Parametric polymorphism and Generics
Today’s Lecture Outline

• Parametric polymorphism
• Java generics
  • Declaring and instantiating generics
  • Bounded types: restricting instantiations
  • Generics and subtyping. Wildcards
  • Type erasure
• Java arrays
Polymorphism

• Subtype polymorphism
  • What we discussed... Code can use a subclass $B$ where a superclass $A$ is expected
  • E.g., Code $A\ a;$ $\ldots\ a.m()$ is “polymorphic”, because $a$ can be of many different types at runtime: it can be a $A$ object or an $B$ object. Code works with $A$ and with $B$ (with some caveats!)

• Standard in object-oriented languages
Polymorphism

- Parametric polymorphism
  - Code takes a type as a parameter
  - Implicit parametric polymorphism
  - Explicit parametric polymorphism

  - Standard in functional programming languages

- Overloading typically referred to as “ad-hoc” polymorphism
Implicit Parametric Polymorphism

• There is no explicit type parameter(s).
Code is “polymorphic” because it works with many different types. E.g.:

```python
def intersect(sequence1, sequence2):
    result = []
    for x in sequence1:
        if x in sequence2:
            result.append(x)
    return result
```

• As long as sequence1 and sequence2 are of some iterable type, `intersect` works!
Implicit Parametric Polymorphism

• In Python, Lisp, Scheme, others languages
• There is no explicit type parameter(s); the code works with many different types

• Usually, there is a single copy of the code, and all type checking is delayed until runtime
  • If the arguments are of type as expected by the code, code works
  • If not, code issues a type error at runtime
Explicit Parametric Polymorphism

• In Ada, C++, Java
• There is an explicit type parameter(s)
• Explicit parametric polymorphism is also known as genericity
• E.g. in C++ we have templates:

```cpp
template<class V>
class list_node {
    list_node<V>* prev;
    ...
}
```

```cpp
template<class V>
class list {
    list_node<V> header;
    ...
}
```
Explicit Parametric Polymorphism

• Instantiating classes from previous slide with \texttt{int}:

\begin{verbatim}
typedef list_node<int> int_list_node;
typedef list<int> int_list;
\end{verbatim}

• Usually templates are implemented by creating \texttt{multiple copies} of the generic code, one for each concrete type argument, then compiling

• Problem: if you instantiate with the “wrong” type argument, C++ compiler gives us long, cryptic error messages referring to the generic (templated) code in the STL :)
Explicit Parametric Polymorphism

- Java generics work differently from C++ templates: more type checking on generic code
- OO languages usually have both: **subtype polymorphism**
  - through inheritance: A extends B or A implements B
- and **explicit parametric polymorphism**
  - referred to as generics or templates
Using Java Generics

List<AType> list = new ArrayList<AType>();

AType is the type argument. We instantiated generic (templated) class ArrayList with concrete type argument AType

List<String> names = new ArrayList<String>();

names.add(“Ana”);

names.add(“Katarina”);

String s = names.get(0); // what happens here?

Point p = names.get(0); // what happens here?

Point p = (Point) names.get(0); // what happens?
Defining a Generic Class

class MySet<T> {
    // rep invariant: non-null,
    // contains no duplicates
    List<T> theRep;
    T lastLookedUp;
}

Declaration of type parameter

Use of type parameter
Defining a Generic Class

// generic (templated, parameterized) class
public class Name<TypeVar, ... TypeVar> {

• Convention: TypeVar is 1-letter name such as T for Type,
    E for Element, N for Number, K for Key, V for Value
• Class code refers to the type parameter
    • E.g., E
• To instantiate a generic class, client supplies type arguments
    • E.g., String as in List<String> name;
    • Think of it as invoking a “constructor” for the generic class
Example: a Generic Interface

// Represents a list of values
public interface List<E> {
    public void add(E value);
    public void add(int index, E value);
    public E get(int index);
    public int indexOf(E value);
    public boolean isEmpty();
    public void remove(int index);
    public void set(int index, E value);
    public int size();
}

public class ArrayList<E> implements List<E> {

    public class LinkedList<E> implements List<E> {

}
Generics Clarify Your Code

• Without generics
  • This is known as “pseudo-generic containers”

```java
interface Map {
    Object put(Object key, Object value);
    Object get(Object key);
}
```

Client code:
```
Map nodes2neighbors = new HashMap();
String key = …
HashSet value = …
nodes2neighbors.put(key, value);
```

```java
HashSet neighbors = (HashSet) nodes2neighbors.get(key);
```

Casts in client code. Clumsy. If client mistakenly puts non-HashSet value in map, ClassCastException at this point.
Generics Clarify Your Code

• With generics

```java
interface Map<K,V> {
    V put(K key, V value);
    V get(K key);
}
```

Client code:

```java
Map<String,HashSet<String>> nodes2neighbors = new HashMap<String,HashSet<String>>();
String key = …
nodes2neighbors.put(key,value);
HashSet<String> neighbors = nodes2neighbors.get(key);
```

Today’s Lecture Outline

• Parametric polymorphism
• Java generics
  • Declaring and instantiating generics
  • Bounded types: restricting instantiations
  • Generics and subtyping. Wildcards
  • Type erasure
• Java arrays
Bounded Types Restrict Instantiation by Client

```java
interface MyList1<E extends Object> { ... }
```

`MyList1` can be instantiated with any type. Same as

```java
interface MyList1<E> { ... }
```

```java
interface MyList2<E extends Number> { ... }
```

`MyList2` can be instantiated only with type arguments that are `Number` or subtype of `Number`

```java
MyList1<Date> // OK
MyList2<Date> // what happens here?
```
Why Bounded Types?

- Generic code can perform operations permitted by the bound

```java
class MyList1<E extends Object>
    void m(E arg) {
        arg.intValue(); //compile-time error; Object does not have intValue()
    }
}

class MyList2<E extends Number>
    void m(E arg) {
        arg.intValue(); //OK. Number has intValue()
    }
}
```
Another Example

```java
public class Graph<N> implements Iterable<N>
{
    private final Map<N, Set<N>> node2neighbors;
    public Graph(Set<N> nodes,
                 Set<Tuple<N, N>> edges) {
        ...
    }
}

public interface Path<N, P extends Path<N,P>>
    extends Iterable<N>, Comparable<Path<?,?>> {
    public Iterator<N> iterator(); ...
    P extend(N n);
}
```
Bounded Type Parameters

<Type extends SuperType>
• An upper bound, type argument can be SuperType or any of its subtypes

<Type super SubType>
• A lower bound, type argument can be SubType or any of its supertypes
Exercise

• Given this hierarchy with X, Y and Z:

• What are valid instantiations of generics

```java
class A<T extends X> { ... } ?
class A<T extends Z> { ... } ?
class A<T super Z> { ... } ?
class A<T super X> { ... } ?
```
Declaring a Generic Method

class MyUtils {
    <T extends Number>
    T sumList(Collection<T> l) {
        ...
    }
}

Declaration of type parameter

Uses of type parameter
Generic Method Example: Sorting

```java
public static <T extends Comparable<T>>
    void sort(List<T> list) {
        // use of get & T.compareTo<T>
        // T e1 = list.get(...);
        // T e2 = list.get(...);
        // e1.compareTo(e2);
        ...
    }
```

We can use T.compareTo<T> because T is bounded by Comparable<T>!
Another Generic Method Example

```java
public class Collections {
    
    public static <T> void copy(List<T> dst, List<T> src) {
        for (T t : src) {
            dst.add(t);
        }
    }
}
```

When you want to make a single (often static) method generic in a class, precede its return type by type parameter(s).
More Bounded Type Examples

\[
<T \text{ extends } \text{Comparable}<T>>

\>
\]
\[T \text{ max(Collection}<T> \ c);\]

\[
<T> \text{ void copy(List<T2 super } T > \text{ dst,}
\]
\[\text{List<T3 extends } T > \text{ src);} ;\]
\] (actually, must use wildcard \(\_\_\_\) --- more on this later:

\[
<T> \text{ void copy(List<} ? \text{ super } T > \text{ dst,}
\]
\[\text{List<} ? \text{ extends } T > \text{ src);} ; \text{)} ; \]

\[
<T \text{ extends } \text{Comparable}<T2 \text{ super } T>>

\]
\[\text{void sort(List<T> list)} ;\]
\] (same, must use \(\_\_\_\) with super: \(<? \text{ super } T>)\)
Today’s Lecture Outline

• Parametric polymorphism

• Java generics
  • Declaring and instantiating generics
  • Bounded types: restricting instantiations
  • Generics and subtyping. Wildcards
  • Type erasure

• Java arrays
Generics and Subtyping

- `Integer` is a subtype of `Number`

- Is `List<Integer>` a subtype of `List<Number>`?
Use Function Subtyping Rules to Find Out!

interface List<Number> {
    boolean add(Number elt);
    Number get(int index);
}

interface List<Integer> {
    boolean add(Integer elt);
    Integer get(int index);
}

• Function subtyping: subtype must have supertype parameters and subtype return!
What is the Subtyping Relationship Between List<Number> and List<Integer>

• Thus, List<Number> and List<Integer> are unrelated through subtyping!
Immutable Lists

interface ImmutableList<Number> {
    Number get(int index);
}

interface ImmutableList<Integer> {
    Integer get(int index);
}

ImmutableList<Number> ?

ImmutableList<Integer> ?
Write-only Lists

interface WriteOnlyList<Number> {
    boolean add(Number elt);
}
interface WriteOnlyList<Integer> {
    boolean add(Integer elt);
}

• Is WriteOnlyList<Integer> subtype of WriteOnlyList<Number>  
  • NO!

• Is WriteOnlyList<Number> subtype of WriteOnlyList<Integer> 
  • YES!
Getting Stuff Out of WriteList

```java
interface WriteList<Number> {  
    boolean add(Number elt);
    Number get(int index);
}
interface WriteList<Integer> {  
    boolean add(Integer elt);
    Object get(int index);
}

• Is WriteList<Number> subtype of WriteList<Integer>
  • YES!

Contravariant subtyping: because the subtyping relationship between the composites (WriteList<Number> is subtype of WriteList<Integer>) is the opposite of the subtyping relationship between their type arguments (Integer is subtype of Number)
Invariance is Restrictive (Because it Disallows Subtyping)

- Java solution: wildcards

```java
interface Set<E> {
    // Adds all elements in c to this set
    // if they are not already present.
    void addAll(Set<E> c);
    void addAll(Collection<E> c);
    void addAll(Collection<? extends E> c);
    <T extends E> void addAll(Collection<T> c);
}
```

Not good. Can’t have
Set<Number> s; List<Number> l;
s.addAll(l); // List & Set unrelated

Not good either. Can’t have
Set<Number> s; List<Integer> l;
s.addAll(l);

This is because of invariance: List<Integer> is a
subtype of Collection<Integer> but Collection<Integer>
is not a subtype of Collection<Number>!

Solution: wildcards.

? Is the wildcard.
Java Wildcards

- A wildcard is essentially an anonymous type variable
  - Use ? if you’d use a type variable exactly once
- ? appears at the instantiation site of the generic (also called use site)
  - As opposed to declaration site (also called definition site: where type parameter is declared)
- Purpose of the wildcard is to make a library more flexible and easier to use by allowing limited subtyping
Using Wildcards

class HashSet<E> implements Set<E> {
    void addAll(Collection<? extends E> c) {
        // What does this give us about c?
        // i.e., what can code assume about c?
        // What operations can code invoke on c?
    }
}

• Wildcard appears at instantiations (uses) of generics
• There is also <? super E>
• Intuitively, why <? extends E> makes sense here?

This is instantiation of the generic Collection
Covariance

List<Apple> apples = new ArrayList<Apple>();
List<? extends Fruit> fruits = apples;

• ? Introduces covariant subtyping
• Apple is a subtype of Fruit
• List<Apple> is subtype of List<? extends Fruit>

List<Apple> apples = new ArrayList<Apple>();
List<? Extends Fruit> fruits = apples;
apples.add(new Strawberry());

• Won’t compile
• fruits.add(new Fruit()); won’t compile either
Covariance

- the `? extends T` wildcard tells the compiler that we're dealing with a subtype of the type T, but we cannot know which one.
- Since there's no way to tell, and we need to guarantee type safety, you won't be allowed to put anything inside such a structure.
- Since we know that whichever type it might be, it will be a subtype of T, we can get data out of the structure with the guarantee that it will be a T instance.
- Fruit get = fruits.get(0);
Legal Operations on Wildcards

Object o;
Number n;
Integer i;
PositiveInteger p;

List<? extends Integer> lei;

First, which of these is legal?
lei = new ArrayList<Object>();
lei = new ArrayList<Number>();
lei = new ArrayList<Integer>();
lei = new ArrayList<PositiveInteger>();
lei = new ArrayList<NegativeInteger>();

Which of these is legal?
lei.add(o);
lei.add(n);
lei.add(i);
lei.add(p);
lei.add(null);
o = lei.get(0);
n = lei.get(0);
i = lei.get(0);
p = lei.get(0);
Contravariance

• List<Fruit> fruits = new ArrayList<Fruit>();
• List<? super Apple> = fruits;
• fruits is a reference to a List of something that is a supertype of Apple.
• Apple and any of its subtypes will be assignment compatible with fruits
• These will compile
  • fruits.add(new Apple());
  • fruits.add(new GreenApple());
• These will not
• fruits.add(new Fruit());
• fruits.add(new Object());
Contravariance

• Since we cannot know which supertype it is, we aren't allowed to add instances of any.
• The only thing you can get out of it will be Object instances since we cannot know which supertype it is
  • the compiler can only guarantee that it will be a reference to an Object, since Object is the supertype of any Java type.
Legal Operations on Wildcards

Object o;
Number n;
Integer i;
PositiveInteger p;

List<? super Integer> lsi;

First, which of these is legal?
lsi = new ArrayList<Object>();
lsi = new ArrayList<Number>();
lsi = new ArrayList<Integer>();
lsi = new ArrayList<PositiveInteger>();

Which of these is legal?
lsi.add(o);
lsi.add(n);
lsi.add(i);
lsi.add(p);
lsi.add(null);

o = lsi.get(0);
n = lsi.get(0);
i = lsi.get(0);
p = lsi.get(0);
How to Use Wildcards

• Use `<? extends T>` when you get (read) values from a **producer**
• Use `<? super T>` when you add (write) values into a **consumer**
• E.g.:
  <T> void copy(List<? super T> dst, List<? extends T> src)

• **PECS**: **P**roducer **E**xtends, **C**onsumer **S**uper
• Use neither, just `<T>`, if both **add** and **get**
PECS

• Use the ? extends wildcard if you need to retrieve object from a data structure.
• Use the ? super wildcard if you need to put objects in a data structure.
• If you need to do both things, don't use any wildcard.
Wildcards allow Subtyping for Generics

Object

Number

Integer

List<Object>

List<Number>

List<Integer>

List<? extends Number>

List<?>

ArrayList<Integer>

List<Integer>

List<Double>

List<? extends Number>

List<?>

List<Integer>
Today’s Lecture Outline

• Parametric polymorphism
• Java generics
  • Declaring and instantiating generics
  • Bounded types: restricting instantiations
  • Generics and subtyping. Wildcards (briefly)
  • Type erasure
• Arrays
Type Erasure

• All type arguments become Object when compiled
  • Reason: backward compatibility with old bytecode
  • At runtime all generic instantiations have same type

```java
List<String> lst1 = new ArrayList<String>();
List<Integer> lst2 = new ArrayList<Integer>();
lst1.getClass() == lst2.getClass() // true
```

• Cannot use `instanceof` to find type argument

```java
Collection<?> cs = new ArrayList<String>();
if (cs instanceof Collection<String>) {
    // compile-time error
```

• Must use `equals()` on elements of generic type
Equals for a Generic Class

class Node<E> {
    ...
    
    @Override
    public boolean equals(Object obj) {
        if (!(obj instanceof Node<E>))
            return false;
        Node<E> n = (Node<E>) obj;
        return this.data().equals(n.data());
    }
}

At runtime, JVM has no knowledge of type argument. Node<String> is same as Node<Elephant>. Instanceof is a compile-time error.
Equals for a Generic Class

class Node<E> {

    ...

    @Override
    public boolean equals(Object obj) {
        if (!(obj instanceof Node<E>)) {
            return false;
        } // Normalize
        Node<E> n = (Node<E>) obj;
        return this.data().equals(n.data());
    }

}
Today’s Lecture Outline

• Parametric polymorphism
• Java generics
  • Declaring and instantiating generics
  • Bounded types: restricting instantiations
  • Generics and subtyping. Wildcards (briefly)
  • Type erasure
• Arrays
Arrays and Subtyping

• \texttt{Integer} is subtype of \texttt{Number}
• Is \texttt{Integer[]} a subtype of \texttt{Number[]}?
• Use our subtyping rules to find out
  (Just like with \texttt{List<Integer>} and \texttt{List<Number>})
• Again, the answer is NO!
• Different answer in Java: in Java \texttt{Integer[]} is a Java subtype \texttt{Number[]}!
  • The Java subtype is not a true subtype!
  • Known as “problem with Java’s covariant arrays”
Integer[] is a Java subtype of Number[]

```java
Number n;
Number[] na;
Integer i;
Integer[] ia;
na[0] = n;
na[1] = i;
n = na[0];
i = na[1]; //what happens?
ia[0] = n; //what happens?
ia[1] = i;
n = ia[0];
i = ia[1];
```

```
ia = na; //what happens here?
// OK!
i = ia[2]; //what happens here?
```
Writing a Generic Class

- Start by writing a concrete class
- Make sure it is correct
  - Reasoning
  - Testing
- Think about writing a second concrete class
- Generalize by adding type parameters (generics)
  - Which are the same, which differ
  - Compiler will find errors
- With practice, it gets easier to start with generics