CSE 331
Software Design & Implementation
Topic: Generics; Event-driven Programming

💬 Discussion: What can you do to make a team work smoothly?
Reminders

- If you don’t know where to start, read answers-hw6.txt!
- Don’t add generics to HW6

Upcoming Deadlines

- HW6 due Thursday (7/28)
Last Time...

- Intro to Generics
- Generic Methods
- Generics and Subtyping
- Arrays
- Type Bounds

Today’s Agenda

- Wildcards
- Type Erasure
- Event-driven Programming
Recall: 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>
```
Recall: Type Parameters

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

interface Map<K, V> {
    V put(K key, V value);
    ...
}

• Generics always make the client code easier to read and safer
• Generics usually clarify the implementation
  – (but sometimes uglify: wildcards, arrays, instantiation)
class Utilities {
    public static <T extends Number> double sumList(List<T> lst) {
        double result = 0.0;
        for (T n : lst) { // T also works
            result += n.doubleValue();
        }
        return result;
    }
    public static <T> T choose(List<T> lst) {
        int i = ... // random number < lst.size
        return lst.get(i);
    }
}
Recall: Generics + Subtyping

If $A$ and $B$ are different, then $\text{GenericClass}<A>$ is not a subtype of $\text{GenericClass}<B>$

For example, $\text{List<Integer>}$ and $\text{List<Number>}$ are not subtype-related

- Example: If $\text{HeftyBag}$ extends $\text{Bag}$, then
  - $\text{HeftyBag<Integer>}$ is a subtype of $\text{Bag<Integer>}$
  - $\text{HeftyBag<Number>}$ is a subtype of $\text{Bag<Number>}$
  - $\text{HeftyBag<String>}$ is a subtype of $\text{Bag<String>}$
Recall: Generics + Subtyping

If $A$ and $B$ are different, then $\text{GenericClass}<A>$ is not a subtype of $\text{GenericClass}<B>$

For example, $\text{List}<\text{Integer}>$ and $\text{List}<\text{Number}>$ are not subtype-related

- Example: If $\text{HeftyBag}$ extends $\text{Bag}$, then
  - $\text{HeftyBag}<\text{Integer}>$ is a subtype of $\text{Bag}<\text{Integer}>$
  - $\text{HeftyBag}<\text{Number}>$ is a subtype of $\text{Bag}<\text{Number}>$
  - $\text{HeftyBag}<\text{String}>$ is a subtype of $\text{Bag}<\text{String}>$

If $B$ is a subtype of $A$, then $B[]$ is a Java subtype of $A[]$
However, it is not a true subtype! Java will not give you a compiler warning
  - storing a supertype into an index causes $\text{ArrayStoreException}$ (at run time)
Recall: Type Bounds

Instead of this:

```java
<T> void copyTo(List<T> dst, List<T> src) {
    for (T t : src)
        dst.add(t);
}
```

We can now do this:

```java
<T1, T2 extends T1> void copyTo(List<T1> dst, List<T2> src) {
    for (T2 t : src)
        dst.add(t);
}
```

What is the difference between these two?
Where are we?

- basics of generic types for classes and interfaces
- basics of bounding generics
- generic methods [not just using type parameters of class]
- generics and subtyping
- related digression: Java's array subtyping
- using bounds for more flexible subtyping
- using wildcards for more convenient bounds
- Java realities: type erasure
  * unchecked casts
  * equals interactions
  * creating generic arrays
Examples

[Compare to earlier version]

```java
interface Set<E> {
    void addAll(________ c);
}
```

• First version:
  ```java
  void addAll(Collection<E> c);
  ```

• Better version:
  ```java
  <T extends E> void addAll(Collection<T> c);
  ```
Examples

[Compare to earlier version]

```java
interface Set<E> {
    void addAll(_______ c);
}
```

• First version:
  ```java
  void addAll(Collection<E> c);
  ```

• Better version:
  ```java
  <T extends E> void addAll(Collection<T> c);
  ```

• Most idiomatic version:
  ```java
  void addAll(Collection<? extends E> c);
  ```
Wildcards

Syntax: for a type-parameter instantiation (inside the <...>), can write:
  - `? extends Type`, some unspecified subtype of `Type`
  - `?` is shorthand for `? extends Object`

A wildcard is essentially an **anonymous type variable**
  - each `?` stands for some possibly-different unknown type
? versus Object

? indicates a particular but unknown type

```java
void printAll(List<?> lst) {...}
```

Difference between `List<?>` and `List<Object>`:
- can instantiate ? with any type: `Object, String, ...`
- `List<Object>` much more restrictive:
  - e.g., wouldn't take a `List<String>`

Difference between `List<Number>` and `List<? extends Number>`:
- can instantiate ? with `Number, Integer, Double, ...`
- first version is much more restrictive
Non-example

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

No change because `T` used *more than once*
- must choose a name to say that two types must match
Wildcards

Syntax: for a type-parameter instantiation (inside the <...>), can write:
- `? extends Type`, some unspecified subtype of `Type`
- `?` is shorthand for `? extends Object`

A wildcard is essentially an **anonymous type variable**
- each `?` stands for some possibly-different unknown type
- use a wildcard when you would use a type variable only once (no need to give it a name)
- communicates to readers of your code that the type’s “identity” is not needed anywhere else
Wildcards

Syntax: for a type-parameter instantiation (inside the <...>), can write:
- `? extends Type`, some unspecified subtype of `Type`
- `?` is shorthand for `? extends Object`
- `? super Type`, some unspecified superclass of `Type`

Wildcard can have lower bounds instead of upper bounds!
- says that `?` must be `Type` or a superclass of `Type`
Type Bounds

Upper Bound
? extends Number

Lower bound
? super Number
Revisit copy method

First version:

```java
<T> void copyTo(List<T> dst, List<T> src) {
    for (T t : src)
        dst.add(t);
}
```

More general version:

```java
<T1, T2 extends T1> void copyTo(List<T1> dst, List<T2> src) {
    for (T2 t : src)
        dst.add(t);
}
```
More examples

Let’s rewrite this using wildcards:

```java
<T> void copyTo(List<? super T> dst, List<? extends T> src) {
    for (T t : src)
        dst.add(t);
}
```

Why this works:
- lower bound of `T` for where callee puts values
- upper bound of `T` for where callee gets values
- callers get the subtyping they want
  - Example: `copy(numberList, integerList)`
  - Example: `copy(stringList, stringList)`
Should you use `extends` or `super` or neither?

- use `? extends T` when you get values (from a producer)
  - no problem if it’s a subtype
  - (the co-variant subtyping case)
- use `? super T` when you put values (into a consumer)
  - no problem if it’s a supertype
  - (the contra-variant subtyping case)
- use neither (just T, not ?) if you both get and put
  - can’t be as flexible here

```java
<T> void copyTo(List<? super T> dst, List<? extends T> src);
```
More on lower bounds

- As we’ve seen, lower-bound ? super T is useful for “consumers”

- Upper-bound ? extends T could be rewritten without wildcards, but wildcards preferred style where they suffice

- But lower-bound is only available for wildcards in Java
  - this does not parse:
    
    ```java
    <T super Foo> void m(Bar<T> x);
    ```
  - no good reason for Java not to support such lower bounds except designers decided it wasn’t useful enough to bother
    
    • ¯\_(ツ)_/¯
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>();
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>();
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?
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<? 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?
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<? 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?
o = lei.get(0);
n = lei.get(0);
i = lei.get(0);
p = lei.get(0);
lei.add(o);
lei.add(n);
lei.add(i);
lei.add(p);
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?
o = lei.get(0);
n = lei.get(0);
i = lei.get(0);
p = lei.get(0);
lei.add(o);
lei.add(n);
lei.add(i);
lei.add(p);
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?
o = lei.get(0);
n = lei.get(0);
i = lei.get(0);
p = lei.get(0);
lei.add(o);
lei.add(n);
lei.add(i);
lei.add(p);
lei.add(null);
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>;
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>;

CSE 331 Summer 2022
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);
Legal operations on wildcard types

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

List<? super Integer> lsi;

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

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>;

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);
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);
Where are we?

- basics of generic types for classes and interfaces
- basics of *bounding* generics
- generic *methods* [not just using type parameters of class]
- generics and *subtyping*
- related digression: Java's *array subtyping*
- using *bounds* for more flexible subtyping
- using *wildcards* for more convenient bounds
- Java realities: type erasure
  - unchecked casts
  - `equals` interactions
  - creating generic arrays
Type erasure

All generic types become type `Object` once compiled

```
List<String> lst = new ArrayList<String>();
```

at runtime, becomes

```
List<Object> lst = new ArrayList<Object>();
```

Generics are purely a `compiler` feature!
Type erasure example

```java
import java.util.*;

public class Erasure {

    public static void foo() {
        List<String> lst = new ArrayList<String>();
        lst.add("abc");
        lst.add("def");
    }
}
```
Type erasure example

Compile-time signature is \texttt{add(String)} but the bytecodes say...

```
public static void foo();

Code:
0: new    #7                  // class java/util/ArrayList
3: dup
4: invokespecial #9          // Method java/util/ArrayList."<init>":(()V
7: astore_0
8: aload_0
9: ldc    #10                // String abc java/util/List.add:(Ljava/lang/Object;)Z
11: invokeinterface #12, 2   // InterfaceMethod java/util/List.add:(Ljava/lang/Object;)Z
16: pop
17: aload_0
18: ldc    #18                // String def java/util/List.add:(Ljava/lang/Object;)Z
20: invokeinterface #12, 2   // InterfaceMethod java/util/List.add:(Ljava/lang/Object;)Z
25: pop
26: return
```
Type erasure

All generic types become type `Object` once compiled
- gives backward compatibility (a selling point at time of adoption)
- at run-time, all generic instantiations have the same type

Cannot use `instanceof` to discover a type parameter

```java
Collection<?> cs = new ArrayList<String>();
if (cs instanceof Collection<String>) { // illegal
    ...
}
```
Generics and casting

Casting to generic type results in an important warning

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

Compiler gives a warning because the runtime system *will not check for you*

Usually, if you think you need to do this, you're wrong

- a real need to do this is extremely rare

`Object` can also be cast to any generic type 😐

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

• 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 if no unchecked cast warnings are shown
  – 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:
  1. You’re violating good style/design/subtyping/generics
  2. You’re risking difficult debugging
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<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);
    }
    ...
}
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);
    }
    ...
}
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);
    }
    ...
}
Generics and arrays

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

    public Foo() {
        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 is not available at runtime
public class Foo<T> {
    private T aField;
    private T[] anArray;

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

You can declare variables of type T, accept them as parameters, return them, or create arrays by casting Object[].

- casting to generic types is not type-safe (hence the warning)
- Effective Java: use ArrayList instead
Consider the following sorting method:

```java
public static void sort(List<Integer> lst) {
    for (int i = 0; i != n; i++) {
        for (int j = 0; j != n - 1; j++) {
            if (lst.get(j) > lst.get(j + 1)) {
                swap(lst, j, j + 1);
            }
        }
    }
}
```

What could we improve about this?
Consider the following sorting method:

```java
public static void sort(List<?> lst) {
    for (int i = 0; i != n; i++) {
        for (int j = 0; j != n - 1; j++) {
            if (lst.get(j) > lst.get(j + 1)) {
                swap(lst, j, j + 1);
            }
        }
    }
}
```

But wait - this doesn’t compile! Why?
Achievement unlocked: Callbacks

- Even though we are the implementer, we may need the client to help us
  - previously, we have seen clients provide data that we can process
  - now, we will see how clients can provide code that can be executed

**Callback pattern**: “Code” provided by client to be used by library
- In JS etc., pass a function as an argument
- In Java, pass an object with the “code” in a method

**Synchronous** callbacks:
- Useful when library needs the callback result immediately

**Asynchronous** callbacks (i.e. event-driven programming):
- Useful for performing an action when some interesting event occurs later
A sorting example...

First, we can define:

```java
public interface Comparable<T> {
    public int compareTo(T other);
}
```

Every object that implements this interface must provide some code that informs us which of two objects is bigger.

- returns -1 if this is smaller than other
- returns 0 if this is equal to other
- returns 1 if this is bigger than other
A sorting example...

```java
public static <T extends Comparable<T>> void sort(List<T> lst) {
    for (int i = 0; i != n; i++) {
        for (int j = 0; j != n - 1; j++) {
            if (lst.get(j).compareTo(lst.get(j + 1)) > 0) {
                swap(lst, j, j + 1);
            }
        }
    }
}
```

We can use the callback pattern to ask the client how to compare to objects.

Relying on client code to sort
How are callbacks used in practice?

• Clients sit around waiting for events like:
  – mouse move/drag/click, button press, button release
  – keyboard: key press or release, sometimes with modifiers like shift/control/alt/etc.
  – finger tap or drag on a touchscreen
  – window resize/minimize/restore/close
  – timer interrupt (including animations)
  – network activity or file I/O (start, done, error)
    • (we will see an example of this shortly)
Achievement unlocked: Observers

This is the **observer pattern**
- Objects can be *observed* via *observers/listeners* that are *notified* via *callbacks* when an *event* (of interest) occurs
- **Pattern**: Something used over-and-over in software, worth recognizing when appropriate and using common terms
- Widely used in public libraries
- Useful for “visual” programs like web applications

More examples of “observers” coming later...
Event-driven programming

An *event-driven* program is designed to wait for events:
- program initializes then enters the *event loop*
- abstractly:
  ```
  do {
      e = getNextEvent();
      process event e;
  } while (e != quit);
  ```

Contrast with most programs we have written so far
- they perform specified steps in order and then exit
- that style is still used, just not as frequently
  - example: computing Page Rank or other Big Data work
Before next class...

1. Ask questions about HW6 and do answers-hw6.txt early
   - Implement your specification from HW5
   - Probably shouldn’t use generics yet (we do this in HW7)

2. Next time, we will start looking at HTML, CSS, and JS
   - HW8 and HW9 will use these instead of Java