Polymorphism (generics)

CSE 331
University of Washington

Michael Ernst

Varieties of abstraction

- Abstraction over computation: procedures

```
int x1, y1, x2, y2;
Math.sqrt(x1*x1 + y1*y1);
Math.sqrt(x2*x2 + y2*y2);
```

- Abstraction over data: ADTs (classes, interfaces)

```
Point p1, p2;
```

- Abstraction over types: polymorphism (generics)

```
Point<Integer>, Point<Double>
```

  - 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

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

… and many, many more

```
interface List<E> {
  boolean add(E n);
  E get(int index);
}
```

Declares a new **variable**, called a **formal parameter**

Instantiate by passing an **Integer**:
l.add(7);
myList.add(myInt);

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

Declares a new **type variable**, called a **type parameter**

Instantiate by passing a type argument:
List<Float>
List<List<String>>
List<T>

The type of **List** is Type → Type

Using generics (supplying type arguments)

```
List<AType> name = new ArrayList<AType>();
```

- The type that is passed (***AType***) is called the *type parameter*

```
List<String> names = new ArrayList<String>();
names.add("Boris");
names.add("Natasha");
String spy = names.get(0);  // OK
Point oops1 = names.get(1); // compiler error
Point oops1 = (Point)names.get(1);
                    // run-time error
```

- Use of the “raw type” **`ArrayList`** is error-prone
  – Compiler will warn you (can suppress the warning if desired)

Type variables are types

Declaration

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

Use

Restricting instantiation by clients

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

Upper bound

```
interface MyList1<E extends Object> {…}
interface MyList2<E extends Number> {…}
MyList1<Date>       // OK
MyList2<Date>       // compile-time error
```

Declaring and instantiating generics

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

- Convention 1-letter name such as **`T`** for **`Type`**, **`E`** for **`Element`**, **`N`** for **`Number`**, **`K`** for **`Key`**, **`V`** for **`Value`**, or **`M`** for **`Murder`**

- The class's code refers to the type parameter
  - e.g., **`E`**
- To instantiate the abstraction, a client supplies type arguments
  - e.g., **`String`** as in Name<**`String`**>
  - Analogous to 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> { ...
```

Using type variables

Code can perform any operation permitted by the bound

```
interface MyList1<E extends Object> {
  void m(E arg) {
    arg.asInt();  // compiler error
  }
}

interface MyList2<E extends Number> {
  void m(E arg) {
    arg.asInt();  // OK
  }
}
```

Another example

```
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();
}
```

Bounded type parameters

`<Type extends SuperType>`

An upper bound; accepts the given supertype or any of its subtypes

Works for multiple superclass/interfaces with **`&`**

`<Type extends ClassA & InterfaceB & InterfaceC & ...>`

`<Type super SuperType>`

A lower bound; accepts the given supertype or any of its supertypes

Example

```
// TreeSet works for any comparable type
public class TreeSet<T extends Comparable<T>> {
    ...
}
```

Not all generics are for collections

```
class MyUtils {
  static Number sumList(List<Number> l) {
    Number result = 0;
    for (Number n : l) {
      result += n;
    }
    return result;
  }
}
```

Signature of a generic method

```
class MyUtils {
  static
  T sumList(Collection<T> l) {
    // ... black magic within ...
  }
}
```

Type uses
Where is this type
variable declared?

Declaring a method's type parameter

```
class MyUtils {
  static
  <T extends Number> T sumList(Collection<T> l) {
    // ... black magic within ...
  }
}
```

How to declare a type parameter to a method

Sorting

```
public static
<T extends Comparable<T>>
void sort(List<T> list) {
 // … use list.get() and T.compareTo(T)
}
```

Actually:

```
<T extends Comparable<? super T>>
```

Generic methods

```
public static <Type> returnType name(params) {
```

When you want to make just a single (often static) method generic in a class, precede its return type by type parameter(s)

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

More bounded type examples

```
<T extends Comparable<T>>
T max(Collection<T> c)
```

Find max value in any collection (if the elements can be compared)

```
<T>
void copy(List<T2 super T> dst, List<T3 extends T> src)
```

Copy all elements from **`src`** to **`dst`**

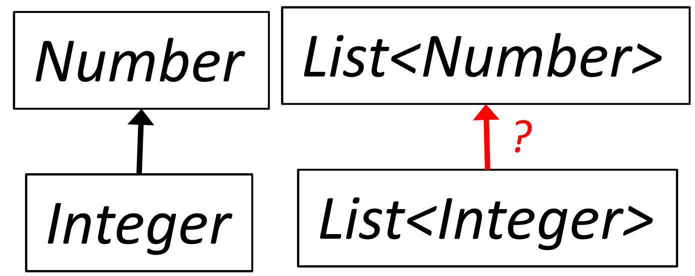
`dst` must be able to safely store anything that could be in **`src`**

This means that all elements of **`src`** must be of **`dst`**'s element type or a subtype

```
<T extends Comparable<T2 super T>>
void sort(List<T> list)
```

Sort any list whose elements can be compared to the same type or a broader type

Generics and subtyping



Integer is a subtype of Number

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

Use our subtyping rules to find out

List<Number> and List<Integer>

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

Java subtyping is invariant with respect to generics

Immutable lists

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

Why would we want this?

Write-only lists

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

Why would we want this?

{In,Co,Contra}variant subtyping

Number ← Integer

Bag ← HeftyBag

Comparing two things, x and f(x).
If y>x, then what is the relationship between f(y) and f(x)?

Covariant subtyping	Contravariant subtyping	Invariant subtyping
ImmutableList<Number> ↑ *ImmutableList<Integer>*	*WriteOnlyList<Number>* ↓ *WriteOnlyList<Integer>*	*List<Number>* *List<Integer>*
Bag<Integer> ↑ *HeftyBag<Integer>*	*Bag<Integer>* ↑ *HeftyBag<Integer>*	*Bag<Integer>* ↑ *HeftyBag<Integer>*

Invariant subtyping is restrictive
Solution: wildcards

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

Unrelated to invariant subtyping

Problem 1:

```
Set<Number> s;
List<Number> l;
s.addAll(l);
```

Caused by invariant subtyping

Problem 2:

```
Set<Number> s;
List<Integer> l;
s.addAll(l);
```

A wildcard is essentially an anonymous type variable

Use it when you would use a type variable exactly once

It appears at the use site; nothing appears at the declaration site

Using wildcards

```
class HashSet<E> implements Set<E> {
  void addAll(Collection<? extends E> c) {
    // What can this code assume about c?
    // What operations can this code invoke on c?
    ...
  }
}
```

Wildcards are written at type argument uses

- Within a parameter declaration

A missing extends clause means "`extends Object`"

There is also "`? super E`"

Legal operations on wildcard types

```
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);`~~

Legal operations on wildcard types

```
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>;`~~
~~`lsi = new ArrayList<NegativeInteger>;`~~

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);`~~

Equals for a parameterized class

```
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<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());
  }
  …
}
```

Erasure: at run time, the JVM has no knowledge of type arguments

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());
  }
  …
}
```

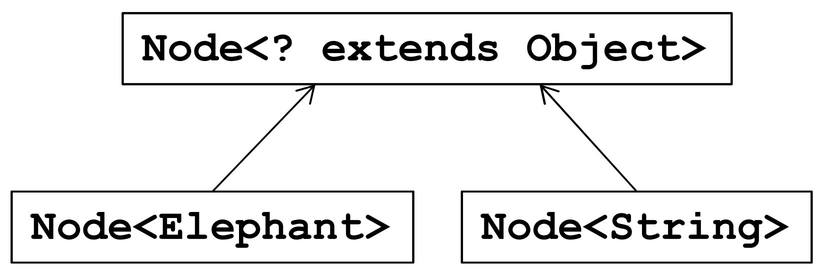
Erasure again.
At run time, equivalent to

```
Node<Elephant> type =
  (Node<String>) obj;
```

Equals for a parameterized class

```
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());
  }
  …
}
```

Works if the type of obj is **Node<Elephant>** or **Node<String>** or …



no subtyping relationship

Wildcards

- **`?`** indicates a wild-card type parameter, one that can be any type
  `List<?> list = new List<?>();   // anything`

- Difference between **`List<?>`** and **`List<Object>`**
  - **`?`** can become any particular type; **`Object`** is just one such type
  - **`List<Object>`** is restrictive; wouldn't take a **`List<String>`**

- Difference between **`List<Foo>`** and **`List<? extends Foo>`**
  - The latter binds to a particular **`Foo`** subtype and allows ONLY that
    - Ex: **`List<? extends Animal>`** might store only **`Giraffes`** but not **`Zebras`**
  - The former allows anything that is a subtype of **`Foo`** in the same list
    - Ex: **`List<Animal>`** could store both **`Giraffes`** and **`Zebras`**

PECS: Producer Extends, Consumer Super

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

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

```
<T> void copy(
    List<? super T> dst,
    List<? extends T> src)
```

Subtyping for generics

Object

Number

List<?>

List

List<Object>

Integer

List<? extends Number>

List<Number>

List<Integer>

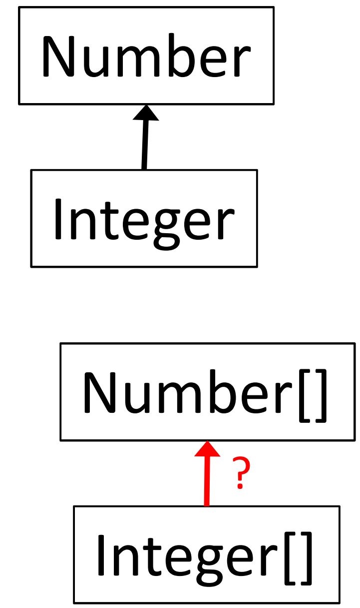
List<Double>

ArrayList<Integer>

LinkedList<Integer>

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)

Arrays and subtyping



Integer is a subtype of Number

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

Use our subtyping rules to find out

(Same question as with Lists)

Same answer with respect to true subtyping

Different answer in Java!

- **`Integer[]`** is a Java subtype of **`Number[]`**
- Java subtyping disagrees with true subtyping

Integer[] is a Java subtype of Number[]

```
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;
na[2] = d;
i = ia[2];
```

Why did the Java designers do this?

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 `int`s are the same, nor are all `String`s
   - 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);
}
```

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

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

Cost: More complicated declarations and instantiations, added compile-time checking

Generics usually clarify the implementation

sometimes ugly: wildcards, arrays, instantiation

Generics always make the client code prettier and safer

Java practicalities

Type erasure

- All generic types become type **`Object`** once compiled
  - Big reason: backward compatibility with old byte code
  - So, at runtime, all generic instantiations have the same type

```
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

```
Collection<?> cs = new ArrayList<String>();
if (cs instanceof Collection<String>) {
  // illegal
}
```

Generics and casting

- Casting to generic type results in a warning

```
List<?> lg = new ArrayList<String>();  // ok
List<String> ls = (List<String>) lg;   // warn
```

- The compiler gives an unchecked warning, since this isn't something the runtime system is going to check for you
- Usually, if you think you need to do this, you're wrong
  - (Unless you're implementing things like **`ArrayList`** – and then be sure you understand why you're getting the warning)

- The same is true of type variables:

```
public static <T> T badCast(T t, Object o)
{
    return (T) o;   // unchecked warning
}
```

Generics and arrays

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

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

- You cannot create objects or arrays of a parameterized type

Generics/arrays: a hack

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

    @SuppressWarnings("unchecked")
    public Foo(T param) {
        aField = param;                      // ok
        T[] a2 = (T[]) (new Object[10]); // ok
    }
}
```

- 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
  - You almost surely don't need this in common situations!

Comparing generic objects

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

- When testing objects of type **`E`** for equality, must use **`equals`**