Polymorphism (generics)

CSE 331
University of Washington

Michael Ernst
Varieties of abstraction

• Abstraction over computation: procedures (methods)
  
  ```java
  int x1, y1, x2, y2;
  Math.sqrt(x1*x1 + y1*y1);
  Math.sqrt(x2*x2 + y2*y2);
  ```

• Abstraction over data: ADTs (classes, interfaces)
  
  ```java
  Point p1, p2;
  ```

• Abstraction over types: polymorphism (generics)
  
  ```java
  Point<Integer>, Point<Double>
  ```

  – Type abstraction applies to both computation and data
Why we ♥ abstraction

• Hide details
  – Avoid distraction
  – Permit the details to change later

• Give a meaningful name to a concept

• Permit reuse in new contexts
  – Avoid duplication: error-prone, confusing
  – Programmers hate to repeat themselves
A collection of related abstractions

```java
interface ListOfNumbers {
    boolean add(Number elt);
    Number get(int index);
}
interface ListOfIntegers {
    boolean add(Integer elt);
    Integer get(int index);
}

… and many, many more
```

```java
// Type abstraction
// abstracts over element type E
interface List<E> {
    boolean add(E n);
    E get(int index);
}
```

// Use types

```java
List<Number> List<Number>
List<Integer> List<Integer>
List<String> List<String>
```

**add** declares a new variable, called a formal parameter

```
add declares a new variable, called a formal parameter
```

**List** declares a new type variable, called a type parameter

```
List declares a new type variable, called a type parameter
```

use types

```java
// Instantiate by passing an Integer argument:
l.add(7);
myList.add(myInt);
```

```
The type of add is Integer → boolean
```

```java
List<Number> l;
```

```
The kind of List is Type → Type
```

```java
List<Number> myList;
```

```
(Never use List on its own.
Only instantiate it with a type.)
```
Declaring generics

class MyClass<TypeVar1, ..., TypeVarN> {...}
interface MyInterface<TypeVar1, ..., TypeVarN> {...}

Convention: Type variable has a one-letter name such as:
E for Element
K for Key
V for Value
Avoid T for Type which is uninformative
Type variables are types

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

Declaration

Use
Restricting instantiation by clients

```java
boolean add1(Object elt);
boolean add2(Number elt);
add1(new Date()); // OK
add2(new Date()); // compile-time error
```

```java
interface MyOList<E extends Object> {...}
interface MyNList<E extends Number> {...}
MyOList<Date> // OK
MyNList<Date> // compile-time error
```

“Upper bound” Default is Object
Using type variables

Code can perform any operation permitted by the bound

class Foo<T extends Object> {
    void m(T arg) {
        arg.asInt(); // compiler error: T might not support asInt
    }
}

class Bar<N extends Number> {
    void m(N arg) {
        arg.asInt(); // OK: Number and its subtypes support asInt
    }
}
More generic classes

```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();
}

Do not paste into your project unless you understand it and it is what you want.
• Basics of generic types for classes and interfaces
• Basics of bounding generics
• Generic methods [not just using type parameters of class]
  • Generics and subtyping
  • Using bounds for more flexible subtyping
  • Using wildcards for more convenient bounds
• Related digression: Java’s array subtyping
• Java realities: type erasure
  – Unchecked casts
  – equals interactions
  – Creating generic arrays
Not all generics are for collections

```java
class Util {
    public static Object choose(List<Object> lst) {
        int i = new Random().nextInt(lst.size());
        return lst.get(i);
    }
}
```

Class **Util** is not generic, but the method should be generic (**Util** is a collection of static methods, not an ADT that represents something. 😞)

Invariant collection typing: `List<String>` is not a subtype of `List<Object>`

Poor approximation to return value

Cannot pass `List<String>`
Signature of a generic method

class Util {
    public static
        E choose(List<E> lst) {
            int i = new Random().nextInt(lst.size());
            return lst.get(i);
        }
}
Declaring a method’s type parameter

class Util {
    public static <E> E choose(List<E> lst) {
        int i = new Random().nextInt(lst.size());
        return lst.get(i);
    }
}

class Collections {
    public static <E> void copy(List<T> dst, List<T> src) {
        for (E elt : src) {
            dst.add(elt);
        }
    }
}
Using generic methods

• Instance methods can use type parameters of the class

• Methods can have their own type parameters
  – If so, called a “generic method”

• Callers to generic methods need not explicitly instantiate the method’s type parameters
  – Compiler usually figures it out for you
  – Type inference
class Util {
    public static double sumList(List<Number> lst) {
        double result = 0;
        for (Number n : lst) {
            result += n.doubleValue();
        }
        return result;
    }
}
class Util {
    public static double sumList(List<N> lst) {
        double result = 0;
        for (Number n : lst) {
            result += n.doubleValue();
        }
        return result;
    }
}

Type use. Intention: only for numbers.
Declaring a method’s type parameter

class Util {
    public static <N extends Number> double sumList(List<N> lst) {
        double result = 0;
        for (Number n : lst) {
            result += n.doubleValue();
        }
        return result;
    }
}

A type parameter may have a bound

modifiers... <TypeVar [super/extends] Bound> returnType name(params) {...}
public static
<E extends Comparable<E>>
void sort(List<E> list) {
    // ... use list.get() and E.compareTo(E)
}

Actually:
<E extends Comparable<? super E>>
which we will learn about shortly.
More bounded type examples

```java
<E extends Comparable<E>>
E max(Collection<E> c)
    Find max value in any collection (if the elements can be compared)

<E extends Object, E2 extends E>
void copy(List<E> dst, List<E2> src)
    Copy all elements from `src` to `dst`
    `dst` must be able to safely store anything that could be in `src`
    All elements of `src` must be of `dst`'s element type or a subtype
    Equivalently: `<E, E2 extends E>`

<E extends Comparable<E2 super E>>
void sort(List<E> list)
    Sort any list whose elements can be compared to the same type or a broader type
    Actually: `<E extends Comparable<? super E>>`
```
Bounds with type qualifiers

<E extends @NonNull Object>
void foo(List<E> arg) { ... }

<E extends @Nullable Object>
void bar(List<E> arg) { ... }

Set:@NonNull Object> sNnO;
Set:@Nullable Object> sNbleO;

Set:@NonNull Date> sNnD;
Set:@Nullable Date> sNbleD;

Which calls are legal?
1. foo(sNnO)
2. foo(sNbleO)
3. bar(sNnO)
4. bar(sNbleO)
5. foo(sNnD)
6. foo(sNbleD)
7. bar(sNnD)
8. bar(sNbleD)
Outline

- Basics of generic types for classes and interfaces
- Basics of bounding generics
- Generic methods [not just using type parameters of class]
- Generics and subtyping
- Using bounds for more flexible subtyping
- Using wildcards for more convenient bounds
- Related digression: Java’s array subtyping
- Java realities: type erasure
  - Unchecked casts
  - `equals` interactions
  - Creating generic arrays
Generics and subtyping

Integer is a subtype of Number

Is List<Integer> a subtype of List<Number>?

Use our subtyping rules to find out
What is the subtyping relationship between `List<Number>` and `List<Integer>`?

Java subtyping is **invariant** with respect to generics. If \( A \neq B \), then \( C<A> \) has **no** subtyping relationship to \( C'<B> \).
Immutable lists

```java
interface ReadOnlyList<Number> {
    Number get(int index);
}
interface ReadOnlyList<Integer> {
    Integer get(int index);
}
```

Covariance
(True subtyping, but Java forbids)
Write-only lists

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

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

WriteOnlyList<Eagle> hotelCalifornia;

Number

WOList<Integer>

Integer

WOList<Number>

Contravariance
(True subtyping, but Java forbids)
{In, Co, Contra} variant subtyping

If $X \subseteq Y$, then what is the relationship between $C<X>$ and $C<Y>$?
Generic types and subtyping

List<Integer> and List<Number> are not subtype-related

Generic types can have subtyping relationships

Example: If HeftyBag extends Bag, then

- HeftyBag<Integer> is a subtype of Bag<Integer>
- HeftyBag<Number> is a subtype of Bag<Number>
- HeftyBag<String> is a subtype of Bag<String>

... But HeftyBag<Integer> is unrelated to Bag<Number>
Outline

• Basics of generic types for classes and interfaces
• Basics of *bounding* generics
• Generic *methods* [not just using type parameters of class]
• Generics and *subtyping*
• Using *bounds* for more flexible subtyping
• Using *wildcards* for more convenient bounds
• Digression: Java’s unsound *array subtyping*
• Java realities: type erasure
  – Unchecked casts
  – *equals* interactions
  – Creating generic arrays
Invariant subtyping is restrictive
Solution: wildcards

```java
interface Set<E> {
    // Add each element of collection c
    // to this set, if not already present.
    void addAll(Set<E> c);
    void addAll(Collection<E> c);
    <E2 extends E> void addAll(Collection<E2> c);
    void addAll(Collection<? extends E> c);
}
```

Collection<? extends Number> means:
At run time, the value is a collection of some concrete type
```
new List<Number>() , new Set<Integer>() , new Queue<Double>()
```
At compile time, we don’t know which one it was
Code must accommodate all possibilities

Problem 1:
Set<Number> s;
List<Number> l;
s.addAll(l);

Problem 2:
Set<Number> s;
List<Integer> l;
s.addAll(l);

Solution: wildcards

Unrelated to invariant subtyping
Caused by invariant subtyping
What a wildcard represents

A type is a set of values

Wildcards increase flexibility, usability

double sum(Collection<? extends Number> lst) {...}

List<Number> ln; sum(ln);
List<Integer> li; sum(li);
List<Double> ld; sum(ld);
Set<Number> sn; sum(sn);
Set<Integer> si; sum(si);
Set<Double> sd; sum(sd);
LinkedList<BigDecimal> llbd; sum(llbd);
Wildcard = anonymous type variable

Use a wildcard when you would use a type variable once

```java
interface Set<E> extends Collection<E> {
    void addAll(Collection<? extends E> c);
    <E2 extends E> void addAll(Collection<E2> c);
}
```

Wildcards are written at type argument use sites within a parameter declaration.

Nothing appears at the declaration site.

Missing extends clause: "<?" = "<? extends Object>"

There is also "? super E"

Expressible only as a wildcard; no equivalent syntax
Copying a collection

```java
class Collections {
    <E> void copy(List<E> dest, List<E> src) {
        for (E elt : src) {
            dst.add(t);
        }
    }
}
```

`Collections.copy(stringList, stringList);` ✔️
`Collections.copy(numberList, integerList);` ✗

Version with wildcards permits more clients
Legal operations on wildcard types

Object o;
Number n;
Integer i;
@Positive Integer 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<>;
lei = new ArrayList<@Positive Integer>;
lei = new ArrayList<@Negative Integer>;

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);
Legal operations on wildcard types

Object o;
Number n;
Integer i;
@Positive Integer 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<@Positive Integer>;
lsi = new ArrayList<@Negative Integer>;

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);
PECS: _Producer_ Extends, _Consumer_ Super

Where should you insert wildcards? Should you use `extends` or `super` or neither?

- Use `? extends E` when you *get* values from a producer
- Use `? super E` when you *put* values into a consumer
- Use neither (just `E`, not `?`) if you both *get* and *put*

Example:

```java
<T> void copy(List<? super E> dst, List<? extends E> src)
```
Subtyping for generics

Subtyping requires *invariant* type arguments

Exception: *super* wildcard is a supertype of what it matches

Don’t use raw types like `List`! (CSE 331 forbids it)

Object

Number

Integer

ArrayList<Integer>

LinkedList<Integer>

List<Object>

List<Number>

List<Integer>

List<Double>

List<?>

List<? extends Number>

List<? extends Integer>

List<? extends Double>

List<LinkedLIST>
Arrays and subtyping

Is `Integer[]` a subtype of `Number[]`?

Think of `Array` as an ADT with `get` and `set` ops:

```java
class Array<E> {
    public E get(int i) { ... }
    public E set(E newVal, int i) { ... }
}
```

Same answer as Lists with respect to true subtyping

`Integer[]` and `Number[]` are unrelated

Different answer in Java!

`Integer[]` is a Java subtype of `Number[]`
Java subtyping disagrees with true subtyping
Unsound: a program that type-checks can crash
Rationale: enabled code reuse (e.g., sorting routines) before generics
Backwards compatibility means it’s here to stay 😞
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];

ia[0] = n;
ia[1] = i;
n = ia[0];
i = ia[1];

ia = na;

Double d = 3.14;
na = ia;
n[2] = d;
i = ia[2];
```
void maybeSwap(LibraryHolding[] arr) {
    if (arr[17].dueDate() < arr[34].dueDate()) {
        // ... swap arr[17] and arr[34]
    }
}

// client with array that is a Java subtype
Book[] books = ...;
maybeSwap(books);
What can happen: the bad

void replace17(LibraryHolding[] arr, 
        LibraryHolding h) {
    arr[17] = h;
}

// client with array that is a Java subtype
Book[] books = …;
LibraryHolding theWall = new CD("Pink Floyd", 
        "The Wall", …);

replace17(books, theWall);
Book b = books[17]; // holds a 
b.getChapters(); // fails, viol

ArrayStoreException in replace17 at run time.
The exception prevents 
execution of the following code.

Every Java array-update includes this run-time check 
Reads don’t have to (why?)
Tips when writing a generic class

1. Start by writing a concrete instantiation
2. Get it correct (testing, reasoning, etc.)
3. Consider writing a second concrete version
4. Generalize it by adding type parameters
   – Think about which types are the same & different
   – Not all ints are the same, nor are all Strings
   – The compiler will help you find errors
Eventually, it will be easier to write the code generically from the start
   – but maybe not yet
Parametric polymorphism

“Parametric polymorphism” means: identical code and behavior, regardless of the type of the input

- Applies to procedures and types
- One copy of the code, many instantiations
- Utilizes dynamic dispatch

Types of parametric polymorphism

- Dynamic (e.g., Lisp)
- static (e.g., ML, Haskell, Java, C#, Delphi)
- C++ templates are similar; both more and less expressive

In Java, called “generics”

- Most commonly used in Java with collections
- Also used in reflection and elsewhere

Lets you write flexible, general, type-safe code
Generics clarify your code

interface Map { 
    Object put(Object key, Object value);
    equals(Object other);
} 

Generics usually clarify the implementation
   But sometimes ugly: wildcards, arrays, instantiation
Generics always make the client code prettier and safer

interface Map<Key,Value> { 
    Value put(Key key, Value value);
    equals(Object other); 
} 

plus casts in client code 
→ possibility of run-time errors

Cost: More complicated declarations and instantiations, added compile-time checking
Java practicalities
Type erasure

All generic types become type `Object` once compiled

- Big reason: backward compatibility with old byte code
- So, at run time, all generic instantiations have the same type

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

You cannot use `instanceof` to discover a type parameter

```java
Collection<?> cs = new ArrayList<String>();
if (cs instanceof Collection<String>) // illegal ...
```
Recall equals

class Node {

    ...

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

    ...
}
Equals for a parameterized class

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

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

More erasure.
At run time, cannot check.
Equivalent to
Node<Elephant> type = (Node<String>) obj;
class Node\<E\> { 

    ... 

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

    ... 

} 

This test should distinguish Node\<Elephant\> from Node\<String\>. 

Works if the type of obj is Node\<Elephant\> or Node\<String\> or ... 

no subtyping relationship between Node\<Elephant\> and Node\<String\>
Generics and casting

• Casting to generic type results in a compiler warning
  ```java
  List<?> lg = new ArrayList<String>(); // ok
  List<String> ls = (List<String>) lg; // warn
  ```

• The compiler gives an unchecked warning, since the runtime system *cannot* check this

• If you think you need to do this, you are probably wrong
  (Unless you’re implementing `ArrayList` – and then be sure you understand the warning.)

• `Object` can also be cast to any generic type 😞
  ```java
  public static <T> T badCast(T t, Object o) {
    return (T) o; // unchecked warning
  }
  ```
Generics and arrays

```java
public class Foo<T> {
    private T aField;           // ok
    private T[] anArray;        // ok

    public Foo(T param) {
        aField = new T();         // compile-time error
        anArray = new T[10];      // compile-time error
    }
}
```

• You cannot create objects or arrays of a parameterized type (type info not available at run time)
Generics + arrays: a hack

```java
public class Foo<T> {
    private T aField;                   // ok
    private T[] anArray;                // ok

    @SuppressWarnings("unchecked")
    public Foo(T param) {
        aField = param;                   // ok
        anArray = (T[]) (new Object[10]); // ok
    }
}
```

- You cannot create objects or arrays of a parameterized type (type info not available at run time)
- You can create variables of that type, accept them as parameters, return them, or create arrays by casting `Object[]`
  - Casting to generic types is not type-safe, so it generates a warning
  - Rare to need an array of a generic type (e.g., use `ArrayList`
Comparing generic objects

public class ArrayList<E> {
    ...
    public int indexOf(E value) {
        for (int i = 0; i < size; i++) {
            // if (elementData[i] == value) { // wrong
            if (elementData[i].equals(value)) {
                return i;
            }
        }
        return -1;
    }
}

• When testing objects of type E for equality, use equals
Guarantees and lack of guarantees

• Java guarantees a `List<String>` variable always holds a (subtype of) the raw type `List`

• Java does not guarantee a `List<String>` variable always has only `String` elements at run-time
  – Will be true unless unchecked casts involving generics are used
  – Compiler inserts casts to/from `Object` for generics
    • If these casts fail, hard-to-debug errors result
    • Often far from where conceptual mistake occurred

• So, two reasons not to ignore warnings:
  – You’re violating good style/design/subtyping/generics
  – You’re risking difficult debugging