantiated with concrete types
- Compiler checks that types fit to all used operations

```
1 template <class T> class Node {
2     public:
3         T item;
4         Node<T>* next;
5         T getLast() {
6             Node<T>* c = this;
7             while (c->next != nullptr)
8                 c = c->next;
9             return c->item;
10        }
11 }
```

```
1 int main(int argc, char* argv[]) {
2     Node<int>* n = new Node<int>();
3     n->item = 5;
4     n->next = new Node<int>();
5     n->next->item = 6;
6     cout
7         << "Last item: "
8         << n->getLast() << endl;
9 }
```



# C++ Templates Challenge

! Problems in template not noticed until instantiated

```
1 template <class T> class Node {
2     public:
3         T item;
4         Node<T>* next;
5         // ...
6         void printCout() {
7             Node<T>* c = this;
8             cout << c->item;
9             while (c->next != nullptr) {
10                 c = c->next;
11                 cout << "," << c->item;
12             }
13             cout << endl;
14         }
15 }
```

- Works fine

```
1 int main(int argc, char* argv[]) {
2     Node<int>* n = new Node<int>();
3     n->item = 5;
4     n->printCout();
5 }
```

```
1 typedef struct { int a; int b; } Pair;
2
3 int main(int argc, char* argv[]) {
4     Node<Pair>* n = new Node<Pair>();
5     n->item = (Pair) { .a=2, .b=3 };
6     n->printCout();
7 }
```

- Compile error: cannot print `Pair`



# Benefits of Parametric Polymorphism

- Type parameters constrain possible implementations

```
1 def f[T](xs: Seq[T]) : Int
```

- Only 1 sensible implementation: length of sequence

```
1 def f(xs: Seq[Int]) : Int
```

- Many possible implementations: length of sequence, minimum, first element, median, sum, ...



# Given Types

- Define ordering of elements

```
1 trait Ord[T]:  
2   def compare(x: T, y: T): Int  
3 end Ord
```

## Subtyping polymorphism

- Use as types in arguments

```
1 def max[T](x: Ord[T], y: Ord[T]): Ord[T] =  
2   if x.compare(y) > 0 then x else y
```

- Add to classes statically

```
1 class MyInt extends Ord[MyInt]
```

## Parametric polymorphism

- Define type bounds

```
1 def max(x: T, y: T)(using ord: Ord[T]): T =  
2   if ord.compare(x,y) > 0 then x else y
```

- Use `given` instance of type `Ord[T]`

```
1 given Ord[Int]:  
2   def compare(x: Int, y: Int): Int = x - y
```



## Summoning Types

- Extend ordering on elements to ordering on containers (requires Scala 3.6 or newer)

```
1 given [T: Ord] => Ord[List[T]]:  
2 // def listOrd[T](using ordT: Ord[T]): Ord[List[T]] = new Ord[List[T]]:  
3   def compare(xs: List[T], ys: List[T]): Int =  
4     (xs, ys) match  
5       case (Nil, Nil) => 0  
6       case (Nil, _)  => -1  
7       case (_, Nil)  => 1  
8       case (x :: xtail, y :: ytail) =>  
9         val firstComp = summon[Ord[T]].compare(x, y) // ordT.compare(x, y)  
10        if firstComp != 0 then firstComp  
11        else compare(xtail, ytail)
```



# Summoning Types

- Find the maximum among a sequence of orderable items

```
1 def maximum[T : Ord](xs: Seq[T]): T =  
2   xs.reduceLeft(max) // forwards the type bound to max
```

- Define a descending order based on an ascending order ( `using` is explicit named type bound)

```
1 def descending[T](using asc: Ord[T]): Ord[T] = new Ord[T]:  
2   def compare(x: T, y: T): Int = asc.compare(y, x)
```

- Find the minimum among a sequence of orderable items

```
1 def minimum[T : Ord](xs: List[T]): T =  
2   maximum(xs)(using descending) // pass descending for the type bound of maximum
```



# Summary: Parametric Polymorphism

## Type Parameters

- Independence between container class and types of contained objects
- Type safety for reading and writing
- Not all languages retain type parameters at runtime (Java: compiler knows generics, JVM does not)
- Functional languages use type parameters to shift "code correctness" to the compiler

## C++ templates

- Template itself has no executable form: `List<T>`
- Executable generated for each instantiation: `List<int>`, `List<string>`
- Compile problems hidden until template gets instantiated