



Victoria University
of Wellington, New Zealand
*Te Whare Wananga o te
Upoko o te Ika a Maui
Aotearoa*



SWEN221: Software Development 8: Polymorphism II

David J. Pearce & Marco Servetto & Nicholas Cameron & James
Noble & Petra Malik

Computer Science, Victoria University

Inheritance + Constructors

- Constructors are **not** inherited
- Constructors use **super** in first line to forward construction to super class
- If the programmer does not explicitly write the super call, this call is **added by the compiler**.

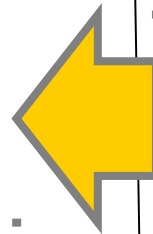
Sneaky code inserted by Java

```
class A { }  
class B extends A {  
    B(){  
        System.out.println("B constr");  
    }  
}
```



How you code looks like

```
class A extends Object { A(){super();} }  
class B extends A {  
    B(){  
        super();  
        System.out.println("B constr");  
    }  
}
```



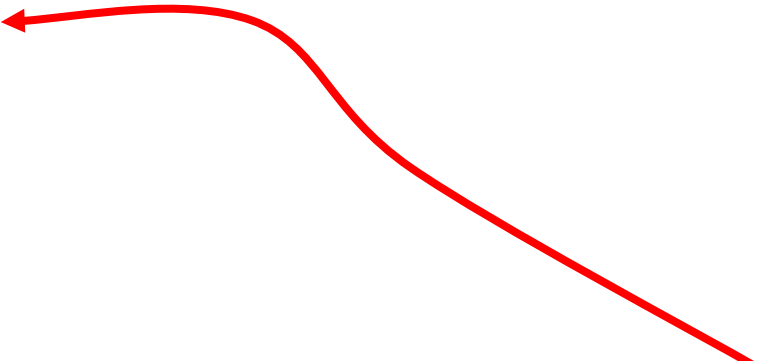
What is added
by Java

Java code-Quiz: it compiles?

```
class A {  
    int f;  
    A(int f){  
        this.f=f;  
    }  
}
```

```
class B extends A {  
    B(){  
        System.out.println("B constr");  
    }  
}
```

Constructor of A
takes 1 parameter

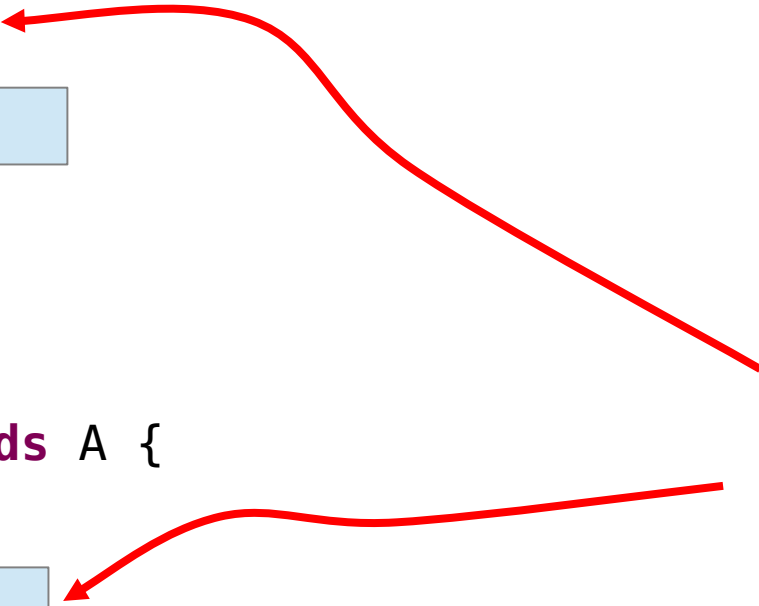


Java code-Quiz: it compiles? NO

```
class A extends Object {  
    int f;  
    A(int f){  
        super();  
        this.f=f;  
    }  
}
```

```
class B extends A {  
    B(){  
        super();  
        System.out.println("B constr");  
    }  
}
```

Constructor of A
takes 1 parameter

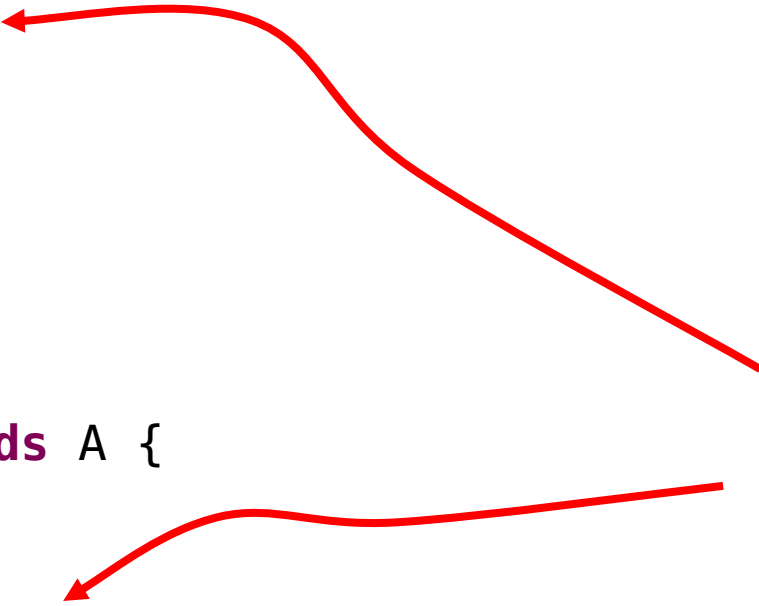


Java code-Quiz: Possible fix

```
class A {  
    int f;  
    A(int f){  
        this.f=f;  
    }  
}
```

```
class B extends A {  
    B(){  
        super(0);  
        System.out.println("B constr");  
    }  
}
```

Constructor of A
takes 1 parameter



Why inheritance?

- Allows to create a hierarchy of classes/interfaces, and to model our problem domain.
- Dynamic dispatch (overriding) ensures subclass can change behaviour as needed
- For example, method toString()
 - allows any possible object,
 - of any possible class,
 - included the one that still does not exist, to **decide** how to convert into a String

Why dynamic dispatch? (overriding)

```
class Log{  
    static void print(Object o){  
        System.out.println("Log: "+o.toString());  
    }  
}
```

```
Point p=new Point(1,2);  
Log.print(p)
```

- Point: class created in 2015
- Log: class created in 2005
- Thanks to dynamic dispatch, the code of `print` can call `toString`, so code wrote in 2005 can call code wrote in 2015!
- Not trivial with conventional imperative programming.

Why dynamic dispatch?

```
class HashSet implements Collection { ... }  
class ArrayList implements Collection { ... }  
class Vector implements Collection { ... }  
class TreeSet implements Collection { ... }
```

```
int sumElems(Collection<Integer> elems) {  
    int sum=0;  
    for(Integer i : elems) { sum += i; }  
    // use elems.iterator()  
    return sum;  
    // for(int i=0;i<elems.size();i++){sum+=elems.get(i);}  
    //use elems.get() and elems.size()  
}
```

Dynamic dispatch + Subtyping

Methods can operate on many different objects

Behaviour depends upon types of objects

Powerful mechanism for code reuse

Dispatch special cases

Access control: not all the methods are visible at any points, so while looking for method descriptors, look only for visible methods.

Static/private methods: while most methods can be overridden, static and private methods can not.

Remember, no dynamic dispatch for static and private methods. Dispatch decided at compile time.

Recompilation: adding-modifying overloaded versions can provide “interesting behaviour”, just **recompile all** often (eclipse: project->clean)

Overloading/Overriding

- Overloading: just syntactic convenience
 - understanding how it behave in all the cases is very hard
- Overriding: the most important feature in OO!
 - understanding how it behave is easy, but design programs taking full advantage of it, requires a lifetime programming experience.
- Interaction between the two is hard to understand
- Just distinguishing between the two can be hard

Overloading is not Overriding

- **Overloading:** multiple methods with the same name, but different parameters type.
- **Overriding:** redefinition of the same method in the same method in a subtype.

```
class James{  
    String sayHiTo(String name){...}  
    String sayHiTo(String name,String surname){...}  
}
```

Is there overloading/overriding here?

A:Only Overloading, B:Only Overriding, C: both, D: none

Overloading is not Overriding

- **Overloading:** multiple methods with the same name, but different parameters type.
- **Overriding:** redefinition of the same method in the same method in a subtype.

```
class James{  
    String sayHiTo(String name){...}  
  
}
```

```
class JamesBond extends James{  
    String sayHiTo(String name){...}  
  
}
```

Is there overloading/overriding here?

A: Only Overloading, B: Only Overriding, C: both, D: none

Overloading is not Overriding

- **Overloading:** multiple methods with the same name, but different parameters type.
- **Overriding:** redefinition of the same method in the same method in a subtype.

```
class James{  
    String sayHiTo(String name){...}  
  
}
```

```
class JamesBond extends James{  
  
    String sayHiTo(String name,String surname){...}  
}
```

Is there overloading/overriding here?

A:Only Overloading, B:Only Overriding, C: both, D: none

Overloading is not Overriding

- **Overloading:** multiple methods with the same name, but different parameters type.
- **Overriding:** redefinition of the same method in the same method in a subtype.

```
class James{  
    String sayHiTo(String name){...}  
  
}
```

```
class JamesBond extends James{  
    String sayHiTo(String name){...}  
    String sayHiTo(String name,String surname){...}  
}
```

Is there overloading/overriding here?

A:Only Overloading, B:Only Overriding, C: both, D: none

Overloading:: Quiz

- Really simple: 1 parameter methods

```
class Foo{
```

```
    String bar(int i){return "i";} (A)
```

```
    String bar(String s){return "s";} (B)
```

```
    String bar(Point p){return "p";} (C)
```

```
    String bar(ColoredPoint cp){return "cp";} (D)
```

```
void print(){
```

```
    System.out.println(bar(new Point()));
```

```
    System.out.println(bar("foo"));
```

```
    System.out.println(bar(1));
```

```
    System.out.println(bar(new ColoredPoint()));
```

```
    System.out.println(bar((Point)new ColoredPoint()));
```

```
    System.out.println(bar(null));}}
```


Overloading:: Good uses of overloading

- Different number of parameters
 - Quite simple, only one is "syntactically" applicable

Overloading:: Good uses of overloading

- Really isomorphic behaviour.
 - **PrintStream.println** //as in `System.out.println(1);`
 - byte, short, int, long, float, double, boolean, char,
 - Just print it!
 - object
 - Just print the result of `.toString()`
 - object IF null
 - Just print “null”
 - **StringBuffer.append** is similar!

Questions

```
abstract class Parent {  
    int x=100;  
    Parent (int x) {this.x=x;}  
    abstract int m(int x);  
    static Parent k (Parent x) { return x;}  
    int f (int x) { return m(x+1);}}
```

```
class Heir extends Parent {  
    static int x=200;  
    Heir (int x) { super (x+1);}  
    Heir () {this(3);}  
    int m(int x) { return x+2;}  
    int n (int x) { return this.x + x; }}
```

```
public class Es1 {  
    public static void main(String[] argv){  
        Parent p = new Heir();  
        Heir h = new Heir();  
        System.out.println(p.f(h.x));           //a  
        System.out.println((p.k(new Parent()))).x); //b  
        System.out.println((p.k(h)).x);         //c  
        System.out.println((p.n(5)).x);         //d  
        new Heir().x=5;                          //e1  
        System.out.println((h.n(p.x)));         //e2  
    }}
```

First point//step 0

```
abstract class Parent {  
    int x=100;  
    Parent (int x) {this.x=x;}  
    abstract int m(int x);  
    static Parent k (Parent x) { return x;}  
    int f (int x) { return m(x+1);}}  
  
public class Es1 {  
    public static void main(String[] argv){  
        Parent p = new Heir();  
        Heir h = new Heir();  
        System.out.println(p.f(h.x));  
    }}
```

//a

```
class Heir extends Parent {  
    static int x=200;  
    Heir (int x) { super (x+1);}  
    Heir () {this(3);}  
    int m(int x) { return x+2;}  
    int n (int x) { return this.x + x; }}
```

First point//step 1

```
abstract class Parent {  
    int x=100;  
    Parent (int x) {this.x=x;}  
    abstract int m(int x);  
    static Parent k (Parent x) { return x;}  
    int f (int x) { return m(x+1);}}  
  
public class Es1 {  
    public static void main(String[] argv){  
        Parent p = //some Heir instance x=?  
        Heir h = // some Heir instance x=?  
        System.out.println(p.f(h.x));  
    }}
```

//a

```
class Heir extends Parent {  
    static int x=200;  
    Heir (int x) { super (x+1);}  
    Heir () {this(3);}  
    int m(int x) { return x+2;}  
    int n (int x) { return this.x + x; }}
```

First point//step 2

```
abstract class Parent {  
    int x=100;  
    Parent (int x) {this.x=x;}  
    abstract int m(int x);  
    static Parent k (Parent x) { return x;}  
    int f (int x) { return m(x+1);}}  
  
public class Es1 {  
    public static void main(String[] argv){  
        Parent p = //some Heir instance x=?  
        Heir h = // some Heir instance x=?  
        System.out.println(p.f(200));  
    }}
```

//a

```
class Heir extends Parent {  
    static int x=200;  
    Heir (int x) { super (x+1);}  
    Heir () {this(3);}  
    int m(int x) { return x+2;}  
    int n (int x) { return this.x + x; }}
```

First point//step 3

```
abstract class Parent {  
    int x=100;  
    Parent (int x) {this.x=x;}  
    abstract int m(int x);  
    static Parent k (Parent x) { return x;}  
    int f (int x) { return m(x+1);}}  
  
public class Es1 {  
    public static void main(String[] argv){  
        Parent p = //some Heir instance x=?  
        Heir h = // some Heir instance x=?  
        System.out.println(p.m(200+1));  
    }}
```

```
class Heir extends Parent {  
    static int x=200;  
    Heir (int x) { super (x+1);}  
    Heir () {this(3);}  
    int m(int x) { return x+2;}  
    int n (int x) { return this.x + x; }}
```

First point//step 4

```
abstract class Parent {  
    int x=100;  
    Parent (int x) {this.x=x;}  
    abstract int m(int x);  
    static Parent k (Parent x) { return x;}  
    int f (int x) { return m(x+1);}}  
  
public class Es1 {  
    public static void main(String[] argv){  
        Parent p = //some Heir instance x=?  
        Heir h = // some Heir instance x=?  
        System.out.println((200+1)+2);  
    }}
```

```
class Heir extends Parent {  
    static int x=200;  
    Heir (int x) { super (x+1);}  
    Heir () {this(3);}  
    int m(int x) { return x+2;}  
    int n (int x) { return this.x + x; }}
```