



INF3110 – Programming Languages Object Orientation and Types, part II

Eyvind W. Axelsen

eyvinda@ifi.uio.no | [@eyvindwa](https://twitter.com/eyvindwa) 

<http://eyvinda.at.ifi.uio.no>

Slides adapted from previous years' slides
made by Birger Møller-Pedersen

birger@ifi.uio.no

Object Orientation and Types

Lecture I

- From predefined (simple) and user-defined (composite) types
 - via
- Abstract data types
 - to
- Classes
 - Type compatibility
 - Subtyping <> subclassing
 - Class compatibility
 - Covariance/contravariance
 - Types of parameters of redefined methods

Lecture II - Today

- Type systems (very briefly!)
- Polymorphism
 - Generics
- "Advanced" oo concepts
 - Specialization of behavior?
 - Multiple inheritance - alternatives
 - Inner classes
- **Mandatory exercise II out now!**
 - **Deadline** October 26th

Repetition

Remember: *syntax* (program text) and *semantics* (meaning) are two separate things.

Types and **type systems** help to ascribe *meaning* to programs:

- What does `"Hello" + " World"` mean?
- Which operation is called when you write `System.out.println("INF3110")`?
- What does the concept of a `Student` entail?

Repetition - What is a type?

- A set of values that have a set of operations in common
 - 32 bit integers, and the arithmetic operations on them
 - Instances of a Person class, and the methods that operate on them
- How is a type *identified*?
 - By its *name* (e.g. Int32, Person, Stack): nominal type checking
 - By its *structure* (fields, operations): structural type checking
- Does this cover everything a type might be? No.
 - Alternative definition of “type”: *A piece of the program to which the type system is able to assign a label.*
 - *(but don't worry too much about this now)*

Repetition - Classification of types

- Predefined, simple types (not built from other types)
 - boolean, integer, real, ...
 - pointers, pointers to procedures
 - string
- User-defined simple types
 - enumerations, e.g. `enum WeekDay { Mon, Tue, Wed, ... }`
- Predefined composite types
 - Arrays, lists/collections (in some languages)
- User-defined, composite types
 - Records/structs, unions, abstract data types, classes
- Evolution from simple types, via predefined composite types to user-defined types that reflect parts of the application domain.

What is a type system?

- One possible definition
 - *“A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute” [Pierce, 2002]*

What is a type system?

- One possible definition
 - “A *type system* is a tractable syntactic method for proving the absence of certain **program** behaviors by classifying phrases according to the kinds of values they compute” [Pierce, 2002]
 - We are interested in type systems in relation to *programs* and *programming languages*, and not other kinds of type systems
 - The idea of type systems (or *type theory*) predates programming languages, and type theory has other applications as well

What is a type system?

- One possible definition
 - “A type system is a tractable *syntactic method* for proving the absence of certain program behaviors by *classifying phrases according to the kinds* of values they compute” [Pierce, 2002]
 - The type system deals with syntactic phrases, or terms, in the language, and assigns labels (types) to them.
 - This applies to static type systems
 - Dynamic type systems, on the other hand, label and keep track of data at *runtime*.

What is a type system?

- One possible definition
 - “A type system is a tractable syntactic method for *proving the absence of certain program behaviors* by classifying phrases according to the kinds of values they compute” [Pierce, 2002]
 - The goal of the type system is to prove the absence of certain undesirable behaviors
 - There are hard limits to what kind of undesirable behaviors a type system can prove things about, e.g. (non)termination
 - “The fundamental purpose of a type system is to prevent the occurrence of execution errors during the running of a program” [Cardelli, 2004]
 - But what constitutes an execution error? `ArrayIndexOutOfBoundsException`? `NullPointerException`?

What is a type system?

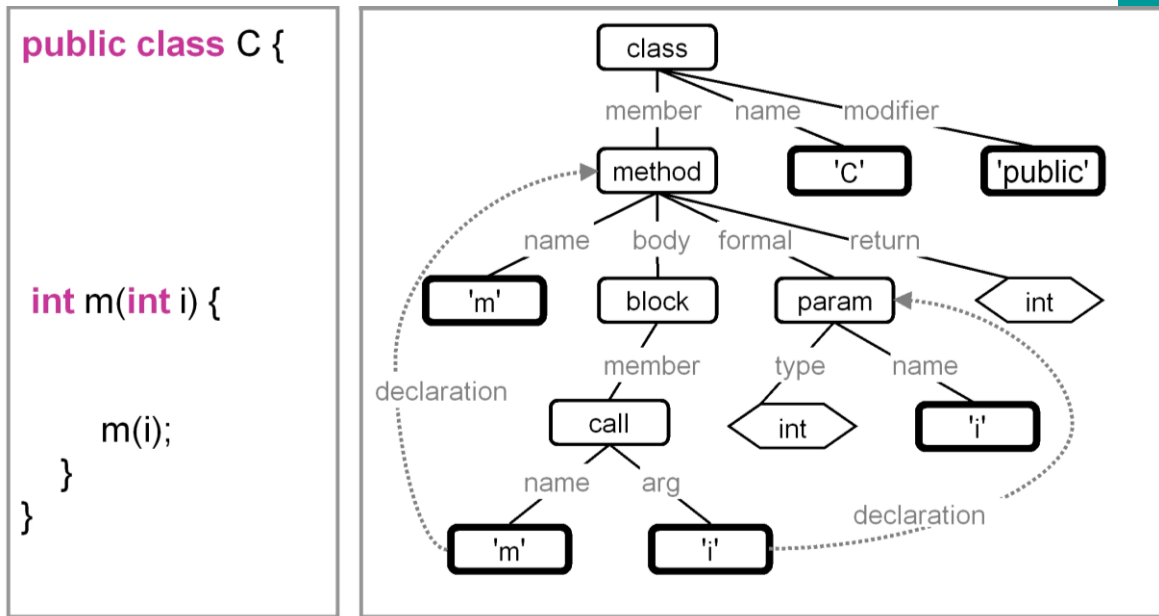
- One possible definition
 - “A type system is a *tractable* syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute” [Pierce, 2002]
 - In order to attain its goal, the type system should preferably be computationally tractable
 - Tractable = polynomial time, with regard to length of the program
 - In practice, the degree of the polynomial should not be too high

Main categories for programming language type systems

- Untyped
 - There are no types (e.g. everything is just a bit pattern)
 - Or, if you will, everything has the same single type
- Statically typed
 - Types checking is a syntactic process at compile-time
 - Rejects programs that do not type check before they can run
- Dynamically typed (or: dynamically checked)
 - Types are checked at runtime
 - By a runtime system, or
 - By code inserted by a compiler
- Categories are not mutually exclusive
 - Most “real-world” languages are somewhere in between, with elements from more than one category
 - There is a *tension* between safety and expressivity that must be resolved by the language/type system designer

Static type systems

- Types are assigned to syntactical elements of a program (prior to running it)
 - Types annotations can be specified explicitly in the source code by the programmer, “ALGOL-style”, as in Java, C++, etc
 - Or they can be inferred by the compiler, as in ML, Haskell, etc, Hindley-Milner style
- An AST (*abstract syntax tree*) is typically created from the source code using the language’s grammar
 - Some of the nodes in the tree will be declarations of types, or type annotations
- Uses the language’s semantics to establish relationships between expressions and types
 - Thus type checking the program
 - Checks *structural* or *nominal* conformance according to language semantics



Static type systems [cont.]

- Static type systems are always *conservative*
 - They cannot (in general) prove the presence of errors, only the absence of certain bad behaviors
 - They are therefore bound to potentially reject “correct” programs

```
if( < complex runtime condition that always evaluates to true > )  
  < valid code >  
else  
  < type error >
```

- Mainstream languages typically concede to *tradeoffs* between flexibility and type safety
 - E.g. covariant array conversions, null-references, runtime contract checking
 - Escape hatches to circumvent the type system:
 - `Unchecked_` constructs in Ada
 - `unsafe { ... }` in C#
 - `Obj.magic` in Ocaml
 - “*license to kill [the type system]*” – anonymous stackoverflow.com user
 - Foreign Function Interfaces in most languages, e.g. ML, JavaScript, Python, Java, etc

Dynamically typed languages

- Type checks at *runtime*
 - As long as the receiver supports the requested operation, everything is fine
 - Errors due to type-incorrect operations will be caught* at runtime
 - * if the language is *safe*, otherwise, anything could happen
- Never need to reject a correct program
 - But may indeed end up running many faulty ones
 - Extensive testing/TDD *may* find the errors that a compiler would otherwise have found
 - A test suite can find an *upper* bound on correctness, while (static) type systems find a *lower* bound

Dynamically typed languages [cont.]

- Freedom of expression where static type system cannot (at present?) correctly type the program

- Can have meta-object protocols with sophisticated behavior

- Responding to method calls or not depending on runtime environment, e.g.:

```
def methodMissing(name, args) {  
    if(name.startsWith("get") && App.User.IsAuthorized())  
        return OtherClass.metaClass.invoke(name, args);  
    else  
        throw new MessageNotUnderstoodException();  
}
```

- Effortlessly create proxies at runtime

- Create and cache new methods from business rules defined by users, e.g. in an internal DSL

- Etc

- Classes and objects can be adapted at runtime

- Add or remove methods or fields, swap out classes, etc.

- Used a great deal by e.g. Flickr, Facebook and Gmail [Vitek 2009]

Mark Mannasse: “The fundamental problem *addressed* by a type theory [aka type system] is to ensure that programs have meaning.

*The fundamental problem **caused** by a type theory is that meaningful programs may not have meanings ascribed to them.*

*The **quest** for richer type systems results from this tension.”*
[as quoted by Pierce 2002, p 208]

Words of wisdom?

“Static typing is great because it keeps you out of trouble.

Dynamic typing is great because it gets out of your way”

– Zack Grossbart (author, blogger,)

Polymorphism – a single interface usable for instances of different types

- **Ad hoc polymorphism:** functions/methods with the same name that can be applied to different parameter types and arities
 - Typically called *overloading*
- **Parametric polymorphism:** "when the type of a value contains one or more *type variables*, so that the value may adopt any type that results from substituting those variables with concrete types"
[<https://wiki.haskell.org/Polymorphism>].
 - In OOP communities, this is typically called *generics*.
 - In FP communities, this is typically called just *polymorphism*.
- **Subtype polymorphism** (*subtyping*): an instance of a subtype can be substituted where a supertype is expected
 - In OOP communities, this is often simply referred to as *polymorphism*.

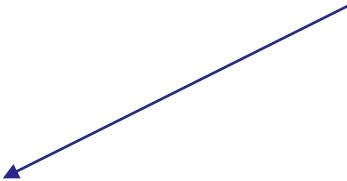
Generics/parametric polymorphism

- Type constructors, of types of types
 - E.g. `List<T>` can be used to construct `List<String>`, `List<Person>`, etc.
- Different languages offer different degrees of expressiveness
 - What can be said about `T`?
 - Can we constrain what it can be?
 - Can we be sure that whatever is in our `List<String>` is really only strings?
 - What about subtype hierarchies?
 - To which extent is the generic type type safe?
 - Can the generic type be analyzed on its own, independently of any use-cases?

Constraining type parameters

- C++ polymorphic sort function

C++ templates provide no support for constraints on template parameters



```
template <typename T>
void sort( int count, T arr[] ) {
    for (int i=0; i<count-1; i++)
        for (int j=i+1; j<count-1; j++)
            if (arr[j] < arr[i])
                swap(arr[i], arr[j]);
}
```

- What parts of the implementation depend on what property of T?
Usage, meaning and implementation of <

Java lists without and with generics

```
// create new list
List myIntList = new LinkedList();
// add an integer to the list
myIntList.add(new Integer(0));
// take an integer out from the list
Integer x = (Integer)myIntList.iterator().next();
```

```
// the same, this time with generic list
List<Integer> myIntList = new LinkedList<Integer>();
myIntList.add(new Integer(0));
Integer x = myIntList.iterator().next();
```

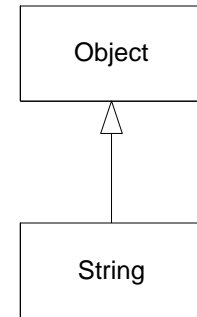
Generics and subtyping

- String subtype of Object ~~⇒~~ List<String> subtype of List<Object> ?

```
List<String> ls = new ArrayList<String>();  
List<Object> lo = ls;  
lo.add(new Object());  
String s = ls.get(0);
```

compile-time
error

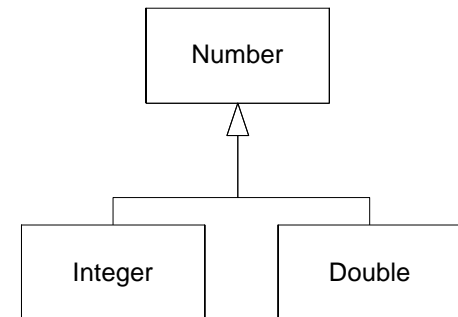
attempts to assign
an Object to a String



- Integer subtype of Number ~~⇒~~ List<Integer> subtype of List<Number> ?

```
List<Integer> ints = Arrays.asList(1,2);  
List<Number> nums = ints;  
nums.add(3.14);
```

compile-time
error



But look out! Arrays and subtyping

String subtype of Object → String[] subtype of Object[]?

```
String[] myStrings = new String [10];  
myStrings[0] = "Hello";  
myStrings[1] = "World!";
```

```
Object[] myObjects = myStrings;    // ???  
myObjects[3] = new Object(); // !!!
```

Try it out in Java and/or C#!

Unbounded polymorphism - Wildcards - I

Goal: Write code to print the elements of **any** collection:

```
void printCollection(Collection c) {  
    Iterator i = c.iterator();  
    for (k = 0; k < c.size(); k++)  
        System.out.println(i.next());  
}  
  
void printCollection(Collection<Object> c) {  
    for (Object e : c)  
        System.out.println(e);  
}  
  
void printCollection(Collection<?> c) {  
    for (Object e : c)  
        System.out.println(e);  
}
```

Collection<any type>
is **not** a subtype of
Collection<Object>

Collection<any type>
is a subtype of
Collection<?>

Bounded polymorphism - Wildcards - II

```
public abstract class Shape {  
    public abstract void draw(Canvas c);  
}
```

```
public class Circle extends Shape {  
    private int x, y, radius;  
    public void draw(Canvas c) { ... }  
}
```

```
public class Rectangle extends Shape {  
    private int x, y, width, height;  
    public void draw(Canvas c) { ... }  
}
```

```
public class Canvas {  
    public void draw(Shape s) { s.draw(this); }  
}
```

Write code to draw a list of any kind of shape →

Bounded polymorphism - Wildcards - III

```
// in class Canvas:  
public void drawAll(List<Shape> shapes) {  
    for (Shape s: shapes)  
        s.draw(this);  
}
```

Cannot draw e.g. a
list of rectangles or
circles

```
public void drawAll(List<? extends Shape> shapes) {  
    for (Shape s: shapes)  
        s.draw(this);  
}
```

- `List<S>` subtype of `List<? extends Shape>` for every `S` being a subtype of the (concrete) type `Shape`
- `List<S>` subtype of `List<? extends T>` for every `S` being a subtype of (the generic parameter) `T`

Generic methods

```
static void fromArrayToColl(Object[] a, Collection<?> c) {  
    for (Object o: a)  
        c.add(o);    // compile time error - why?  
}
```

```
static <T> void fromArrayToColl(T[] a, Collection<T> c) {  
    for (T o: a)  
        c.add(o);    // works - why?  
}
```

Generic methods - Copy collections

```
class Collections {  
    public static <T> void copy(  
        List<T> dest, List<? extends T> src) {  
        for (T s : src)  
            dest.add(s);  
        }  
    }  
}
```

-- or:

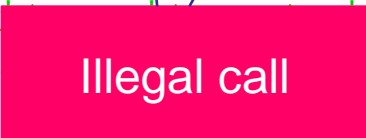
```
class Collections {  
    public static <T, S extends T> void copy(  
        List<T> dest, List<S> src) {  
        for (S s : src)  
            dest.add(s);  
        }  
    }  
}
```

Generic parameters – write to sinks

```
interface Sink<T> {
    flush(T t); // flush might for instance write stuff to disk
}

// writeAll writes everything in coll to disk using sink.flush
public static <T> T writeAll(Collection<T> coll, Sink<T> snk){
    T last;
    for (T t : coll) {
        last = t;
        snk.flush(last);
    }
    return last; // return last element written
}

...
Sink<Object> s = ...; // a sink that can write any object
Collection<String> cs = ... ; // can it write strings...?
String str = writeAll(cs, s); //?
```



The diagram shows a pink box labeled "Illegal call" with two blue arrows pointing to the `Sink<Object> s` and `Sink<T> snk` parameters in the `writeAll` method signature. A red line connects the `//?` comment in the `String str = writeAll(cs, s);` line to the pink box.

```
Sink<Object> s;  
Collection<String> cs;
```

```
public static <T> T writeAll(  
    Collection<? extends T>, Sink<T>) {  
    ...  
}
```

```
String str = writeAll(cs, s); //?
```

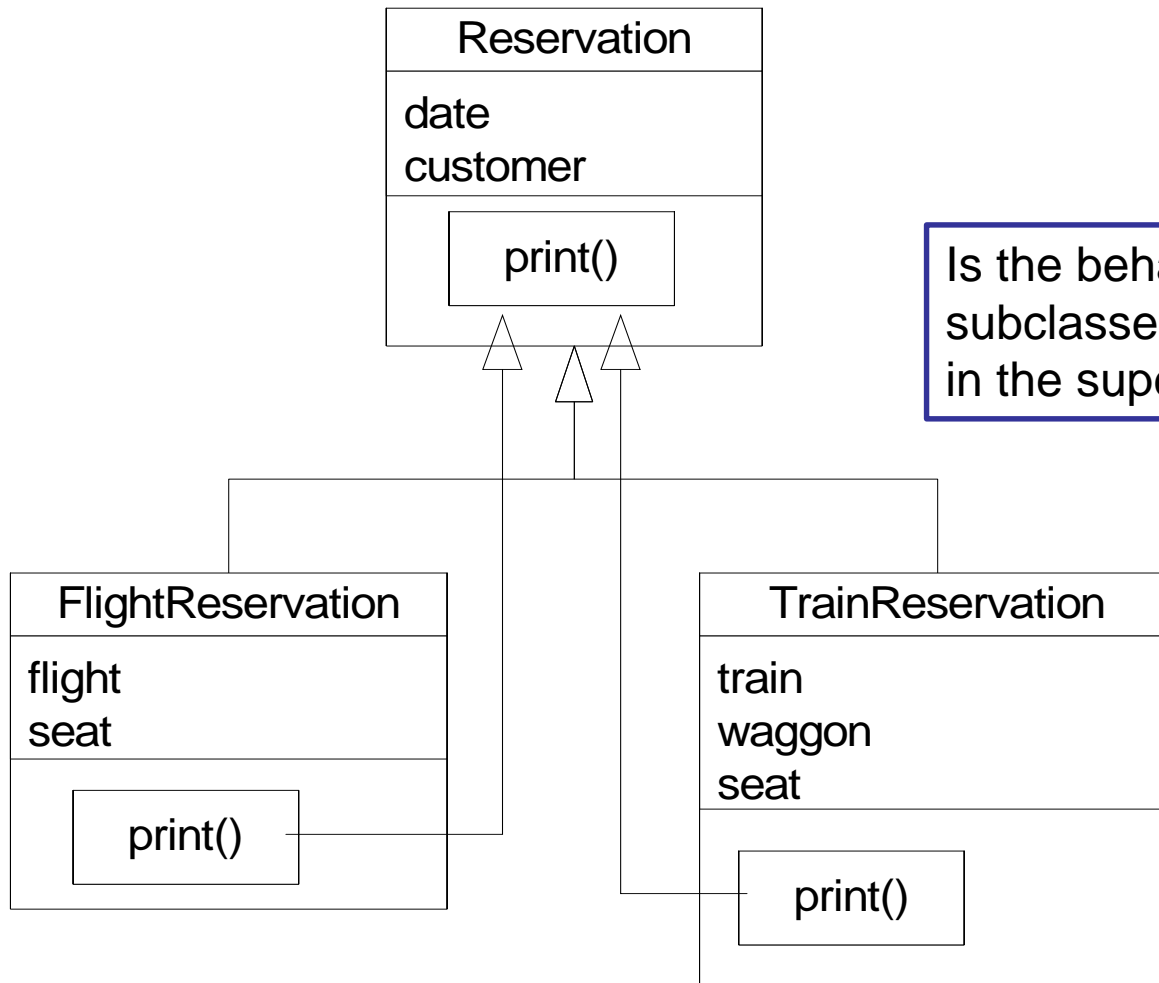
call ok, but
wrong return
type:
T which is
Object

```
public static <T> T writeAll(  
    Collection<T> coll, Sink<? super T> snk) {  
    ...  
}
```

```
String str = writeAll(cs, s); //?
```

Yes: returns T
which is now
String

Subtyping of behaviour specification?



Is the behavior of `print` in the subclasses a behavioral subtype of that in the superclass?

'Subtyping' for behaviour – the super style

```
class Reservation {  
    date . . . ;  
    customer . . . ;  
    void print() {  
        // print date and Customer  
    }  
}
```

```
class FlightReservation extends Reservation {  
    flight . . . ;  
    seat . . . ;  
    void print {  
        super.print();  
        // print Flight and Seat  
    }  
}
```

We depend on the developer of FlightReservation to do the "right thing"

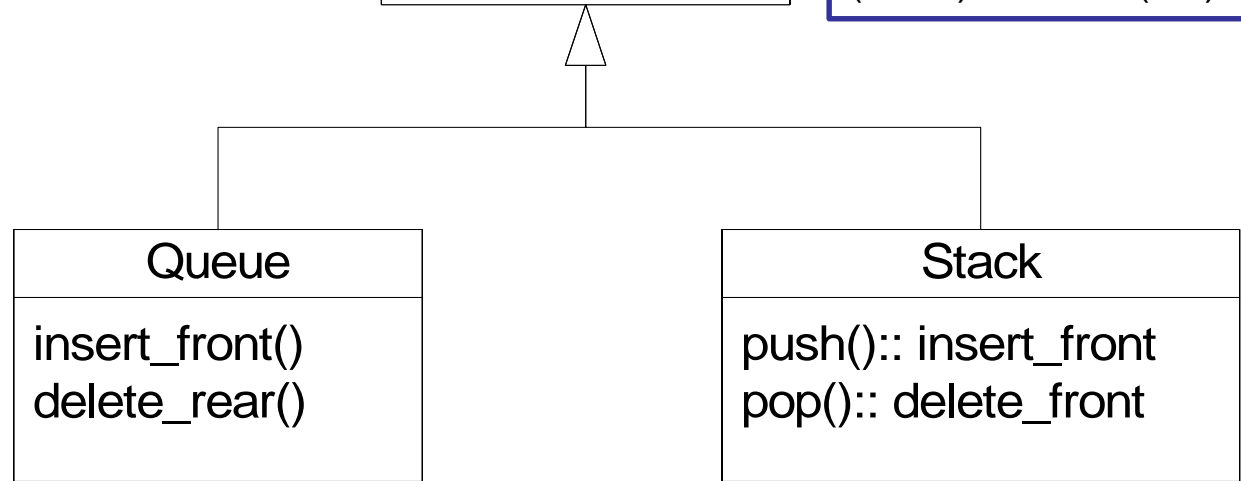
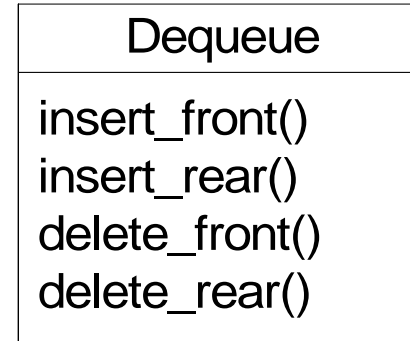
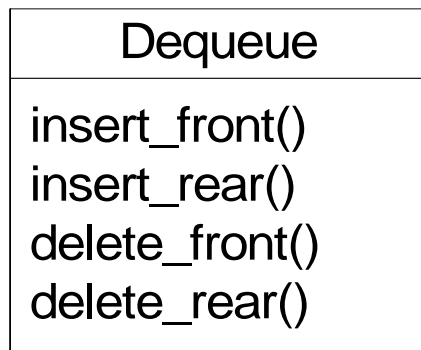
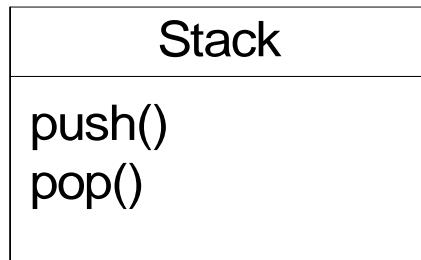
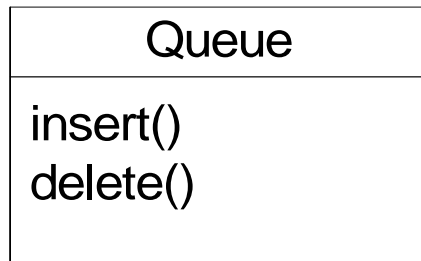
Subtyping for behaviour – the inner style

```
class Reservation {  
    date . . . ; customer . . . ;  
    void print() {  
        // print Date and Customer  
        inner;  
    }  
}
```

```
class FlightReservation  
    extends Reservation {  
    flight . . . ; seat . . . ;  
    void print extended {  
        // print flight and seat  
        inner;  
    }  
}
```

- Does the inner style give behavioral compatibility?
- No, still only structural compatibility, but structure in terms of sequence of statements, in addition to signature (number of types of parameters)!

Subtyping = subclassing??

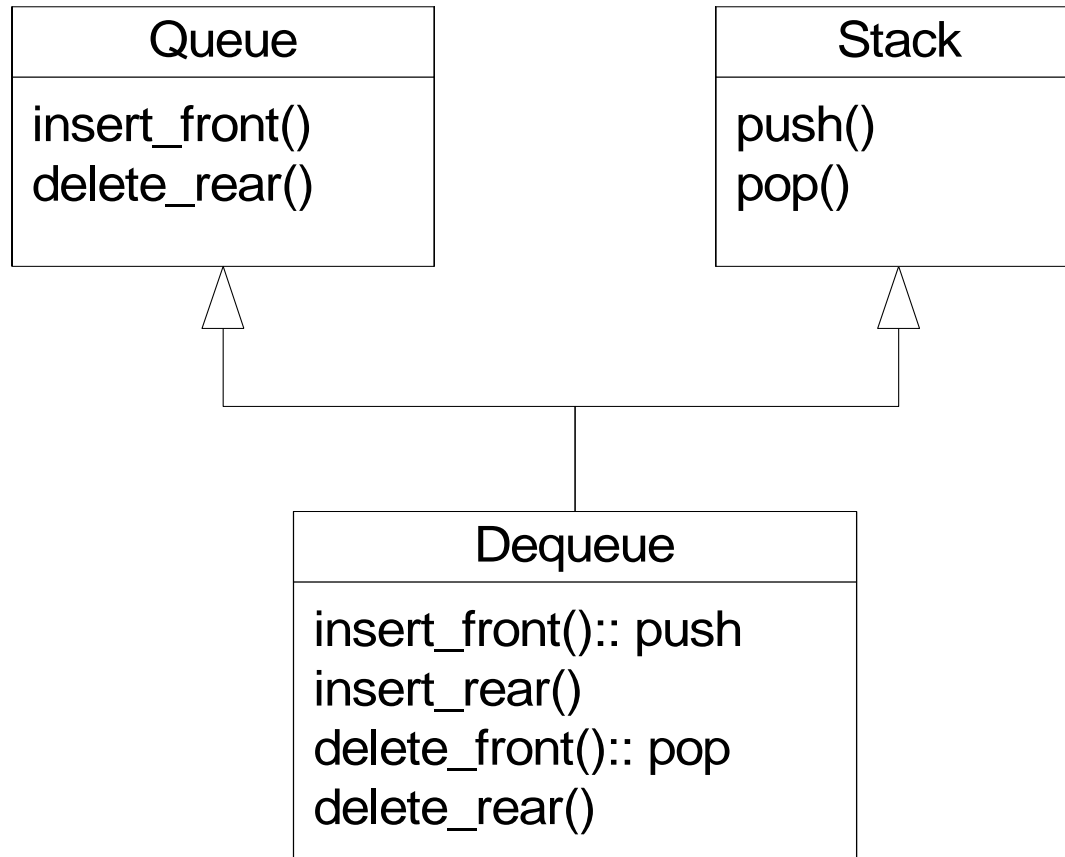


A double-ended queue (dequeue, often abbreviated to deque, pronounced deck) is an abstract data type that generalizes a queue, for which elements can be added to or removed from either the front (head) or back (tail)

```

Deque d; Stack s; Element e;
void f(Deque dp, Element ep) {
    dp.insert_front(ep); dp.insert_rear(ep)
}
...
f(s, e)
  
```

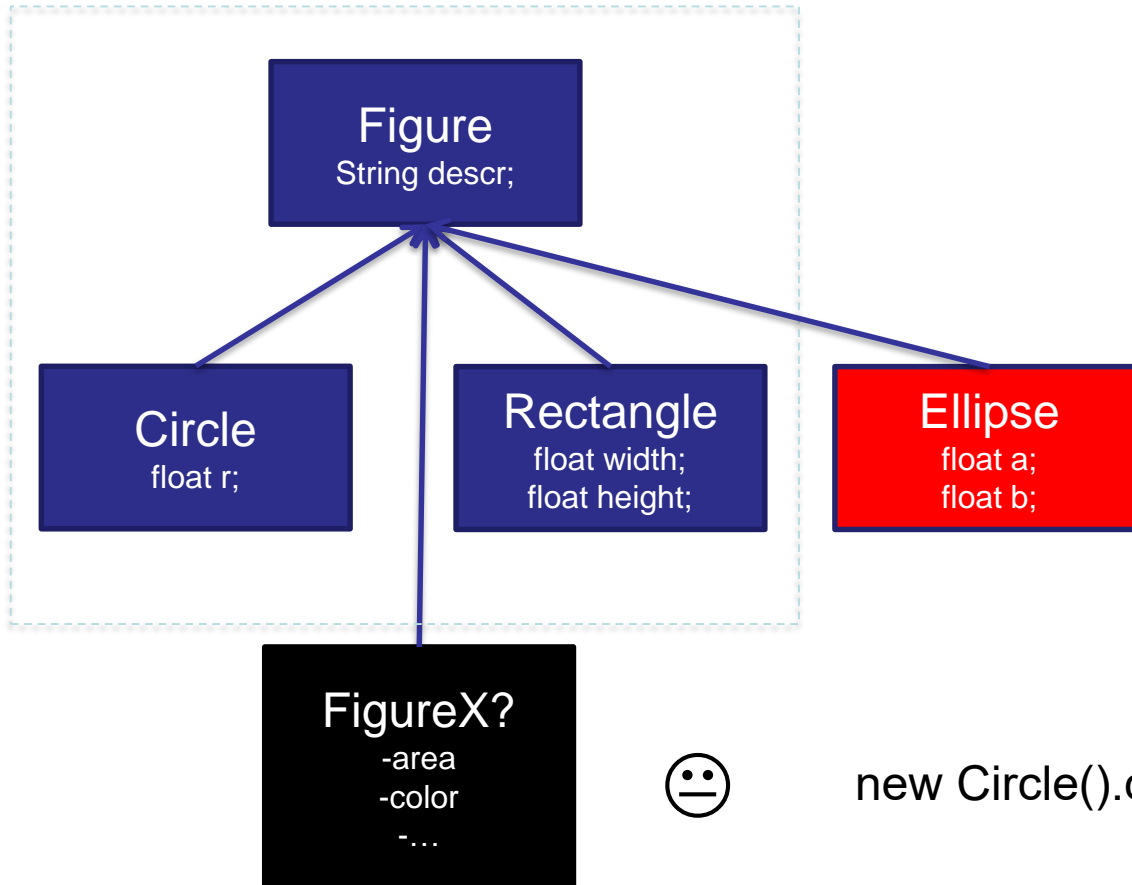
The opposite any better?



Dequeue can take the place of both a Queue and a Stack (via different references).

A context where it is used as a stack cannot be sure that it behaves like a stack.

Inheritance and extension



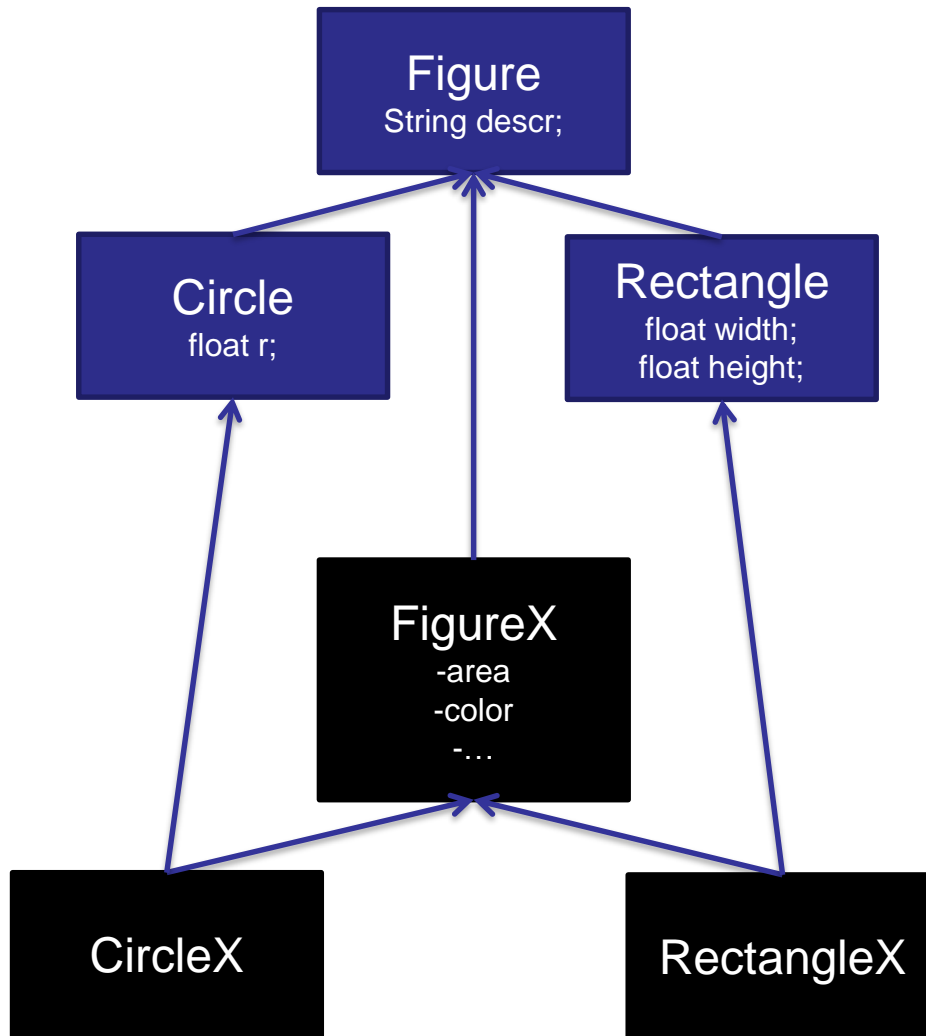
We want to extend this with new figures, and new properties for figures



new Circle().color= ...;

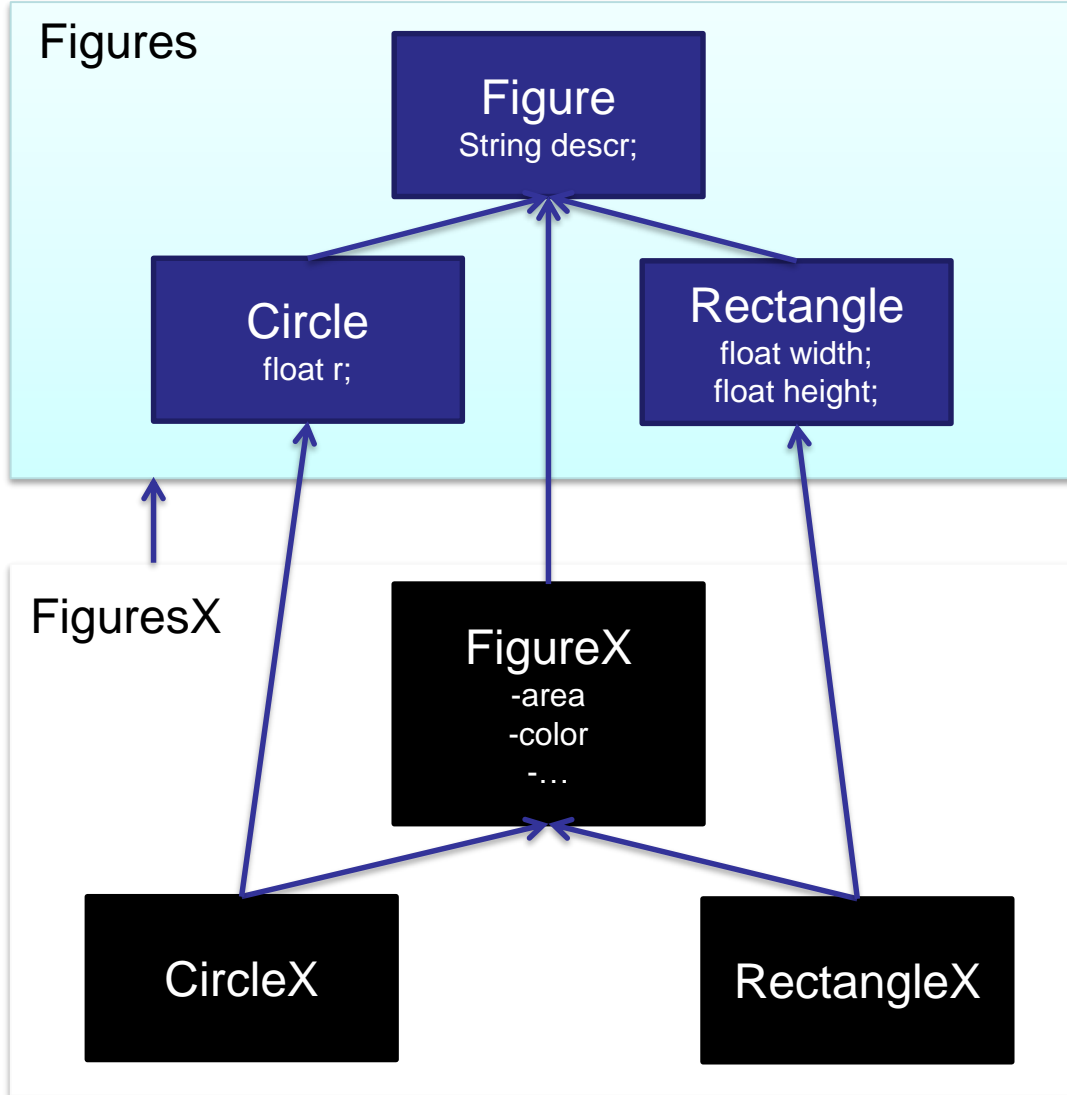


Multiple inheritance



- Solves the problem (kind of) but:
- Complex hierarchy for simple problem
 - One or two description variables from figure? («Diamond problem»)
 - Difficult with overrides
 - Runtime complexity

Virtual classes



Solves the problem (kind of) but:

- Complex hierarchy for simple problem?
- Runtime complexity
- Not type safe (typically)

Composition / Encapsulation?

```
class Apartment {  
    Kitchen theKitchen = new Kitchen();  
    Bathroom theBathroom = new Bathroom();  
    Bedroom theBedroom = new Bedroom ();  
    FamilyRoom theFamilyRoom =  
        new FamilyRoom ();  
    . . .  
    Person Owner;  
    Address theAddress = new Address()  
}
```

```
myApartment.theKitchen.paint(); ...
```

```
myApartment.theKitchen = otherAppartment.theKitchen; // ?
```

Where are Kitchen,
Bathroom, Bedroom,
FamilyRoom defined?

Do they belong to the
apartment?

Inner classes - locally defined classes

```
class Apartment {
    Height height;
    Kitchen theKitchen = new Kitchen();

    // define inner class
    // reuse the height of this specific Apartment:
    class ApartmentBathroom extends Bathroom {... height ...}
    // then use it
    // can't be mixed with another Apartment's rooms:
    ApartmentBathroom Bathroom_1 = new ApartmentBathroom ();
    ApartmentBathroom Bathroom_2 = new ApartmentBathroom ();

    Bedroom theBedroom = new Bedroom ();
    FamilyRoom theFamilyRoom = new FamilyRoom ();
    . . .
    Person Owner;
    Address theAddress = new Address()
}
```


Virtual classes

(made-up syntax ahead)

```
class Apartment {  
    virtual class ApartmentBathroom < Bathroom  
        ...  
};
```

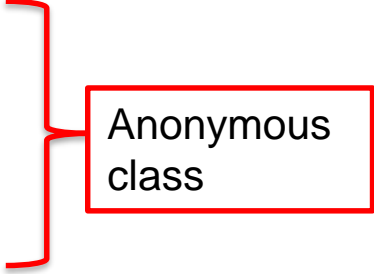
```
class SpecialApartment extends Apartment {  
    class ApartmentBathroom:: PinkBathroom  
        // PinkBathroom defined somewhere else  
}
```

```
class MoreSpecialApartment extends Apartment {  
    class ApartmentBathroom:: PinkBathroom {...}  
}
```

Singular objects (singleton class)

- anonymous classes

```
...  
Button btn = new Button();  
btn.setText("Say 'Hello World'");  
btn.setOnAction(  
    new EventHandler<ActionEvent>() {  
        public void handle(ActionEvent event) {  
            System.out.println("Hello World!");  
        }  
    }  
);  
...
```



Anonymous
class

```
interface HelloWorld {  
    public void greet();  
    public void greetSomeone(String someone);  
}
```

```
HelloWorld norwegianGreeting = new  
    HelloWorld() {  
        String name = "Verden";  
        public void greet() {  
            greetSomeone("Verden");  
        }  
        public void greetSomeone(String someone) {  
            name = someone;  
            System.out.println("Hallo " + name);  
        }  
    };
```

Anonymous
class

```

public static void printPersons(
    List<Person> roster, CheckPerson tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p.printPerson();
        }
    }
}

```

```

interface CheckPerson {
    boolean test(Person p);
}

```

Functional interface

```

printPersons(
    roster,
    new CheckPerson() {
        public boolean test(Person p) {
            return p.getGender() == Person.Sex.MALE
                && p.getAge() >= 18
                && p.getAge() <= 25;
        }
    }
);

```

Anonymous
class

```

public static void printPersons(
    List<Person> roster, CheckPerson tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p.printPerson();
        }
    }
}

```

```

interface CheckPerson {
    boolean test(Person p);
}

```

Functional interface

```

printPersons(
    roster,
    (Person p) ->
        p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25
);

```

Anonymous
function

Coming up!

- Daniel, with a couple of more ML-lectures
- Mandatory 2 out now

