class NewSet<T> implements Set<T> {

    // rep invariant:
    // non-null, contains no duplicates
    // ...
    List<T> theRep;
    T lastItemInserted;

    ...

}"
Generics and subtyping

```java
interface List<T> extends Collection<T> {
    boolean add(T elt);
    T get(int index);
}
```

Java subtyping is **invariant** with respect to generics
- Not covariant and not contravariant
- Neither `List<Number>` nor `List<Integer>` subtype of other
Consequences of invariant subtyping

class Utils {
    static double sumList(List<Number> lst) {
        double result = 0.0;
        for (Number n : lst) {
            result += n.doubleValue();
        }
        return result;
    }
}

Cannot pass List<Double> to this method!
   - List<Double> is not a subtype of List<Number>
Generic Methods

class Utils {
    static <T extends Number>
    double sumList(List<T> lst) {
        double result = 0.0;
        for (Number n : lst) { // T also works
            result += n.doubleValue();
        }
        return result;
    }
}

*Can* now pass List<Double> to this method
- Java can see that this is safe by checking the method body
- generic methods work around limitations of generic classes
Where are we?

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

• Now:
  – generics and subtyping
  – generic methods [not just using type parameters of class]
  – 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
More verbose first

Last Time:
- how to use *type bounds* to write reusable code despite invariant subtyping
- elegant technique using generic methods
- general guidelines for making code as reusable as possible
  • (though not always the most important consideration)

Today: *Java wildcards*
- essentially provide the same expressiveness
- *less verbose*: No need to declare type parameters that would be used only once
- *better style* because Java programmers recognize how wildcards are used for common idioms
  • easier to read (?) once you get used to it
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);
```

Can pass a `List<Integer>` to `addAll` for a `Set<Number>`
- `List<Integer>` is a subtype of `Collection<Integer>`
- `Collection<Integer>` is allowed above
  - have `T = Integer` and `E = Number`
Where are we?

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

• Now:
  – generics and subtyping
  – generic methods [not just using type parameters of class]
  – 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
Examples

[Compare to earlier version]

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

- More idiomatic (but equally powerful) compared to
  ```java
  <T extends E> void addAll(Collection<T> c);
  ```

- More powerful than `void addAll(Collection<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

    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

Object

Number

Integer

Object

Number

Integer
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

```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)`
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 ? 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:
    <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?
\[
\begin{align*}
lei &= \text{new ArrayList<Object>();} \\
lei &= \text{new ArrayList<Number>();} \\
lei &= \text{new ArrayList<Integer>();} \\
lei &= \text{new ArrayList<PositiveInteger>();} \\
lei &= \text{new ArrayList<NegativeInteger>();} \\
\end{align*}
\]
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;

Which of these is legal?
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<? 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

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<? 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>;

CSE 331 Fall 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>;

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\(<? \ super \ Integer> \) \( lsi \);

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?
\[ lsi = \text{new ArrayList<Object>}; \]
\[ lsi = \text{new ArrayList<Number>}; \]
\[ lsi = \text{new ArrayList<Integer>}; \]
\[ lsi = \text{new ArrayList<PositiveInteger>}; \]
\[ lsi = \text{new ArrayList<NegativeInteger>}; \]
Legal operations on wildcard types

First, which of these is legal?

```java
List<? super Integer> lsi;
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?

```java
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[\texttt{]}]} and \texttt{Type2[\texttt{]}]} 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 ¯\_(ツ)_/¯
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
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

```
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 $\text{add(String)}$ but the bytecodes say...

```java
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
11: invokeinterface #12, 2 // InterfaceMethod java/util/List.add:(Ljava/lang/Object;)Z
16: pop
17: aload_0
18: ldc    #18 // String def
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<>(); // 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);
    }
    ...
}
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);
  }
  ...
}

More erasure: At run time, 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);
    }

    // Leave it to here to “do the right thing” if this and n differ on element type
}

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

Node<? extends Object>
Generics and arrays

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