

Generics

School of Computing
Clemson University
Clemson, SC 29634 USA

CPSC 2150: Wednesday, Feb. 15, 2023

Slide Sources:

- 1 Previous/Current CPSC 2150 Instructors
- 2 The lecture slides have also benefitted from instructors at Ohio State: Paolo Bucci, Wayne Heym, Paul Sivilotti, and Bruce Weide.

Last Class

- 1 The “Extends” Relation
- 2 Inheritance
- 3 Abstract Classes

General Announcements

1 Exam #1 - Feb. 22, 2023

- ▶ Please complete the Lockdown Browser Practice Quiz as part of your participation grade when it opens up. More information will be announced on Canvas.
- ▶ Will post review slides soon!

2 New Quiz: “Interfaces and Generics”

- ▶ **Available:** Today @ 12:00pm
- ▶ **Due:** Sunday, Feb. 19 @ 11:59pm

Homework Assignment

- 1 Programming assignment 2 is posted on your lab Canvas page.

- ▶ Generics in Java (and templates in C++) make interfaces and classes reusable by parameterizing them
- ▶ Users can employ same interfaces and classes in different situations by instantiating them appropriately
- ▶ **Key idea:** Development, testing, and verification of generic modules happen once in their lifetime; the cost is amortized through many uses

Generics

- ▶ What are we parameterizing?
 - ▶ A data type used somewhere in the `class` or `interface`
- ▶ We've already been using a generic `interface`:
 - ▶ `List` can store a list of any data type:
 - ▶ `List<Integer>`
 - ▶ `List<String>`
 - ▶ `List<BoardPosition>`

A Generic Interface

```
public interface IStack<T> {  
  
    public void push(T x);  
    public T pop();  
    public int depth();  
    public void clear();  
    public int maxDepth();  
  
}
```

A Generic Interface

```
public interface IStack<T> {  
  
    public void push(T x);  
    public T pop();  
    public int depth();  
    public void clear();  
    public int maxDepth();  
  
}
```

- ▶ `T` is a stand in for our data type.

A Generic Interface

```
public interface IStack<T> {  
  
    public void push(T x);  
    public T pop();  
    public int depth();  
    public void clear();  
    public int maxDepth();  
  
}
```

- ▶ **T** is a stand in for our data type.
- ▶ We can use **T** to refer to that data type passed in as a parameter during variable declaration.
 - ▶ **Ex:** Passed in **String** for **T**.
 - ▶ Now we can use can only store and retrieve **Strings**.

Generics

- ▶ Note that `IStack` is a **generic type** (also called a **parameterized type**)
- ▶ The actual type of the entries is selected only later by the client when the type `IStack` is used to declare or instantiate a variable, e.g.:

```
IStack<Integer> si = new Stack1<Integer>(100);
```

Generics

- ▶ Note that `IStack` is a **generic type** (also called a **parameterized type**)
- ▶ The actual type of the entries is selected only later by the client when the type `IStack` is used to declare or instantiate a variable, e.g.:

```
IStack<Integer> si = new Stack1<Integer>(100);
```

- ▶ The **formal type parameter** was called `T`; here, the **actual type** (or **argument type**) is `Integer`.

Generics

- ▶ Note that `IStack` is a **generic type** (also called a **parameterized type**)
- ▶ The actual type of the entries is selected only later by the client when the type `IStack` is used to declare or instantiate a variable, e.g.:

```
IStack<Integer> si = new Stack1<Integer>(100);
```

- ▶ The **formal type parameter** was called `T`; here, the **actual type** (or **argument type**) is `Integer`.
- ▶ As of Java 7, generic arguments in a constructor call are inferred from the declared type:

```
IStack<Integer> si = new Stack1<>(100);
```

- ▶ The **diamond operator** means the same thing as the constructor with explicit generic arguments.

Generic Type Parameters

- ▶ Can have more than one!
 - ▶ **Example:** `Map<K, V>`
- ▶ Convention is to use one letter
 - ▶ `E` - element
 - ▶ `K` - key
 - ▶ `V` - value
 - ▶ `T` - type
 - ▶ `N` - number
- ▶ No requirement that it has to be one letter
 - ▶ Distinguishes from a normal class name

A Generic Interface

```
public interface IStack<Type> {  
  
    public void push(Type x);  
    public Type pop();  
    public int depth();  
    public void clear();  
    public int maxDepth();  
  
}
```

- Is `Type` the generic argument, or a named class?

Reference Types

- ▶ Note the use of `Integer` here, not `int`
`IStack<Integer> si = new Stack1<Integer>(100);`
- ▶ Java demands that generic arguments must be **reference types**
- ▶ This is what allows generics to work
 - ▶ Java doesn't really have to know the data type, it just knows it's a memory location
 - ▶ Memory locations are stored in our stack
 - ▶ Memory locations are passed into our methods
 - ▶ Memory locations are returned from our methods

Wrapper Types

- ▶ Each of the primitive types has a corresponding **wrapper type** that is a reference type (i.e., an `Object`, in part to satisfy this requirement)
 - ▶ This is a big part of why the differences are important
 - ▶ Reference types also always have `equals` and `toString` methods defined, since they inherit them from `Object` class.
 - ▶ `List`'s `contains` uses the reference type's `equals` method to check for equality.
- ▶ Wrapper types are also **immutable**
 - ▶ This is helpful, but not required

Recall: Wrapper Types

Primitive Type	Wrapper Type
<code>boolean</code>	<code>Boolean</code>
<code>char</code>	<code>Character</code>
<code>int</code>	<code>Integer</code>
<code>double</code>	<code>Double</code>

- ▶ Java compiler can automatically convert between them.
 - ▶ **Autoboxing** (or just **boxing**) is converting a primitive types to its corresponding object wrapper classes.
 - ▶ **Unboxing** is converting from an object wrapper class to its primitive type.

A Generic Implementation

```
/**
 * @invariant ...
 *
 * @correspondence ...
 */
public class Stack2<T> implements IStack<T> {

    private int top;
    private T contents[];

    public Stack2(int d) {
        top = 0;
        contents = (T[]) new Object[d];
    }

    ...

}
```

A Generic Implementation

```
/**
 * @invariant ...
 *
 * @correspondence ...
 */
public class Stack2<T> implements IStack<T> {

    private int top;
    private T contents[];

    public Stack2(int d) {
        top = 0;
        contents = (T[]) new Object[d];
    }

    ...

}
```

- We have use `T` in both `Stack2` and `IStack` to clarify that both need to have the same type for `T`

A Generic Implementation

```
/**
 * @invariant ...
 *
 * @correspondence ...
 */
public class Stack2<T> implements IStack<T> {

    private int top;
    private T contents[];

    public Stack2(int d) {
        top = 0;
        contents = (T[]) new Object[d];
    }

    ...

}
```

- ▶ We have use **T** in both **Stack2** and **IStack** to clarify that both need to have the same type for **T**
- ▶ You cannot create an array of **Ts** using the **new** keyword.
- ▶ However, all classes extend **Object**...

A Generic Implementation

push

```
/**
 * @invariant ...
 *
 * @correspondence ...
 */
public class Stack2<T> implements IStack<T> {

    private int top;
    private T contents[];

    ...

    public void push(T x) {
        contents[top++] = x;
    }

    ...

}
```

Aliasing Concerns

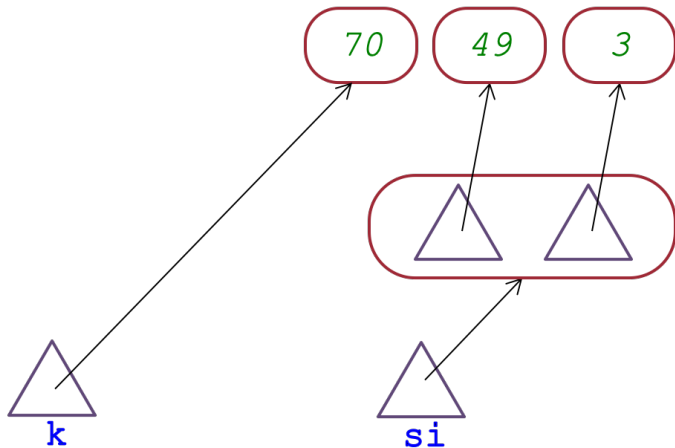
- ▶ Since the parameterized data type is always a reference type, we need to be careful about aliasing.
- ▶ Consider our stack and its `push` method

Updated push Specification

```
public interface IStack<T> {  
  
    /**  
     * @pre    length of self, |self| < MAX_DEPTH  
     * @post   self = x added to the front of #self  
     */  
    public void push(T x);  
  
    ...  
  
}
```

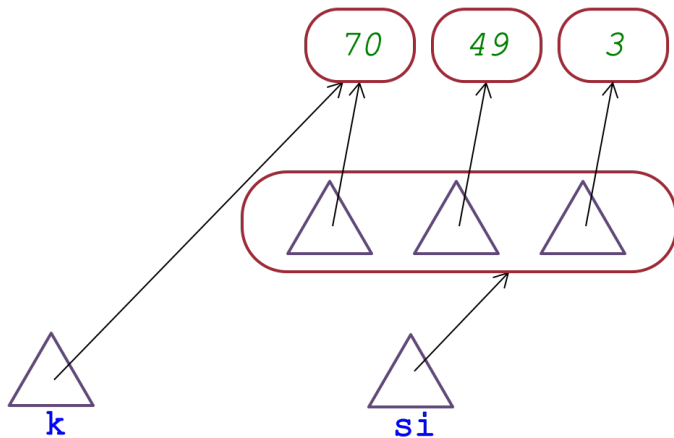
Meaning of “Aliases: ...”

Before `push`



Meaning of “Aliases: ...”

After `push`



Meaning of “Aliases: ...”

- ▶ We now have two pointers to the same memory location!
 - ▶ The tracing table notation with $\rightarrow$ gives us no easy way to describe this situation
 - ▶ The picture is, however, a handy way to see what's going on, so draw pictures!
- ▶ Since `Integer` is immutable, there is **no consequence** to this case of aliasing
 - ▶ But consider: `IStack<Pencil> sp = ...`

No Consequences?

- ▶ Recall, if a class is immutable, then the value of that object cannot change.
 - ▶ So an `Integer` object is a reference to the value, and that value cannot directly change
- ▶ We change `Integers` by changing the reference, not the value.
- ▶ This happens in boxing and unboxing.

Boxing and Unboxing of Integers

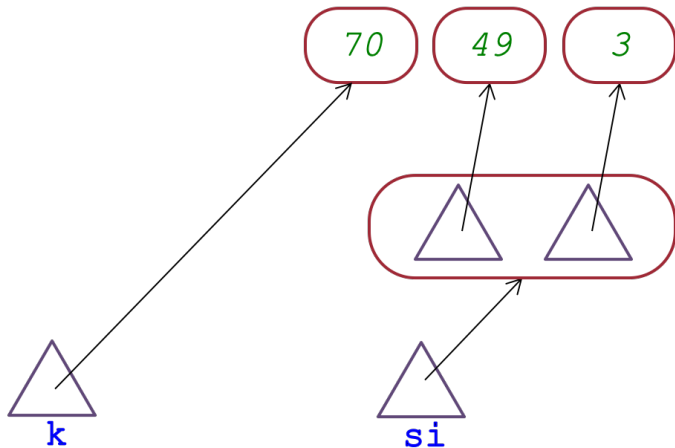
- ▶ `Integer x = new Integer(43);`
 - ▶ This creates a reference to the memory location that stores 43.
 - ▶ `x` stores a memory location (For example: **0x042**)
- ▶ `x = 37;`
 - ▶ Boxing and unboxing happens automatically
 - ▶ `x = new Integer(37);`
 - ▶ `x` now stores a new memory location. (For example: **0x078**)
 - ▶ ...and garbage collection cleans up the inaccessible original object if there are no more references to it.

Example Tracing Table

Code	State
	$si = \langle 49, 3 \rangle$ $k = 70 \text{ // } 0x065$
<code>si.push(k);</code>	
	$si = \langle 70, 49, 3 \rangle$ $k = 70 \text{ // } 0x065$
<code>k = 37;</code>	
	$si = \langle 70, 49, 3 \rangle$ $k = 37 \text{ // } 0x056$

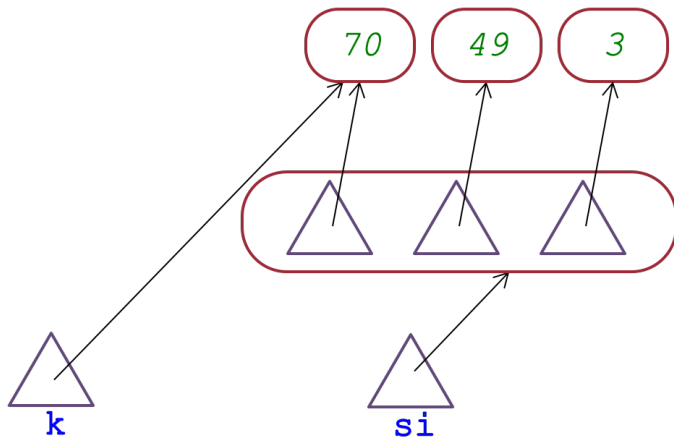
Meaning of “Aliases: ...”

Before `push`



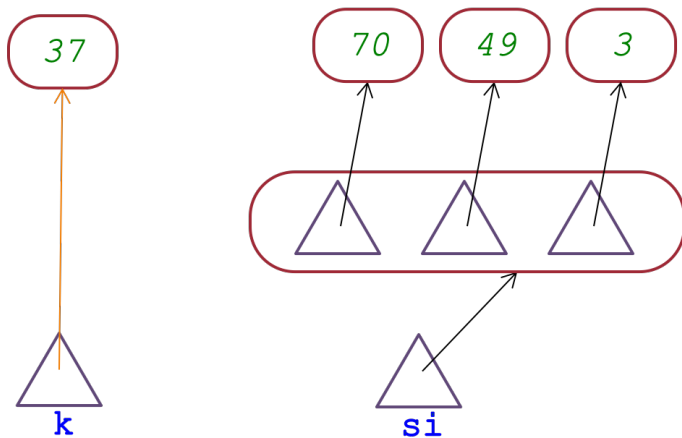
Meaning of “Aliases: ...”

After `push`



Meaning of “Aliases: ...”

After $k = 37$



Stack of Mutable Objects?

- Now we want a stack of `Pencils` (which we defined).

```
/**
 * This class is used to store the information of a pencil.
 *
 * @invariant hasEraser iff length >= 10 AND
 *             [color is valid] AND length >= 0
 */
public class Pencil {

    private boolean hasEraser;
    private String color;
    private int length;

    /**
     * ...
     */
    public Pencil(String c, int l) {
        color = c;
        length = l;
        hasEraser = (length >= 10); //enforce the invariant!
    }

    /**
     * ...
     */
    public int sharpen(int amount) {
        length = length - amount;
        hasEraser = (length >= 10);

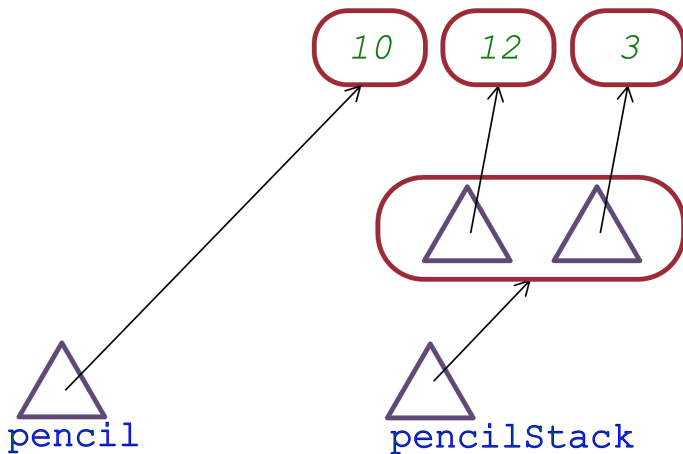
        return length;
    }
}
```

Stack of Mutable Objects?

- ▶ `Pencil` is not an immutable class
 - ▶ Both the value in and outside of the stack will be changed.
 - ▶ So we can change the value, which leads to an aliasing issue.
- ▶ **Note:** When you use the `final` keyword on a reference type, it means that the memory address can't change. However, you might be able to change the data stored at that memory address if the object has mutable data fields.

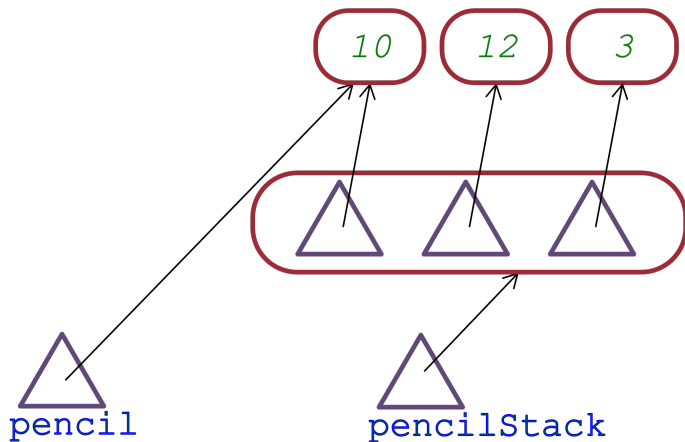
Meaning of “Aliases: ...”

Before `push`



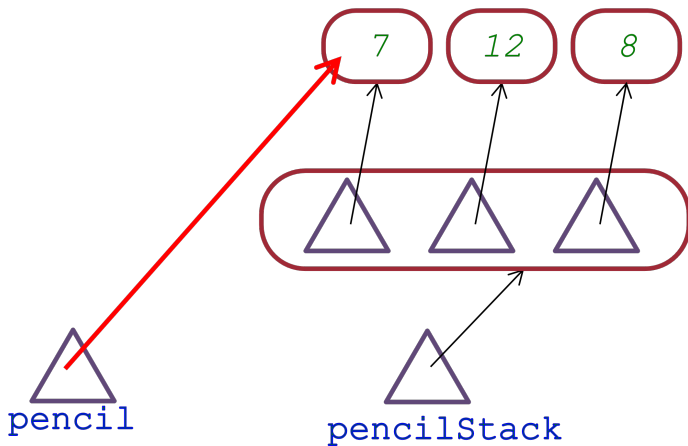
Meaning of “Aliases: ...”

After `push`



Meaning of “Aliases: ...”

After `pencil.sharpen(3)`



Generics

Summary

- ▶ Generics allow us to parameterize our classes and interfaces
- ▶ We can make a data type used in our classes and interfaces be decided when an object is declared.
- ▶ This allows us to reuse code, instead of making new classes for each different data type
- ▶ This relies on a few things:
 - ▶ All non-primitive types are reference types
 - ▶ Reference types extend `Object`.
 - ▶ Allows for casting
 - ▶ Reference types have an implementation for many methods, such as `equals` and `toString`.
 - ▶ They may not do what we expect, but they will at least compile

General Announcements

1 Exam #1 - Feb. 22, 2023

- ▶ Please complete the Lockdown Browser Practice Quiz as part of your participation grade when it opens up. More information will be announced on Canvas.
- ▶ Will post review slides soon!

2 New Quiz: “Interfaces and Generics”

- ▶ **Available:** Today @ 12:00pm
- ▶ **Due:** Sunday, Feb. 19 @ 11:59pm

Homework Assignment

- 1 Programming assignment 2 is posted on your lab Canvas page.

Summary

- 1 Generics
- 2 A generic `interface`
- 3 Reference Types vs. Wrapper Types
 - ▶ Boxing
 - ▶ Unboxing
- 4 A generic implementation
- 5 Meaning of “Aliases...”