Polymorphism

“poly” = many
“morph” = shape

Allow a variable to contain values with different types
Subtype polymorphism

- OO languages support subtype polymorphism
- A variable can contain an instance of a subtype of the declared type

```java
Number x;
x = new Integer(17);
x = new Float(2.718);
```
Parametric polymorphism

• Allow code to be parametrized on types

• Introduce **type variables** or **type parameters**
  ```java
class Pair<A, B> { A fst; B snd; }
```

• Different languages allow one to parametrize classes, interfaces, methods, functions, expressions
In functional languages

• Can parametrize algebraic data types:

  datatype 'a list = nil | cons of 'a * 'a list
  val xs: int list = cons(1, cons(2, nil))

• More on this later in the course.
Parametric polymorphism in OO

- Many OO languages support parametrized classes, interfaces, and methods
- Called *generics*

- Issues:
  - What can be parametrized?
  - What types can be used to instantiate a type variable?
  - Bounded polymorphism
  - Variance
  - Implementation
Templates

• C++ supports templates

• Similar to generics, but very different semantics and implementation

• Will discuss later
Types parametrized on types

- Can parametrize classes and interfaces/traits:
  ```java
class Pair<A,B> { A fst; B snd; }
```

- Can parametrize methods:
  ```java
  <A> boolean equal(Pair<A,A> p) {
      return p.fst == p.snd;
  }
  ```
Instantiation

- Type parameters are instantiated on other types.

- As if substitution was performed:

- Consider:

  ```java
class Pair<A,B> { A fst; B snd; }
```

- `Pair<String,Integer>` is as if had written `PairSI` with the class declaration:

  ```java
class PairSI { String fst; Integer snd; }
```
Instantiation

• What types can a parameter be instantiated upon?

• Java:
  • Any reference type ( <: java.lang.Object)

• Scala:
  • Any type

• C#:
  • Any type
Bounded polymorphism

• Type parameters can be bounded when declared

• Subtype bounds:
  • Java:
    ```java
    interface Comparable<T extends Comparable<T>> {...}
    ```
  • Scala:
    ```scala
    def max[A <: Ordered[A])(a: A, b: A) =
      if (a < b) b else a
    ```

• Structural bounds:
  • PolyJ:
    ```scala
    class SortedList[T]
      where T { int compareTo(T) } {...}
    ```
Variance

• What is the relationship between List[Integer] and List[Number]?

• Should List[Integer] be a subtype of List[Number]?

• For C[X]:
  • x is a **covariant parameter** of C if C[A] <: C[B] when A <: B
  • x is a **contravariant parameter** of C if C[A] <: C[B] when B <: A
Suppose `List[Integer] <: List[Number]`

```java
ints: List[Integer] = new List[Integer]();
ints.add(new Integer(23));
nums: List[Number] = ints;
x: Number = ints.get(0);
```

What about `nums.add`?
Covariant parameters

Suppose \( \text{List}[\text{Integer}] \subseteq \text{List}[\text{Number}] \)

```java
ints: List[Integer] = new List[Integer]();
ints.add(new Integer(23));
nums: List[Number] = ints;
x: Number = ints.get(0);

nums.add(new Float(56.78));
```
Covariant parameters

• Cannot call methods that take a covariant parameter as an argument.

• A type system that allows this (without a run-time check) is does not enforce type safety.
Covariant parameters

• In Java, why is `List<Integer>` not a subtype of `List<Number>`?

• What happens if we substitute `Integer` and `Number` for `A` in `List<A>`?

```java
class List<Number> {
    Number get(int i) {...}
    void add(Number v) {...}
}
class List<Integer> {
    Integer get(int i) {...}
    void add(Integer v) {...}
}
```
Covariant parameters

• Why is \texttt{List<Integer>} not a subtype of \texttt{List<Number>}?

• What happens if we substitute \texttt{Integer} and \texttt{Number} for \texttt{A} in \texttt{List\<A\>}? 

```java
class List<Integer> {
    Integer get(int i) {...}
    void add(Integer v) {...}
}

class List<Number> {
    Number get(int i) {...}
    void add(Number v) {...}
}
```

Covariant method parameters
Covariant parameters

- Scala declares variance at the definition:

  ```scala
class List[+A] {...}
  ```

- A is a covariant parameter of `List`

- The compiler checks that a covariant parameter does not occur as a method parameter type.

- => `List.add(A)` is not allowed.
Wildcards

- Java declares variance at the `use`:
  ```java
class List<A> {...}
new List<? extends A>()
```

- These are called **wildcards**.
Covariant wildcards

class List<A> { ... }

List<? extends Number> aList;
List<Integer> iList = ...; aList = fList;
List<Float> fList = ...; aList = iList;

• The type of each element of aList is some statically unknown subtype of Number
Covariant wildcards

class List<A> {...}

List<? extends Number> aList;
List<Integer> iList = ...; aList = fList;
List<Float> fList = ...; aList = iList;

• Compiler knows that the type parameter is some subtype of Number, but doesn’t know which subtype!
Covariant wildcards

class List<A> { ...

List<? extends Number> aList;
List<Integer> iList = ...; aList = fList;
List<Float> fList = ...; aList = iList;

• Cannot call methods that take an A, e.g., add
Contravariant wildcards

class List<A> {...}

List<? super Float> aList;
List<Object> oList = ...; aList = oList;

• The type of each element of aList is some statically unknown supertype of Float
Contravariant wildcards

class List<A> {...}

List<? super Float> aList;
List<Object> oList = ...; aList = oList;

• Compiler knows that the type parameter is some supertype of Float, but doesn’t know which one!
Contravariant wildcards

class List<A> {...}

List<? super Float> aList;
List<Object> oList = ...; aList = oList;

• Must assume methods that return an A, e.g., get, return Object.
Wildcard usability

Want a list of **Birds**

Should one declare a variable as:

- `List<Bird>`  
- `List<? extends Bird>`  
- `List<? super Bird>`  

Hard to know which to do.
List<Bird> birds

• Cannot do:
  birds = new List<Penguin>();

• Can do:
  birds = new List<Bird>();
birds.add(new Bird());
birds.add(new Penguin());
Bird b = birds.get(0);
Wildcard usability

List<? extends Bird> birds

• Can do:
  birds = new List<Penguin>();
  Bird b = birds.get(0);

• Cannot do:
  birds = new List<Object>();
  birds.add(new Bird());
  birds.add(new Penguin());
Wildcard usability

List<? super Bird> birds

• Can do:

```java
birds = new List<Object>();
birds.add(new Bird());
birds.add(new Penguin());
Object o = birds.get(0);
```

• Cannot do:

```java
birds = new List<Penguin>();
Bird b = birds.get(0);
```
Variance in Scala

- Scala declares variance at the class definition:
  ```scala
class Map[[-K, +V] {...}
```

- **Contravariant** in `K`
  - Can use `V` in return types

- **Covariant** in `V`
  - Can use `K` in argument types
Variance in C#

• C# does not support variant parameters.

• Why?
Variance in C#

- C# does not support variant parameters.
- Why? Simplifies the language.
Implementation

• Two main approaches:
  • Polymorphic translation
  • Template instantiation
Polymorphic translation

• Translate a parametrized class into a non-parametrized class
• Insert casts as needed

```java
class C<T> {
    T x;
}
C<String> c = new C<String>();
c.x = "abc";

class C {
    Object x;
}
C c = new C();
c.x = "abc";
String s = (String) c.x;
```
Template instantiation

- Create a new copy of the parametrized class for every instantiation
- Can be done at compile-time or at run-time.

```java
class C<T> {
    T x;
}

C<String> c = new C<String>();
c.x = "abc";
String s = c.x;

class C$S {
    String x;
}

C$S c = new C$S();
c.x = "abc";
String s = c.x;
```
Comparison

- Polymorphic translation
  - usually simpler to implement
  - **much** less space overhead
  - Java does this

- Template instantiation
  - can generate code specialized to the type argument
  - C# does this (at run-time)

- Scala does both:
  - polymorphic translation for reference types
  - template instantiation for primitives
C++ templates

- Do template instantiation on the source code

- With generics, type-checking of parametrized class done before instantiation

- With templates, type-checking does after instantiation

- C++ compiler needs source code of parametrized classes to instantiate them
  - No separate compilation when templates used.
  - Instantiation can fail with bizarre errors.
C#

- CLR generics implemented as run-time template instantiation

- Can be done directly by .NET bytecode. No source required.

- C# compiler checks parametrized code so instantiation should never fail.
Type erasure

• Java **erases** type arguments from the translation

• This was done to support backward compatibility with older Java code:
  
  • Code that uses `Vector` can still use `Vector<T>`
Erasure and run-time types

• Since types are erased, cannot be used for run-time type checking

• Cannot do:
  
  x instanceof List<Integer>

• Can only do:

  x instanceof List
Erasure and run-time types

Casts are unchecked.

```java
List<Integer> xs =
    (List<Integer>)
    (List) new List<String>();
    succeeds!

Integer i = xs.get(0);
    fails with a ClassCastException
Translated to:
    Integer i = (Integer) xs.get(0);
```
Erasure and variance

Because types are erased, can side-step variance issues:

```java
List<Number> xs =
    new List<Integer>();  // illegal!

List<Number> ys =
    (List) new List<Integer>();  // ok!
```

Unchecked cast to List will always succeed. But not also:

```java
List<Number> ys =
    (List) new List<String>();  // ok!
```
Summary

- Parameterized types let you reuse code at multiple types

- Found in most modern statically typed languages

- Issues:
  - What can be parametrized?
  - What types can one instantiate a parameter with?
  - Variance and bounds
  - Implementation: polymorphic or templates