

Testing, and Debugging

Some reflections from TAs in office hours regarding struggling students:

- “I feel like they get stuck trying to figure it out by looking at their code instead of trying to get proactive and poke at their code”
- “They're checking that their code makes sense; sometimes they can't see the bug until it's being run.”

In lab 3, we talked about how debugging should be a scientific process.

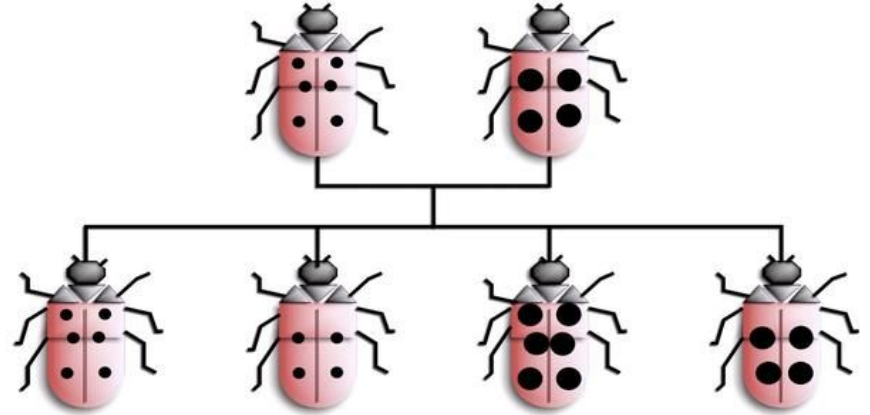
- Reading and re-reading your code is very very slow. It's like trying to chop down a tree with a nail file. Yes, you'll eventually succeed, but there are much better tools.

Using the debugger (especially with tests!) is **incredibly important**.

CS61B, 2021

Lecture 9: More Inheritance!

- Implementation Inheritance: Extends
- Encapsulation
- Casting
- Higher Order Functions in Java



Implementation

Inheritance: Extends

The Extends Keyword

When a class is a hyponym of an interface, we used **implements**.

- Example: `SLList<Blorp> implements List61B<Blorp>`

instead of an interface

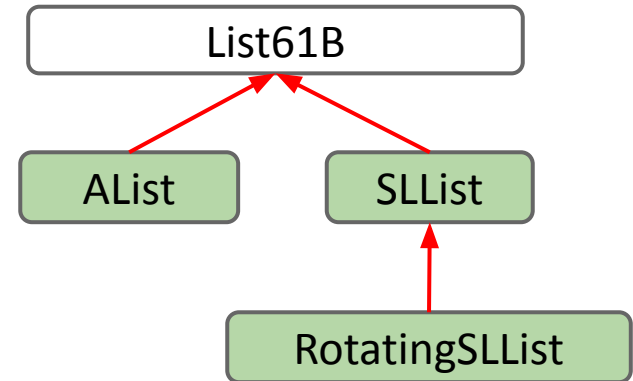
If you want one class to be a hyponym of another *class*, you use **extends**.

We'd like to build `RotatingSLList` that can perform any `SLList` operation as well as:

- `rotateRight()`: Moves back item the front.

Example: Suppose we have `[5, 9, 15, 22]`.

- After `rotateRight`: `[22, 5, 9, 15]`



RotatingSLList

```
public class RotatingSLList<Blorp> extends SLList<Blorp>{  
    public void rotateRight() {  
        Blorp oldBack = removeLast();  
        addFirst(oldBack);  
    }  
}
```

Because of **extends**, RotatingSLList inherits all members of SLList:

- All instance and static variables.
- All methods.
- All nested classes.

... but members may be private and thus inaccessible! More after midterm.

Constructors are not inherited.

Another Example: VengefulSLList

Suppose we want to build an SLList that:


- Remembers all Items that have been destroyed by `removeLast`.
- Has an additional method `printLostItems()`, which prints all deleted items.

```
public static void main(String[] args) {  
    VengefulSLList<Integer> vs1 = new VengefulSLList<Integer>();  
    vs1.addLast(1);  
    vs1.addLast(5);  
    vs1.addLast(10);  
    vs1.addLast(13);      /* [1, 5, 10, 13] */  
    vs1.removeLast();     /* 13 gets deleted. */  
    vs1.removeLast();     /* 10 gets deleted. */  
    System.out.print("The fallen are: ");  
    vs1.printLostItems(); /* Should print 10 and 13. */  
}
```

Another Example: VengefulSLList

```
public class VengefulSLList<Item> extends SLList<Item> {  
    private SLList<Item> deletedItems;  
    public VengefulSLList() {  
        deletedItems = new SLList<Item>();  
    }  
  
    @Override  
    public Item removeLast() {  
        Item oldBack = super.removeLast();  
        deletedItems.addLast(oldBack);  
        return oldBack;  
    }  
  
    public void printLostItems() {  
        deletedItems.print();  
    }  
}
```

calls
Superclass's
version of
removeLast()



Note: Java syntax disallows super.super. For a nice description of why, see [this link](#).

Constructor Behavior Is Slightly Weird

Constructors are not inherited. However, the rules of Java say that **all constructors must start with a call to one of the super class's constructors** [[Link](#)].

- Idea: If every VengefulSLList is-an SLList, every VengefulSLList must be set up like an SLList.
 - If you didn't call SLList constructor, sentinel would be null. Very bad.
- You can explicitly call the constructor with the keyword `super` (no dot).
- If you don't explicitly call the constructor, Java will automatically do it for you.

```
public VengefulSLList() {  
    deletedItems = new SLList<Item>();  
}
```

```
public VengefulSLList() {  
    super(); ← must come first!  
    deletedItems = new SLList<Item>();  
}
```

These constructors are exactly equivalent.

Calling Other Constructors

If you want to use a super constructor other than the no-argument constructor, can give parameters to super.

```
public VengefulSLList(Item x) {  
    super(x); ← calls SLList(Item x)  
    deletedItems = new SLList<Item>();  
}
```

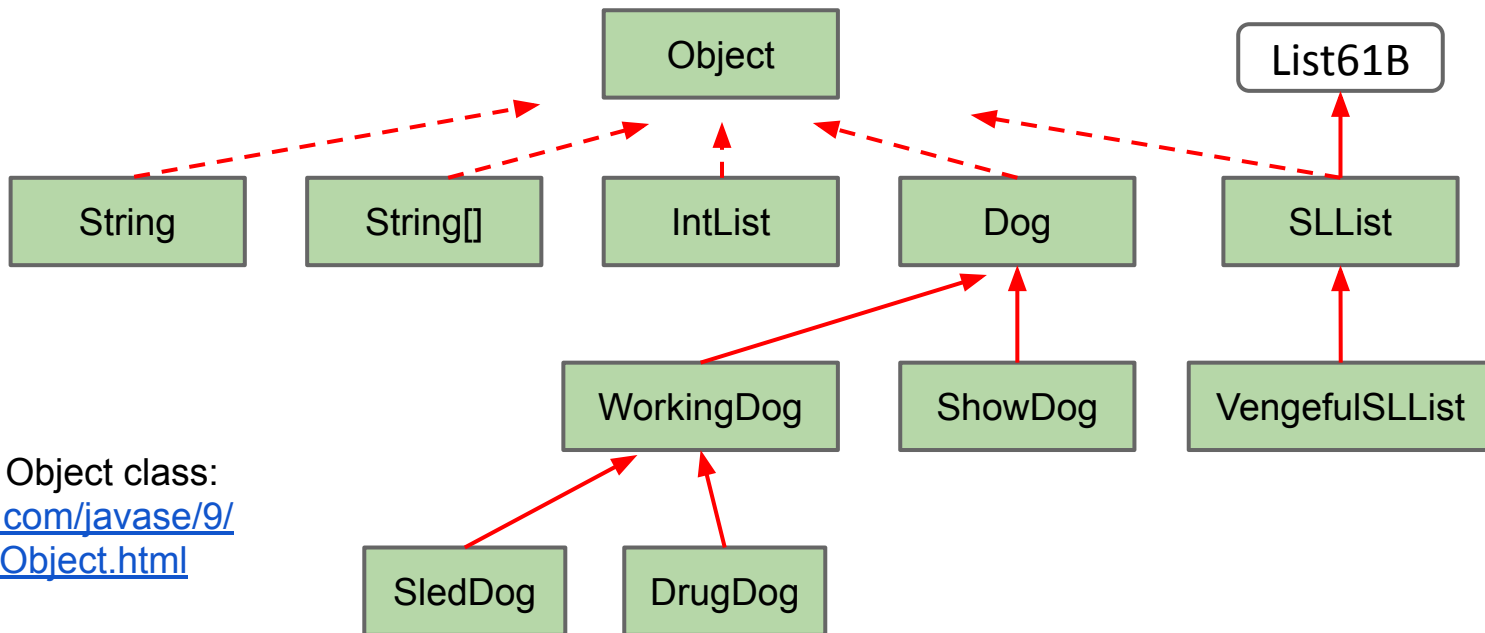
Not equivalent! Code to the right makes implicit call to super(), not super(x).

```
public VengefulSLList(Item x) {  
    deletedItems = new SLList<Item>();  
}
```

The Object Class

As it happens, every type in Java is a descendant of the Object class.

- VengefulSLList extends SLList.
- SLList extends Object (implicitly).



Documentation for Object class:
<https://docs.oracle.com/javase/9/docs/api/java/lang/Object.html>

Interfaces don't extend Object:

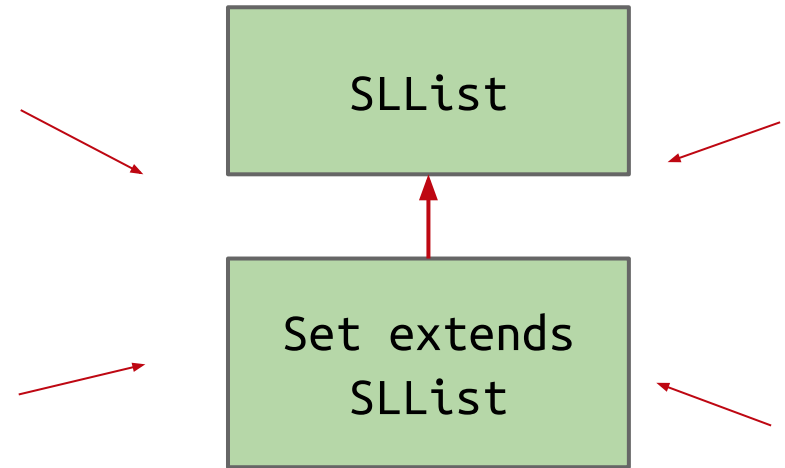
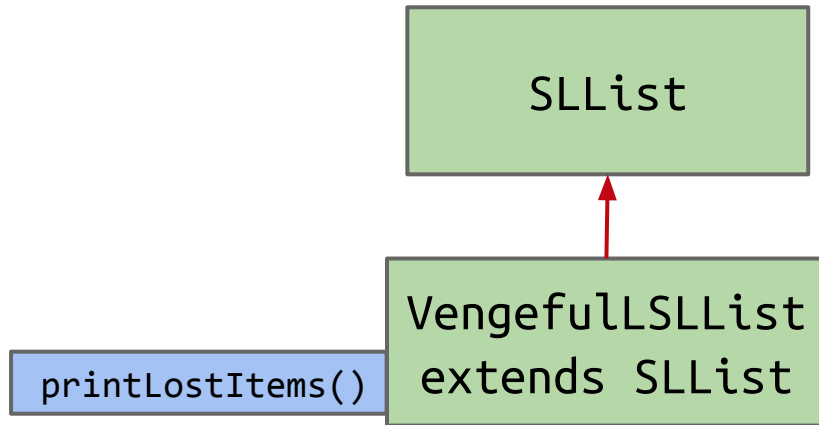
<http://docs.oracle.com/javase/specs/jls/se7/html/jls-9.html#jls-9.2>

Is-a vs. Has-A

Important Note: extends should only be used for **is-a** (hypernymic) relationships!

Common mistake is to use it for “**has-a**” relationships. (a.k.a. meronymic).

- Possible to subclass SLList to build a Set, but conceptually weird, e.g. get(i) doesn't make sense, because sets are not ordered.



This is an abomination.

Encapsulation

Complexity: The Enemy

When building large programs, our enemy is complexity.

Some tools for managing complexity:

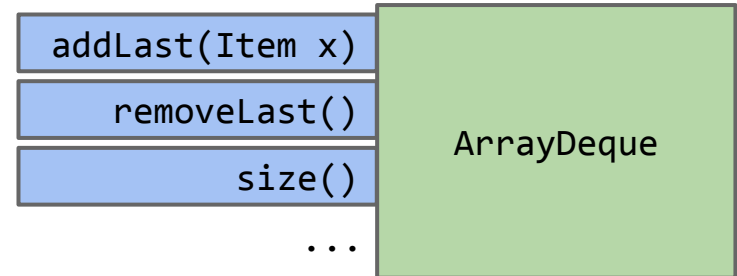
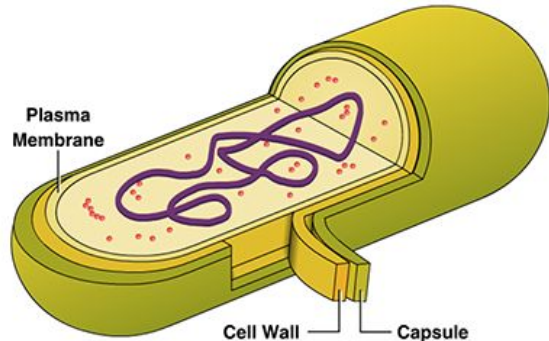
- Hierarchical abstraction.
 - Create **layers of abstraction**, with clear abstraction barriers!
- “Design for change” (D. Parnas)
 - Organize program around objects.
 - Let objects decide how things are done.
 - **Hide information** others don't need.

Managing complexity supremely important for large projects (e.g. project 2).

Modules and Encapsulation [[Shewchuk](#)]

Module: A set of methods that work together as a whole to perform some task or set of related tasks.

A module is said to be **encapsulated** if its implementation is completely hidden, and it can be accessed only through a documented interface.



A Cautionary Tale

Interesting Piazza questions from [proj1gold](#) from a few years ago.

How can we check the length of StudentArrayDeque?

I am trying to find a bug about resizing method, but i don't know how to see the length of the studentArrayDeque.

`StudentArrayDeque.length()` nor `StudentArrayDeque.length` is not working..... so i don't know how to check whether the Array can expand to double of its capacity or not.

Private access in given classes

I wanted to test whether the resizing and downsizing is working properly, but when I try to call `array.items.length`, the compiler yells at me, saying `items` is a private variable. Is there any way around this, or should we just not test this?

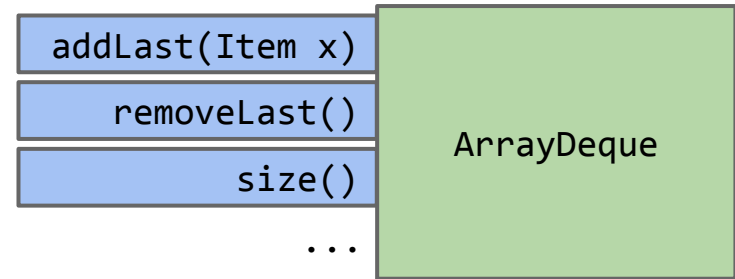
Can we assume these things about studentarraydeque?

Can we assume the studentarraydeque implementation uses `nextfront = 4` `nextlast = 5`, and starting size array 8?

Abstraction Barriers

As the user of an ArrayDeque, you cannot observe its internals.

- Even when writing tests, you don't (usually) want to peer inside.

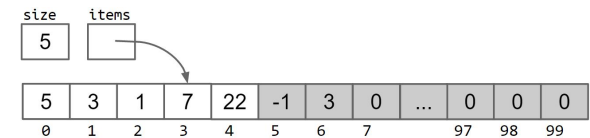


Java is a great language for enforcing abstraction barriers with syntax.

`{5, 3, 1, 7, 22}`



Implementation

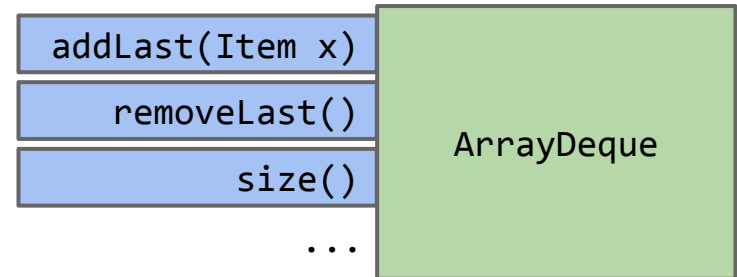
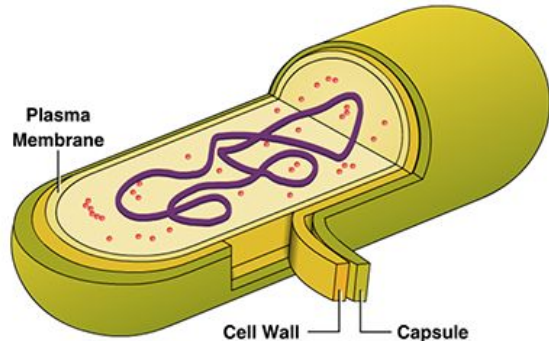


Modules and Encapsulation [[Shewchuk](#)]

Module: A set of methods that work together as a whole to perform some task or set of related tasks.

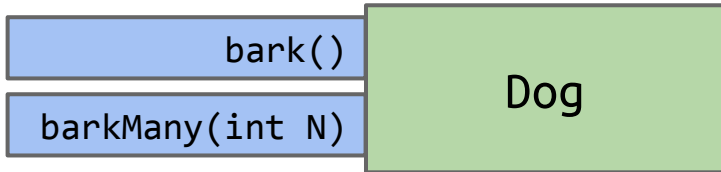
A module is said to be **encapsulated** if its implementation is completely hidden, and it can be accessed only through a documented interface.

- Instance variables private. Methods like `resize` private.
- As we'll see: Implementation inheritance (e.g. `extends`) breaks encapsulation!



Implementation Inheritance Breaks Encapsulation

Suppose we have a Dog class with the two methods shown.

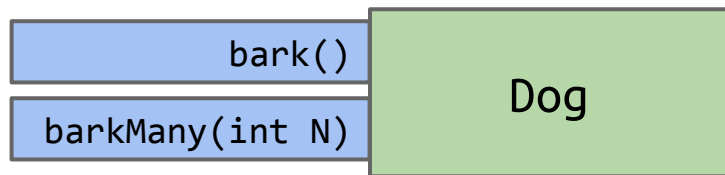


```
public void bark() {  
    System.out.println("bark");  
}  
  
public void barkMany(int N) {  
    for (int i = 0; i < N; i += 1) {  
        bark();  
    }  
}  
  
Dog.java
```

Implementation Inheritance Breaks Encapsulation

We could just as easily have implemented methods as shown below.

- From the outside, functionality is exactly the same, it's just a question of aesthetics.



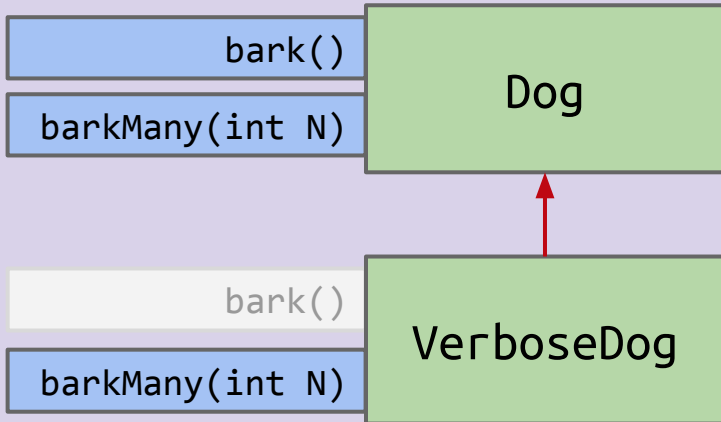
```
public void bark() {  
    barkMany(1);  
}  
  
public void barkMany(int N) {  
    for (int i = 0; i < N; i += 1) {  
        System.out.println("bark");  
    }  
}
```

Dog.java

What would vd.barkMany(3) output?

- a. As a dog, I say: bark bark bark
- b. bark bark bark
- c. Something else.

(assuming vd is a Verbose Dog)



```
public void bark() {
    System.out.println("bark");
}

public void barkMany(int N) {
    for (int i = 0; i < N; i += 1) {
        bark();
    }
}
```

Dog.java

```
@Override
public void barkMany(int N) {
    System.out.println("As a dog, I say: ");
    for (int i = 0; i < N; i += 1) {
        bark(); ← calls inherited bark method
    }
}
```

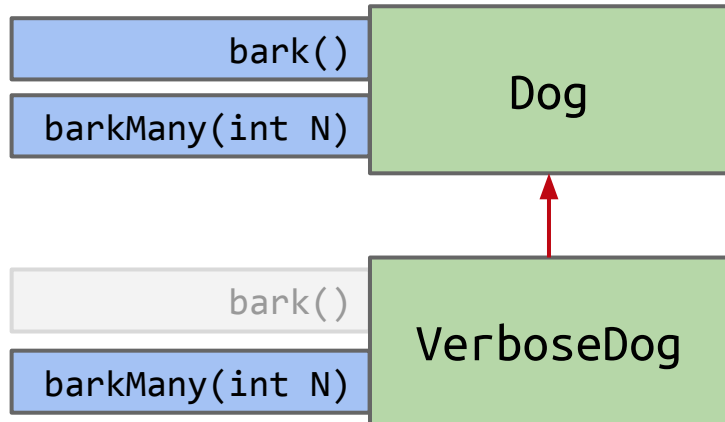
VerboseDog.java

Implementation Inheritance Breaks Encapsulation

What would vd.barkMany(3) output?

- a. **As a dog, I say: bark bark bark**
- b. bark bark bark
- c. Something else.

(assuming vd is a Verbose Dog)



```
public void bark() {
    System.out.println("bark");
}

public void barkMany(int N) {
    for (int i = 0; i < N; i += 1) {
        bark();
    }
}
```

Dog.java

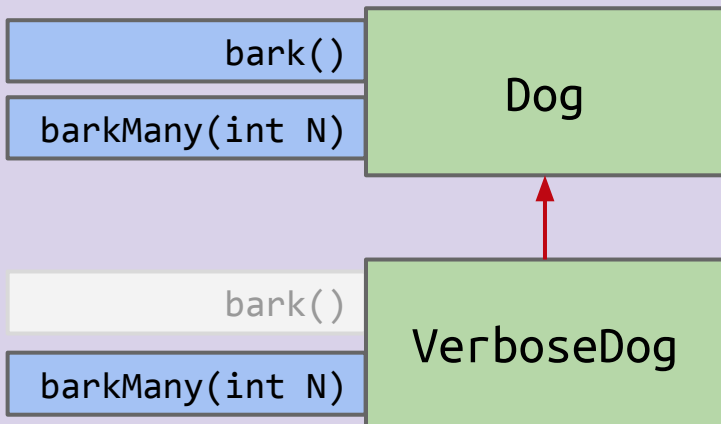
```
@Override
public void barkMany(int N) {
    System.out.println("As a dog, I say: ");
    for (int i = 0; i < N; i += 1) {
        bark(); ← calls inherited bark method
    }
}
```

VerboseDog.java

What would `vd.barkMany(3)` output?

- a. As a dog, I say: bark bark bark
- b. bark bark bark
- c. Something else.

(assuming `vd` is a Verbose Dog)



```
public void bark() {
    barkMany(1);
}

public void barkMany(int N) {
    for (int i = 0; i < N; i += 1) {
        System.out.println("bark");
    }
}
```

Dog.java

```
@Override
public void barkMany(int N) {
    System.out.println("As a dog, I say: ");
    for (int i = 0; i < N; i += 1) {
        bark(); ← calls inherited bark method
    }
}
```

VerboseDog.java

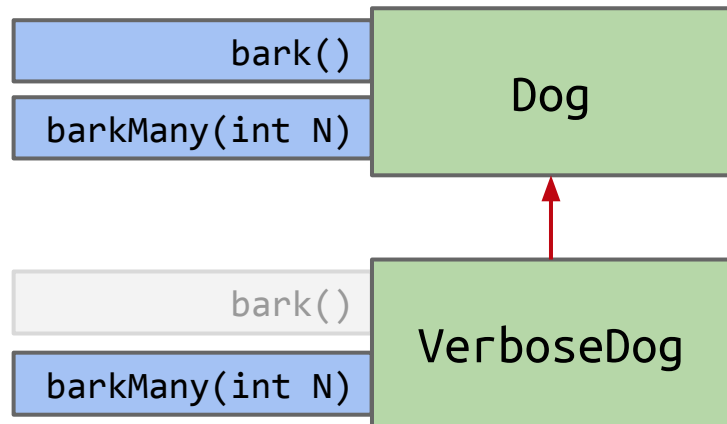
Implementation Inheritance Breaks Encapsulation

What would `vd.barkMany(3)` output?

c. **Something else.**

- Gets caught in an infinite loop!

(assuming `vd` is a Verbose Dog)



```
public void bark() {
    barkMany(1);
}

public void barkMany(int N) {
    for (int i = 0; i < N; i += 1) {
        System.out.println("bark");
    }
}
```

Dog.java

```
@Override
public void barkMany(int N) {
    System.out.println("As a dog, I say: ");
    for (int i = 0; i < N; i += 1) {
        bark(); ← calls inherited bark method
    }
}
```

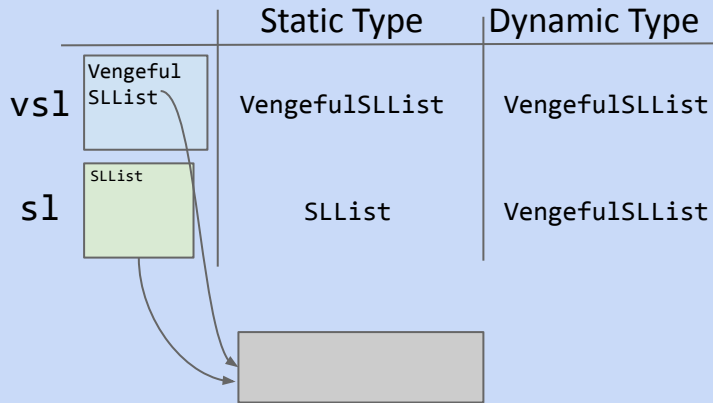
VerboseDog.java

Type Checking and Casting

Dynamic Method Selection and Type Checking Puzzle

For each line of code, determine:

- Does that line cause a compilation error?
- Which method does dynamic method selection use?



Reminder: VengefulSLList overrides `removeLast` and provides a new method called `printLostItems`.

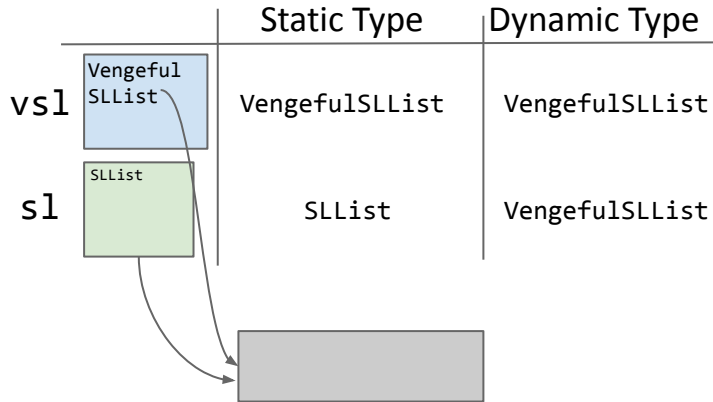
```
public static void main(String[] args) {  
    VengefulSLList<Integer> vs1 =  
        new VengefulSLList<Integer>(9);  
    SLList<Integer> s1 = vs1;  
  
    s1.addLast(50);  
    s1.removeLast();  
  
    s1.printLostItems();  
    VengefulSLList<Integer> vs12 = s1;  
}
```

Reminder: Dynamic Method Selection

Also called
dynamic type.

If overridden, decide which method to call based on **run-time** type of variable.

- sl's runtime type: VengefulSLList.



```
public static void main(String[] args) {  
    VengefulSLList<Integer> vs1 =  
        new VengefulSLList<Integer>(9);  
    SLList<Integer> sl = vs1;  
  
    sl.addLast(50);  
    sl.removeLast();  
}
```

VengefulSLList
doesn't override,
uses SLList's.

Uses VengefulSLList's.

Reminder: VengefulSLList overrides
removeLast and provides a new method called
printLostItems.

Compile-Time Type Checking

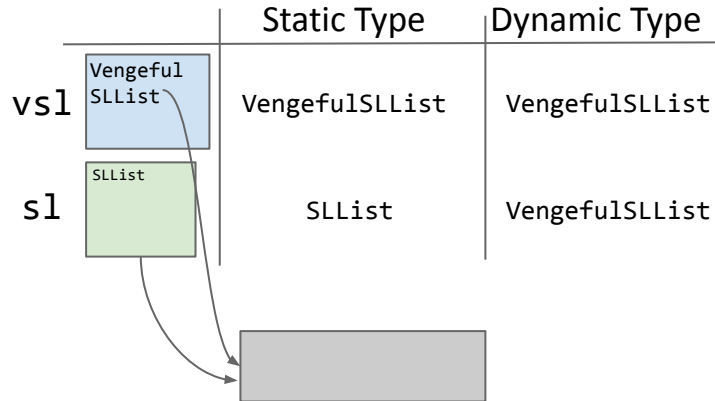
Also called
static type.

Compiler allows method calls based on **compile-time** type of variable.

- `sl`'s runtime type: `VengefulSLList`.
- But cannot call `printLostItems`.

```
public static void main(String[] args) {  
    VengefulSLList<Integer> vs1 =  
        new VengefulSLList<Integer>(9);  
    SLList<Integer> sl = vs1;  
  
    sl.addLast(50);  
    sl.removeLast();  
  
    sl.printLostItems();  
}
```

Compilation
error!



Reminder: `VengefulSLList` overrides `removeLast` and provides a new method called `printLostItems`.

Compile-Time Type Checking

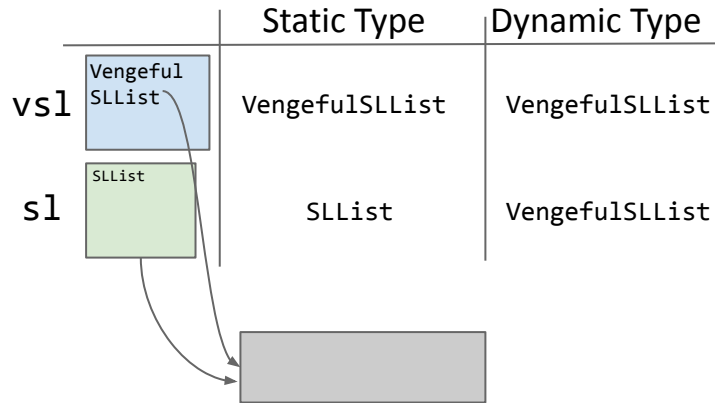
Also called
static type.

Compiler allows method calls based on **compile-time** type of variable.

- `sl`'s runtime type: `VengefulSLList`.
- But cannot call `printLostItems`.

```
public static void main(String[] args) {  
    VengefulSLList<Integer> vs1 =  
        new VengefulSLList<Integer>(9);  
    SLList<Integer> sl = vs1;  
  
    sl.addLast(50);  
    sl.removeLast();  
  
    sl.printLostItems();  
    VengefulSLList<Integer> vs12 = sl;  
}
```

Compilation
errors!



Compiler also allows assignments based on compile-time types.

- Even though `sl`'s runtime-type is `VengefulSLList`, cannot assign to `vs12`.
- Compiler plays it as safe as possible with type checking.

Compile-Time Types and Expressions

Expressions have compile-time types:

- An expression using the new keyword has the specified compile-time type.

```
SLList<Integer> s1 = new VengefulSLList<Integer>();
```

- Compile-time type of right hand side (RHS) expression is VengefulSLList.
- A VengefulSLList is-an SLList, so assignment is allowed.

```
VengefulSLList<Integer> vs1 = new SLList<Integer>();
```

- Compile-time type of RHS expression is SLList.
- An SLList is not necessarily a VengefulSLList, so compilation error results.

Compilation error!

Compile-Time Types and Expressions

Expressions have compile-time types:

- Method calls have compile-time type equal to their declared type.

```
public static Dog maxDog(Dog d1, Dog d2) { ... }
```

- Any call to maxDog will have compile-time type Dog!

Example:

```
Poodle frank = new Poodle("Frank", 5);  
Poodle frankJr = new Poodle("Frank Jr.", 15);  
  
Dog largerDog = maxDog(frank, frankJr);  
Poodle largerPoodle = maxDog(frank, frankJr);
```

Compilation error!

RHS has
compile-time type
Dog.

Casting

Java has a special syntax for specifying the compile-time type of any expression.

- Put desired type in parenthesis before the expression.
- Examples:

- Compile-time type Dog:

```
maxDog(frank, frankJr);
```

- Compile-time type Poodle:

```
(Poodle) maxDog(frank, frankJr);
```

Tells compiler to pretend it sees a particular type.

Compilation OK!
RHS has compile-time type Poodle.

```
Poodle frank = new Poodle("Frank", 5);  
Poodle frankJr = new Poodle("Frank Jr.", 15);  
Dog largerDog = maxDog(frank, frankJr);  
Poodle largerPoodle = (Poodle) maxDog(frank, frankJr);
```

Casting

Casting is a powerful but dangerous tool.

- Tells Java to treat an expression as having a different compile-time type.
- In example below, effectively tells the compiler to ignore its type checking duties.
- Does not actually change anything: sunglasses don't make the world dark.

```
Poodle frank = new Poodle("Frank", 5);  
Malamute frankSr = new Malamute("Frank Sr.", 100);  
  
Poodle largerPoodle = (Poodle) maxDog(frank, frankSr);
```

If we run the code above, we get a `ClassCastException` at runtime.

- So much for .class files being verifiably type checked...

Dynamic Method Selection and Casting Puzzle

Is it Overriding? Overloading?

```
public class Bird {  
    public void gulgate(Bird b) {  
        System.out.println("BiGulBi");  
    }  
}
```

```
public class Falcon extends Bird {  
    public void gulgate(Falcon f) {  
        System.out.println("FaGulFa");  
    }  
}
```

```
Bird bird = new Falcon();  
Falcon falcon = (Falcon) bird;  
bird.gulgate(falcon);  
falcon.gulgate(falcon);
```

What gets printed?

- a. BiGulBi BiGulBi
- b. BiGulBi FaGulFa
- c. FaGulFa BiGulBi
- d. FaGulFa FaGulFa

Is it Overriding? Overloading?

```
public class Bird {  
    public void gulgate(Bird b) {  
        System.out.println("BiGulBi");  
    }  
}
```

Casting causes no change to the bird variable, nor to the object the bird variable points at!

```
public class Falcon extends Bird {  
    public void gulgate(Falcon f) {  
        System.out.println("FaGulFa");  
    }  
}
```

```
Bird bird = new Falcon();  
Falcon falcon = (Falcon) bird;  
bird.gulgate(falcon);  
falcon.gulgate(falcon);
```

What gets printed?

b. BiGulBi FaGulFa

Why does BiGulBi get printed first?

```
public class Bird {  
    public void gulgate(Bird b) {  
        System.out.println("BiGulBi");  
    }  
}
```

An earlier version of this slide said “since there is no overriding.”

```
public class Falcon extends Bird {  
    public void gulgate(Falcon f) {  
        System.out.println("FaGulFa");  
    }  
}
```

```
Bird bird = new Falcon();  
Falcon falcon = (Falcon) bird;  
bird.gulgate(falcon);
```

Remember: The compiler chooses the most specific matching method signature from the static type of the invoking class.

- Falcon is overloading the gulgate method, not overriding.
- Compiler basically thinks “does Bird class have a gulgate method? Yes! I’ll use that”. Since there is no overriding, no dynamic method selection occurs.

Higher Order Functions (A First Look)

Higher Order Functions

Higher Order Function: A function that treats another function as data.

- e.g. takes a function as input.

Example in Python:

```
def tenX(x):  
    return 10*x  
  
def do_twice(f, x):  
    return f(f(x))  
  
print(do_twice(tenX, 2))
```

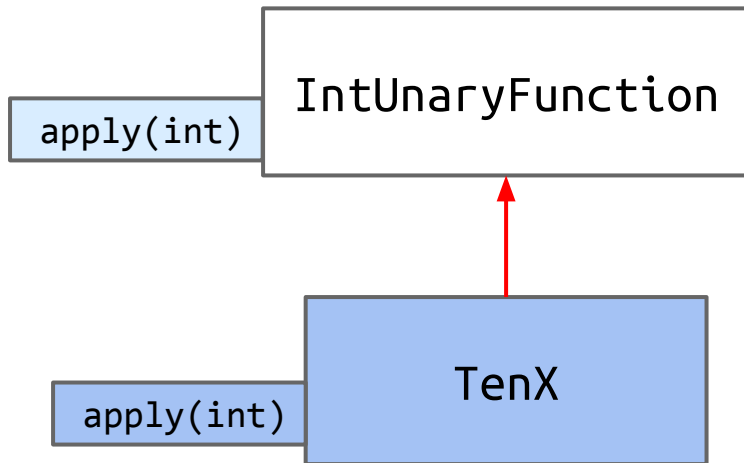
200

Higher Order Functions in Java 7

Old School (Java 7 and earlier)

- Fundamental issue: Memory boxes (variables) cannot contain pointers to functions.

Can use an interface instead. Let's try it out.



```
def tenX(x):  
    return 10*x  
  
def do_twice(f, x):  
    return f(f(x))  
  
print(do_twice(tenX, 2))
```

Higher Order Functions in Java 7

Old School (Java 7 and earlier)

- Fundamental issue: Memory boxes (variables) cannot contain pointers to functions.

Can use an interface instead: Java code below is equivalent to given python code.

```
public interface IntUnaryFunction {  
    int apply(int x);  
}
```

```
public class TenX implements IntUnaryFunction {  
    public int apply(int x) {  
        return 10 * x;  
    }  
}
```

```
def tenX(x):  
    return 10*x
```


Example: Higher Order Functions Using Interfaces in Java

```
public interface IntUnaryFunction {  
    int apply(int x);  
}
```

```
public class TenX implements IntUnaryFunction {  
    public int apply(int x) {  
        return 10 * x;  
    }  
}
```

```
public class HoFDemo {  
    public static int do_twice(IntUnaryFunction f, int x) {  
        return f.apply(f.apply(x));  
    }  
    public static void main(String[] args) {  
        System.out.println(do_twice(new TenX(), 2));  
    }  
}
```

```
def tenX(x):  
    return 10*x
```

```
def do_twice(f, x):  
    return f(f(x))
```

```
print(do_twice(tenX, 2))
```

Example: Higher Order Functions in Java 8 or Later

In Java 8, new types were introduced: now can can hold references to methods.

- You're welcome to use these features, but we won't teach them.
- Why? The old way is still widely used, e.g. Comparators (see next lecture).

```
public class Java8HoFDemo {  
    public static int tenX(int x) {  
        return 10*x;  
    }  
    public static int doTwice(Function<Integer, Integer> f, int x) {  
        return f.apply(f.apply(x));  
    }  
    public static void main(String[] args) {  
        int result = doTwice(Java8HoFDemo::tenX, 2);  
        System.out.println(result);  
    }  
}
```

Implementation Inheritance Cheatsheet

VengefulSLList extends SLList means a VengefulSLList is-an SLList. Inherits all members!

- Variables, methods, nested classes.
- Not constructors.
- Subclass constructor must invoke superclass constructor first.
- Use super to invoke overridden superclass methods and constructors.

Invocation of overridden methods follows two simple rules:

- Compiler plays it safe and only lets us do things allowed by **static** type.
- For overridden methods the actual method invoked is based on **dynamic** type of invoking expression, e.g. Dog.maxDog(d1, d2).bark();
- Can use casting to overrule compiler type checking.

Does not apply to
overloaded methods!



Extra Problem Just For Fun

Type Checking Quiz!

```
ShowDog dogC = new ShowDog("Franklin", "Malamute", 180, 6);  
ShowDog dogD = new ShowDog("Gargamel", "Corgi", 44, 12);
```

```
Dog.maxDog(dogC, dogD);
```

1. What is the static type of `Dog.maxDog(dogC, dogD)`?

2. Which (if any), will compile:

```
Dog md = Dog.maxDog(dogC, dogD);  
ShowDog msd = Dog.maxDog(dogC, dogD);
```

3. How many memory boxes are there in the code below? What are the dynamic types of their contents?

```
Object o = new Dog("Hammy", "Beagle", 15);  
Dog d = new Dog("Ammo", "Labrador", 54);  
Object stuff[] = new Object[5];  
stuff[0] = o;  
stuff[1] = d;  
stuff[2] = null;
```

Citations

<https://wikids-life.wikispaces.com/file/view/LadybirdInheritance.jpg/160451153/604x297/LadybirdInheritance.jpg>

Actual truth:

