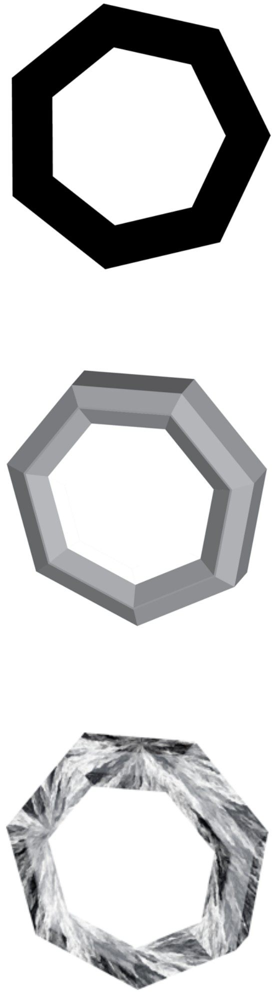
CS61B

Lecture 9: Subtype Polymorphism

- OOP and Polymorphism
- Inheritance (is-a) in Java
- Dynamic Method Selection
- Subtleties



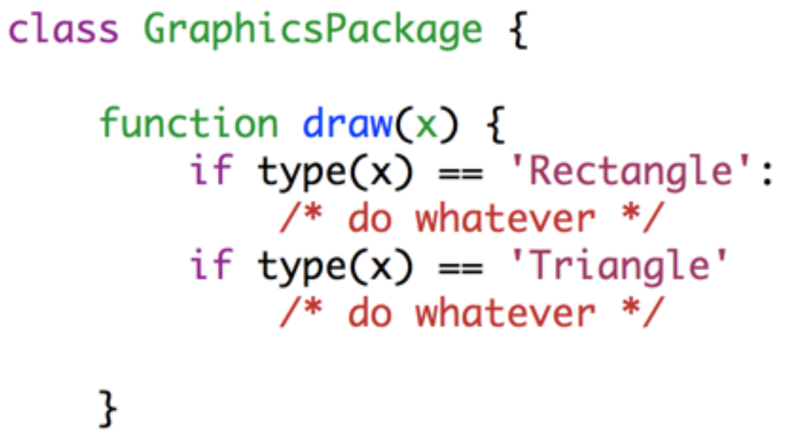
Motivation: Operating on Many Types

Let’s design part of a programming language.

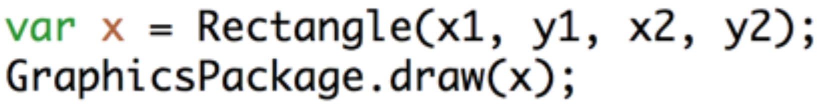
Goal: Given a shape stored in variable x, what syntax should our language provide to draw x?

- Assume x may be one of many different types.
- Assume drawing is fairly different for each type (no easy code reuse).

Approach #1: Check Type Manually



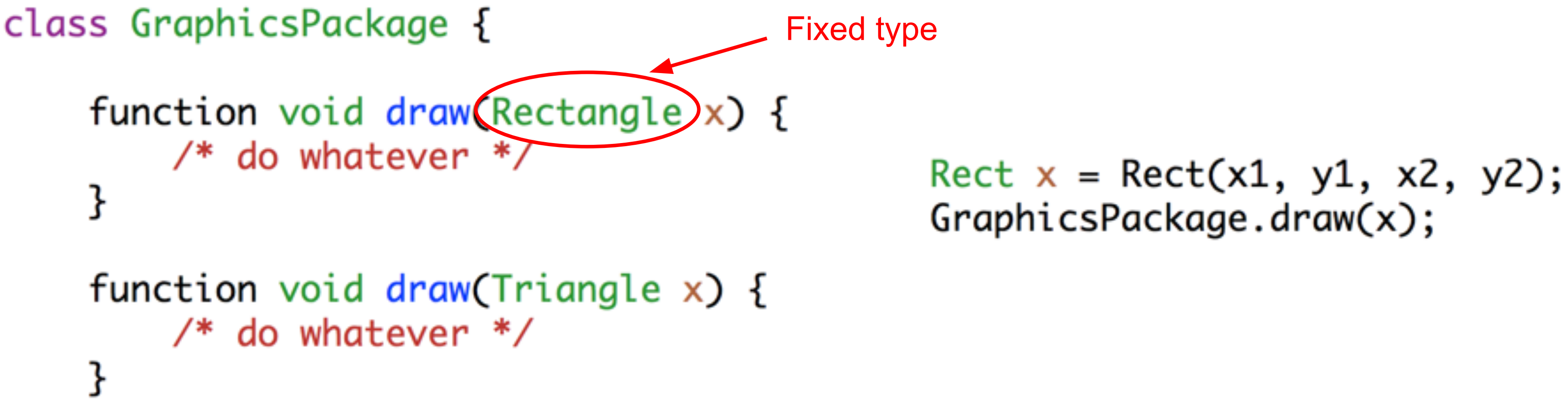
```
class GraphicsPackage {

    function draw(x) {
        if type(x) == 'Rectangle':
            /* do whatever */
        if type(x) == 'Triangle'
            /* do whatever */

    }
```



```
var x = Rectangle(x1, y1, x2, y2);
GraphicsPackage.draw(x);
```

- Bad: Requires manual type checking (yuck).
- Bad: Change in Rectangle means you have to go change GraphicsPackage.

Approach #2: Function Overloading (a.k.a. ad hoc polymorphism)



```
class GraphicsPackage {

    function void draw(Rectangle x) {
        /* do whatever */
    }

    function void draw(Triangle x) {
        /* do whatever */
    }
```

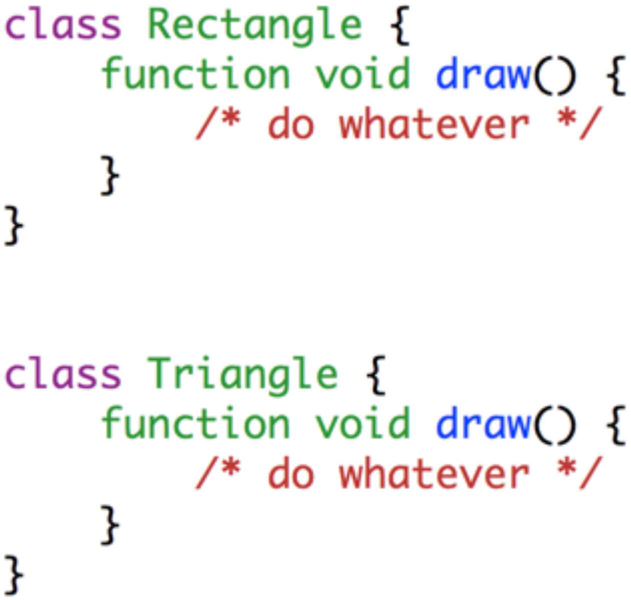
Fixed type

```
Rect x = Rect(x1, y1, x2, y2);
GraphicsPackage.draw(x);
```

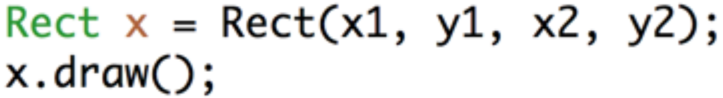
In a language with fixed container types.

- Bad: Change in Rectangle means you have to change GraphicsPackage (and any other client of Rectangle).

Approach #3: Let the Objects Handle the Problem



```
class Rectangle {
    function void draw() {
        /* do whatever */
    }
}


class Triangle {
    function void draw() {
        /* do whatever */
    }
}
```



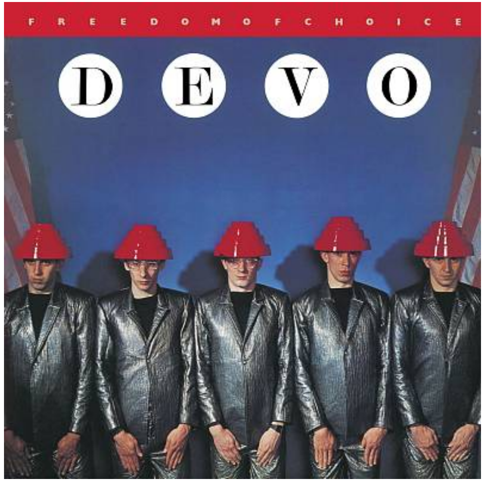
```
Rect x = Rect(x1, y1, x2, y2);
x.draw();
```

Complexity Is the Enemy

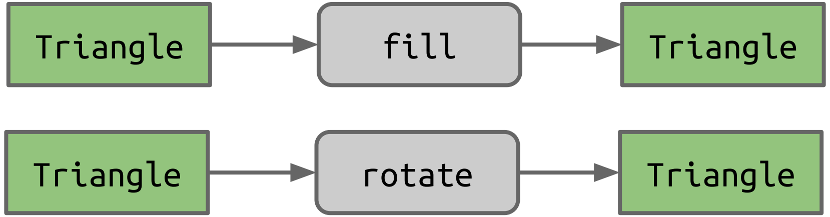
Chapter 2 of HFJ has a cute narrative displaying the power of functions

Tools for managing complexity.

- Hierarchical abstraction.
- “Design for change” (D. Parnas)
  - **Organize program around objects**.
  - **Let objects decide** how things are done.
  - **Hide information** others don’t need.



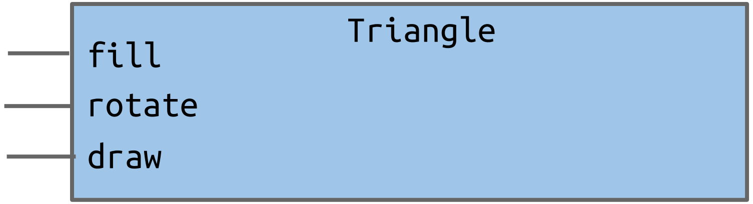
Function based:



Triangle → fill → Triangle

Triangle → rotate → Triangle

Object based:



Triangle

fill

rotate

draw

Public methods

Redundancy and Generality

Organizing around objects is not quite enough:

- Similar types of objects have redundant implementations.
- Functions that operate on similar types of objects are redundant.

```
class Rect {
  Rect(String filename)
  function void draw()
  function Rect rotatedCopy(...)
}
class Triangle {
  Triangle(String filename)
  function void draw()
  function Triangle rotatedCopy(...)
  function boolean isIsoceles()
}
```

```
function void drawTwo(Rectangle x) {
    x.draw();
    x.rotatedCopy(90).draw();
}
function void drawTwo(Triangle x) {
    x.draw();
    x.rotatedCopy(90).draw();
}

Rect r = new Rect(1, 1, 5, 5);
drawTwo(r);
```

Python/Scheme/Matlab approach: Don’t assign fixed types to containers.

Inheritance

Common approach in many programming languages:

- Introduce an ‘is-a’ relationship between classes.
- Examples:
  - Rect is-a Shape
  - SList is-a List (Wednesday)

```
class Rect is-a Shape {
    ...
}
```

```
function void drawTwo(Shape x) {
    x.draw();
    x.rotatedCopy(90).draw();
}

Rect x = new Rect(1, 1, 5, 5);
drawTwo(x);
```

Subtype Polymorphism

The behavior of drawTwo depends on the particular subtype of Shape that is passed to the function.

- This idea is known as ‘subtype polymorphism’.

```
class Rect is-a Shape {
    ...
}
```

```
function void drawTwo(Shape x) {
     x.draw();
     x.rotatedCopy(90).draw();
}

Rect x = new Rect(1, 1, 5, 5);
drawTwo(x);
```

Python/Scheme/Matlab approach: Don’t assign fixed types to containers.

Inheritance (Is-A) in Java

The MaxSList

Suppose we want to add the ability to retrieve the max item from a list.

- Could add max instance variable, and any time an item is inserted, update the max.

Alternate approach: Create a new subclass of SList called MaxSList.

- Create a new class MaxSList that inherits all the basic properties of an SList.
- Add the additional properties needed to support the max() operation.

Subclassing

Subclasses can modify or augment superclass in many ways, including:

- Adding new fields.
- Adding new methods.
- Adding new constructors.
- Overriding existing methods with new implementations.

Let’s try to implement MaxSList by extending our SList from lecture 6.

- If you want to try on your own, see https://github.com/Berkeley-CS61B/lectureCode/tree/master/lec9/exercises for starter code.

Constructor Behavior (Is Slightly Weird)

Constructors are not inherited. However, the rules of Java say that all subclass constructors must **start** with a call to a constructor for SList.

- Idea: If every MaxSList is an SList, every MaxSList must be set up like an SList.
- To call a constructor, use the keyword super, which can take parameters, e.g. super(x). Appropriate constructor is called based on type and number of parameters.

Slightly Weird Things:

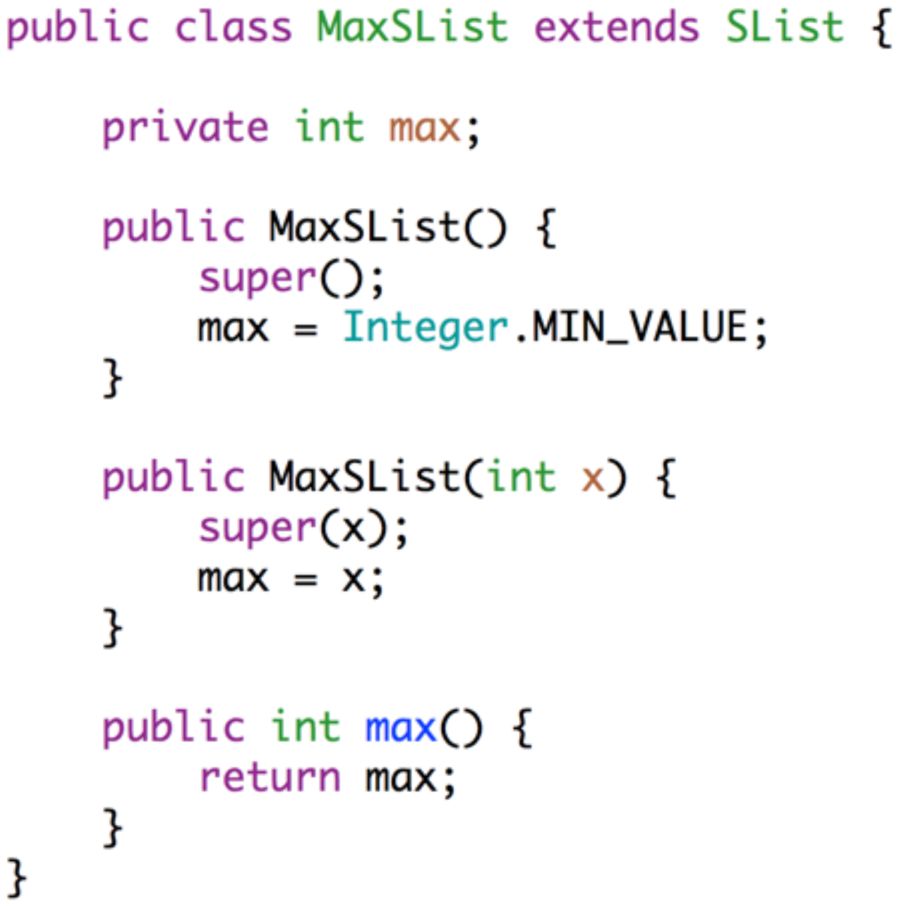
- If (and only if) you don’t include any constructors, there is an implicit no-argument constructor that just calls super().
- If you don’t use super at all, there is an implicit call to super() with no parameters.

Override

To the right, we see that we have:

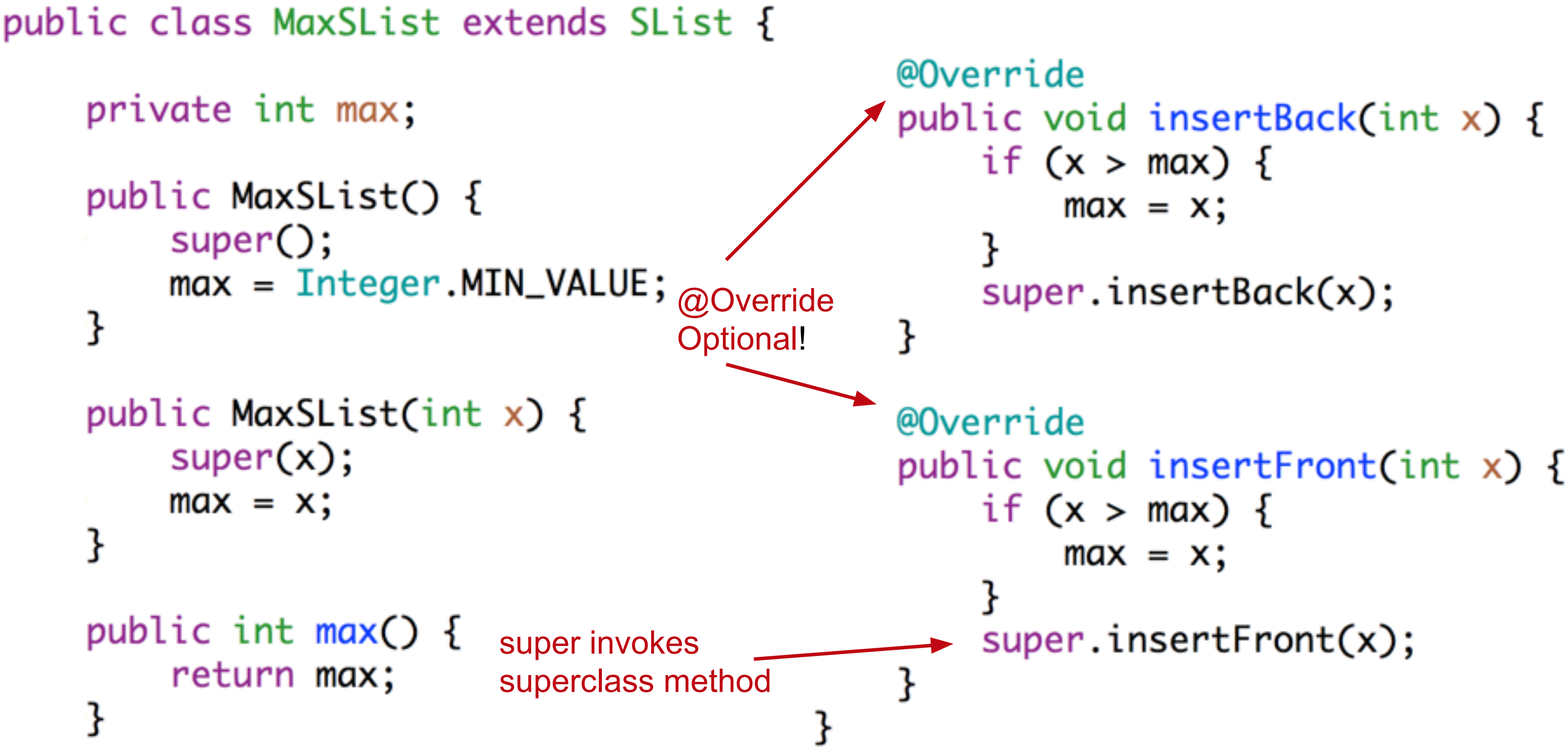
- New fields (a.k.a. variables).
- New constructors.
- New methods.

To complete MaxSList, we need to override the insertFront() and insertBack() methods.



```
public class MaxSList extends SList {

    private int max;

    public MaxSList() {
        super();
        max = Integer.MIN_VALUE;
    }

    public MaxSList(int x) {
        super(x);
        max = x;
    }

    public int max() {
        return max;
    }
}
```

The Glorious MaxSList



```java
public class MaxSList extends SList {

    private int max;

    public MaxSList() {
        super();
        max = Integer.MIN_VALUE;
    }

    public MaxSList(int x) {
        super(x);
        max = x;
    }

    public int max() {
        return max;
    }

    @Override
    public void insertBack(int x) {
        if (x > max) {
            max = x;
        }
        super.insertBack(x);
    }

    @Override
    public void insertFront(int x) {
        if (x > max) {
            max = x;
        }
        super.insertFront(x);
    }
}
```

@Override
Optional!

super invokes
superclass method

Syntax Summary

The extends keyword indicates that a class is a subclass of another.

- Example: MaxSList extends SList

The super keyword is used for:

- Invoking superclass constructor, e.g. super(9);
- Invoking superclass method., e.g. super.insertFront(5);
- Using superclass fields (bleh x 10000000!!)

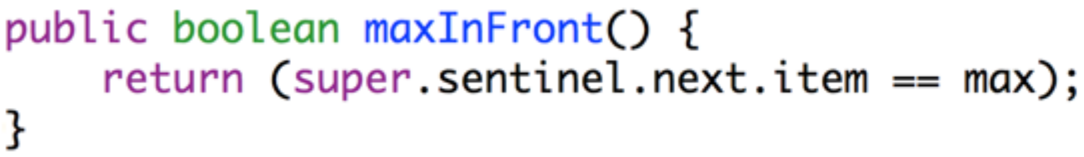
The @Override annotation is good for two things:

- Compiler verifies that you are actually overriding a method.
  - Prevents mistakes like `insertFornt(int x)`
- Slightly easier to read for other programmers.

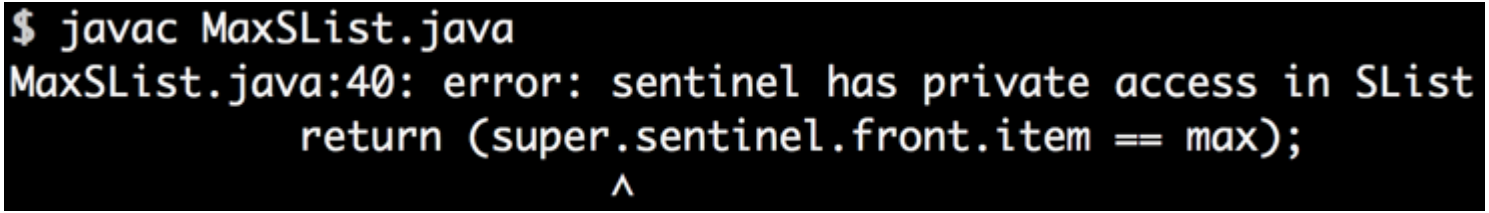
Private Access

The private keyword prevents any other class from accessing data, even subclasses.

Suppose we want a MaxSList method that returns whether the max is in front:



```
public boolean maxInFront() {
    return (super.sentinel.next.item == max);
}
```

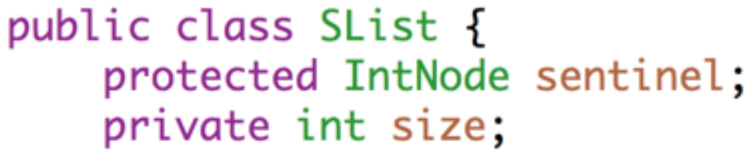


```
$ javac MaxSList.java
MaxSList.java:40: error: sentinel has private access in SList
            return (super.sentinel.front.item == max);
                         ^
```

The Protected Keyword

To allow only subclasses to use a field or method, declare it protected.

- Example below: Any subclass of SList may access sentinel.



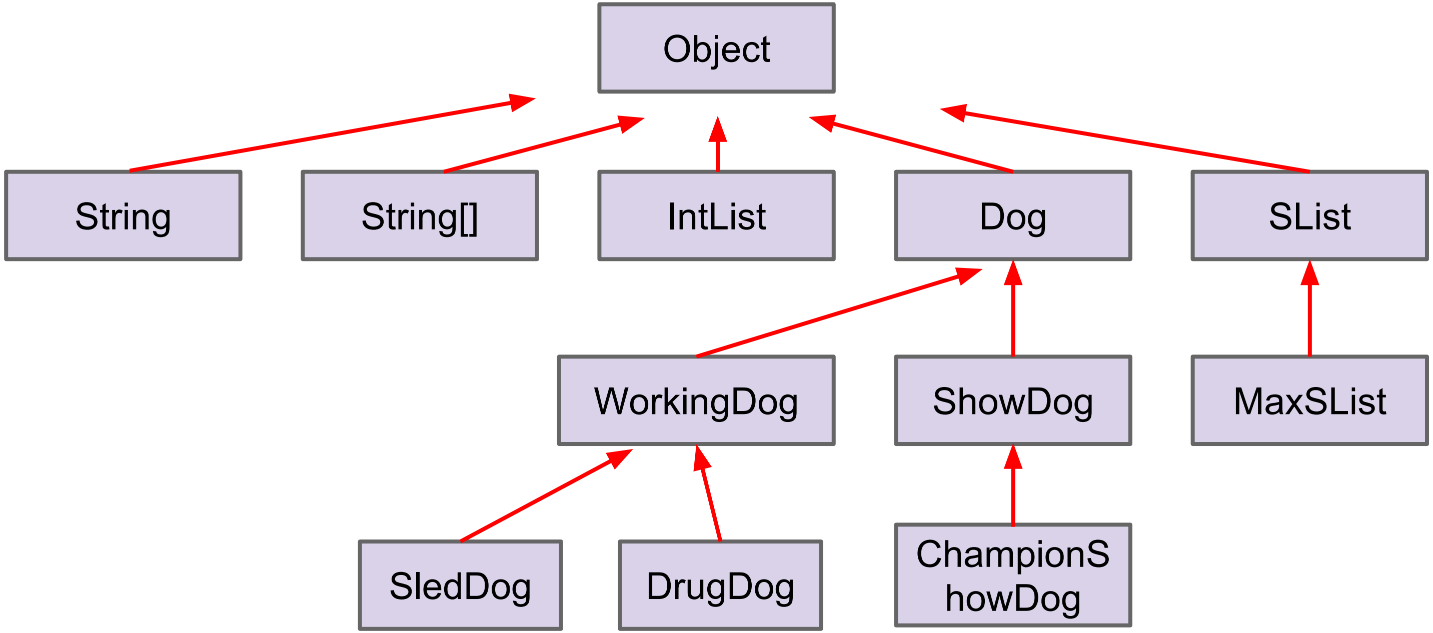
```
public class SList {
    protected IntNode sentinel;
    private int size;
```

Dynamic Method Selection

Containers and Type Hierarchies

Reference types form a type hierarchy.

i.e. non-primitive types

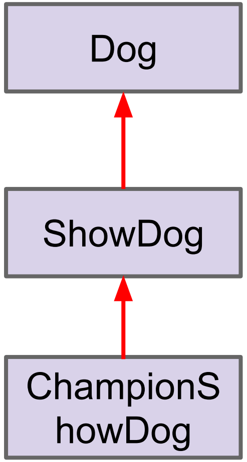


Extremely Important Point of the Day!!!

Every ShowDog **is a** Dog.

- Java allows a Dog variable to point at a ShowDog.
  - Simple container of type Dog can hold address of a ShowDog.

```
/* two lines of valid Java code */
Dog d3 = new ShowDog("Ralph", "Husky");
d3.bark();
```



But not every Dog is a ShowDog.

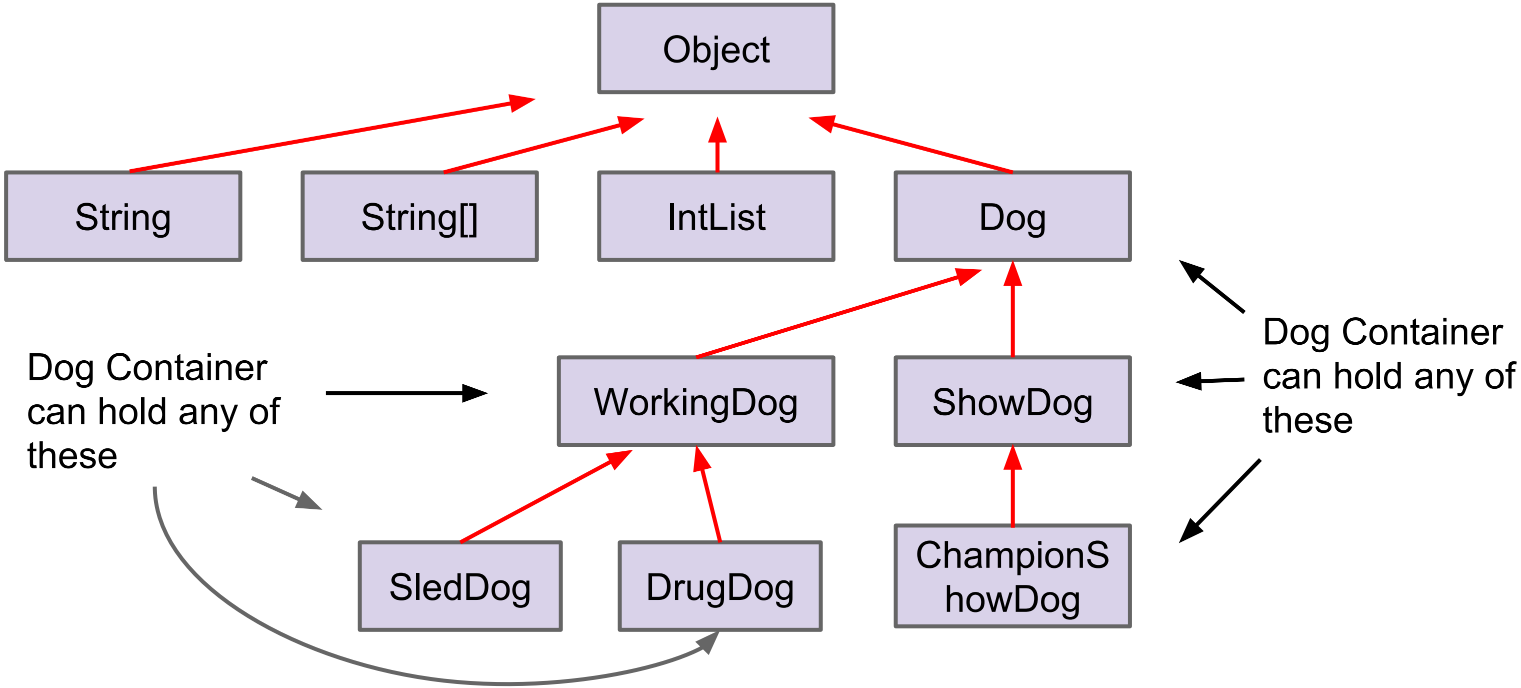
- Code below results in a compile time warning:

```
/* Invalid Java code */
ShowDog sd = new Dog("Ralph", 15);
```

Containers and Type Hierarchies

Reference types form a type hierarchy.

i.e. non-primitive types



Static Vs. Dynamic Type

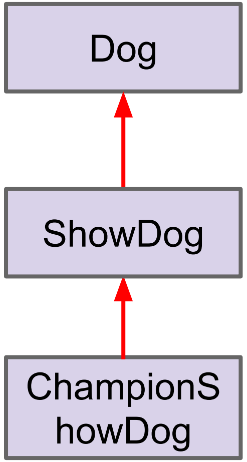
Important Terms:

- ***Static type*** of d3 is Dog.
- ***Dynamic type*** of d3 is ShowDog.
    - Value in d3 points to a ShowDog object.

```
/* two lines of valid Java code */
Dog d3 = new ShowDog("Ralph", "Husky");
d3.bark();
```

Other extremely important point of the day.

- When an overridden method is invoked, the actual method that executes is based on dynamic type, not static type.



Output Prediction

What will be the output of the code below?

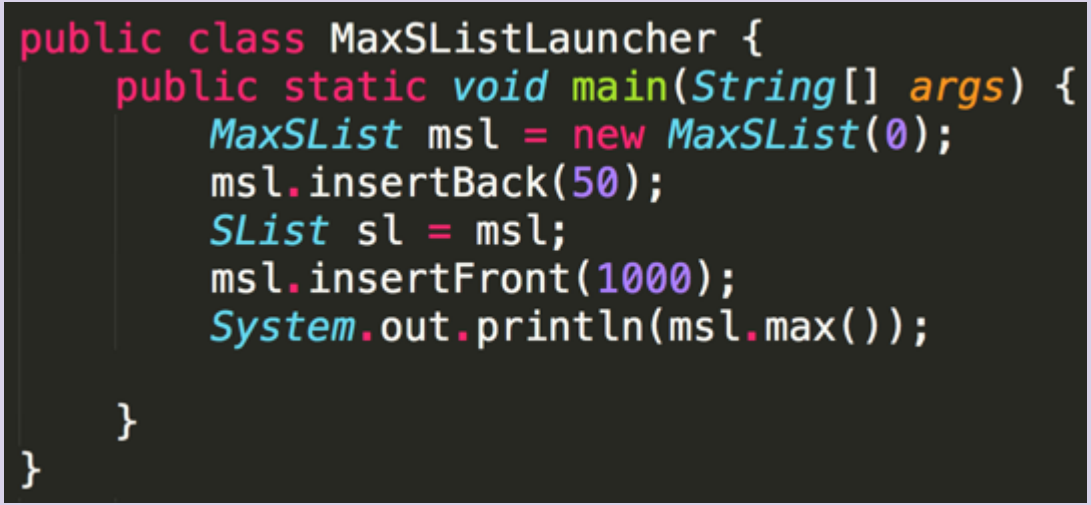
A. -2147483648

B. 0

C. 50

D. 1000

E. Compilation Error.

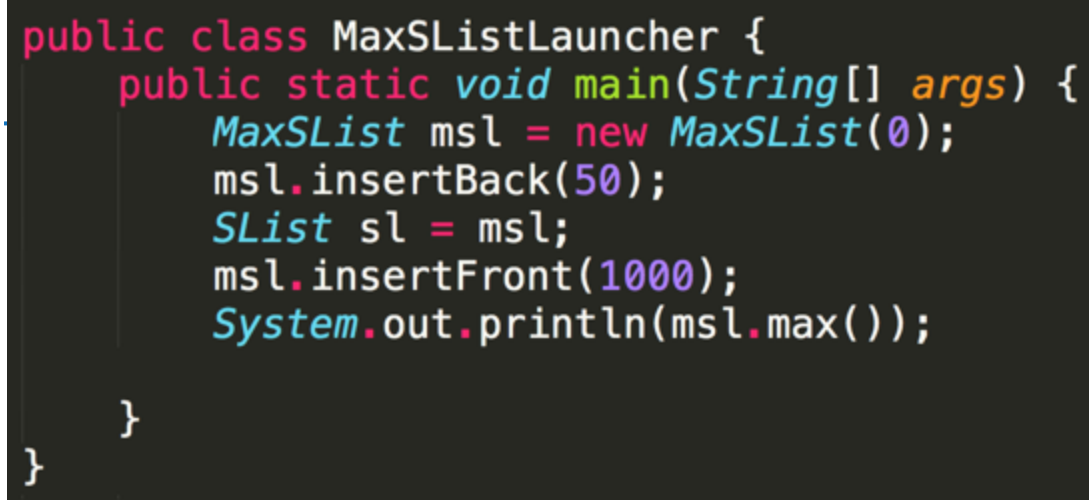


```
public class MaxSListLauncher {
    public static void main(String[] args) {
        MaxSList msl = new MaxSList(0);
        msl.insertBack(50);
        SList sl = msl;
        msl.insertFront(1000);
        System.out.println(msl.max());

    }
}
```

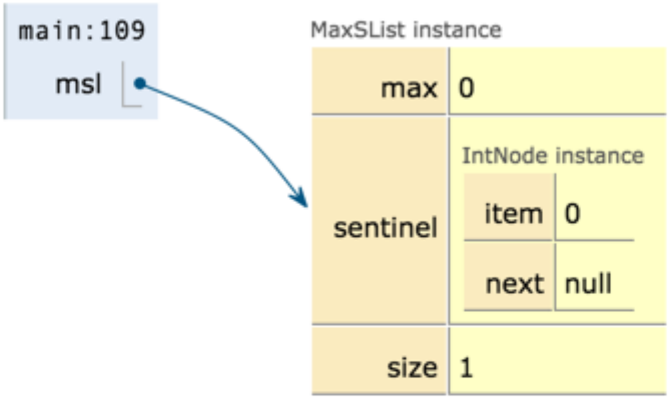
Output Prediction

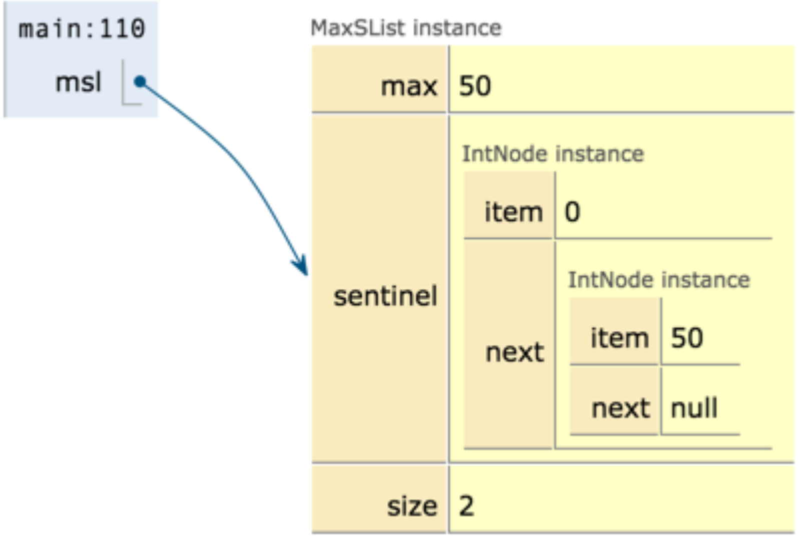
What will be the output?

D.   1000



```
public class MaxSListLauncher {
    public static void main(String[] args) {
        MaxSList msl = new MaxSList(0);
        msl.insertBack(50);
        SList sl = msl;
        msl.insertFront(1000);
        System.out.println(msl.max());

    }
}
```

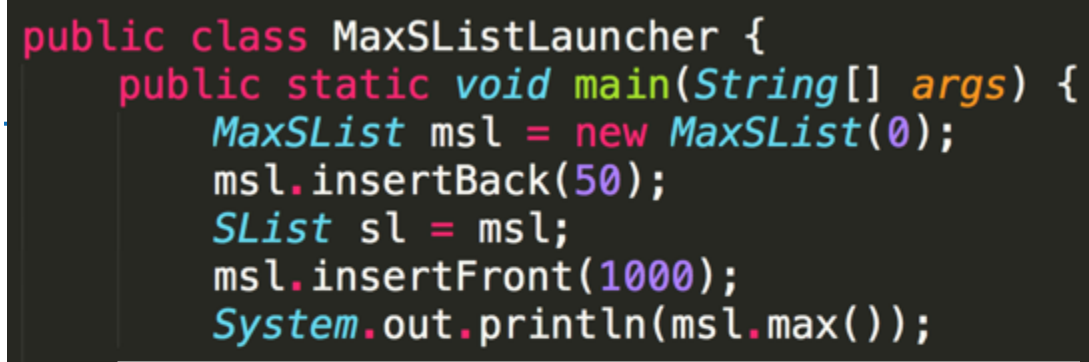




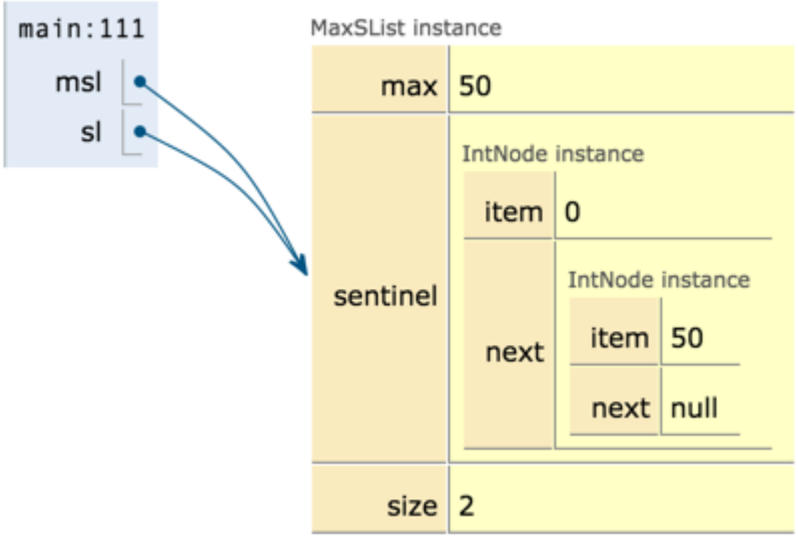
Output Prediction

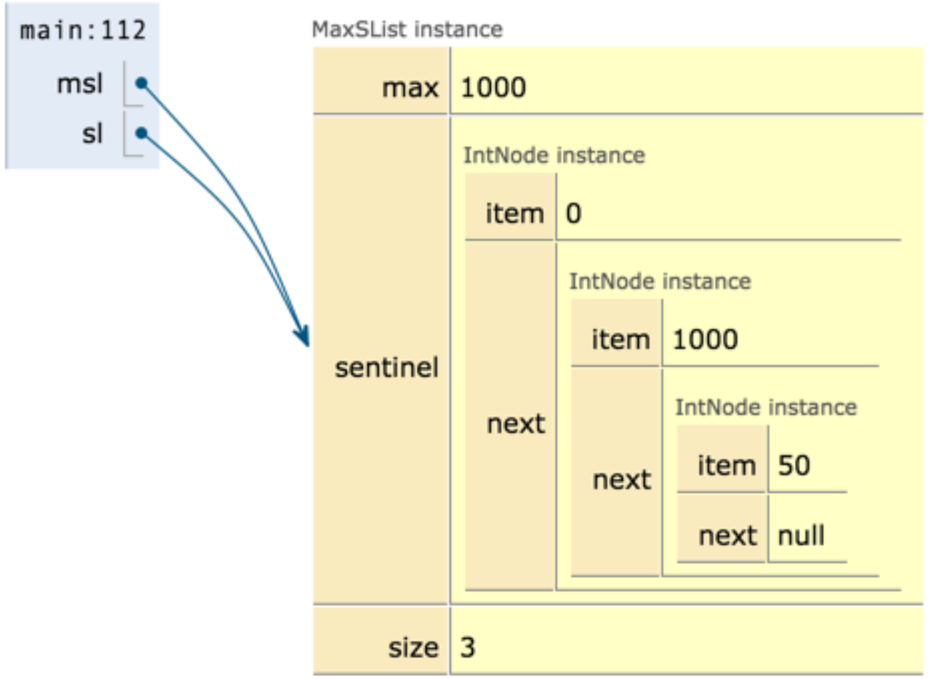
What will be the output?

D. 1000



```
public class MaxSListLauncher {
    public static void main(String[] args) {
        MaxSList msl = new MaxSList(0);
        msl.insertBack(50);
        SList sl = msl;
        msl.insertFront(1000);
        System.out.println(msl.max());
}
```



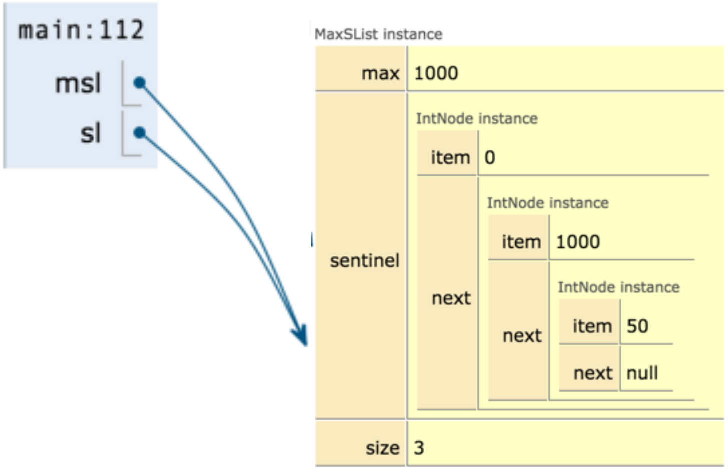


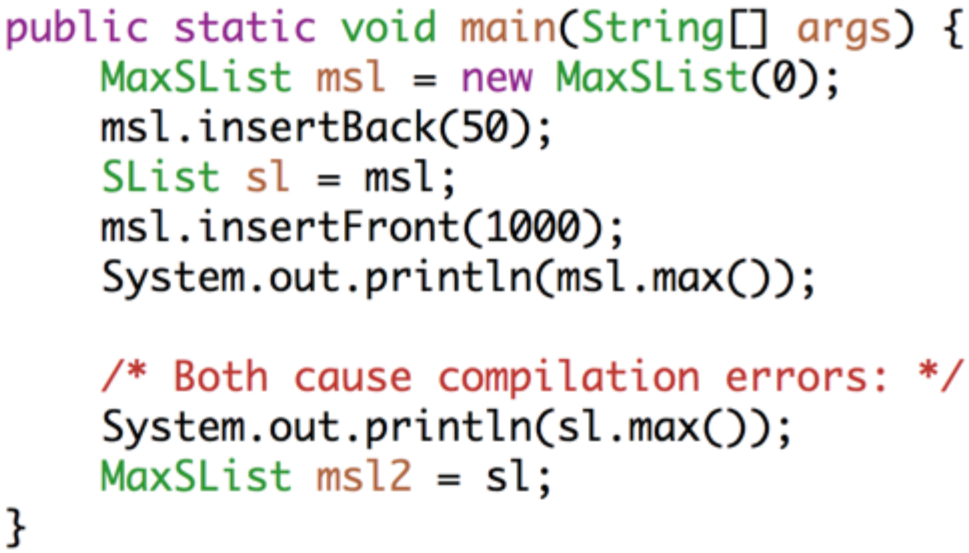
Subtleties

Static Type Checking

Whether or not you can call a method is based solely on static type of variable.

- sl’s dynamic type MaxSList.
- But cannot call max.





```
public static void main(String[] args) {
    MaxSList msl = new MaxSList(0);
    msl.insertBack(50);
    SList sl = msl;
    msl.insertFront(1000);
    System.out.println(msl.max());

    /* Both cause compilation errors: */
    System.out.println(sl.max());
    MaxSList msl2 = sl;
}
```

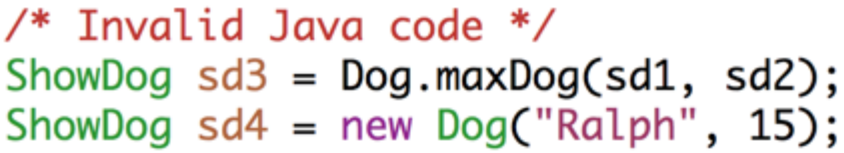
Assignments also allowed based on static type.

- Even though sl’s dynamic type is MaxList, cannot assign to msl2.
- Compiler plays it as safe as possible with type checking.

Static Types and Expressions

Expressions have static types:

- Method calls have static type equal to return type.
- Result of creation using new keyword has appropriate static type.



```
/* Invalid Java code */
ShowDog sd3 = Dog.maxDog(sd1, sd2);
ShowDog sd4 = new Dog("Ralph", 15);
```

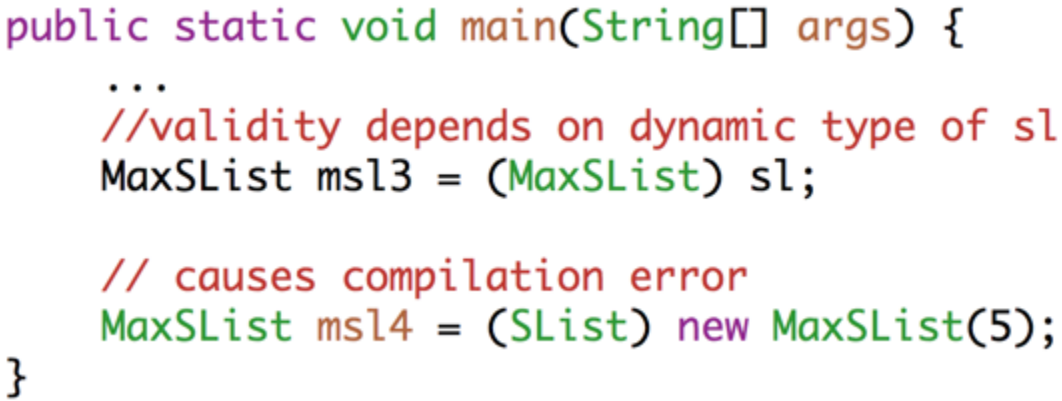
Examples above:

- maxDog has static type Dog (even though arguments are ShowDogs).
- Instantiation has static type Dog.
- Since not all Dogs are Showdogs, these assignments fail.

Casting

Casting is a powerful but dangerous tool.

- Tells Java to treat an expression as having a different static type.
- Effectively tells the compiler to ignore its type checking duties.



```
public static void main(String[] args) {
    ...
    //validity depends on dynamic type of sl
    MaxSList msl3 = (MaxSList) sl;

    // causes compilation error
    MaxSList msl4 = (SList) new MaxSList(5);
}
```

If sl’s dynamic type is not MaxSList, we get a ClassCastException at runtime.

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

Hiding

What happens if we ‘override’ a variable or static method?

- This is called ‘hiding’, not overriding.
- No dynamic selection process.
- Determined entirely by static type.

Example:

- Adding a flavor field for MaxSLists and SLists (see FlavorTest.java on github).

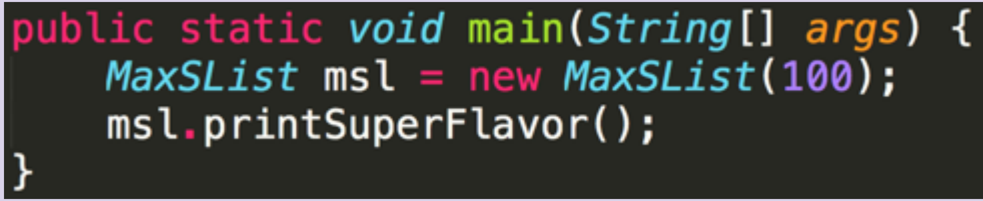
Hiding

Which flavor is output?

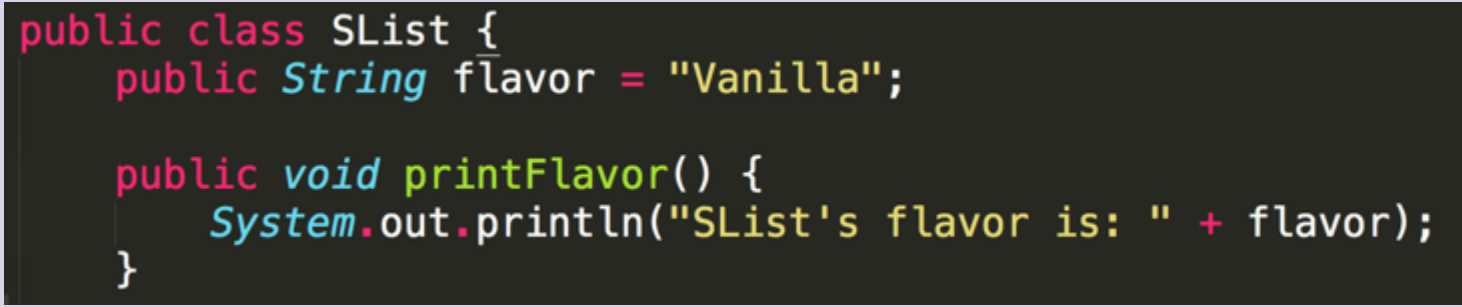
A. MaxSList flavored.

B. Vanilla.

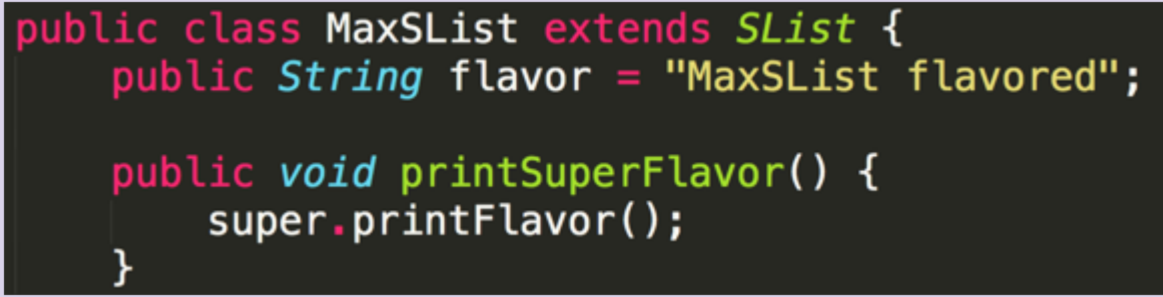
C. Compile error.



```java
public static void main(String[] args) {
    MaxSList msl = new MaxSList(100);
    msl.printSuperFlavor();
}
```



```java
public class SList {
    public String flavor = "Vanilla";

    public void printFlavor() {
        System.out.println("SList's flavor is: " + flavor);
    }
```



```java
public class MaxSList extends SList {
    public String flavor = "MaxSList flavored";

    public void printSuperFlavor() {
        super.printFlavor();
    }
```

Hiding

Hiding is super confusing. Almost never a good idea.

- Java has somewhat intuitive rules for how hiding is handled.
- Discussion this week will give you the full (horrible) picture.
  - Sorry.

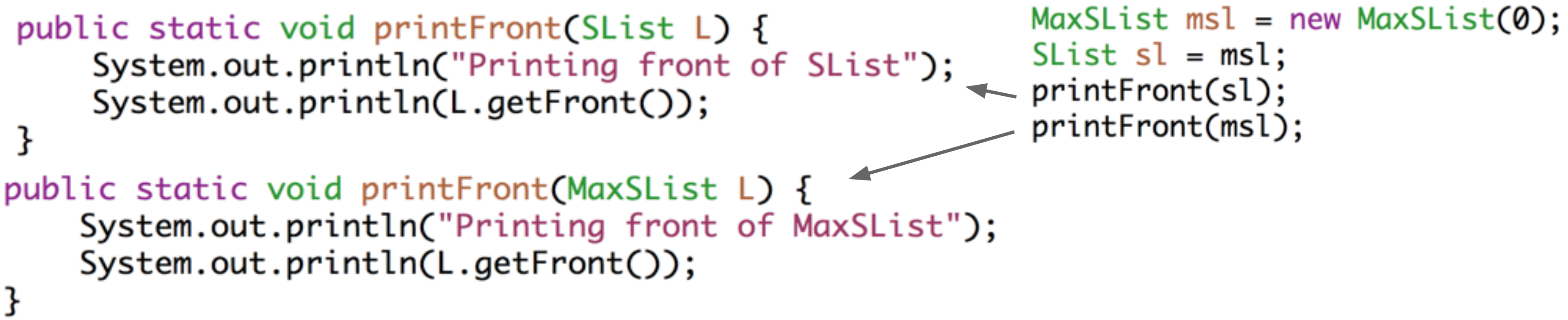
Overloading vs. Overriding

Dynamic method selection happens only for overridden methods, i.e.:

- Superclass and subclass share method with the exact same signature.

Does not apply for overloaded methods, i.e.:

- Two methods of the same class with different signatures.



```
public static void printFront(SList L) {
    System.out.println("Printing front of SList");
    System.out.println(L.getFront());
}
public static void printFront(MaxSList L) {
    System.out.println("Printing front of MaxSList");
    System.out.println(L.getFront());
}
```

```
MaxSList msl = new MaxSList(0);
SList sl = msl;
printFront(sl);
printFront(msl);
```

Definition: The ‘signature’ of a method is its name and parameter types.

The Rules

Behavioral Summary for Inheritance

MaxSList extends SList means a MaxSList is-an SList. Inherits all properties!

Invocation of overridden methods follows two simple rules:

- Compiler plays it safe and only lets us do things allowed by ***static*** type.
- The actual method invoked is based on **dynamic** type.

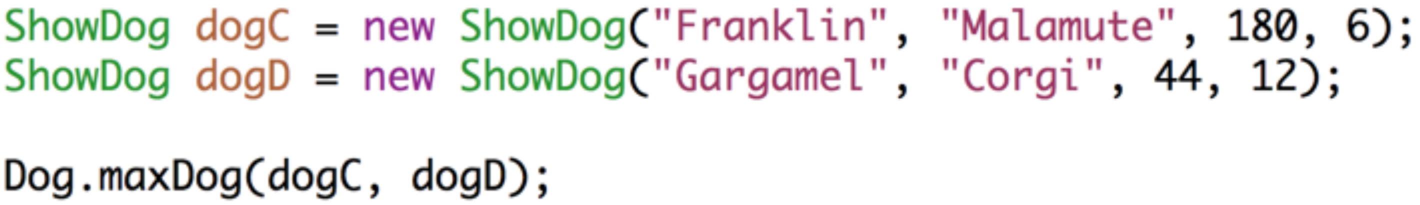
Does not apply to overloaded methods!

Understanding of these top two rules is important. Stuff below, not so much.

Invocation of hidden static methods or hidden variables follows a simple rule:

- Actual method invoked or variable accessed is based on ***static*** type.
- See discussion worksheet for how things get more confusing when we throw non-static methods into the mix with hiding.

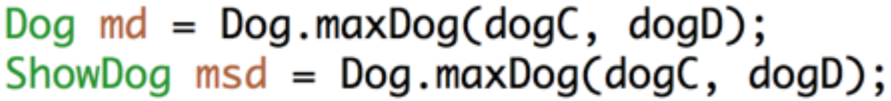
Type Checking Quiz!



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

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

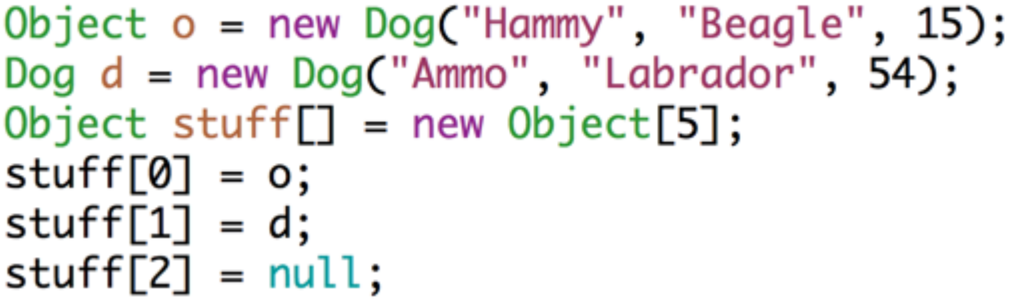
1. What is the static type of `Dog.maxDog(dogC, dog D)`?

2. Which (if any), will compile:



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

3. How many containers 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;
```

Extra Slides I:
Deeper Look at Type Checking
(In Case You’re Curious)

Quick Terminology Reminders

A variable name is a name for a container.

We store values in containers.

Values may be:

- Primitives (int, long…)
- References to Objects

Values themselves never change, only contents of containers.

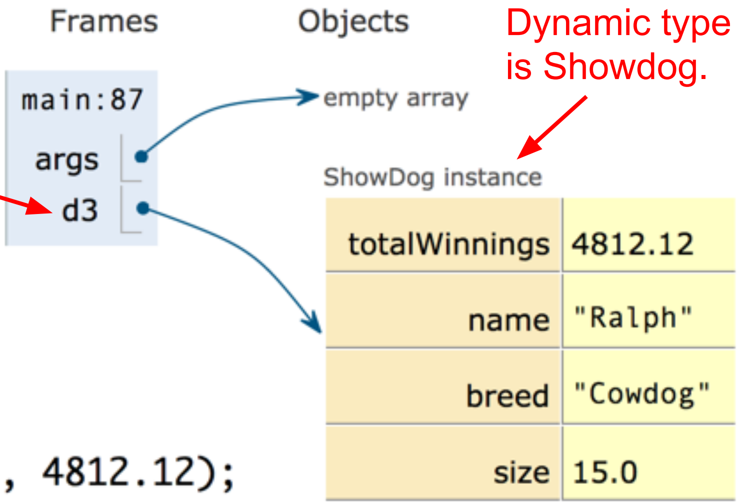
Java Typing: Dynamic vs. Static Types

Every value has a type, known as its ‘dynamic’ type.

Every container has a type, known as its ‘static’ type.

- Compiler’s goal: Dynamic type of value should always be subtype of static type of container.

Static type of d3 container is Dog.



totalWinnings	4812.12
name	"Ralph"
breed	"Cowdog"
size	15.0

Dynamic type is Showdog.

```
/* two lines of valid Java code */
Dog d3 = new ShowDog("Ralph", "Cowdog", 15, 4812.12);
d3.bark();
```

Java Typing: Dynamic vs. Static Types

I own the rights to use 3 spaces in a billion space storage yard.

- I do not own specific spots, but may use up to 3 at once.
- I rent these spaces out for parking cars.
- As an eccentric, I record which I’m using with 3 buckets.

Someone brings in a 2001 Ford Fiesta, and I park it in @2453f89f.

- I write “@2453f89f” on paper and toss it in bucket #2.

Static type of the bucket is Car.

Dynamic type of @2453f89f is 2001 Ford Fiesta.

Java Typing: Dynamic vs. Static Types

Static type of the containers is Car.

Dynamic type of @2453f89f is 2001 Ford Fiesta.

CAR1
CAR2
CAR3
@2453f89f

The Compiler

My coworker, Compiler checks types for me.

- If I try to write @2453f891, and @2453f891 contains an Elephant, she’ll stop me!

She doesn’t even have to check the spot!

- She knows before I even try to write the number down.

Going to have to abandon this analogy to understand why.

Type Checking is Done at Compile Time!

How do we achieve this?

Containers and type hierarchies

Given: Container with static type TC and a value of dynamic type TX.

- The container may hold the value if the value **is a TC**, or in other words:
- The container may hold the value if **TX is a subtype of TC**.

Can an Explosive container hold a value of dynamic type Dynamite?

- Yes, Dynamite is a subtype of Explosive.

All types are subtypes of themselves.

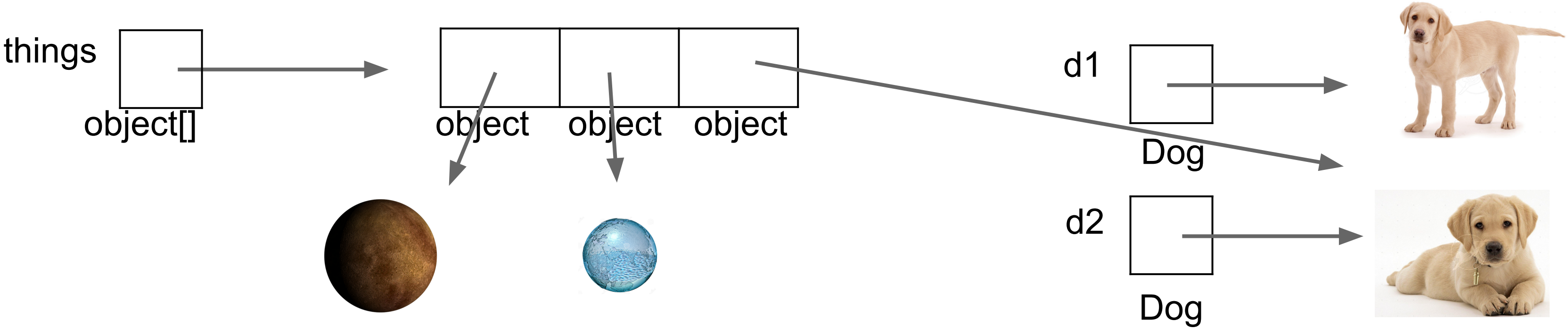
- Otherwise the “TX is a subtype of TC” rule would not work.

Type Checking

Every variable (container) has a static type.

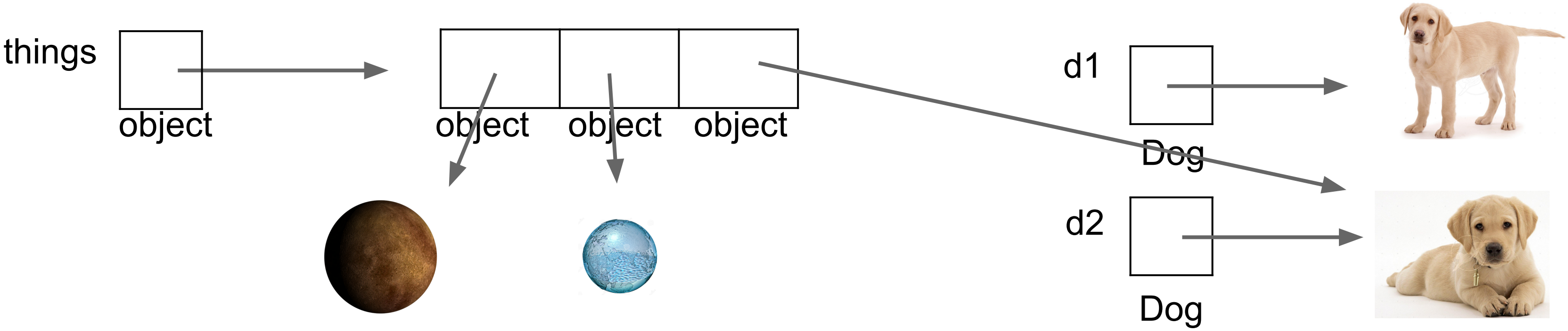
- Corollary: Can derive a static type for every expression.

```
Object[] things = new Object[3];
things[0] = new Planet(6e26, 6.67e9);
things[1] = new Planet(9e31, 4.32e9);

Dog d1 = new Dog("Frank", "Labrador", 30);
Dog d2 = new ShowDog("Frank Jr.", "Labrador", 35, 12312.42);
things[2] = Dog.maxDog(d1, d2);
```



Type Checking

```
Object[] things = new Object[3];
things[0] = new Planet(6e26, 6.67e9);
things[1] = new Planet(9e31, 4.32e9);

Dog d1 = new Dog("Frank", "Labrador", 30);
Dog d2 = new ShowDog("Frank Jr.", "Labrador", 35, 12312.42);
things[2] = Dog.maxDog(d1, d2);
```



Questions: What is static type of things[2]? Static type of Dog.maxDog(d1, d2)? Dynamic type of value of d2?