Types

CMPSC 461
Programming Language Concepts
Penn State University
Fall 2016

Function performs an operation for different values ***of the same type***

```
class IntArrayList {
  …
  boolean add (int i);
  int remove(int idx);
}
```

```
class DoubleArrayList{
  …
  boolean add (double d);
  double remove(int idx);
}
```

(Polymorphic) Functions takes values of different types

Java used to do this (before Java 1.5):

```
class ArrayList {
  …
  boolean add (Object o);
  Object remove (int idx);
}
```

Does it work?

```
List langList = new ArrayList();
langList.add(new Language("Java"));
Language java = (Language) langList.remove(0);
langList.add(new System("Linux")); // Whoops!
Language c = (Language) langList.remove(0);// Error
```

Compiler doesn't know langList should only contain values of type Language

Generics

A facility added to Java 5.0, which allows “a type or method to operate on objects of various types while providing compile-time type safety.”

Not an essentially object-oriented idea

Not originated in Java

Polymorphism: function with multiple forms

Parametric polymorphism (fun. with a set of types)

- Explicit para. polymorphism (Generics, aka. Templates in C++)

Java

```
class ArrayList<T> {
  …
  boolean add (T o);
  T remove (int idx);
}
```

C++

```
template <class T>
T GetMax (T a, T b) {
  T result;
  result = (a>b)? a : b;
  return (result);
}
```

Polymorphism

Parametric polymorphism (fun. with a set of types)

- Explicit para. polymorphism
- Implicit para. polymorphism (Lisp, Scheme, ML)

```
(define (min a b) (if (< a b) a b))
```

No mention of type

In Haskell

```
min a b = if a < b then a else b
```

Type Parameterization

```
ArrayList<Integer> intLst = new ArrayList<Integer>();
ArrayList<String>  strLst = new ArrayList<String>();
```

ArrayList<Integer>

- an application of a type-level function (ArrayList)
- to the type parameter Integer
- gives an array list of integers

The idea of generics is to allow user-defined parameterized types

User-Defined Parameterized Type

Formal type parameter

Java

```
class ArrayList<T> {
  …
  boolean add (T o);
  T remove (int idx);
}
```

Formal type parameter

C++

```
template <class T>
class ArrayList {
  …
  bool add (T o);
  T remove (int idx);
}
```

Intuitively, we defined a type-level function

Generic Functions

Java

Formal type parameter

```
T max<T> (T a, T b) {
  T result;
  result = (a.compareTo(b)
            >=0)? a : b;
  return (result);
}
```

C++

Formal type parameter

```
template <class T>
T GetMax (T a, T b) {
  T result;
  result = (a>b)? a : b;
  return (result);
}
```

Intuitively, a function that takes a type, and parameters

Generics Enforce Type-Safety

```
List<Language> langList = new ArrayList<Language>();
langList.add(new Language("Java")); // OK
Language java = (Language) langList.remove(0);
langList.add(new System("Linux")); // Type error!
Language c = (Language) langList.remove(0);
```

Compiler knows langList only contain values of type Language (no run-time errors)

Implementing Generics

Static mechanism (Ada, C++)

- Compiler generates separate copy for every unique instance
- Each copy is type-checked separately

```
template <class T>
T GetMax (T a, T b) {
  T result;
  result = (a>b)? a : b;
  return (result);
}
int i = GetMax(1,2);
char c = GetMax('c','e');
```

Defined by programmer

```
int GetMax (int a,int b) {
  int result;
  …
}
char GetMax (char a, char b) {
  char result;
  …
}
int i = GetMax(1,2);
char c = GetMax('c','e');
```

Generated/Checked by the compiler

Implementing Generics

Dynamic mechanism (aka. type erasure in Java)

- Type system checks type safety
- All instances share one code, without generics!

Type Erasure in Java

```
class ArrayList<T> {
  …
  boolean add (T o) {…};
  T remove (int idx) {…};
}
ArrayList<Language> langLst = …;
langLst.add(new Language("Java"));
Language java=langLst.remove(0);
```

Defined by programmer

```
class ArrayList {
  …
  boolean add (Object o) {…};
  Object remove (int idx) {…};
}
ArrayList langLst = …;
langLst.add(new Language("Java"));
Language java=
  (Language)langLst.remove(0);
```

Executed by JVM

Type system ensures all dynamic casts are safe

Pros and Cons of Type Erasure

Pros

- Backward compatibility
- One code copy shared by all instances

Cons

- Cannot use type parameter at run time

```
class Example<T> {
  void method(Object item) {
    if (item instanceof T) { … } // cannot compare to T
    T anotherItem = new T(); // cannot use constructor
    T[] itemArray = new T[10]; // cannot create T[10]
}}
```