CSC 347 - Concepts of Programming Languages

Parametric Polymorphism

Instructor: James Riely

Learning Objectives

❓ Inheritance for container classes?

- Compare inheritance and type parameters

Monomorphic Linked Lists (C)

- Implement a linked list

```c
typedef struct Node Node;
struct Node {
  int*  item;
  Node* next;
};

int* get_last (Node* xs) {
  while (xs->next != NULL) {
    xs = xs->next;
  }
  return xs->item;
}
```

❓ Benefits? Downsides?

❗ Code duplication

```c
typedef struct FloatNode FloatNode;
struct FloatNode {
  float*  item;
  FloatNode* next;
};

float* get_last (FloatNode* xs) {
  while (xs->next != NULL) {
    xs = xs->next;
  }
  return xs->item;
}
```

Generic Linked Lists (C)

- Use type `(void*)` in C

```c
typedef struct Node Node;
struct Node {
  void* item;
  Node* next;
};

void* get_last (Node* xs) {
  while (xs->next != NULL) {
    xs = xs->next;
  }
  return xs->item;
}
```

❓ Benefits? Downsides?

❗ No static protection against casts

```c
int main () {
  int p     = (int*) malloc (sizeof(int));
  *p        = 2123456789;
  Node* xs  = (Node*) malloc (sizeof(Node));
  xs->next  = NULL;
  xs->item  = p;                // store pointer
  double* q = get_last(xs);     // alias of p
  printf ("q=%f\n", *q);        // unsafe access
}
```

```
$ clang -m32 parametric-03.c && ./a.out
q=9662106905734617826804919238843065958 4.000000
```

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 }
```

❓ Benefits? Downsides?

❗ 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 }
```

Generic Linked Lists (Java)

- Generic list using parametric polymorphism

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

- 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);
```

- 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

```java
class Node<X> {
  X       item;
  Node<X> next;
  X getLast () {
    Node<X> xs = this;
    while (xs.next != null)
      xs = xs.next;
    return xs.item;
  }
}
```

Scala (imperative)

```scala
class Node[X]
    (val item: X, val next: Node[X]):
  def getLast () : X =
    var xs = this
    while xs.next != null do
      xs = xs.next
    xs.item
  end getLast
end Node
```

Scala (ADT and recursive)

```scala
enum List[+X]:
  case Nil
  case Node(item: X, next: List[X])

  def lastOption: Option[X] = this match
    case Nil              => None
    case Node(item, Nil) => Some(item)
    case Node(_, next)    => next.lastOption
  end lastOption

  def last: X = lastOption.getOrElse(throw ...)
end List
```

Java Type Parameter Erasure

- Scala retains type parameters at runtime (`ClassTag`)

```scala
class ArrayList[X: scala.reflect.ClassTag](val size: Int):
  private val a = new Array[X](size)

  def put(i: Int, item: X) : Unit = a(i) = item
  def get(i: Int)          : X    = a(i)
end ArrayList
```

- Java type parameters are not part of runtime type information

```java
class ArrayList<X> {
  private X[] a;

  public ArrayList(int n) { a = new X[n]; }

  public void put (int i, X item) { a[i] = item; }
  public X    get (int i)         { return a[i]; }
}
```

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

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 `@SupressWarnings` 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

- Generality: one implementation fits many concrete uses
- Specificity: 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, ...

Implicit Arguments and Given Instances

- Define ordering of elements

Subtyping polymorphism

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

- 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

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

- 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 Instances

- 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
```

Summoning Instances

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

```scala
given [T: Ord] => Ord[List[T]]:
// def listOrd[T](using ordT: Ord[T]): Ord[List[T]] = new Ord[List[T]]:
  def compare(xs: List[T], ys: List[T]): Int =
    (xs, ys) match
      case (Nil, Nil) => 0
      case (Nil, _)   => -1
      case (_, Nil)   => 1
      case (x :: xtail, y :: ytail) =>
        val firstComp = summon[Ord[T]].compare(x, y) // ordT.compare(x, y)
        if firstComp != 0 then firstComp
        else compare(xtail, ytail)
```

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