Polymorphism

Method Overloading

Method Overloading

Methods with different parameter lists but the same name.

```
public int max(int num1, int num2)
public double max(double num1, double num2)
```

Overloaded methods must have different parameter types:
you cannot overload methods based on modifiers or return types

Selection (*Binding*) of Overloaded Method Definitions

(single parameter): look at the actual argument type and locate the method with the *most specific* (‘narrowest’) accepting formal parameter

– Example: OverloadedNumbers.java

Method Polymorphism (Overriding)

Method redefines (‘overrides’) a method of the same name in the parent class (e.g. toString() is often overridden)

Note similarity of *overriding* to *variable masking*: we are again re-defining a symbol within a given scope.

Polymorphic Methods

public String toString()

Defined in Object, normally overridden to give text description of object state

- default output is “ClassName@HexAddress”

```
Loan loan = new Loan();
System.out.println(loan) //invokes loan.toString()
====>(output) Loan@15037e5
```

Implementing Overriding in Java

Achieved by redefining an inherited method in a child class. Method signature must be the same.

e.g. in Circle, redefine toString() method inherited from Object:

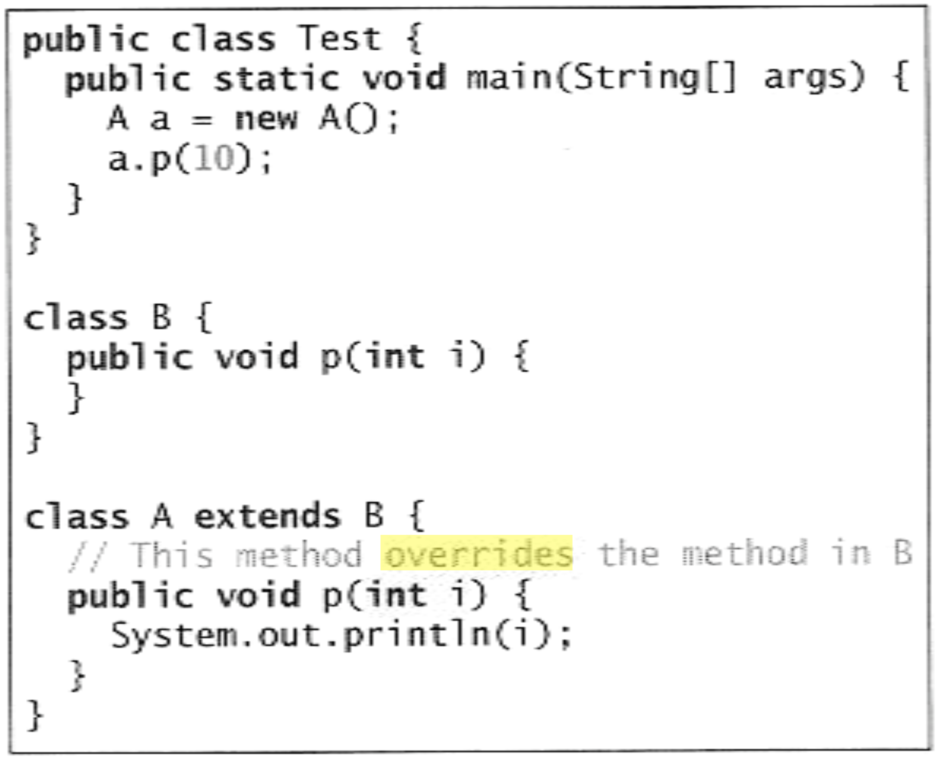
```
public String toString() {
 return “A Circle with color: “ + color +
     “and is filled: “ + filled;}
```

Polymorphism and Method Arguments

Method Arguments in Java

- May be of any subtype of the formal parameter type.
- public static void m(Object x) will accept *any* object x belonging to a subclass of Object (i.e. from any class!)
- public static void p(double x) accepts an x of *any* numeric type (byte, short, int, long, float, double)
  - a *widening type conversion* (cast) will be performed for non-doubles

Example: Overriding (left) vs. Overloading (right)

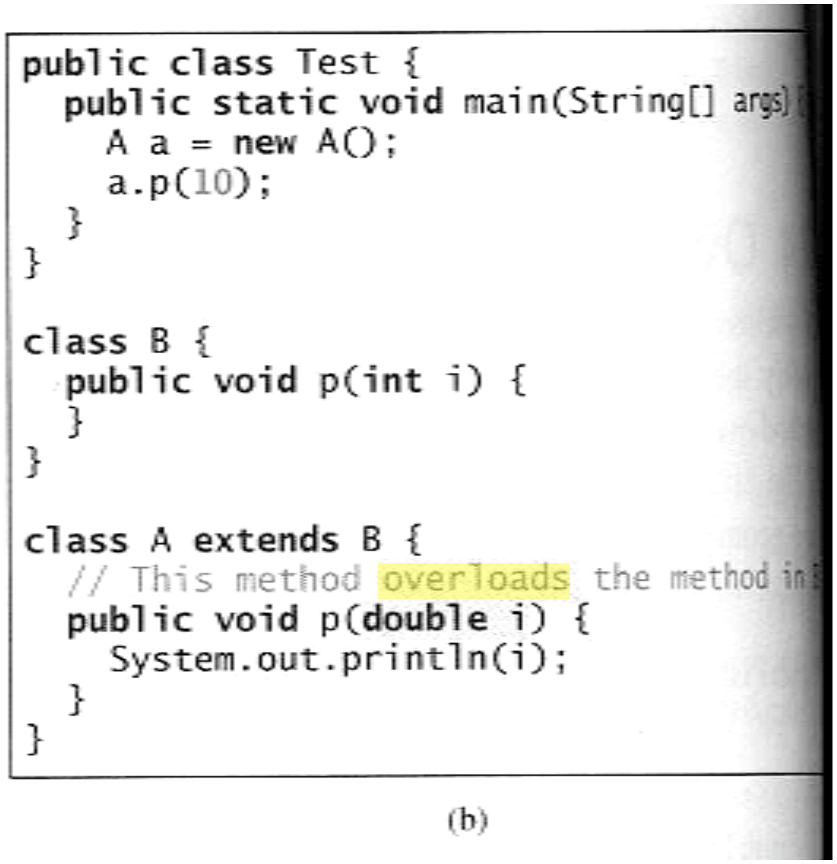


```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
  }
}

class B {
  public void p(int i) {
  }
}

class A extends B {
  // This method overrides the method in B
  public void p(int i) {
    System.out.println(i);
  }
}
```

(a)



```
public class Test {
  public static void main(String[] args)
    A a = new A();
    a.p(10);
  }
}

class B {
  public void p(int i) {
  }
}

class A extends B {
  // This method overloads the method in
  public void p(double i) {
    System.out.println(i);
  }
}
```

(b)

What is the output of each program?

Dynamic Binding (for Method Polymorphism)

Definition

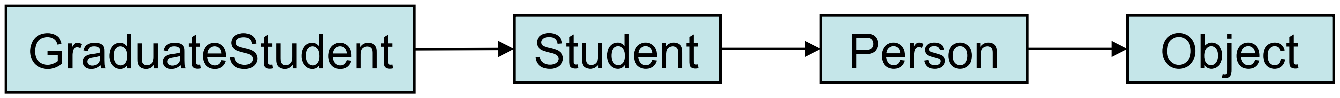
- Selecting the definition of a method to invoke at runtime (i.e. which definition to *bind* to the method call)
- Must match method name; number, order and types for arguments
- Relevant for overridden methods (e.g. toString())

Dynamic Binding In Java

The search for which definition to bind to a method call starts from the actual (constructed) class of an object, or a named class, and proceeds up the inheritance hierarchy towards Object.

Example

PolymorphismDemo.java



Subtle Point: Matching the Method vs. Selecting the Method Definition

Matching the Method Signature (static)

- For objects, the selection of which method signature to use is determined at compile time based on the reference variable type
- Put another way, the type of a reference to an object determines which class contract is active for an object
- If the active class contract does not define or inherit a desired method, it will not be found.
  - e.g. Object o = new Circle(1); o.getRadius() // won't work.
  - Object o = new Circle(1); ((Circle)o).getRadius() // will work.

Selecting the Method Definition (dynamic)

Is done dynamically at runtime (dynamic binding). The constructed *object* type determines the implementation used.

Hiding Data and (Static) Methods

- **Static Methods *and***
- **Static/Instance Data Members**

  cannot be overridden; only hidden. (Avoid this!)

Accessing Hidden Methods and Data

- Using super() in the subclass
- Using a reference variable of the superclass type (i.e. use the superclass type (interface))
- Unlike instance methods, static methods and data members are bound at compile time (“statically”)
- *Example: HidingDemo.java*
- Static methods and fields can always be accessed directly using the class name (*if* it is visible, using *Class.staticMethod()*)