Generics

CSE 331: Software Design and Implementation, Autumn 2026

Your job is to deliver code that you know works.

TODO

- Homework 3: due tonight, 11:59pm
- Section Tuesday: equality and generics
- Midterm: Friday, November 13, in lecture
  - Covers through today's lecture, not Monday's
  - Mock midterm: Tuesday, November 10, 4:30 to 6:20pm, CSE2 G01
  - Midterm review: Thursday, November 12, 4:30 to 6:20pm, CSE2 G01

A queue of anything

- **Problem**: we want a queue of Strings and a queue of Integers
  - **Solution 1**: copy-paste the class
  - **Solution 2**: use Object to stand for "anything"
  - **Solution 3**: a generic class, so the client chooses the element type

ListQueue, with Object for String

```
19  public class ObjectQueue {
        ...
34      private List<Object> queue = new ArrayList<Object>();
        ...
40      public void join(Object o) {
41          if (queue.contains(o)) {
42              throw new IllegalArgumentException(o + " is in the queue");
43          }
44          queue.add(o);
45      }
46
47      public Object next() {
48          if (queue.isEmpty()) {
49              throw new NoSuchElementException("the queue is empty");
50          }
51          return queue.remove(0);
52      }
```

Using `ObjectQueue`

```
20  public static void main(String[] args) {
21      // The queue for office hours. Every element is a NetID.
22      ObjectQueue queue = new ObjectQueue();
23      queue.join("ada");
24      queue.join(1234567);
25      queue.join("eve");
26
27      // Later: help each student in turn.
28      while (queue.size() > 0) {
29          String netid = (String) queue.next();
30          System.out.println("now helping " + netid);
31      }
32  }
```

- `(String)` is a **cast**. Why is it needed?
- What does this program do?
  - Compile time error? Run time error? Runs normally and prints what?

java ObjectQueue

```
$ javac ObjectQueue.java

$ java ObjectQueue
now helping ada
Exception in thread "main" java.lang.ClassCastException:
    class java.lang.Integer cannot be cast to class java.lang.String (…)
 at ObjectQueue.main(ObjectQueue.java:29)
```

```
22  ObjectQueue queue = new ObjectQueue();
23  queue.join("ada");
24  queue.join(1234567);
25  queue.join("eve");
26
27  // Later: help each student in turn.
28  while (queue.size() > 0) {
29      String netid = (String) queue.next();
```

- No message: the code compiles without error
- A run time error. Today's goal: make it a compile time error
- The mistake: `join(1234567)`
- The error is reported at run time, on a different line

"Every element is a NetID"

```
21   // The queue for office hours. Every element is a NetID.
22   ObjectQueue queue = new ObjectQueue();
          ...
29   String netid = (String) queue.next();
```

- **Three ways to know**: testing, tools, reasoning
- `javac`'s type checker is a tool for checking **type safety**:
  - a variable refers only to objects of its declared type
- `(String)` says "trust me" at compile time: checked at run time instead
  - Caught by a test only if `next` returns a non-String
- We want to check this claim at compile time instead
  - Then this mistake is caught even without perfect testing

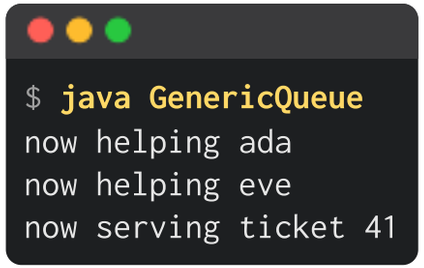
A generic class

```
67  /** A queue kept as one list of the elements in it. */
68  class ListQueue<E> implements Queue<E> {
69      private List<E> queue = new ArrayList<E>();
70
71      public int size() {
72          return queue.size();
73      }
```

- <E> after the name declares a **type parameter**
- Inside the class, E stands for a type the author does not know
- To make the String queue generic: replace String with E everywhere

Using a generic class

```
20  Queue<String> queue = new ListQueue<String>();
21  queue.join("ada");
22  queue.join("eve");
23
24  // Later: help each student in turn.
25  while (queue.size() > 0) {
26      String netid = queue.next();
27      System.out.println("now helping " + netid);
28  }
29
30  // The same class again, for a queue of ticket numbers.
31  Queue<Integer> tickets = new ListQueue<Integer>();
32  tickets.join(41);
33  tickets.join(42);
34  int ticket = tickets.next();
35  System.out.println("now serving ticket " + ticket);
```



- `Queue<String>`: String is the **type argument**, so `E` is String
- `next` returns a String: no cast
- `Queue<Integer>` for the bakery (not `int`!)

Mistake is caught at compile time

- Suppose we try to add an integer to a `Queue<String>`
  - Recall: with `ObjectQueue`, `javac` accepted this!

```
20  Queue<String> queue = new ListQueue<String>();
21  queue.join("ada");
22  queue.join(1234567);
23  queue.join("eve");
```

```
$ javac GenericQueue.java
GenericQueue.java:22: error: incompatible types:
    int cannot be converted to String
        queue.join(1234567);
                   ^
⋮
1 error
```

- This time the error is on the right line.
- Before: caught only by testing or reasoning. Now: by a tool, `javac`

What the code may do with an E

```
75  public void join(E e) {
76      if (queue.contains(e)) {
77          throw new IllegalArgumentException(e + " is in the queue");
78      }
79      queue.add(e);
80  }
```

- The code does not know what `E` is: String here, Integer there
- Only methods from Object may be called
- `join` compares by `equals` (in `contains`), and calls `toString`

A generic method

```
38  public static <P> P cheapest(List<P> list) {
39      P best = list.get(0);
40      for (P p : list) {
41          if (p.getPrice() < best.getPrice()) {
42              best = p;
43          }
44      }
45      return best;
46  }
```

- `<P>` before the return type: a **generic method**
- `p.getPrice()`: does `javac` accept this call?
- No: `P` could be any type, and Object has no `getPrice`
- What if the code needs other methods?

A bound

```
38  public static <P extends Product> P cheapest(List<P> list) {
39      P best = list.get(0);
40      for (P p : list) {
41          if (p.getPrice() < best.getPrice()) {
42              best = p;
43          }
44      }
45      return best;
46  }
```

- `extends Product` is a **bound** on `P`
  - The client may choose only Product, or a Java subtype
  - The body may use a `P` as a Product: `getPrice`
- Now `javac` accepts it

Two calls, two types

```
19  List<Product> cart = new ArrayList<Product>();
20  cart.add(new Product("skittles", 2.0));
        ...
24  List<SaleProduct> deals = new ArrayList<SaleProduct>();
25  deals.add(new SaleProduct("chips", 3.0, 0.5));
        ...
28  Product p = cheapest(cart);
29  System.out.println("cheapest(cart): " + p);
30  SaleProduct s = cheapest(deals);
31  System.out.println("cheapest(deals): " + s);
```



- `cheapest(cart)`: `P` is Product
- `cheapest(deals)`: `P` is SaleProduct, and no cast
- `javac` works `P` out from the argument of each call

A question: lists and subtypes

```
25        List<SaleProduct> deals = new ArrayList<SaleProduct>();
          ...
29        System.out.println("total(cart): $" + total(cart));
30        System.out.println("total(deals): $" + total(deals));
          ...
39    public static double total(List<Product> cart) {
```

- Subtyping: `javac` accepts a SaleProduct where a Product is expected
- **Half sheet, question 1** (2 minutes)
  - `total(deals)`: does `javac` accept a `List<SaleProduct>` here?
- **Ask the compiler**: write a method that takes an A, hand it a B, compile

What javac says to total(deals)

```
$ javac Carts.java
Carts.java:30: error: incompatible types:
    List<SaleProduct> cannot be converted to List<Product>
        System.out.println("total(deals): $" + total(deals));
                                                  ^
⋮
1 error
```

- A `List<SaleProduct>` is not a Java subtype of `List<Product>`
- The is-a test said yes. It misled us before: Square and Rectangle

Substituting generic types

- A **behavioral subtype** must have a stronger specification
- `get` returns a Product. In a `List<SaleProduct>`: a SaleProduct
  - So, by its type, the postcondition of `get` is no weaker
- `add` takes any Product. In a `List<SaleProduct>`: only a SaleProduct
  - So, by its type, the precondition of `add` is **stronger**
- Not a behavioral subtype, so `javac` is right to refuse

Why `javac` is right to refuse

- `total` never adds. So why is `javac` right to refuse `total(deals)`?
- `javac` checks a call against the parameter's type, not the body
- The code of `total` might change later to add a Product

The same mistake, no method in the way

```
21  // The deals: sale products only.
22  List<SaleProduct> deals = new ArrayList<SaleProduct>();
23  deals.add(new SaleProduct("chips", 3.0, 0.5));
24
25  // A second name for the same list, declared a List<Product>.
26  List<Product> all = deals;
27  all.add(new Product("skittles", 2.0));
28
29  SaleProduct second = deals.get(1);
```

- `all = deals`: a second name for the one list
  - A **substitution**, like passing `deals` as a parameter
- `all.add(...)`: the skittles go in, through `all`
- `deals.get(1)`: they come out of `deals`, called a SaleProduct

What `javac` says to `ListCarts`

```
$ javac ListCarts.java
ListCarts.java:26: error: incompatible types:
    List<SaleProduct> cannot be converted to List<Product>
        List<Product> all = deals;
                            ^
1 error
```

- `all.add` and `deals.get` each do what their variable's type allows
  - so `javac` refuses the second name

Invariance of generic types

Rule: the type arguments must be the same

For a generic class or interface `C`,

`C<B>` is accepted where `C<A>` is expected only if

A and B are the same type.

- `List<SaleProduct>` is not a Java subtype of `List<Product>`
- `ArrayList<Product>` *is* a Java subtype of `List<Product>`

`total` with a type parameter

```
20      List<Product> cart = new ArrayList<Product>();
        ...
25      List<SaleProduct> deals = new ArrayList<SaleProduct>();
        ...
29      System.out.println("total(cart): $" + total(cart));
30      System.out.println("total(deals): $" + total(deals));
        ...
39  public static <P extends Product> double total(List<P> cart) {
40      double sum = 0.0;
41      for (Product p : cart) {
42          sum = sum + p.getPrice();
43      }
44      return sum;
45  }
```

```
$ java Carts
total(cart): $3.0
total(deals): $2.0
:
```

- `<P extends Product>`: `cart` is a list of some one type `P`
  - which is Product or a Java subtype of Product
- Now `javac` accepts both calls

How javac checks the body

```
39  public static <P extends Product> double total(List<P> cart) {
40      double sum = 0.0;
41      for (Product p : cart) {
42          sum = sum + p.getPrice();
43      }
44      return sum;
45  }
```

- `for (Product p : cart)`: an element of `cart` has the type `P`
  - `P` is Product or a Java subtype, so `p` may hold it

A wildcard: a type parameter with no name

```
39  public static double total(List<? extends Product> cart) {
40      double sum = 0.0;
41      for (Product p : cart) {
42          sum = sum + p.getPrice();
43      }
44      return sum;
45  }
```

- `total` uses `P` only once, so it needs no name
- `?` is a **wildcard**: a type the method does not name
- `<? extends Product>`: `cart` is a list of some one type
  - which is Product or a Java subtype of Product

`total` may no longer add

```
39  public static double total(List<? extends Product> cart) {
40      cart.add(new Product("bag", 0.1));
41      double sum = 0.0;
```

```
$ javac Carts.java
Carts.java:40: error: incompatible types:
    Product cannot be converted to CAP#1
        cart.add(new Product("bag", 0.1));
                 ^
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Product from capture of ? extends Product
⋮
1 error
```

- CAP#1 is javac's name for the type that ? stands for
  - It might be SaleProduct: so javac refuses the call

The other direction

```
51  public static void addMints(List<? super SaleProduct> list) {
52      list.add(new SaleProduct("mints", 1.0, 0.5));
53  }
```

- A method that only adds asks for less the other way
- `<? super SaleProduct>`: `list` is a list of some one type
  - which is SaleProduct or a Java supertype of SaleProduct
  - A `List<SaleProduct>`, a `List<Product>`, or a `List<Object>`
- A SaleProduct may be added to each of the three
- Through `list`, `get` returns only an Object

Half sheet, question 2 (4 minutes)

`foo` takes these parameters

```
                                  Object  o
                                 Product  p
                             SaleProduct  s
                  List<? extends Product>  from
            List<? super SaleProduct>  to
```

Statement in `foo`	Y/N
`p = from.get(0)`	_____
`to.add(s)`	_____
`o = to.get(0)`	_____
`p = to.get(0)`	_____
`to.add(o)`	_____
`from.add(s)`	_____

- For each call, say whether `javac` accepts it

Being the compiler

- `from` is a `List<? extends Product>`
- `to` is a `List<? super SaleProduct>`

Statement in `foo`	Y/N	Why
`p = from.get(0)`	**Y**	every element is a Product
`to.add(s)`	**Y**	every list it may be takes a SaleProduct
`o = to.get(0)`	**Y**	every element is an Object
`p = to.get(0)`	**N**	`to` may be a `List<Object>`
`to.add(o)`	**N**	`to` may be a `List<SaleProduct>`
`from.add(s)`	**N**	`from` may be a `List<ClearanceProduct>`

- ClearanceProduct: a subclass of SaleProduct, written later

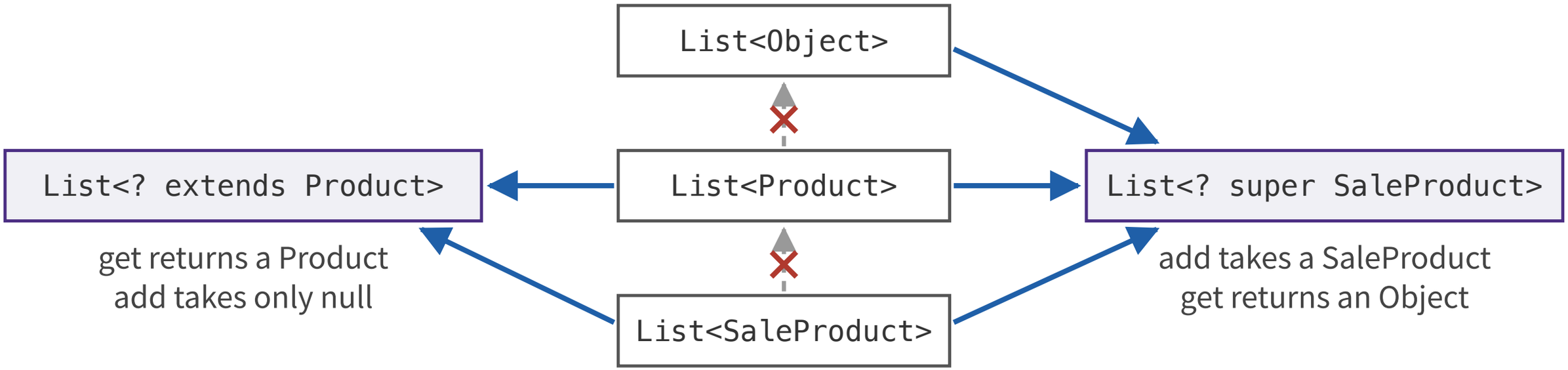
The extremes: Object and null

- `from` is a `List<? extends Product>`
- `to` is a `List<? super SaleProduct>`

Statement in `foo`	Y/N	Why
`o = from.get(0)`	**Y**	every element is an Object
`from.add(null)`	**Y**	`null` may go in a list of any type
`o = to.get(0)`	**Y**	every element is an Object
`to.add(null)`	**Y**	`null` may go in a list of any type

- Through `from`, `add` accepts only `null`
- Through `to`, `get` returns only an Object

Which list is accepted where



- An arrow: `javac` accepts the first where the second is expected
- A crossed arrow: `javac` refuses, since the type arguments differ

Get through extends, add through super

Rule: get through extends, add through super

If a method only gets from a list, declare it a `List<? extends A>`.

If a method only adds to a list, declare it a `List<? super B>`.

If a method does both, declare it a `List<A>`.

- The JDK's `addAll` takes a `Collection<? extends E>`

Naftalin and Wadler, the Get and Put Principle; Bloch, PECS: producer extends, consumer super

What about arrays?

```
16  // The deals: sale products only.
17  SaleProduct[] deals = new SaleProduct[2];
18  deals[0] = new SaleProduct("chips", 3.0, 0.5);
19
20  // A second name for the same array, declared a Product[].
21  Product[] all = deals;
22  all[1] = new Product("skittles", 2.0);
23
24  SaleProduct second = deals[1];
25  System.out.println("second deal: " + second);
```

- `all = deals`: arrays are like lists, right? Does `javac` refuse it?
- No! `javac` accepts it: `SaleProduct[]` is a Java subtype of `Product[]`
- `all[1] = new Product(...)`: what happens?

java ArrayCarts

```
$ java ArrayCarts
Exception in thread "main" java.lang.ArrayStoreException: Product
 at ArrayCarts.main(ArrayCarts.java:22)
```

- An array keeps the class it was made to hold
  - and Java checks every store, when the program runs
- With lists, `javac` refused the second name: the program never ran
- With arrays, the program fails on the run that makes the bad store

When the program runs

```
21  Queue<String> names = new ListQueue<String>();
22  Queue<Integer> tickets = new ListQueue<Integer>();
23  System.out.println("names:   " + names.getClass());
24  System.out.println("tickets: " + tickets.getClass());
25
26  Object o = tickets;
27  System.out.println("Queue<?>: " + (o instanceof Queue<?>));
```

- What do the two calls of `getClass` print?

java Erasure

```
$ java Erasure
names:   class ListQueue
tickets: class ListQueue
Queue<?>: true
```

- `javac` leaves type arguments out of the code that runs: **type erasure**
  - So both queues have the class `ListQueue`
- `o instanceof Queue<String>`: refused
  - In a generic class, `equals` tests `instanceof Queue<?>`
- `new E[10]`: refused. `ListQueue` writes no array: it keeps a `List<E>`

Conclusion

- **A type argument is a claim about every element, and `javac` checks it**
- A generic class leaves a type open: `class ListQueue<E>`
- A bound says what the code needs: `<P extends Product>`
- A `List<SaleProduct>` is not a `List<Product>`: `add` asks for more
- A wildcard asks for less: get through `extends`, add through `super`
- Arrays of objects: Java checks each store when the program runs

Coming up

- **Monday**: storing data so it does not disappear on a crash
- **Section Tuesday**: equality and generics
- Midterm: Friday, November 13, in lecture
  - Mock midterm: Tuesday, November 10, 4:30 to 6:20pm, CSE2 G01
  - Midterm review: Thursday, November 12, 4:30 to 6:20pm, CSE2 G01
- Homework 3: due tonight, 11:59pm
- Hand in your half sheet at the door