# CSC 447 - Concepts of Programming Languages

## Parametric Polymorphism

Instructor: Eric J. Fredericks

# 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;
}
```

❗ 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;
}
```

❓ Benefits? Downsides?

# 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=96621069057346178268049192388430659584.000000
```

# Generic Linked Lists (Java)

- Generic list using subtype polymorphism

```
static class Node {
  Object  item;
  Node    next;
}

static Object getLast (Node xs) {
  while (xs.next != null) {
    xs = xs.next;
  }
  return xs.item;
}
```

Benefits? Downsides?

`ClassCastException` at runtime, but no unsafe memory access

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

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

# Can Inheritance Solve This?

- Container base class

```
abstract class Node {
  Object item;
  Node next;
  Object getLast() {
    Node xs = this;
    while (xs.next != null) xs = xs.next;
    return xs.item;
  }
}
```

- Implementations reuse `getLast`

```
class IntNode extends Node {
  int getLastItem() {
    return (int)getLast(); // cast
  }
}
```

❓ Benefits? Downsides?

❗ static casts shifted into library, but problem not solved

❓ How to adapt a class?

- 💡 Need type parameters

```
class Node {
  ??? item;
  Node next;
  ??? getLast() {
    Node xs = this;
    while (xs.next != null) xs = xs.next;
    return xs.item;
  }
}
```

# Generic Linked Lists (Java)

- Generic list using parametric polymorphism

```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;
  }
}
```

- Create a list of `Integer`

```java
public static void main (String[] args) {
  Node<Integer> xs = new Node<>();
  xs.next    = null;
  xs.item    = Integer.valueOf(37);
```

- Incompatible read: compile error

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

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

# Generic Linked Lists (Scala)

- Generic list using parametric polymorphism

```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
```

- Recursive

```scala
class Node[X](val item: X, val next: Node[X]):
  def getLast () : X =
    if next == null then item
    else next.getLast ()
```

# Java Type Parameter Erasure

- Java type parameters are not part of runtime type information

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

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

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

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

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

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

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

? 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?

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

  @SuppressWarnings("unchecked")
  ArrayList(int n) {
    a = (X[]) new Object[n];
  }

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

```
$ 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

```
ArrayList<String> ss = new ArrayList<>(10);
ArrayList os = ss;
os.put (1, 2123456789);

// ClassCastException when reading
String s = ss.get (1);
```

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

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

# Arrays Checked When Assigned

- Java stores types with arrays

```java
String[] ss = new String[10];
Object[] os = ss;

// ArrayStoreException when writing
os[1] = 2123456789;
String s = ss[1];
```

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

```
$ java Parametric7
ArrayStoreException: Integer
    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

```cpp
template <class T> class Node {
  public:
    T item;
    Node<T>* next;
    T getLast() {
      Node<T>* c = this;
      while (c->next != nullptr)
        c = c->next;
      return c->item;
    }
}
```

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

# C++ Templates Challenge

Problems in template not noticed until instantiated

```
template <class T> class Node {
  public:
    T item;
    Node<T>* next;
    // ...
    void printCout() {
      Node<T>* c = this;
      cout << c->item;
      while (c->next != nullptr) {
        c = c->next;
        cout << "," << c->item;
      }
      cout << endl;
    }
}
```

- Works fine

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

```
typedef struct { int a; int b; } Pair;

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

- Compile error: cannot print `Pair`

# Summary

- Parametric polymorphism allows independence between container class and types of contained objects
- Type safety when reading from and writing to containers
- Not all languages retain type parameters at runtime (Java: compiler knows generics, JVM does not)
- C++ templates are different
  - Template itself has no executable form: `List<T>`
  - Executable generated for each instantiation: `List<int>`, `List<string>`
  - Compile problems hidden until template gets instantiated