Ca' Foscari University of Venice

Generic types

Object oriented programming, module 1

Pietro Ferrara

pietro.ferrara@unive.it

Types as parameters

- We have used vectors many times so far
- When we introduced, we briefly said that a vector is parametrized on a type
  - That is, the type of elements contained in the vector
  - Technically speaking, a template in C++
- We also saw many implementation of lists
  - Singly linked, doubly linked, singular, with tail, …
  - All specific for a type (int or string)
- What about generalizing this implementation?
  - How can we parametrize our implementation on the type of elements of the list?

```
#include <vector>

int main() {
 vector<Token> tok = vector<Token>;
 double number = 0;
 char op = 0;
 cin >> number >> op;
 while(op!='=') {
  tok.push_back(Token{true, number});
  tok.push_back(Token{false, op});
  cin >> number >> op;
 }
 tok.push_back(Token{true, number});
}
```

Generic programming

Generic programming: Writing code that works with a variety of types presented as arguments, as long as those argument types meet specific syntactic and semantic requirements.

- For instance, when I create a vector I specify that
  - I can put only elements of the given type
  - When I retrieve an element, I get an element of the given type
- We can specify templates over classes or methods
- Sometimes called also parametric polymorphism

polymorphism is the provision of a single interface to entities of different types or the use of a single symbol to represent multiple different types (Wikipedia)

```
template<typename T>

class List{

 private:
  struct Cella{
   T info;
   Cella* next;
  };
  Cella* l;    //head
  Cella* last;  //tail
}

template<typename T1>
T1 identity(T1 par) {
 return par;
}
```

From templates to class instances

- Intuitively, when we pass a type to a template the compiler
  - Compiler the generic code with the given type
  - Substitute this new type with the original generic type
- In this way, we can assign a vector<int> to a vector<int>
  - But not a vector<string> or a vector<double> to a vector<int>!

```
List<int> list = List<int>;
```

```
template<typename T>
class List{
 private:
  struct Cella{
   T info;
   Cella* next;
  };
  Cella* l;    //head
  Cella* last;  //tail
}
```

```
class List_int{
 private:
  struct Cella{
   int info;
   Cella* next;
  };
  Cella* l;    //head
  Cella* last;  //tail
}
```

```
List_int list = List_int;
```

Ca' Foscari University of Venice

From C++ templates to Java generics

- Java supports generic types
  - Same purpose of C++ templates
  - Different runtime approach
- Do not create a new class each time a class with generic is instantiated
- Introduced in Java 1.5
- Replaced by runtime checks and casts
- Indeed, a very simple concept:

A generic type is a generic class or interface that is parameterized over types.
https://docs.oracle.com/javase/tutorial/java/generics/types.html

```
public class HashMap<K,V> {
  public V get(K key) {...}
  public V put(K key, V value) {...}
}

public class FuelTypeCache {
  HashMap<String, FuelType> map = ...;
  FuelType getFuelTypeFromName(String n) {
     return map.get(n);
  }
}
```

Generics as parameters

- A generics can be seen as a parameter
  - Passed when the class is instantiated
- Substituted in all the signatures of methods and fields
- We can parameterize our classes on how many generics we want
- Generics can be passed to superclasses
  - class VehicleList extends List<Vehicle>
- Widely used to implement data structures
  - Exactly like C++ templates

```
public class List<V> {
 private V[] elements;
 public void add(V el) {
  int n = elements.length+1;
  elements = Arrays.copyOf(elements, n);
  elements[n-1] = el;
 }
 public boolean contains(V el) {
   for(int i=0; I < elements.length; i++)
    if(elements[i]==el)
     return true;
   return false;
 }
 public V get(int i) {
  return elements[i];
 }
}
```

Subtyping with generics

- Java generics are invariant
- I cannot assign an expression with a generic type to a variable with a different generic type
  - Even if one is subtype of the other one!

```
List<Vehicle> v =
    new List<Vehicle>();
List<Bicycle> b = v;
v.add(new Car(…));
Bicycle b1 = b.get(0);
```

```
List<Bicycle> b =
    new List<Bicycle>();
List<Vehicle> v = b;
v.add(new Car(…));
Bicycle b1 = b.get(0);
```

- Instead arrays are covariant
  - I can execute "Vehicle[] v = new Bicycle[10];"

```
public class List<V> {
  public void add(V el) {…}
  public boolean contains(V el) {…}
  public V get(int i) {…}
}

List<Vehicle> v = new List<Vehicle>();
v.add(new Car(…));
List<Bicycle> b = new List<Bicycle>();
v.add(new Bicycle());
v = b;
b = v;
```

Ca' Foscari University of Venice

Generics for methods

- Parametrize methods with generics
  - Not the whole class
- Declare the generics in the method def
  - Before the return type
  - Wrapping it with < and >
- The generics can then be used for
  - Type of parameters
  - Return type
  - Type of local variables

```
public class List<V> {
 public void add(V el) {…}
 public boolean contains(V el) {…}
 public V get(int i) {…}

 public static <T> List<T> toList(T value) {
  List<T> result = new List<T>();
  result.add(value);
  return result;
 }
 public static <T> T getFirst(List<T> list) {
  return list.get(0);
 }
}
```

Ca' Foscari University of Venice

Type inference on generics

- We do not need to explicitly pass the generic type when instantiating classes or calling methods with generics
  - Type inference will do the job for us
- But it is like we specified it ☺
- Generics inferred from our declared types
  - I assigned a List<> to a List<Vehicle>
  - I called a method with a generics type as first parameter passing a Vehicle object
- Obviously, it's far from perfect…

```
public class List<V> {
  public void add(V el) {…}
  public boolean contains(V el) {…}
  public V get(int i) {…}
  public static <T> List<T> toList(T value) {…}
  public static <T> T getFirst(List<T> list) {...}
}


List<Vehicle> v1 = new List<>();
v1.add(new Car(…));
Vehicle v2 = List.getFirst(v1);
List<Vehicle> v3 = List.toList(new Bicycle(…));
```

Bounded generics

- We might want to restrict the possible generics
  - E.g., to rely on the interface of a type
- Add an “extends …” clause to the generics declaration
  - Only types that are subtype of the given bound are allowed
- When instantiating the class or calling the method the type is checked
- Use all the components of the extended type in the implementation

```
<T extends Vehicle> T race(T v1, T v2, double length) {
 v1.fullStop();
 v2.fullStop();
 double distanceV1 = 0, distanceV2=0;
 while(true) {
  distanceV1 += v1.getSpeed();
  distanceV2 += v2.getSpeed();
  if(distanceV1 >= length || distanceV2 >= length) {
   if(distanceV1 > distanceV2) return v1;
   else return v2;
  }
  v1.accelerate(Math.random()*10.0);
  v2.accelerate(Math.random()*10.0);
 }
}
```

Ca' Foscari University of Venice

Wildcards

- Another option is to use wildcards
  - Pass ? instead of a generic type
- List<?> supertype of List<T> for any T
- Wildcards can be bounded (extends)
- Covariant relations on extends
  - List<? extends Car> is a subtype of List<? extends Vehicle>
- Honestly, IMHO the wrong solution to the right problem ☺️

```
public class List<V> {
  public void add(V el) {…}
  public boolean contains(V el) {…}
  public V get(int i) {…}
}
List<Car> v = new List<Car>();
List<?> q = v;
List<? extends Vehicle> w = v;
…
Vehicle e = q.get(0); // error
q.add(new Car(..)); // error
Vehicle e = w.get(0); // OK
w.add(new Car(..)); // error
v.add(new Truck(…)); // OK
```

Ca' Foscari University of Venice

Parametric polymorphism

- This is nothing else than another form of polymorphism!
- Instead of subtyping, it relies on generics

"a function or a data type can be written generically so that it can handle values identically without depending on their type"

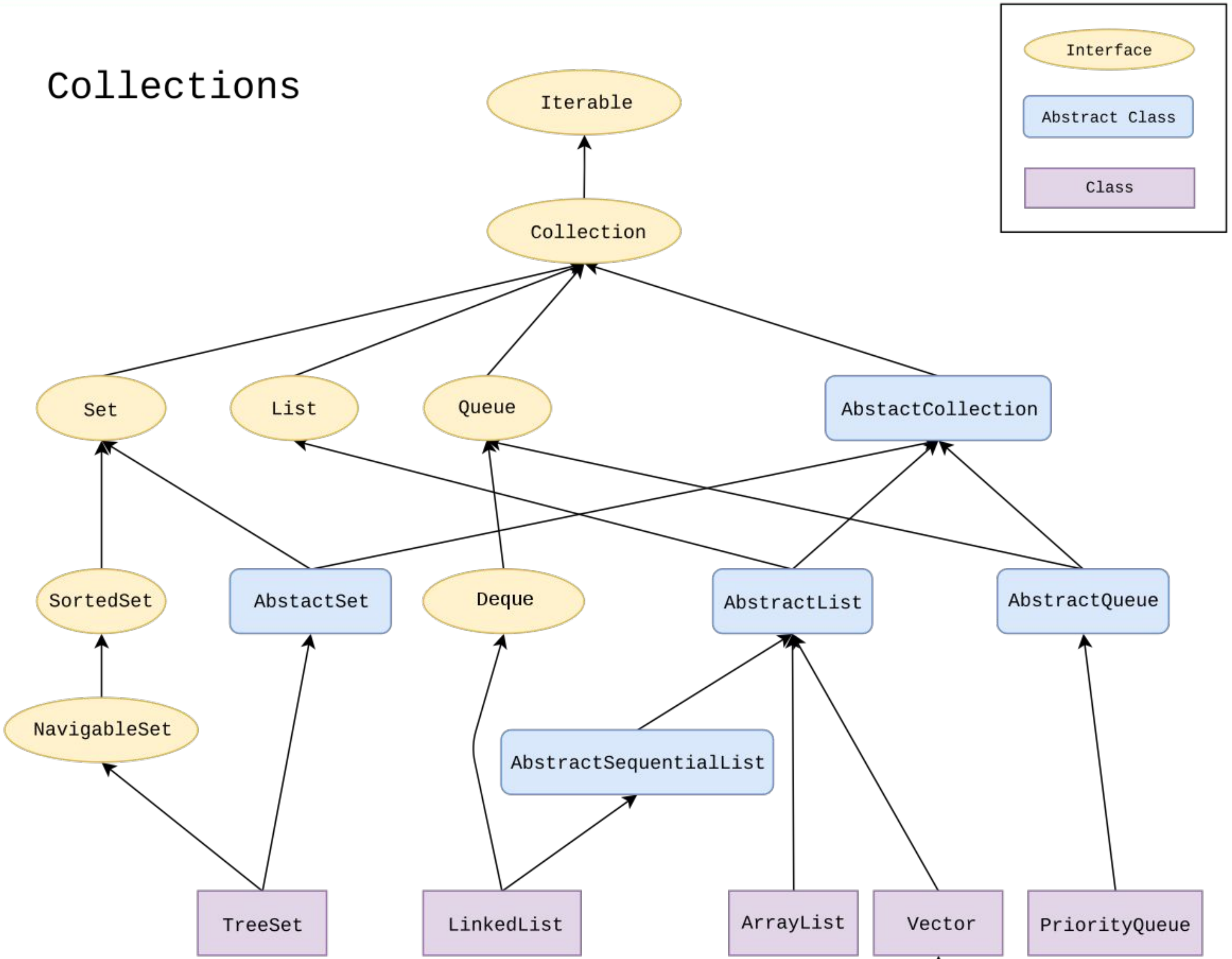
Wikipedia
The Free Encyclopedia

- All mainstream OO programming language supports this
  - In different ways (e.g., C++ templates)
- They were not part of core Java
  - Added only in 2004

Polymorphism: the same symbol (class) has different behaviors

A real-world case study: Collections

- Very complex type hierarchy
- Combine interfaces, abstract classes, and classes
- Widely used in Java
  - Almost by any program
- Deeply studied
  - But other choices might have been possible…



Materials

- Lecture notes: Chapter 11
- Arnold & others: Chapter 11 (no 11.1.2)