Structured Programming

COMP1110/6710

Australian National University

Image Courtesy NASA/JPL-Caltech.

Today: Core OO

That means: **Subtyping**, Dynamic Dispatch, Overriding

Also: Inheritance, Access Control

Australian National University

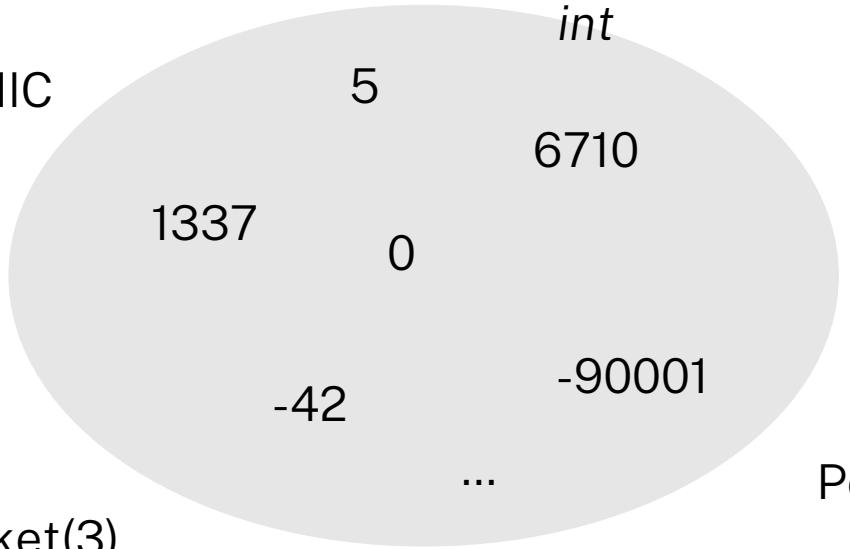
Real Software Concerns

- Working in teams, collaborating with other programmers, other teams
- Software needs to be maintained: fixing bugs, adding features, adapting to changes in libraries/environment
- DRY on larger scale – once written, code should be reusable for other purposes

➔ Software should consist of small components that can easily be swapped out and reused elsewhere.

What's a Type?

A set of possible values!



Role.ACADEMIC

"Hello"

true

[1, 2, 3]

OrangeBasket(3)

Person("Frank", 55)

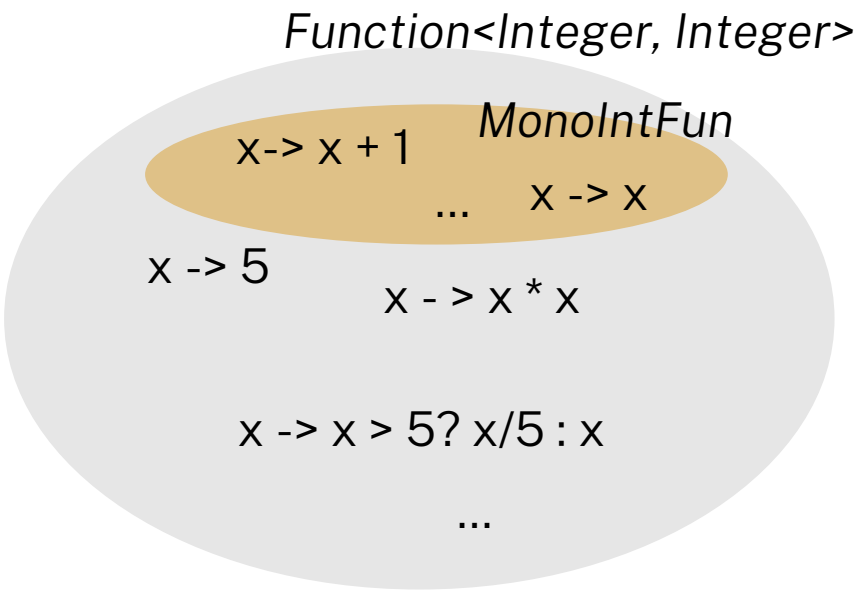
Remember: Humans and Machines

Java's `int`: any integer between -2^{31} and $2^{31} - 1$ (~ -2 billion to +2 billion)

```
/** A PosInt is an int that is greater than 0 */
```

➔ Any integer between 1 and $2^{31} - 1$, but only as part of a human signature. Java does not check this part.

Function Types




```
/**
 * A MonoIntFun is a
 * Function<Integer,Integer>,
 * where, for any MonoIntFun f
 * and any two ints x, y, if
 * x >= y, then f(x) >= f(y)
 */
```


Barbara Liskov
Turing Award 2008

Liskov Substitution Principle:

A type S is a subtype of a type T if any value of type S can be used wherever any value of type T is expected, and it behaves according to T's specification.

➔ *"Behavioral Subtyping"*

Behavioral Subtyping Examples

- Every `PosInt` is also an `int`, hence `PosInt` is a subtype of `int`.
- Similarly, every `MonoIntFun` is also a `Function<Integer,Integer>` …
- In both cases, the reverse is not true.

- `String` values (e.g. “Hello”) cannot be used where `int`s are expected
- Conversely, you also could not write 5.length() , hence neither is a subtype of the other.

Behavioral Subtyping Examples

However, in terms of Java types, `int = PosInt`, and
`MonoIntFun = Function<Integer,Integer>`

Java Interfaces

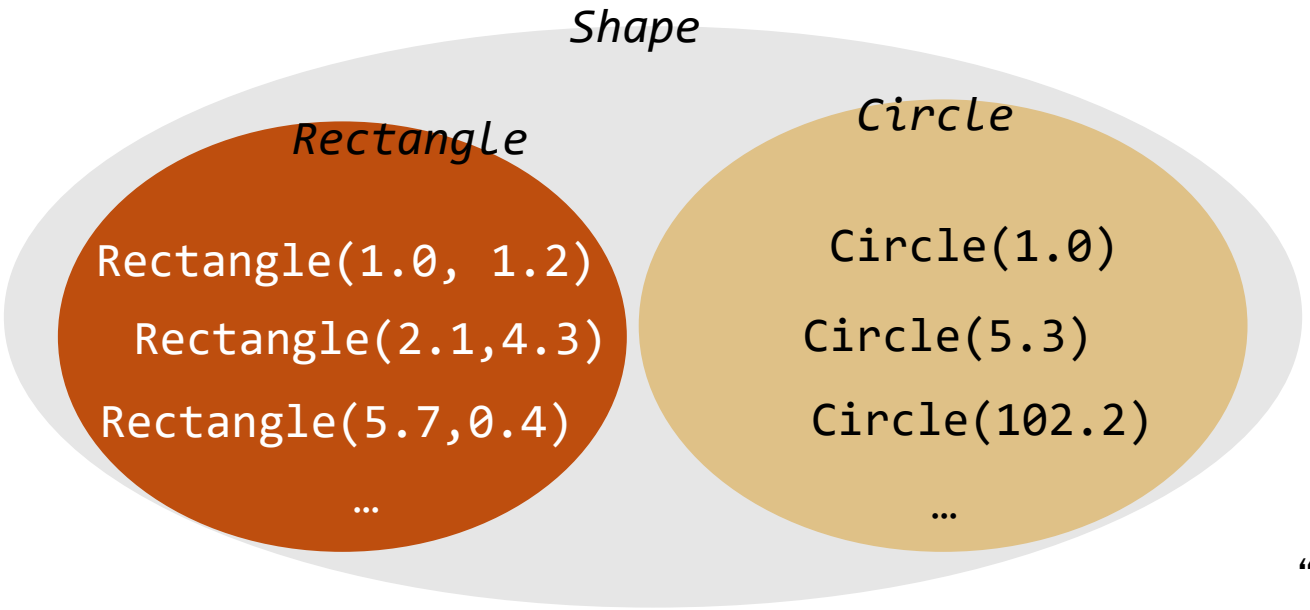
Java's Main Mechanism for Subtyping

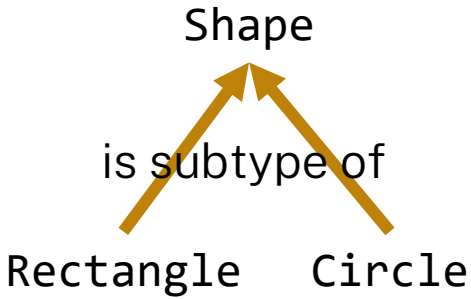
We've seen a special case before:

```
sealed interface Shape permits Rectangle, Circle {}
record Rectangle(double w, double h) implements Shape {}
record Circle(double radius) implements Shape {}
```


Java Interfaces

Java's Main Mechanism for Subtyping





"union". See also HtDC

Java Interfaces

Java's Main Mechanism for Subtyping

Special feature for more functional behaviour:
allows omitting default-case in switch, but relies on knowing all cases

```
sealed interface Shape permits Rectangle, Circle {}
record Rectangle(double w, double h) implements Shape {}
record Circle(double radius) implements Shape {}
```

OO Philosophy: Allow extending programs with new classes implementing existing interfaces all the time

Java Interfaces

Java's Main Mechanism for Subtyping

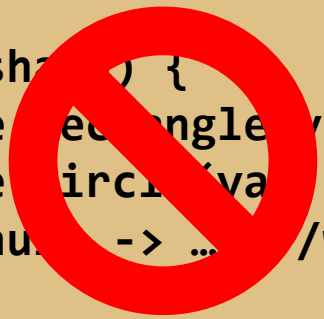
Now there can be arbitrarily many classes/records implementing ConsList

```
interface Shape {}
record Rect                                    ape {}
record Circ
```

Instead, we now use the space between the curly braces!

BUT:

```
switch(sh    ) {
    case  ec  ngle  ar w, var h) -> …;
    case  irc  (va  radius) -> …;
    defau   -> …    /what goes here???
}
```



Java Interfaces

Java's Main Mechanism for Subtyping

```
interface Shape {
    double getArea();
}
```

Interfaces can prescribe arbitrarily many methods that classes implementing them have to provide. Interfaces themselves generally do have code for those methods.

Java Interfaces

Java's Main Mechanism for Subtyping

```
interface Shape {
    double getArea();
}
class Circle implements Shape {
    double radius;
    Circle(double radius) { this.radius = radius; }
    @Override
    double getArea() { return radius * radius * Math.PI; }
}
```

Implementing or replacing (see next) a method in a supertype is called "overriding". Java uses the @Override annotation to check whether there actually is something that is being overridden.

Java Interfaces

Java's Main Mechanism for Subtyping

```
interface Shape {
    double getArea();
}
class Rectangle(double w, double h) implements Shape {
    @Override
    double getArea() { return w * h; }
}
```

Pro tip: if your fields never have to be changed, you can also use records, which saves you some work in writing constructors.

Dynamic Dispatch

How do we know which method to call?

```
double getShapeArea(Shape s) { return s.getArea(); }


void main() {
    System.out.println(getShapeArea(new Rectangle(5,3)));
    System.out.println(getShapeArea(new Circle(3.2)));
}
```

Remember: Objects know themselves. So each individual `s` knows whether it is a rectangle or a circle or …, and calls the right `getArea()`

Java Interfaces vs. Classes

- Interfaces have no instance fields, and no constructors
- The main point of interfaces is to declare (not implement) methods

- **Special features (distinction-level):**
- Interfaces can have static fields and methods; for fields, you can omit the static keyword
- Interfaces can have “default” implementations for instance methods

Design Recipe

Where should things go?

Write once for each method, in interface. Copy to classes/records implementing the interface and overriding the method.

Design Strategy is added to actual implementation, not interface.

Examples/tests should go where they fit best – can be with the interface to give examples/tests for everything together, or with individual classes. There should be at least one example/test per class.

Design Recipe

Key point: Liskov Substitution Principle

The argument types/pre-conditions of a subtype cannot be more strict than those of a supertype. If one of your method implementations makes certain assumptions about input data/the overall state, you need to state that in the interface.

Similarly, the return type/post-condition must be at least as strong as in the interface. You cannot relax any requirements of your signature in a subclass – you need to do that in the interface.

Inheritance

Code Reuse

But only if there is Behavioural Subtyping, because Java conflates the two concepts.

Australian National University

Some things get repetitive

```
interface User {
   String getName();
}
```

```
class Academic implements User {
  String name;
  Academic(String name) { … }
  @Override
  String getName() { return name; }
}
```

```
class Student implements User {
  String name;
  Student(String name) { … }
  @Override
  String getName() { return name; }
}
```

```
class ProfessionalStaff
  implements User {
  String name;
  ProfessionalStaff(String name) { … }
  @Override
  String getName() { return name; }
}
```

Some things get repetitive

Now a class so we can have instance fields

```
class User {
   String name;
   User(String name) { … }
   String getName()
   { return name; }
}
```

Keyword for extending classes is “extends”

```
class Student extends User {
  Student(String name) {
    super(name);
  }
}
```

Constructor needs to explicitly call super-constructor with special keyword “super”, except if there is a parameter-less constructor

```
class Academic extends User {
  Academic(String name) {
    super(name);
  }
}
```

```
class ProfessionalStaff extends User {
  ProfessionalStaff(String name) {
    super(name);
  }
}
```


Abstract Classes

When you can't implement everything right there

```
abstract class User {
  String name;
  User(String name) { … }
  String getName() {
    return name;
  }
  abstract Maybe<Date> nextSalaryIncrease();
}
```

"abstract" classes allow "abstract" methods to exist in them, but in turn disallow directly constructing instances of that class.
I.e. you can't write `new User("Fabian")`
Need to use a non-abstract subclass

"abstract" methods behave as if specified in an interface

Extending Classes

A bigger Example

```
class Student extends User {
  Date birthday;
  Student(String name, Date birthday) {
     super(name);
     this.birthday = birthday;
  }
  @Override
  Maybe<Date> nextSalaryIncrease() {
    return new Nothing<>();
  }
}
```

Can add new fields and constructor arguments

Need to initialize new fields after super-constructor call

Overriding works the same as for interfaces

The Object Class – Recap:

Things we'll cover in this course:

```
"hello".equals("world")
    ⇔ Equals("hello", "world")

new HashMap<>().toString()
    ⇔ ToString(new HashMap<>())

CurrentDate().getHashCode()
```

Things we likely won't cover:

```
"hello".getClass()
"hello".notify()
"hello".notifyAll()
"hello".wait()
```

Every reference value ultimately inherits these from the Object class. Most can be overridden.

Overriding an Existing Method

Implicitly extends Object

```
class User {
  String name;
  User(String name) { … }
  String getName()  { return name; }

  @Override
  String toString() {
    return name + " / " + super.toString();
  }
}
```

Overrides default implementation from Object

Calls default implementation from superclass, in this case Object – you do not have to do that.

Java Interfaces vs. Classes, Part 2

- Anything can extend arbitrarily many interfaces
- But only classes can extend classes, and at most one each (by default: Object)

Access Control

To preserve your ability to change things, hide everything from others that they don’t explicitly need

Australian National University

Supertypes Already Hide Stuff

```
class Student extends User {
  Date birthday;
  Student(String name, Date birthday) {
     super(name);
     this.birthday = birthday;
  }
  @Override
  Maybe<Date> nextSalaryIncrease() {
    return new Nothing<>();
  }
}
```

Since “birthday” is only introduced here, anyone looking at this object as a “User” does not know the field exists.

Visibility Attributes

```
public abstract class User {
  public final int id;
  protected String name;
  private static int counter=0;
  protected User(String name) {
    this.name = name;
    this.id = counter++;
  }
  String getName() { return name; }
}
```

public – accessible from anywhere
protected – accessible from subclasses
private – accessible from within the class
[nothing] – accessible from classes within the same folder
final – not technically about visibility, but requires field to be set in constructor, then disallows any changes afterward (this is how records work)

Inner Classes

To get inside access

```
interface A { String getString(); }
class B {
    private String str;
    …
    private class C implements A() {
      String getString() { return str; }
    }
    A getA() { return new C(); }
}
```

C is an “inner class”. It can only be created from within an instance method of B, and is then associated with that object, hence it has direct access to its fields.

Static Inner Classes

To get inside access

```
interface A { String getString(); }
class B {
   private String str;
   …
   public static class D implements A() {
     B b; D(B b) { this.b = b; }
     String getString() { return b.str; }
   }
   A getA() { return new D(this); }
}
```

D is a “static inner class”. It can be accessed from anywhere, and is only logically contained within B. Given any B, it still has access to its private fields.