CS131: Programming Languages

Lun Liu

05.12.17

Java: Object-Oriented Programming

- Interface
  - A type and a set of associated operations
  - Can typecheck interface with clients without implementation
- Class
  - Implementation

Polymorphism in Java

- Subtype polymorphism
  - `S` is a sub-type of `T` (`S <: T`) implies `S`-values can be used in place of `T`-values

- Parametric polymorphism (like in OCaml)
  - Generics
  - `class List<E> { ... }`
  - `E` is type variable
  - Clients explicit instantiate `E`: `List<String> ls = ...;`

Subtyping & Inheritance

- `S` is a sub-type of `T` (`S <: T`) implies `S`-values can be used in place of `T`-values
  - For example, a function expecting `T`-values would be happy with `S`-values
- Inheritance: re-use of code -- an entity inherits real code from another entity
  - For example, a class inherits (entire code of) another class

Subtyping vs Inheritance

- Subtyping
  - Interface compatibility
  - Relation between types
- Inheritance
  - Implementation and code reuse
  - Relation between implementations

Subtyping vs Inheritance

- `class C1 extends class C2:` C1 is a subtype of C2; C1 inherits from C2
- `class C1 implements interface I1, I2, I3 ...:` C1 objects have type I1, I2, I3…
- `interface I1 extends interface I2, I3, I4 ...:` I1 is a subtype of I2, I3, I4…

Parametric Polymorphism in Java

- Generics

Example: List without generics

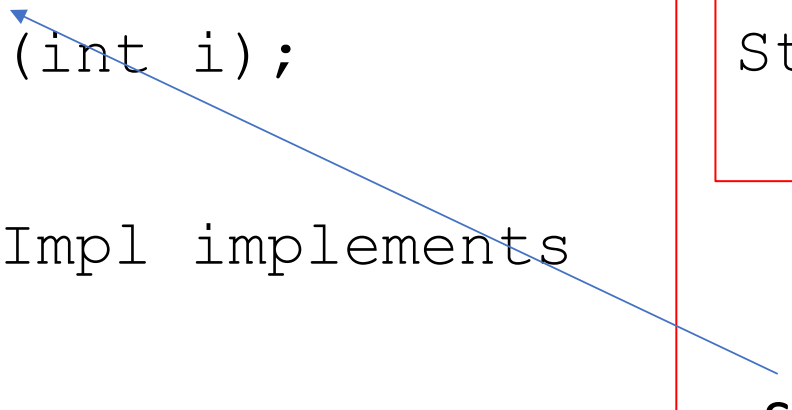
```
interface MyList {
  boolean contains(Object o);
  void add(Object o);
  Object get(int i);
}
class MyListImpl implements
MyList {…}
```

```
MyList l = new MyListImpl()
l.add("lol");
String s = (String) l.get(0);
```

Example: List without generics

```
interface MyList {
  boolean contains(Object o);
  void add(Object o);
  Object get(int i);
}
class MyListImpl implements
MyList {…}
```

```
MyList l = new MyListImpl()
l.add("lol");
String s = (String) l.get(0);
```

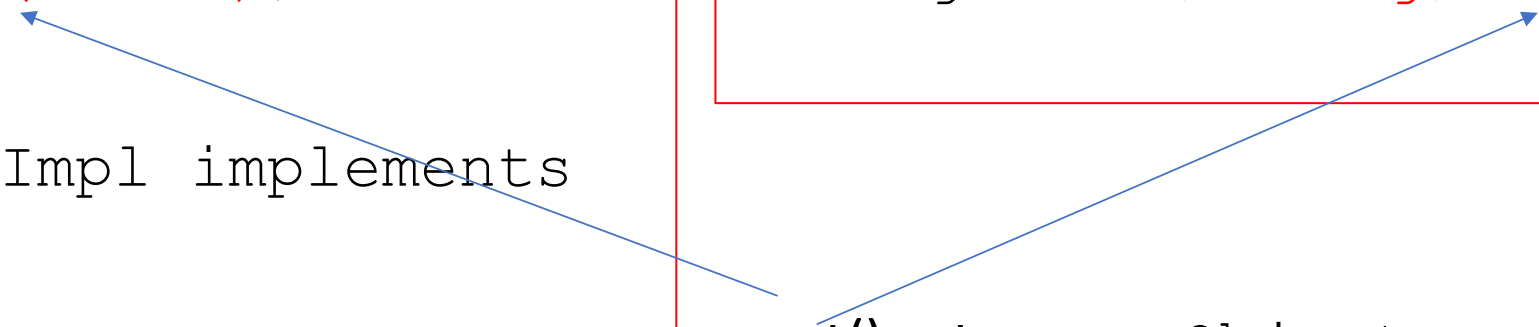


String is a subtype of Object;
subtype polymorphism → typechecked!

Example: List without generics

```
interface MyList {
  boolean contains(Object o);
  void add(Object o);
  Object get(int i);
}
class MyListImpl implements
MyList {…}
```

```
MyList l = new MyListImpl()
l.add("lol");
String s = (String) l.get(0);
```



get() returns an `Object`, want to use as a string; explicit casting needed to shut up compiler

Example: List without generics

```
interface MyList {
  boolean contains(Object o);
  void add(Object o);
  Object get(int i);
}
class MyListImpl implements
MyList {…}
```

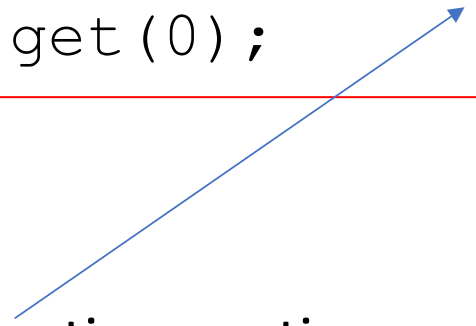
```
MyList l = new MyListImpl()
l.add("lol");
Integer i = (Integer)
l.get(0);
```

Returns Object, explicitly cast to Integer →
Typechecked!

Example: List without generics

```
interface MyList {
  boolean contains(Object o);
  void add(Object o);
  Object get(int i);
}
class MyListImpl implements
MyList {…}
```

```
MyList l = new MyListImpl()
l.add("lol");
Integer i = (Integer)
l.get(0);
```



Runtime casting error

Example: List with generics

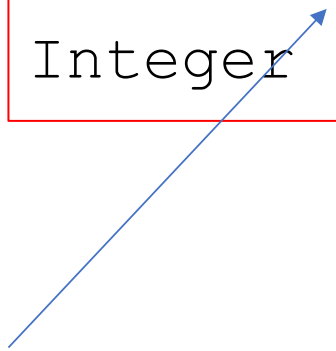
```
interface MyList<E> {
  boolean contains(E e);
  void add(E e);
  E get(int i);
}
class MyListImpl implements
MyList {…}
```

```
MyList<Integer> l = new
MyListImpl<Integer>()
l.add(2);
l.add("lol");
Integer i = l.get(1);
```

Example: List with generics

```
interface MyList<E> {
  boolean contains(E e);
  void add(E e);
  E get(int i);
}
class MyListImpl implements
MyList {…}
```

```
MyList<Integer> l = new
MyListImpl<Integer>()
l.add(2);
l.add("lol");
Integer i = l.get(1);
```



E is instantiated to Integer. static type error

Why Generics (Parametric Polymorphism)

- Stronger type checks at compile time.
  - If a generic type checks, we know it is safe for all instantiation of type variables
- Does not require casting for clients
- Enabling programmers to implement generic algorithms.

Parametric Polymorphism in Java

- Generics
  - Declaration: `class/interface C<G> { ... }` | `G` is just a symbol
  - Bounded Declaration: `class/interface C<G extends T>` | `G` is a symbol, `T` is an actual type (which is the upper-bound)
  - Usage: `C<T> c;` | `T` is an actual type
  - Usage: `C<> c;` | Java 7+ allows this when the generic parameter is clear from the context
- Wildcards

Generics Wildcards

```
void
printCollection(Collection<Objec
t> c) {
   for (Object e : c) {
      System.out.println(e);
   }
}
```

- `Collection<Object>` is **NOT** a supertype of any `Collection` (`Collection<String>` etc.)
  - Why?
  - `Collection<Integer> ci = …`
  - `ci.add("aa");`

```
void printCollection(Collection<?> c) {
   for (Object e : c) {
      System.out.println(e);
   }
}
Collection<?> c = new ArrayList<String>();
printCollection(c);
```

Generics Wildcards

```
void
printCollection(Collection<Objec
t> c) {
   for (Object e : c) {
      System.out.println(e);
   }
}
```

- `Collection<Object>` is **NOT** a supertype of any `Collection` (`Collection<String>` etc.)
  - Why?
  - `Collection<Integer> ci = …`
  - `ci.add("aa");`

```
void printCollection(Collection<?> c) {
   for (Object e : c) {
      System.out.println(e);
   }
}
Collection<?> c = new ArrayList<String>();
printCollection(c);
```

Using `E` instead `?` is also ok in this case

Parametric Polymorphism in Java

- Generics
  - Declaration: `class/interface C<G> { ... }` | `G` is just a symbol
  - Bounded Declaration: `class/interface C<G extends T>` | `G` is a symbol, `T` is an actual type (which is the upper-bound)
  - Usage: `C<T> c;` | `T` is an actual type
  - Usage: `C<> c;` | Java 7+ allows this when the generic parameter is clear from the context
- Wildcards
  - `C<?> cx – ct;`
  - `C<? extends T> cx = ct;`
  - `C<? super T> cx = ct; //Wildcards support both upper and lower bounds, type parameters just support upper bounds.`
  - No `implements` keyword, regardless of whether `T` is a class or an interface
  - (http://stackoverflow.com/questions/4902723/why-cant-a-java-type-parameter-have-a-lower-bound)

Generics (Java) vs Templates (C++)

- Looks similar, actually massively different
- Templates
  - Preprocessor/ macro
  - Basically creating another copy of code
- Generics
  - Type variables used for type checking
  - Java compiler erases all type parameters and replaces each with its first bound if the type parameter is bounded, or `Object` if the type parameter is unbounded. (Type erasure)

https://docs.oracle.com/javase/tutorial/java/generics/genTypes.html

Generics Type Erasure

```
public class Node<T> {
  private T data;
  private Node<T> next;
  public Node(T data, Node<T> next) {
    this.data = data;
    this.next = next;
  }
  public T getData() { return data; }
// ...
}
```

```
public class Node{
  private Object data;
  private Node next;
  public Node(Object data, Node next)
{
    this.data = data;
    this.next = next;
  }
  public Object getData() { return
data; } // ...
}
```

https://docs.oracle.com/javase/tutorial/java/generics/genTypes.html