CS131: Programming Languages

Lun Liu

05.19.17

Java Generics with Wildcards

- `C<? extends T>`
- Q: Why do we need this, can’t we just use C<T> with subtype polymorphism?

Java Generics with Wildcards

```
public void drawAll(List<? extends Shape>
shapes) {
   for (elem : shapes){
     elem.draw();
   }
}
List<Rectangle> lr = …;
drawAll(lr);
```

Java Generics with Wildcards

```
public void drawAll(List<? extends Shape> shapes) {
... }

public void drawAll(List<Shape> shapes) { ... }
List<Rectangle> lr = …;
drawAll(lr);
//won't type check, because List<Rectangle> is not a
subtype of List<Shape>
```

- You don't want to allow the following code

```
List<Rectangle> lr = …
lr.add(new Circle(…));
```

Today

- Java’s memory semantics
- Dynamic dispatch vs Static overloading

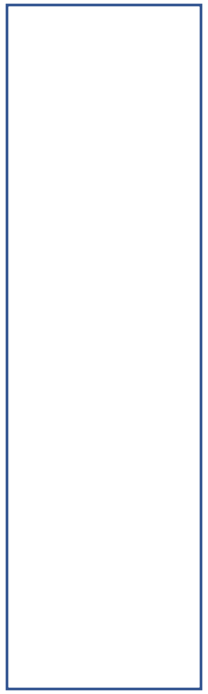
Java's Memory Semantics

- `T i = new T();  //T: Object, Integer, MyClass, etc.`
  - `i` is actually a reference (pointer) to the new `T` object on the heap
- `T r = i;`
  - `r` is also a pointer, and it points to the same `T` object that `i` points to
- `r.foo();  //if foo() modify the object`
- `i.check();  //i can also see the change`

Java’s Parameter Passing

- Still **pass-by-value**
- Just that value is the value of the ”pointer”

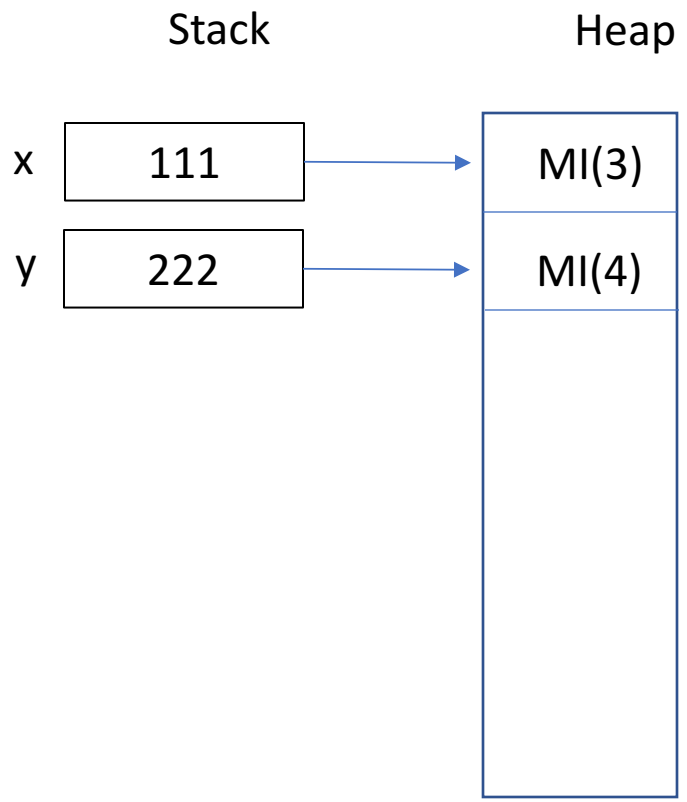
```
Class ParameterPassing{
  void foo(MyInteger a, MyInteger b) {
    a = new MyInteger(7);
    b.setValue(8);
  }
  public static void main(String[] args){
    MyInteger x = new MyInteger(3);
    MyInteger y = new MyInteger(4);
    new ParameterPassing().foo(x, y);
    System.out.println(x);
    System.out.println(y);
  }
}
```



Java's Parameter Passing

- Still **pass-by-value**
- Just that value is the value of the "pointer"

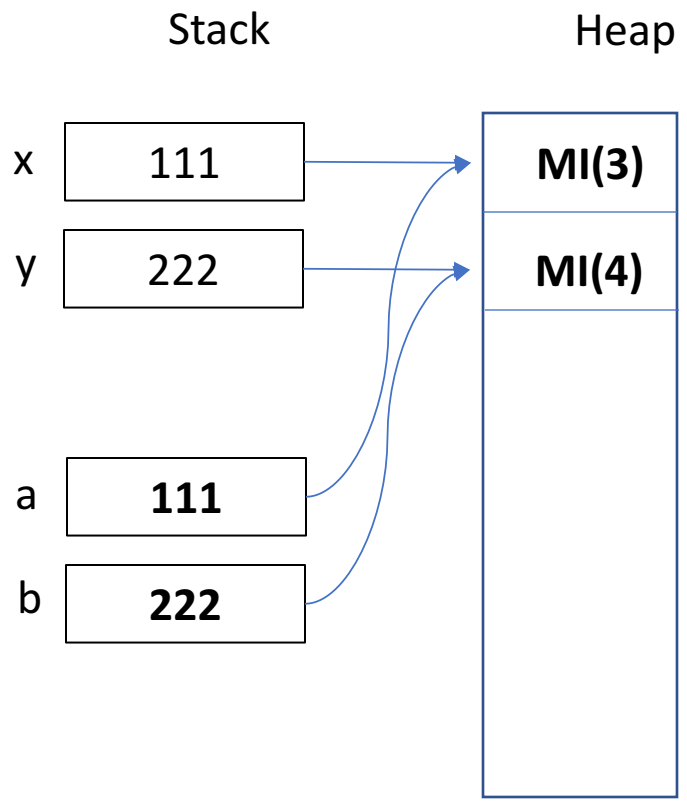
```
Class ParameterPassing{
  void foo(MyInteger a, MyInteger b) {
    a = new MyInteger(7);
    b.setValue(8);
  }
  public static void main(String[] args){
    MyInteger x = new MyInteger(3);
    MyInteger y = new MyInteger(4);
    new ParameterPassing().foo(x, y);
    System.out.println(x);
    System.out.println(y);
  }
}
```



Java's Parameter Passing

- Still **pass-by-value**
- Just that value is the value of the "pointer"

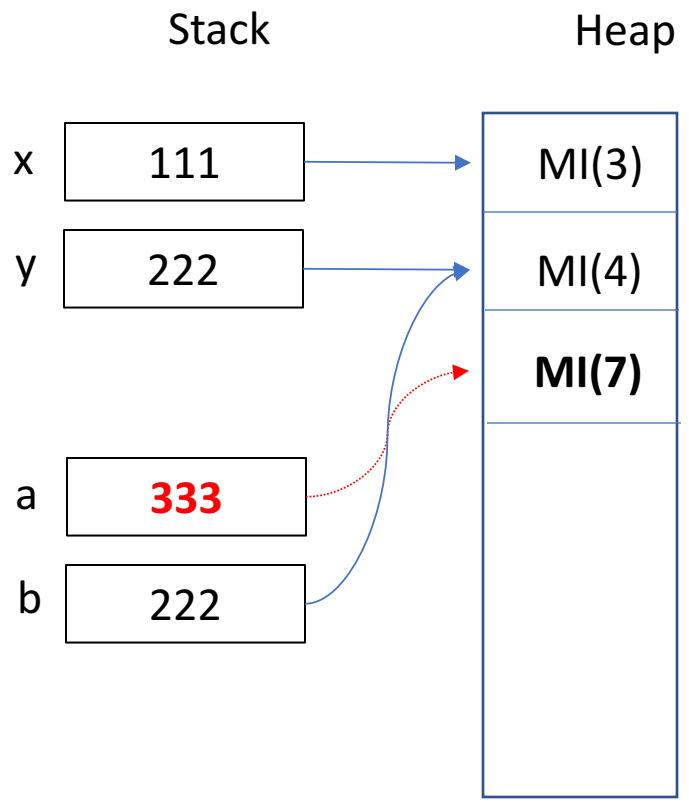
```
Class ParameterPassing{
  void foo(MyInteger a, MyInteger b) {
    a = new MyInteger(7);
    b.setValue(8)
  }
  public static void main(String[] args){
    MyInteger x = new MyInteger(3);
    MyInteger y = new MyInteger(4);
    new ParameterPassing().foo(x, y);
    System.out.println(x);
    System.out.println(y);
  }
}
```



Java's Parameter Passing

- Still **pass-by-value**
- Just that value is the value of the "pointer"

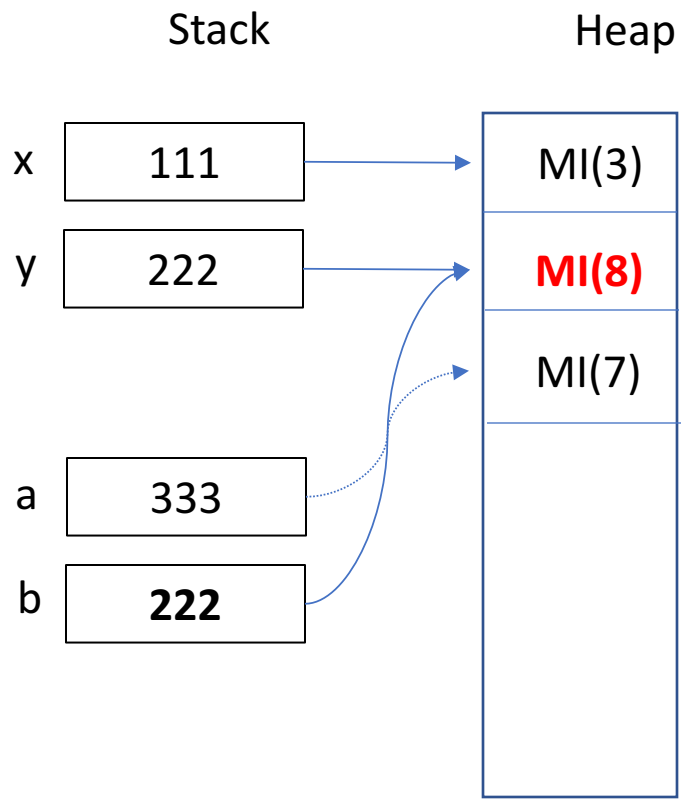
```
Class ParameterPassing{
  void foo(MyInteger a, MyInteger b) {
    a = new MyInteger(7);
    b.setValue(8);
  }
  public static void main(String[] args){
    MyInteger x = new MyInteger(3);
    MyInteger y = new MyInteger(4);
    new ParameterPassing().foo(x, y);
    System.out.println(x);
    System.out.println(y);
  }
}
```



Java's Parameter Passing

- Still **pass-by-value**
- Just that value is the value of the "pointer"

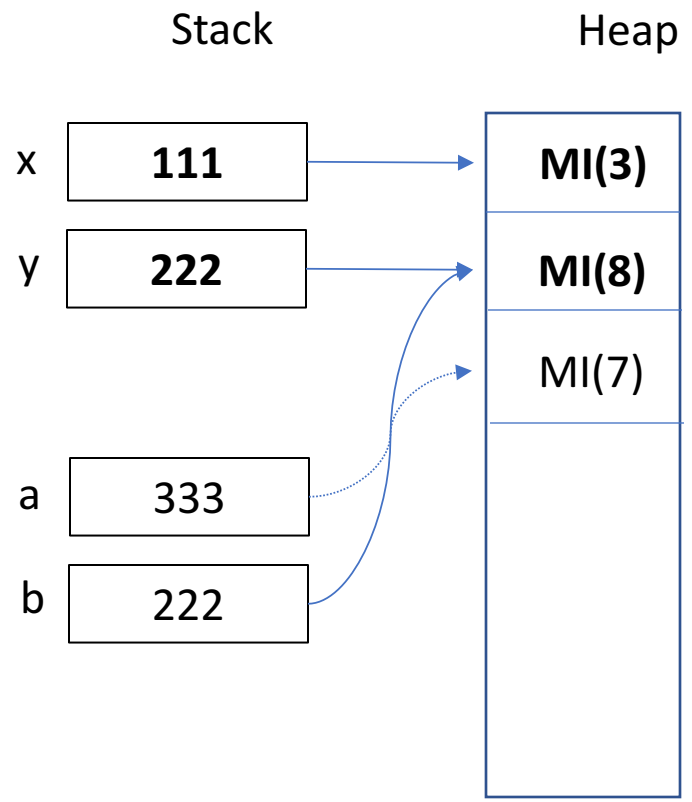
```
Class ParameterPassing{
  void foo(MyInteger a, MyInteger b) {
    a = new MyInteger(7);
    b.setValue(8);
  }
  public static void main(String[] args){
    MyInteger x = new MyInteger(3);
    MyInteger y = new MyInteger(4);
    new ParameterPassing().foo(x, y);
    System.out.println(x);
    System.out.println(y);
  }
}
```



Java's Parameter Passing

- Still **pass-by-value**
- Just that value is the value of the "pointer"

```
class ParameterPassing{
  void foo(MyInteger a, MyInteger b) {
    a = new MyInteger(7);
    b.setValue(8);
  }
  public static void main(String[] args){
    MyInteger x = new MyInteger(3);
    MyInteger y = new MyInteger(4);
    new ParameterPassing().foo(x, y);
    System.out.println(x);   //3
    System.out.println(y);   //8
  }
}
```



Primitives, Boxing & Unboxing

- Unlike Objects, primitives(`int, boolean`, etc.) are still kept on stack directly
- Java automatically box/ unbox

```
//before autoboxing
Integer n = Integer.valueOf(3);
int n_primitive = n.intValue();
//After Java5
Integer n = 3;              //auto boxing
int n_primitive = n;  //auto unboxing
```

Method Call in Java

- Two phases
  - Compile time
    - Determine the type signature based on the static types of the arguments
    - Static overloading
  - Runtime
    - Look up the method in the class (or in its superclass) of the receiver object at run time
    - Dynamic dispatch

Static Overloading

- Methods with **same name** but different **type signatures** (number/ static types of formal parameters)
- compiler will figure out the **type signature** for a specific call during **compile time**
- Depends on **static type information**

Static Overloading

- Overload.java

Dynamic Dispatch

- **Dynamic dispatch**
  - look up the method (with **certain type signature**) in the class of the receiver object at **run time**
  - if it's not there, look in its superclass recursively

Overriding

- Method in subclass can **override** method implementation of method of **same name** and **same type signature** in the super class
- We know which method to execute by **dynamic dispatch**
  - Dynamic dispatch look for methods **in its own class first**, then look for methods in its superclass

Overloading vs Dynamic Dispatch

- Points.java

Thanks