Where are we?

• Done:
  – basics of generic types for classes and interfaces
  – basics of bounding generics

• Now:
  – 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
Review

- `List<Number>` is a subtype of `Collection<Number>`
- `Set<Number>` is a subtype of `Collection<Number>`
- `Integer` is a subtype of `Number`
- `List<Integer>` is not a subtype of `List<Number>`
- `Set<Integer>` is not a subtype of `Set<Number>`

Summary:
- subtyping works in the type if the parameters are the same
- changing the parameters removes any subtype relationships
Best type for `addAll`

```java
interface Set<E> {
    // Adds all elements in c to this set
    // (that are not already present)
    void addAll(_______ c);
}

void addAll(Collection<E> c);
```

Still too restrictive:

- cannot pass a `List<Integer>` to `addAll` for a `Set<Number>`
Best type for `addAll`

```java
interface Set<E> {
    // Adds all elements in c to this set
    // (that are not already present)
    void addAll(_______ c);
}

<T extends E> void addAll(Collection<T> c);
```

The fix: bounded generic type parameter
- *can* pass a `List<Integer>` to `addAll` for a `Set<Number>`
- `addAll` implementations won’t know what element type `T` is, but will know it is a subtype of `E`
<T extends Number> void sumList(List<T> nums) {
    double s = 0;
    for (T t : nums) {
        s += t.doubleValue();
    }
    return s;
}
Revisit copy method

Earlier we saw this:

\[
<T> \text{ void } \text{copyTo}(\text{List}<T> \text{ dst}, \text{List}<T> \text{ src}) \{ \\
\quad \text{for (T t : src)} \\
\qquad \text{dst.add(t)}; \\
\}
\]

Now we can do this (which is more general):

\[
<T1, T2 \text{ extends T1}> \text{ void } \text{copyTo}(\text{List}<T1> \text{ dst}, \text{List}<T2> \text{ src}) \{ \\
\quad \text{for (T2 t : src)} \\
\qquad \text{dst.add(t)}; \\
\}
\]
Where are we?

- **Done:**
  - basics of generic types for classes and interfaces
  - basics of *bounding* generics

- **Now:**
  - 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
Wildcards

[Compare to earlier version]

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

• Equivalent to

  <T extends E> void addAll(Collection<T> c);

• Idiomatic Java
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
More examples

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

• No change because T used more than once
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

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)
  - avoids declaring generic type variables
- communicates to readers of your code that the type’s “identity” is not needed anywhere else
<T> void copyTo(List<? super T> dst, 
        List<? extends T> src);

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)
PECS: _Producer Extends, Consumer Super_

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 \( \texttt{super T} \) is useful for “consumers”

• Upper-bound \( \texttt{extends T} \) could be rewritten without wildcards, but wildcards preferred style where they suffice

• But lower-bound is \textit{only} available for wildcards in Java
  – this does not parse:
    \[
    \langle T \texttt{ super Foo} \rangle \texttt{ void m(Bar\langle T\rangle x);}
    \]
  – no good reason for Java not to support such lower bounds except designers decided it wasn’t useful enough to bother
  • \_\_(ツ)_\_/
? 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<Foo>` and `List<? extends Foo>`:
- In latter, element type is one unknown subtype of `Foo`
  - Example: `List<? extends Animal>` might store only
    Giraffes only (no Zebras)
- Former allows anything that is a subtype of `Foo` in the same list
  - Example: `List<Animal>` could store Giraffes and Zebras
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?
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<? extends Integer> lei;

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);

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<? 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>;
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?

• Done:
  – basics of generic types for classes and interfaces
  – basics of *bounding* generics

• Now:
  – 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
Java arrays

We know how to use arrays:
- declare an array holding Type elements: `Type[]`
- get an element: `x[i]`
- set an element `x[i] = e;`

Java included the syntax above because it’s common and concise

But can reason about how it should work the same as this:
```java
class Array<T> {
    public T get(int i) { ... “magic” ... }
    public T set(T newVal, int i) { ... “magic” ... }
}
```

So: If `Type1` is a subtype of `Type2`, how should `Type1[]` and `Type2[]` be related??
Java Arrays

• Given everything we have learned, if \texttt{Type1} is a subtype of \texttt{Type2}, then \texttt{Type1}[] and \texttt{Type2}[] should be unrelated
  – invariant subtyping for generics
  – because arrays are mutable
Surprise!

- Given everything we have learned, if Type1 is a subtype of Type2, then Type1 [] and Type2 [] should be unrelated
  - invariant subtyping for generics
  - because arrays are mutable

- But in Java, if Type1 is a subtype of Type2, then Type1 [] is a subtype of Type2 [] (covariant subtyping)
  - not true subtyping: the subtype does not support setting an array element to hold a Type2 (spoiler: throws an exception)
  - Java (and C#) made this decision in pre-generics days
    - needed to write reusable sorting routines, etc.
    - also urement of
What can happen: the good

Programmers can use this subtyping to “do okay stuff”

```java
void maybeSwap(LibraryHolding[] arr) {
    if(arr[17].dueDate() < arr[34].dueDate())
        // … swap arr[17] and arr[34]
}

// client with subtype
Book[] books = …;
maybeSwap(books); // relies on covariant
    // array subtyping
```
What can happen: the bad

Something in here must go wrong!

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

// client with subtype

```java
Book[] books = ...;
LibraryHolding theWall = new CD("Pink Floyd", "The Wall", ...);
replace17(books, theWall);
Book b = books[17]; // would hold a CD
b.getChapters(); // so this would fail
```
Java’s choice

• Java normally guarantees run-time type is a subtype of the compile-time type
  – this was violated for the `Book b` variable

• To preserve the guarantee, Java must never get that far:
  – each array “knows” its actual run-time type (e.g., `Book []`)
  – trying to store a supertype into an index causes `ArrayStoreException` (at run time)

• So the body of `replace17` would raise an exception
  – even though `replace17` is entirely reasonable
    • and fine for plenty of “careful” clients
    – every Java array-update includes this run-time check
      • (array-reads never fail this way – why?)
  – beware careful with array subtyping
Where are we?

• Done:
  – basics of generic types for classes and interfaces
  – basics of bounding generics

• Now:
  – 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
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

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

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

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 this is something 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());
    }
    ...
}

Erasure: Type arguments do not exist at runtime
equals for a parameterized class

```java
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 runtime, do not know what E is and will not be checked, so don’t indicate otherwise.
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 ...

Leave it to here to “do the right thing” if this and n differ on element type
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
Necessary array cast

```java
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
Some final thoughts…
Generics clarify your code

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

interface Map<Key, Value> {
    Value put(Key key, Value value);
    ...
}

• Generics usually clarify the implementation
  – (but sometimes uglify: wildcards, arrays, instantiation)
• Generics always make the client code prettier and safer

plus casts in client code
→ possibility of run-time errors
Tips when writing a generic class

• Think through whether you **really need** to make it generic
  – if it’s not really a container, most likely a **mistake**

• Start by writing a concrete instantiation
  – get it correct (testing, reasoning, etc.)
  – consider writing a second concrete version

• Generalize it by adding type parameters
  – think about which types are the same or different
  – the compiler will help you find errors

• It will become easier with practice to write generic from the start