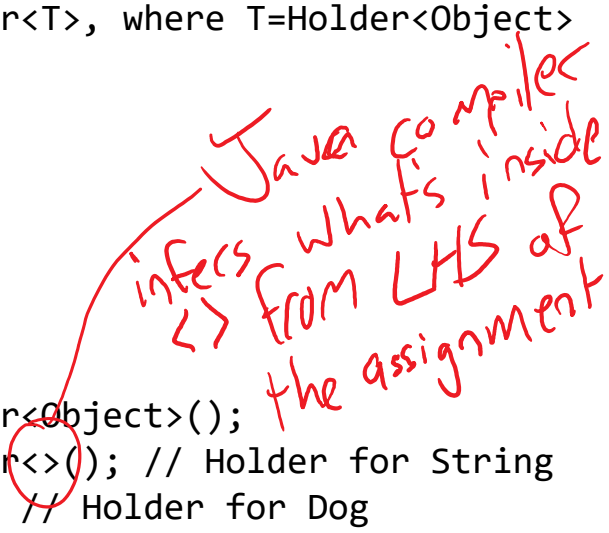
Writing Generic Code (Sec 19.3-19.9)

REVIEW

- Why Generics?
  - Allows us to dodge explicit typing and write generic, reusable code.
  - Imagine if you had to write an ArrayList for all 6,000+ classes in the standard libraries.
  - With generics, we can eliminate the need to store as Objects and cast and still only write one class!
- We can use generically typed classes as follows:
  - `ArrayList<String> list = new ArrayList<String>();`
  - We can also use "<>" or "the diamond" on the right side to infer the type:
    - `ArrayList<String> list = new ArrayList<>();`

CLASSES CAN BE PARAMETERIZED

- The following syntax defines a class with generic parameters:
  - `public class MyClass<A, Type1, Type2, ...> {...}`
- Here is another example:
  - `public class Holder<T> { // What if Holder<T>, where T=Holder<Object>`
  -
```
    private T data;
    public void setData(T data) {
        this.data = data;
     }
    public T getData() {
        return data;
    }
}
```
  - `Holder<Object> myObjectHolder = new Holder<Object>();`
  - `Holder<String> myStringHolder = new Holder<>(); // Holder for String`
  - `Holder<Dog> myDogHolder = new Holder<>(); // Holder for Dog`



- Can have multiple parameter types (i.e. class SomeClass<T1, T2>{...})
- ***To get more instructive diagnostics on generics code***
  - `javac -Xlint Example.java`

But what should we call the parameters?

- Although the generic parameters can be words, by convention they are single, upper case letters that follow this convention:
  - E - Element
  - K - Key
  - N - Number
  - T - Type
  - V - Value
  - S, U, V, etc. - 3rd, 4th, 5th, etc. types

BOUNDED TYPES

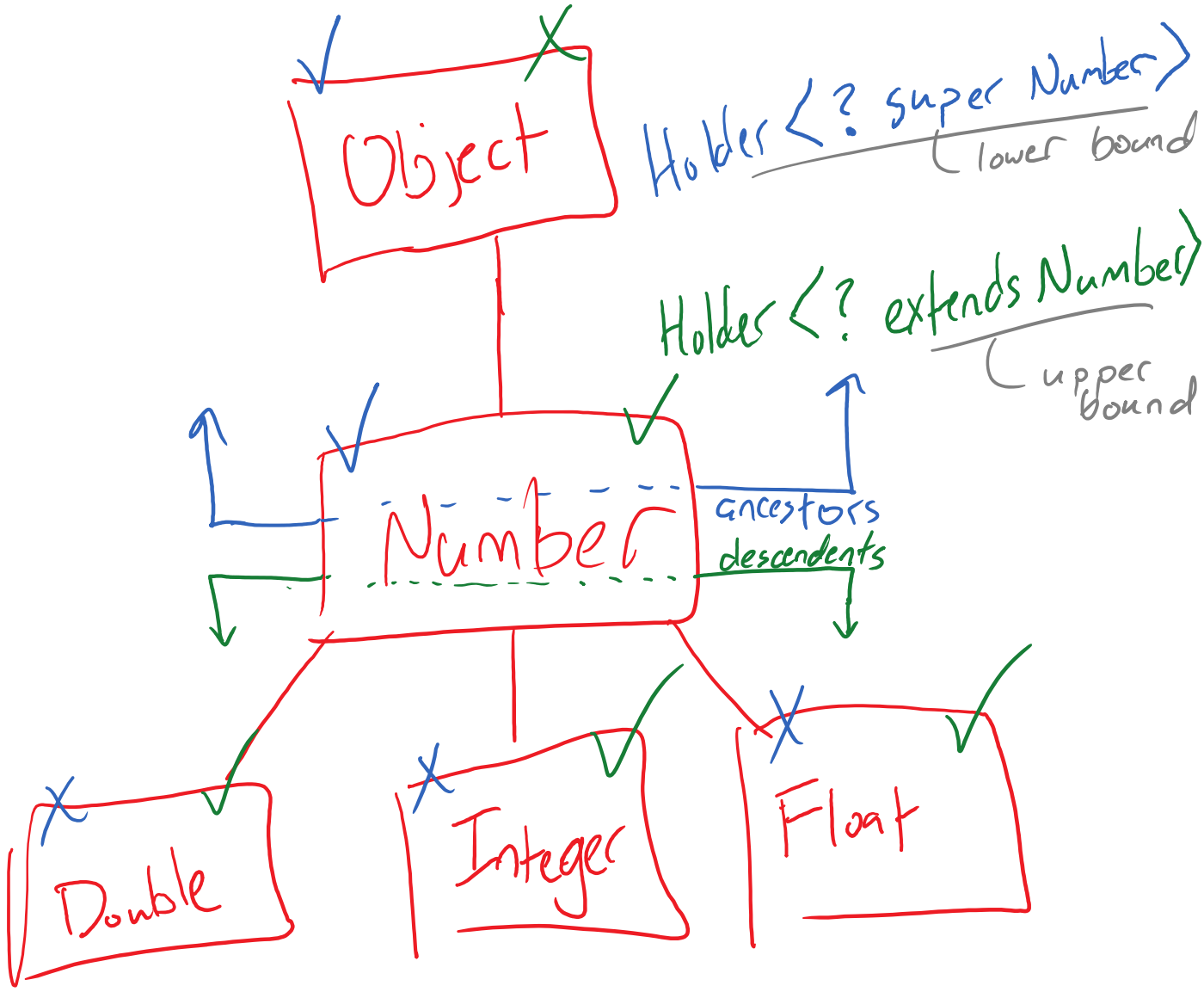
- Generics also let us partially constrain or bound the type.
- This allows us to specify certain functionality must exist without being too specific.
  - e.g. We can specify the type must extend **Number** without explicitly saying it must be **Double**, **Integer**, **Float**, etc.
- Bounded type params allow you to place a restriction on which Objects can validly inhabit your type parameter
  - `Pair<T extends AClass> // Extends works for both classes and interfaces`
  - `Pair<T extends A & B> // Multiple bounds - possibly interfaces`
  - `Pair<T extends Comparable<? super T>>  // Lower bounded "wildcard"`
- **Wildcard** (?): Any class that fulfils the requirement
  - ? ***super*** T: T or a **supertype** of T (establishes lower bound in the class hierarchy)
  - ? ***extends*** T: T or a **subtype** of T (establishes upper bound in the class hierarchy)
  - https://docs.oracle.com/javase/tutorial/java/generics/wildcards.html

GENERIC METHODS

- For ***static*** generic methods, the type parameter section must appear **before** the method's return type. This is because static methods do not require an instance of a class therefore the type parameter is not passed in when an object is constructed, so you must identify the parameterized type before listing it in as parameter list for the method. Here are examples:
  - `public static <T> void doSomething(T var) {}`
  - ```
    public static <T extends Comparable<? super T>>
        void sort(List<T> list) {}
    ```
- https://docs.oracle.com/javase/tutorial/java/generics/index.html

TYPE ERASURE

- The Java compiler **replaces** all type parameters with their bounded types, or **Object** if they are unbounded. This is called type erasure.
- This optimizes code considerably, but it means you cannot use generic types where a **class literal** is required.
- From the Java Tutorial:
  - Replace all type parameters in generic types with their bounds or Object if the type parameters are unbounded. The produced bytecode, therefore, contains only ordinary classes, interfaces, and methods.
  - *Insert type casts if necessary to preserve type safety.*
  - Generate bridge methods to preserve polymorphism in extended generic types.
    ```


Object
Holder<? super Number>
lower bound
Holder<? extends Number>
upper bound
Number
ancestors
descendents
Double
Integer
Float