Subtyping

CSE 331: Software Design and Implementation, Autumn 2026

Your job is to deliver code that you know works.

TODO

- Reading 3: due tonight, 11:59pm
- Homework 3: due Friday, November 6, 11:59pm
- Section Tuesday: exceptions and subtyping

Must the argument's type match?

```
$ javac Products.java
Products.java:17: error: incompatible types:
    String cannot be converted to Product
        cart.add("skittles");
                 ^
⋮
1 error
```

- `cart` is a `List<Product>`, so `add` expects a Product: a String is refused
- Does the argument's type have to equal the parameter's type? **No!**

Product and SaleProduct

```
36  class Product {
37      private String name;
38      private double price;
```

```
63  class SaleProduct extends Product {
64      private double discount;  // the fraction taken off, 0 to 1
```

```
16  List<Product> cart = new ArrayList<>();
17  cart.add(new Product("skittles", 2.0));
18  cart.add(new SaleProduct("m&ms", 2.0, 0.5));
```

- A Product where a Product is expected
- A SaleProduct where a Product is expected

Half sheet, question 1

```
15  public static void main(String[] args) {
16      List<Product> cart = new ArrayList<>();
17      cart.add(new Product("skittles", 2.0));
18      cart.add(new SaleProduct("m&ms", 2.0, 0.5));
19      for (Product p : cart) {
20          System.out.println(p);
21      }
22      System.out.println("total: $" + total(cart));
23  }
```

```
55  /** @return this product's name and price, as "name: $price" */
56  @Override
57  public String toString() {
58      return name + ": $" + price;
59  }
```

```
74   */
75  @Override
76  public double getPrice() {
77      return super.getPrice() * (1 - discount);
78  }
```

- What does this program do?
  - A compile time error
  - A run time error
  - Runs and prints

The run

```
$ java Products
skittles: $2.0
m&ms: $2.0
total: $3.0
```

- The second `add` compiled: a SaleProduct went as a Product
- The m&ms should be half off, but the line shows the full price!
- The total is right: `total` called `getPrice`, and SaleProduct's ran

Java subtype

Definition: Java subtype

A type B is a **Java subtype** of a type A if the compiler accepts a B where an A is expected.

The compiler accepts it when B reaches A by a chain of `extends` and `implements`.

```
36  class Product {
```

```
63  class SaleProduct extends Product {
```

```
17  cart.add(new Product("skittles", 2.0));
18  cart.add(new SaleProduct("m&ms", 2.0, 0.5));
```

- The second add is allowed: SaleProduct extends Product directly
- The rule: Java checks an argument for subtyping, not for type equality

Review: subclassing is copying code

Definition: Subclassing

`class B extends A` makes B a **subclass** of A. B **inherits** A's fields and methods: it has them without writing them out, except the methods it overrides.

```
55  /** @return this product's name and price, as "name: $price" */
56  @Override
57  public String toString() {
58      return name + ": $" + price;
59  }
```

```
26  public static double total(List<Product> cart) {
27      double sum = 0.0;
28      for (Product p : cart) {
29          sum = sum + p.getPrice();
30      }
31      return sum;
32  }
```

- Benefits: less redundancy; a change made once reaches every subclass
- Potential downside: harder to reason about your program!

Four words

- **Inheritance**: a class gets another's fields and methods for free
- **Java subclassing**: `extends`, inheritance and Java subtyping at once
- **Java subtyping**: the compiler accepts a B where an A is expected
- **Behavioral subtyping**: a B keeps every promise that A makes
  - The one the lecture's question is about; defined next
- Java subtyping without inheritance: ListQueue implements HelpQueue
- `total` never read SaleProduct's code, so what did it rely on?

The question for the hour

- Every argument about a call used the called method's specification
  - Never its code (Lecture 2)
- `total` was argued correct against Product, and a SaleProduct arrived
  - `total` never read SaleProduct's code, and could not have
- **How to preserve reasoning across Java subtypes?**
- What would B have to promise to keep `total`'s argument right?

Half sheet, question 2 (3 minutes)

```
48
49  /** Finds a value in an array, if it is there. */
50  interface Finder {
51      /**
52       * @return the smallest index i such that a[i] == x, or -1 if x
53       *     does not occur in a
54       */
55      int indexOf(int[] a, int x);
56  }
```

```
58  /** Finds a value known to be in an array. */
59  interface PresentFinder extends Finder {
60      /**
61       * @requires x occurs in a
62       * @return the smallest index i such that a[i] == x
63       */
64      int indexOf(int[] a, int x);
65  }
```

```
16  public static boolean contains(Finder finder, int[] a, int x) {
17      return finder.indexOf(a, x) != -1;
18  }
```

- A client of Finder, argued correct against Finder's specification
- What happens on `contains(assuming, a, 7)` if 7 is not in the array?

Behavioral subtype

Definition: Behavioral subtype

A type B is a **behavioral subtype** of a type A if:
every object that satisfies B's specification also satisfies A's.
In other words: for each method of A, that method exists in B, and
the B version's specification is at least as strong as the A version's.

- The client's argument used A's specification alone, so a B must imply it
- First of Any: same precondition, and *the smallest index* is *an index*
- PresentFinder of Finder: no. It requires x to occur; Finder promises -1

The argument against Finder, and the run

```
$ java IndexOf
3 in a: true
7 in a: false
3 in a: true
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
    Index 4 out of bounds for length 4
 at ScanAssuming.indexOf(IndexOf.java:83)
 at IndexOf.contains(IndexOf.java:17)
 at IndexOf.main(IndexOf.java:27)
```

- No index is -1, so `contains` is right for every Finder
- Each satisfies its own specification; the fault is the `extends`

The Liskov substitution principle

- "B is a subtype of A" should mean: a B may be used wherever an A is expected

Barbara Liskov, "Data Abstraction and Hierarchy", 1987; Liskov and Wing, 1994.

What the compiler checks

```
21  interface Source {
22      /**
23       * @return a list of the indexes at which x occurs in this
24       *     sequence, in increasing order
25       * @throws IOException if the numbers could not be read
26       */
27      List<Integer> positions(int x) throws IOException;
28  }
```

```
30  /** A sequence of numbers kept in an array in memory. */
31  interface ArraySource extends Source {
32      /**
33       * @return a new ArrayList of the indexes at which x occurs in
34       *     this sequence, in increasing order
35       */
36      ArrayList<Integer> positions(int x);
37  }
```

- `javac` checks an override's **return type**: the same or a subtype
- And its **checked exceptions**: the same, fewer, or narrower
- The parts of a postcondition the types carry; nothing in the Javadoc
- A wider parameter type is a new **overload**, not an override

331 rule about subtyping

Rule: declare only behavioral subtypes

Only declare Java subtypes that are actually behavioral subtypes.

- The compiler cannot check this: it reads the types, not the Javadoc
- The payoff: every client gets to reuse its reasoning on every subtype

Is a Square a Rectangle?

```
14  /**
15   * @modifies r
16   * @effects sets r's width to w; r's height is unchanged
17   * @return the area of r afterward
18   */
19  public static int widen(Rectangle r, int w) {
20      int h = r.getHeight();
21      r.setWidth(w);
22      return w * h;
23  }
```

```
82  /**
83   * @modifies this
84   * @effects sets this square's width and height to w
85   */
86  @Override
87  public void setWidth(int w) {
88      super.setWidth(w);
89      super.setHeight(w);
90  }
```

- Every square is a rectangle, so `class Square extends Rectangle`
- `main` widens a 2 by 3 rectangle, then a 2 by 2 square, to width 5
- Each line prints what `widen` returned, and the area afterward
- What does the square's line print? Which method is to blame?

Not a behavioral subtype

```
57  /**
58   * @modifies this
59   * @effects sets this rectangle's width to w; its height is
60   *     unchanged
61   */
62  public void setWidth(int w) {
63      width = w;
64  }
```

```
82  /**
83   * @modifies this
84   * @effects sets this square's width and height to w
85   */
86  @Override
87  public void setWidth(int w) {
88      super.setWidth(w);
89      super.setHeight(w);
90  }
```

- Nobody's code is wrong: each satisfies its own specification
- Rectangle: *its height is unchanged*. Square: *width and height to w*
- Incomparable: Square is a Java subtype and not a behavioral one
- `extends Rectangle` breaks the rule; the is-a test is not the test

Two ways out

```
49  /**
50   * @return a new rectangle whose width is w and whose height is
51   *     this rectangle's height
52   */
53  public Rectangle withWidth(int w) {
54      return new Rectangle(w, height);
55  }
```

```
96  /**
97   * @modifies this
98   * @effects sets this square's width and height to s
99   */
100 public void setSide(int s) {
101     rect.setWidth(s);
102     rect.setHeight(s);
103 }
```

- **Immutable shapes**: `withWidth` returns a new Rectangle
  - A promise a Square keeps: it inherits every specification whole
- **Hold a Rectangle**: Square keeps one in a field and forwards to it
  - Composition: reuse without subtyping; `widen(s)` is refused
- The JDK's Properties extends Hashtable, and also made this mistake!

Conclusion

- **How to preserve reasoning across Java subtypes?**
- **Ensure Java subtypes are behavioral subtypes.**
  - Every object that satisfies B's specification also satisfies A's
  - Compilers check types, not Javadoc: *we* check this by reasoning
- Mutable interfaces have subtle subtyping properties

Coming up

- Reading 3: due tonight, 11:59pm
- **Monday**: Reading 3 discussion; "modern" JavaScript front ends
- **Wednesday**: equality
- **Section Tuesday**: exceptions and subtyping
- Homework 3: due Friday, November 6, 11:59pm
- Hand in your half sheet at the door