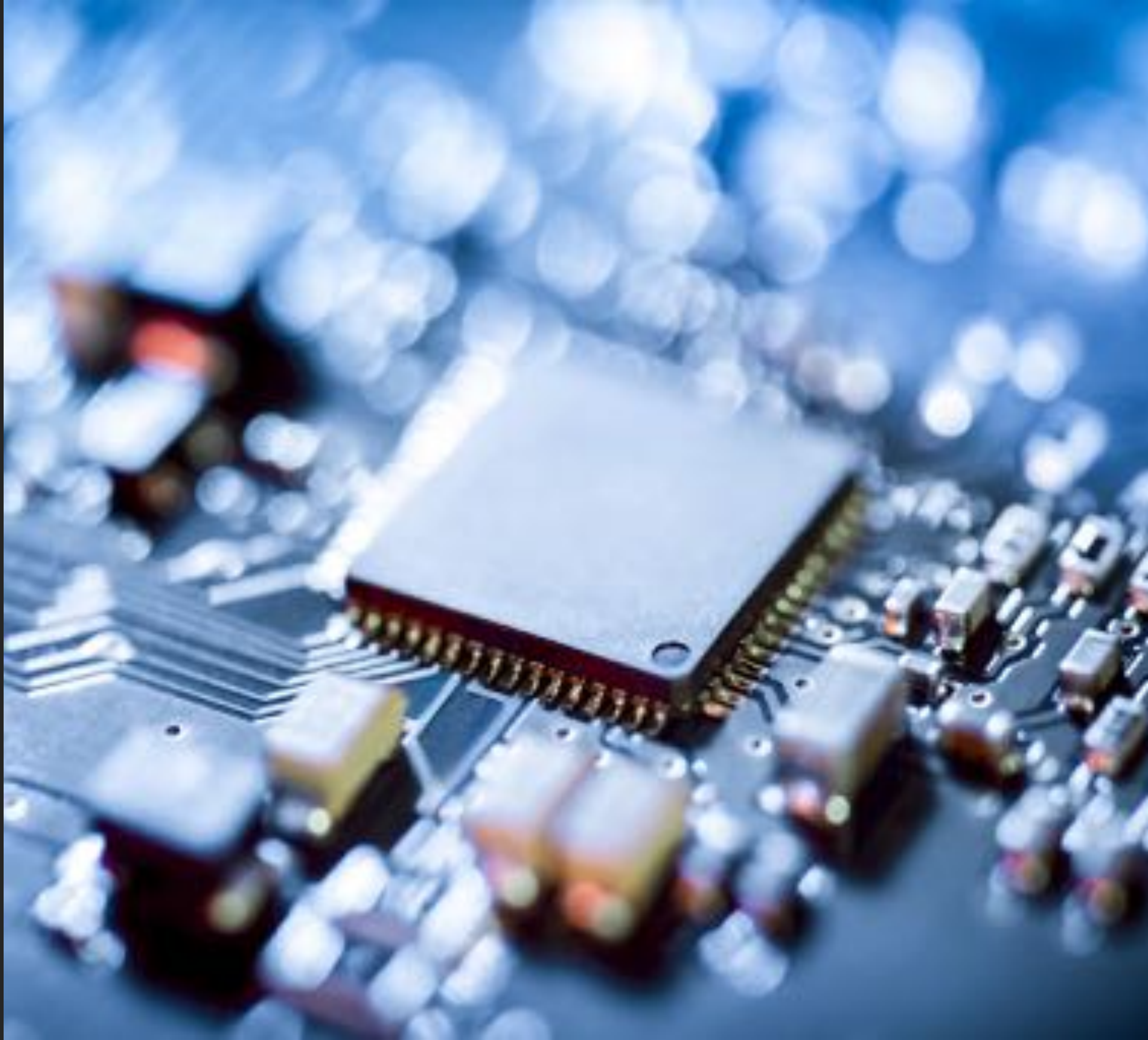
SOFTWARE ENGINEERING

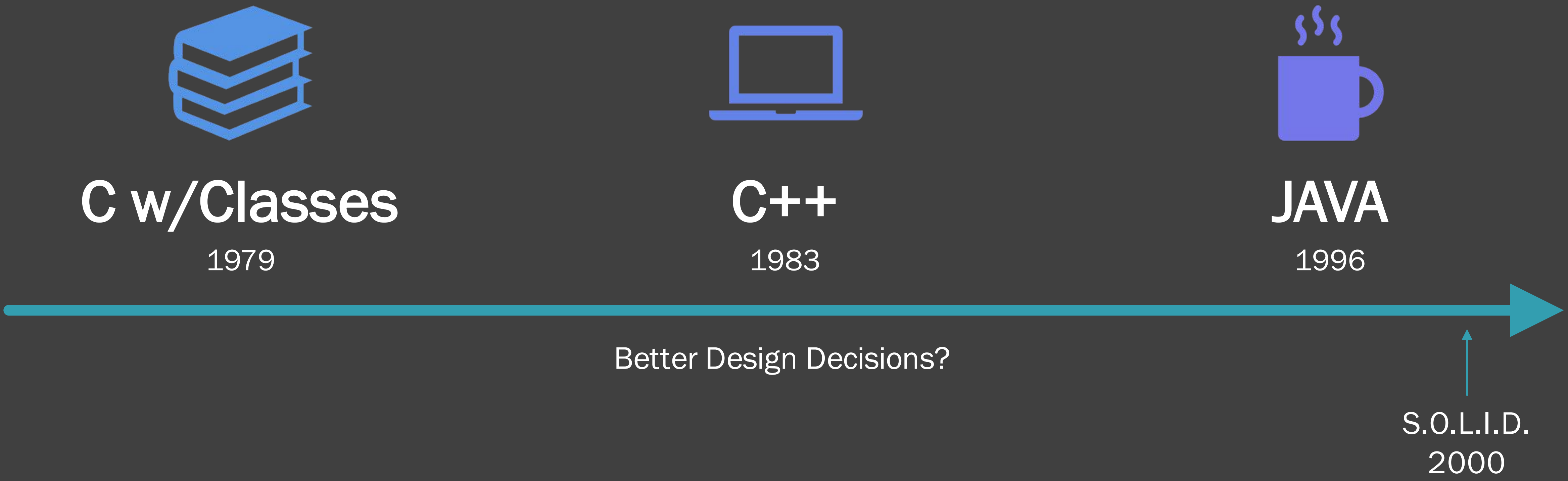
ECE 651

SPRING 2020

JAVA FOR C++ PROGRAMMERS



A TALE OF TWO LANGUAGES



C w/Classes

1979

C++

1983

JAVA

1996

Better Design Decisions?

S.O.L.I.D.
2000

QUICK REMINDER – OO DESIGN PRINCIPLES

- Abstraction, Encapsulation, Inheritance, Polymorphism!
- Effective parallelism between developers requires independent tasks
- Least Surprise
- DRY: Don't Repeat Yourself
- Low Coupling / High Cohesion
- SOLID
- Design for testability

THINK, PAIR, SHARE

I will introduce a difference between C++ and Java, you will think about design principles that the difference might support.

METHOD DISPATCH

C++

- Can request dynamic dispatch
  - Calls to an overridden method is resolved at runtime
- How do we request it?
  - **virtual** on declaration in class of static type (or its parents)

Java

- Dynamic dispatch for every method
  - No other choice

DESIGN PRINCIPLES ADDRESSED

- **Least Surprise:** Might expect dynamic dispatch
  - Especially if you wrote virtual in the child class
- **Open/Close:** Didn't make it virtual to begin with? Need to modify parent
  - What if we we had a different subclass with same method/static dispatch?

MEMORY ALLOCATION FOR OBJECTS

C++

- Objects created in Heap or Stack
- Object in the **frame**
  - Destroyed when function returns
- Heap management
  - The memory containing the object persists until the end of your program, or until you delete the object
  - Uses delete + destructors

JAVA

- All Objects created in the Heap
- No such thing as objects in the **frame**
- Heap management
  - Garbage collection – freeing any object without reference in the method
- No destructors

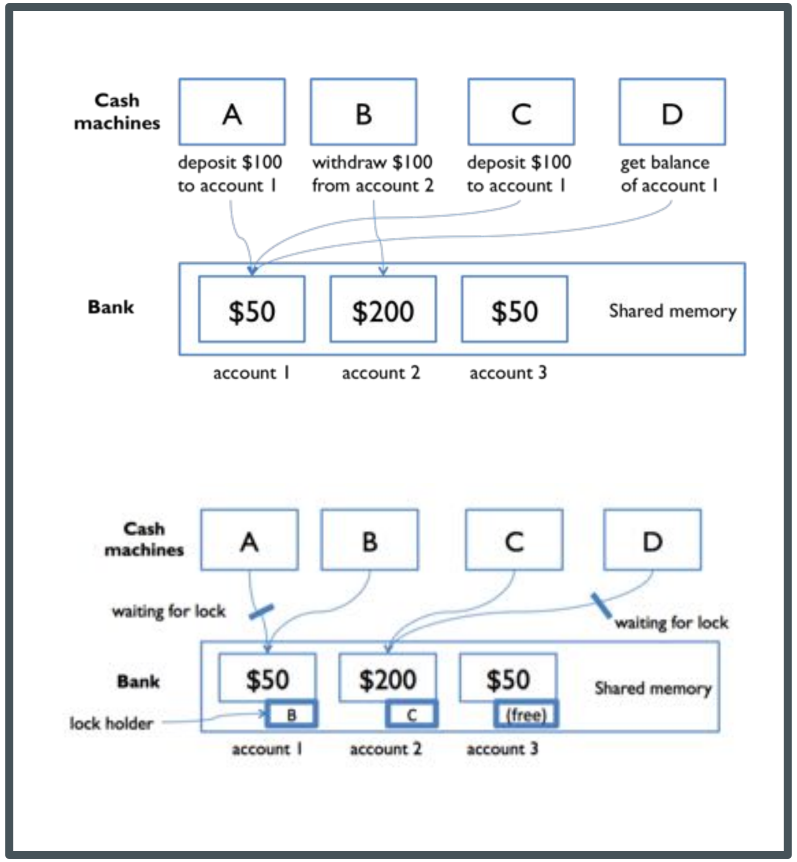
DESIGN PRINCIPLES ADDRESSED

- **Least surprise:** no strange bugs from free-related errors
- **Better abstraction:** don’t need to know where memory is allocated or when to free

NO OBJECTS IN THE FRAME: WHAT'S THE CONSEQUENCE?

- No Resource Acquisition is Initialization (RAII)
- What is RAII?
  - Local object owns resource, responsible for destruction
- How do we handle non-memory resources?

NON-MEMORY RESOURCES (1)



- Reminder: **Multi-threading** means more than one thread of code can run simultaneously
- “**synchronized**” is a Java keyword that locks/unlocks a mutex.

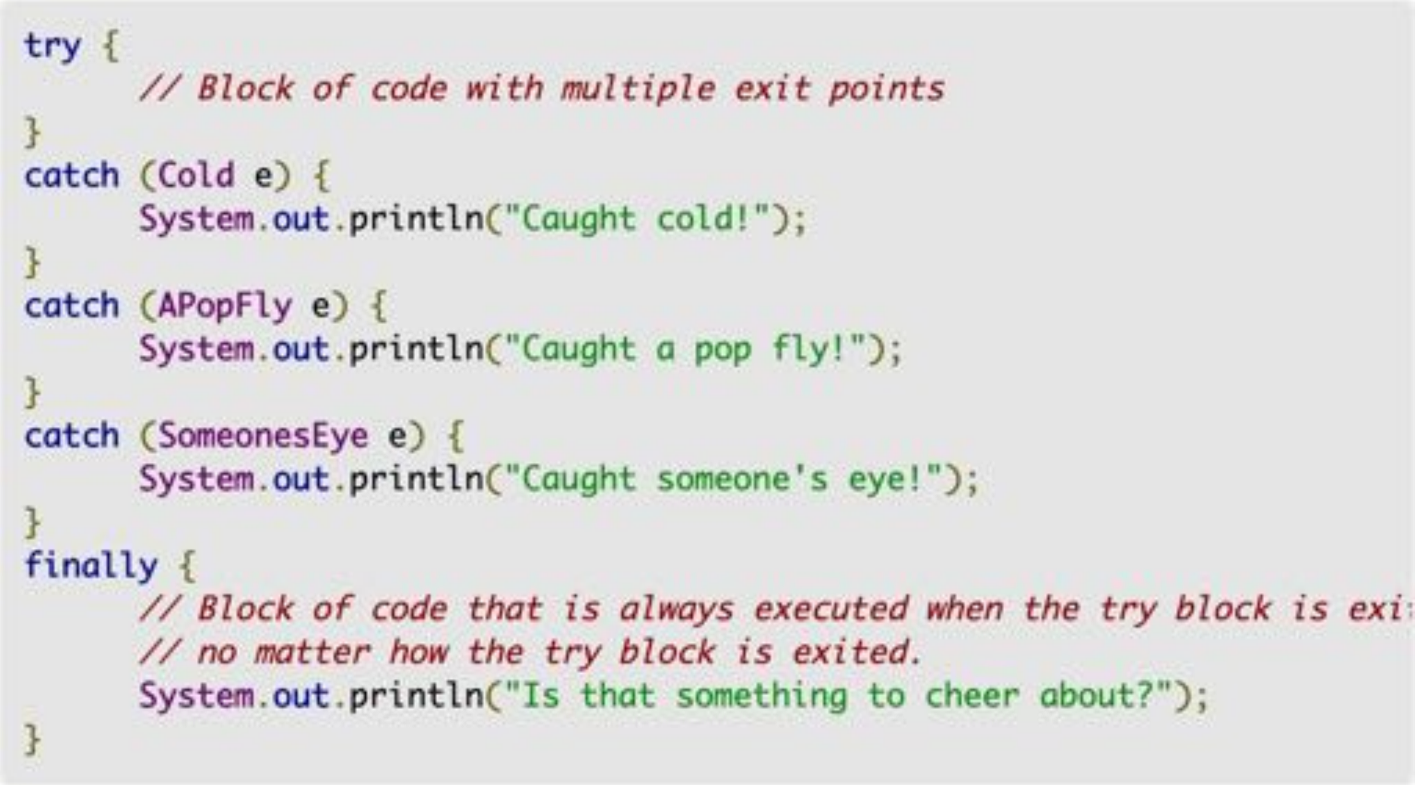
```
synchronized(object) {   //locks mutex in object
    //critical section code
}  //unlocks mutex: even if block is exited by exception
```

- **Mutexes (locks)** are one synchronization technique
  - Mutexes ensure exclusive access (one thread can lock at a time)
  - Written by experts to ensure no problems with hardware re-ordering

Image Source: http://web.mit.edu/6.031/www/fa17/classes/21-locks/

NON-MEMORY RESOURCES (2)

- “finally” block ensures that the JVM will execute the code written within it even if there is an exception in the code
- Avoid having cleanup code bypassed by a return, continue, or break

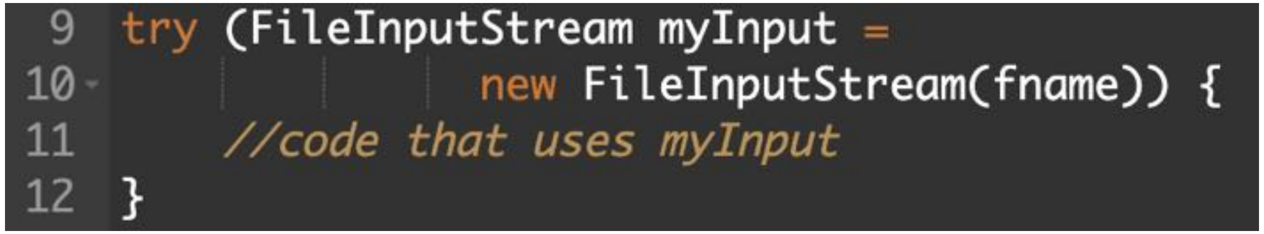


```
try {
        // Block of code with multiple exit points
}
catch (Cold e) {
        System.out.println("Caught cold!");
}
catch (APopFly e) {
        System.out.println("Caught a pop fly!");
}
catch (SomeonesEye e) {
        System.out.println("Caught someone's eye!");
}
finally {
        // Block of code that is always executed when the try block is exi
        // no matter how the try block is exited.
        System.out.println("Is that something to cheer about?");
}
```

Source: https://www.javaworld.com/article/2077609/try-finally-clauses-defined-and-demonstrated.html

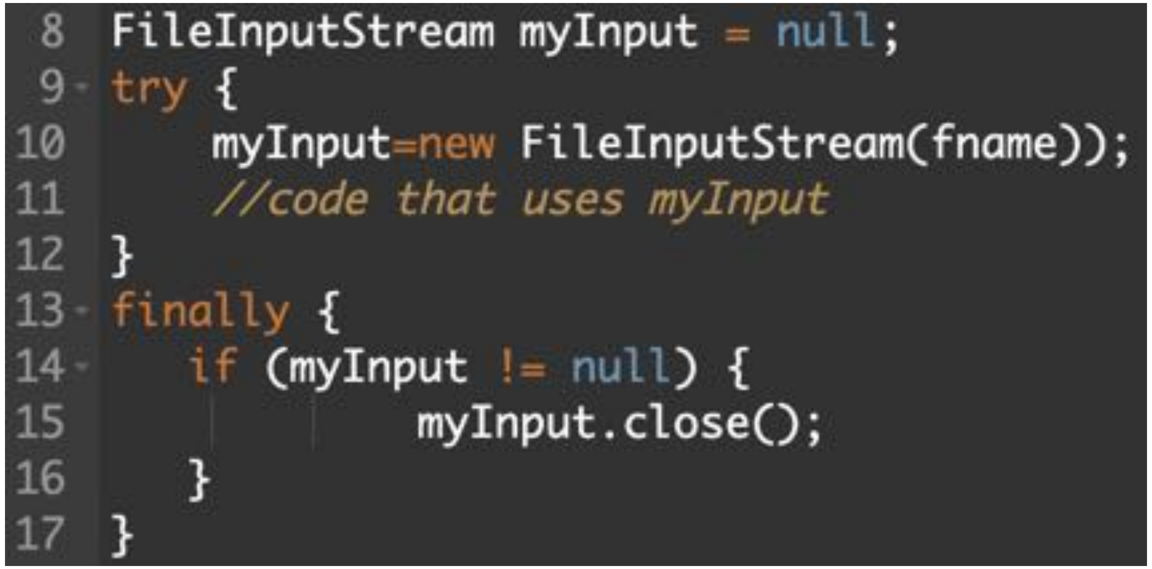
NON-MEMORY RESOURCES (2)

- Try-with-resource:
  - Shorthand for try/finally
  - Where finally just closes resource
- Resource must implement `java.lang.AutoCloseable`



```
try (FileInputStream myInput =
                    new FileInputStream(fname)) {
        //code that uses myInput
}
```

OR

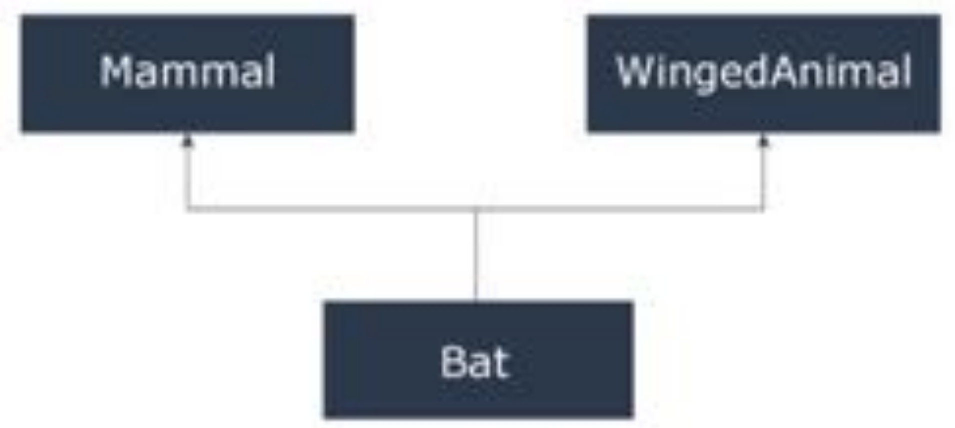


```
FileInputStream myInput = null;
try {
    myInput=new FileInputStream(fname));
    //code that uses myInput
}
finally {
   if (myInput != null) {
            myInput.close();
   }
}
```

INHERITANCE

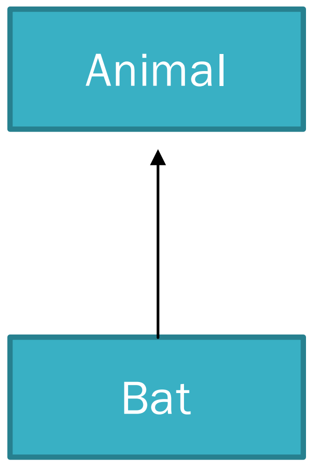
C++

- Multiple Inheritance
  - A subclass can inherit from more than one superclass



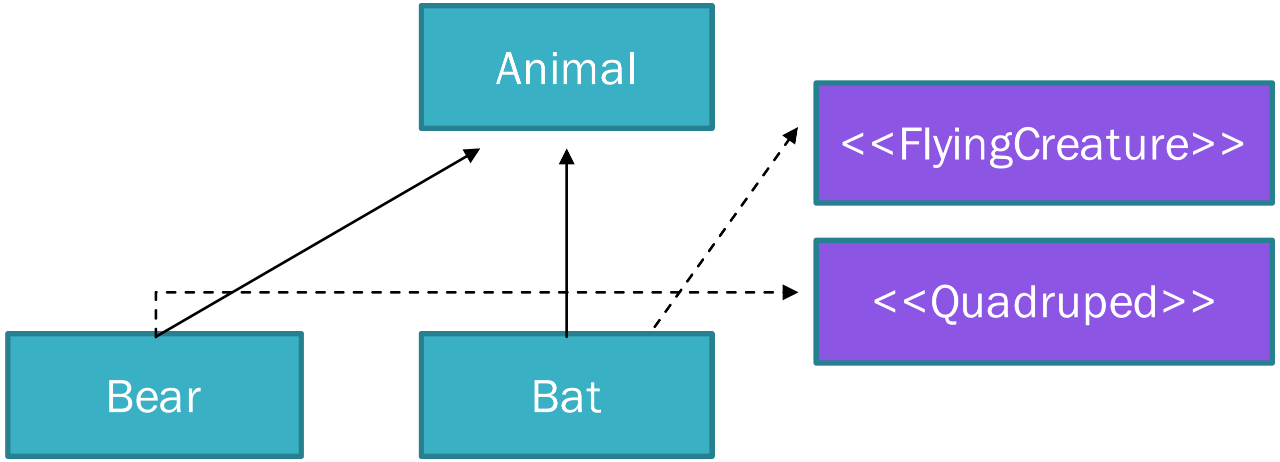
Java

- Single inheritance
- Uses interfaces instead



INTERFACES IN JAVA

- Specify what a class must do (method specification), but not how (implementation)



DESIGN PRINCIPLES ADDRESSED

- **Interface Segregation Principle:** Can split interfaces without complications of multiple inheritance
- **Dependency Inversion Principle:** can depend on just an interface

PARAMETRIC POLYMORPHISM

- A programming language technique that enables the generic definition of entities (classes, functions, methods), to improve code re-use
- Entities are parameterized over one or more types (e.g., “T”)
  - Can be used for any T
  - E.g., LinkedList<int>, LinkedList<String>,…

PARAMETRIC POLYMORPHISM

C++

- Called “templates”
- Recompiled for each T it is used with
- Type checking done at use
- Code must be directly visible at use

Java

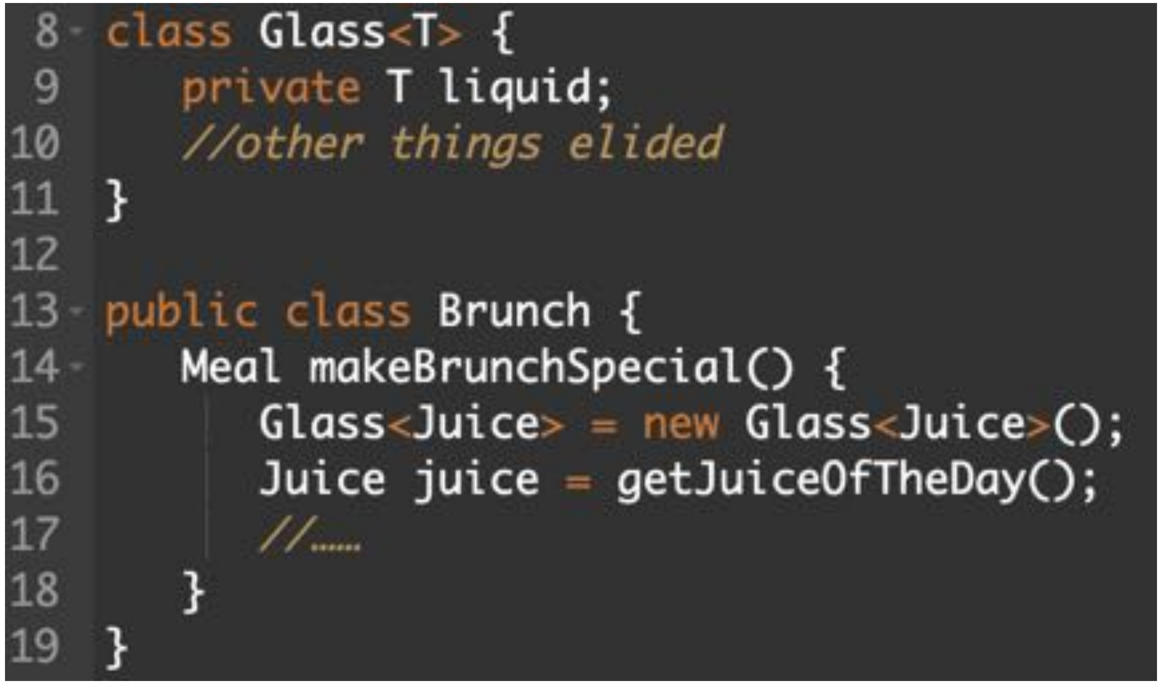
- Called “generics”
- Compiled once, re-used for all Ts
  - T is “erased”: not available at runtime
- Type checking done at definition
- Can use compiled class files normally

PARAMETRIC POLYMORPHISM

- True independence of the type (really for all) is restrictive
  - Want to order things? Not all types are orderable.
  - Want to check for equality? Not all types support equality testing.
  - Want to ….? Not all types support …

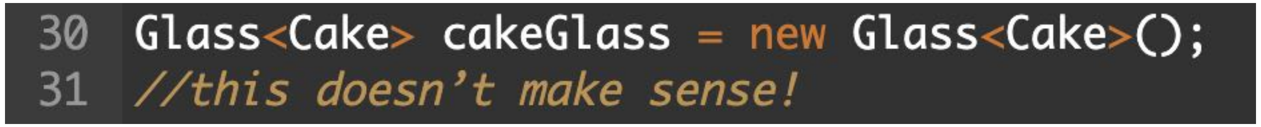
JAVA GENERICS

- Once Glass<T> is compiled works for any type
- Compiles one version of Glass for Ts it is used with
- T goes away, and is turned into Object (Type Erasure)



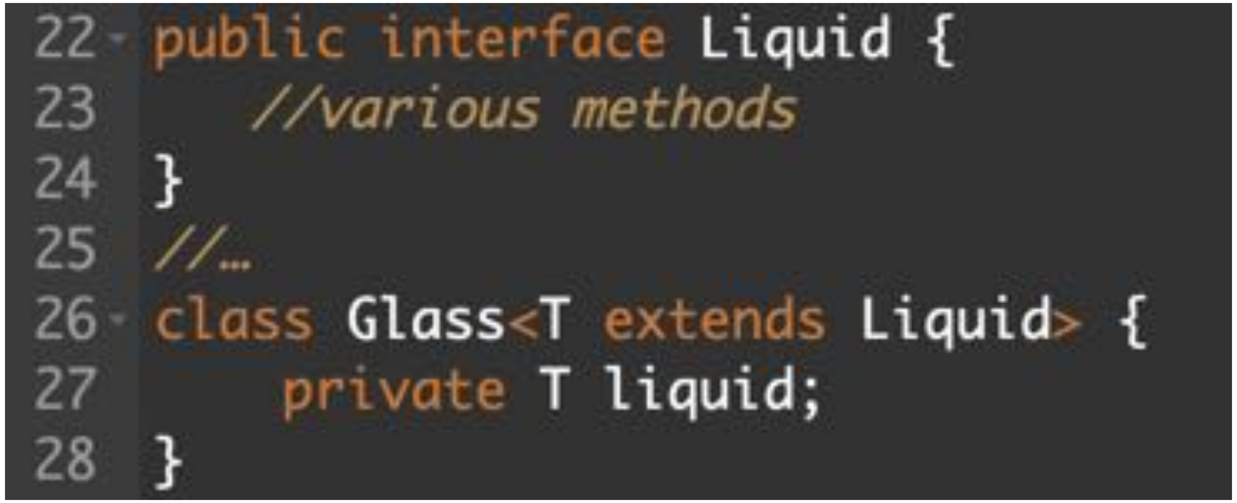
```java
class Glass<T> {
    private T liquid;
    //other things elided
}

public class Brunch {
    Meal makeBrunchSpecial() {
        Glass<Juice> = new Glass<Juice>();
        Juice juice = getJuiceOfTheDay();
        //......
    }
}
```

BOUNDED POLYMORPHISM



```java
Glass<Cake> cakeGlass = new Glass<Cake>();
//this doesn’t make sense!
```

- Could use “Cake” as a parameter, but this is not really what you want.
- Instead can restrict generic to bounded type parameters
- Now glass instantiations will only accept liquids



```java
public interface Liquid {
    //various methods
}
//…
class Glass<T extends Liquid> {
     private T liquid;
}
```

THINK, PAIR, SHARE

- What design principles do generics address?

DESIGN PRINCIPLES ADDRESSED

- **Least surprise:** don’t get compiler errors in a class you’ve used many times
- **Abstraction: clear interface:** know exactly what we need to use as type parameter
- “Parametric analog of Liskov Substitution Principle” – Drew
  - LSP basically says if S is a subtype of T, code works fine if use S where T expected
  - Parametric analog : if code parameterized over <T>, and can pass S in for that parameter, code should work.

WHAT WOULD C++ DESIGNERS ARGUE ARE BETTER ABOUT TEMPLATES?

Type bounded polymorphism is overly constraining!
With templates, I can make a vector of anything. I only need < defined on T if I need to order vector<T>!

C++

WHAT WOULD C++ DESIGNERS ARGUE ARE BETTER ABOUT TEMPLATES?

```
public class OrderableVector<T extends Comparable>
           extends Vector<T>
             implements
Comparable<OrderableVector<T>>
```

Java

WHAT WOULD C++ DESIGNERS ARGUE ARE BETTER ABOUT TEMPLATES?

Well what if I had 5 different things that you might or might not define? 5 different sub-classes? 31 for all combinations?

C++

WHAT WOULD C++ DESIGNERS ARGUE ARE BETTER ABOUT TEMPLATES?

That isn't really common, and if it does come up, maybe you should rethink your design in that case?

Java

WHAT WOULD C++ DESIGNERS ARGUE ARE BETTER ABOUT TEMPLATES?

But I **can** do it if that's my choice.

C++

SML FUNCTORS

- SML has a better solution than either of these, in its **functors**
  - Embodies dependency inversion
- To become an awesome SML hacker and write a compiler
  - Take ECE 553 next year!

OPERATOR OVERLOADING & USER-DEFINED CONVERSIONS

C++

- Allows operator overloading
  - E.g., overload <, ==, etc. to use Standard Template Library
- Many user-defined implicit conversions
  - One argument constructors (how do we prevent implicit use?)
  - operator type()

Java

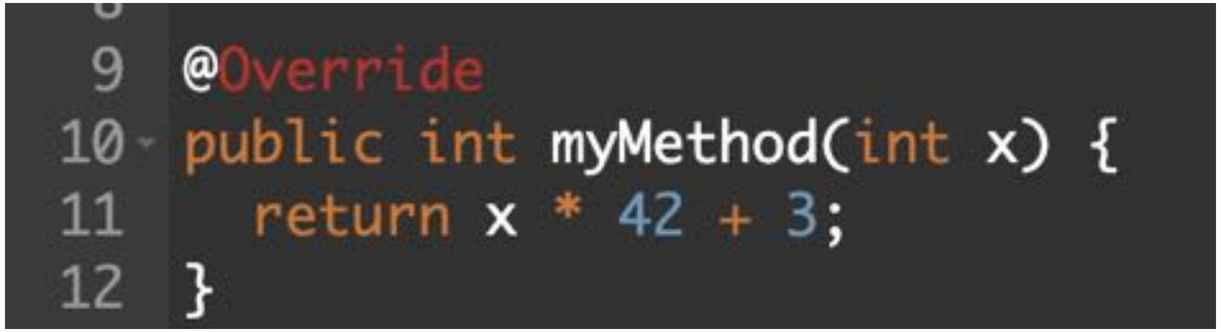
- Allows overload parameter lists on methods
- No user-defined overloading of operators
  - Easily abused
  - Makes confusing code
- No user-defined implicit conversions
  - Good b/c not surprised by them (Least Surprise x 100)

JAVA'S TOSTRING

- Java Object's have
  - public String toString()
  - which specifies how to convert that object to a String
- Does not get used implicitly
  - E.g., cannot pass SomeOtherClass to method that takes String
- May look implicit in certain cases (but not really)
  - Methods that take Object and call toString on them
  - + operator for concatenation

OVERRIDE ANNOTATION

- Use @Override to override a method
- Not required
- Best practice



```
@Override
public int myMethod(int x) {
   return x * 42 + 3;
}
```

- Compiler checks to make sure that you are overriding something in parent
- Helps to avoid errors (e.g., typo, different parameter list, etc)