Principles of Software Construction: Objects, Design and Concurrency

Inheritance, type-checking, and method dispatch

15-214

toad

Fall 2013

Jonathan Aldrich **Charlie Garrod**

School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

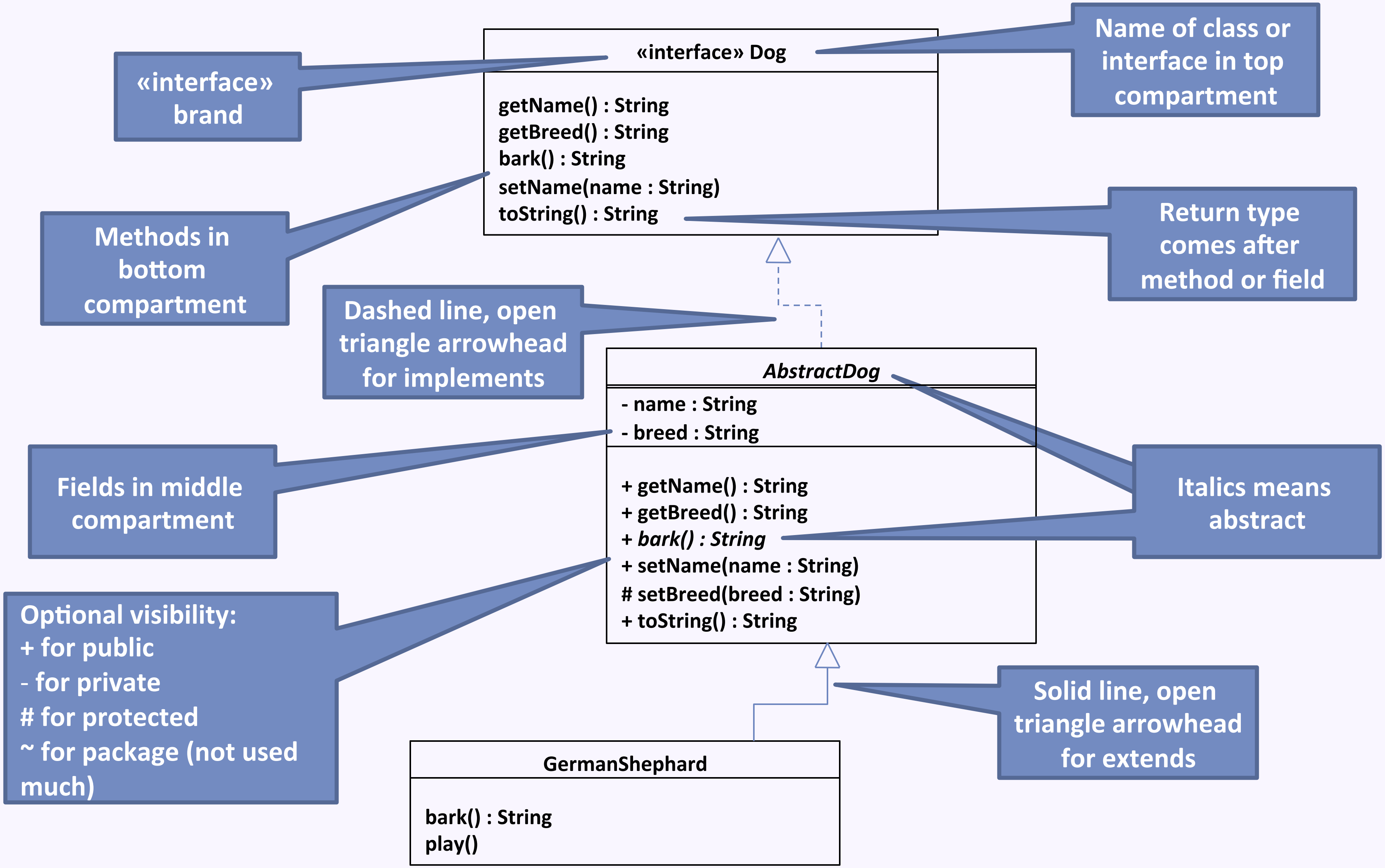
- Homework 1 due Tuesday

Key concepts from Tuesday

Key concepts from Tuesday

- Java packages
- The key encapsulation principle
- Inheritance
  - For code reuse
  - Abstract classes
  - Some design principles
    - Hierarchical modeling
- Static (compile-time) type vs. dynamic (run-time) type

Aside: UML class diagram notation



«interface» brand

«interface» Dog

Name of class or interface in top compartment

getName() : String
getBreed() : String
bark() : String
setName(name : String)
toString() : String

Methods in bottom compartment

Return type comes after method or field

Dashed line, open triangle arrowhead for implements

AbstractDog

- name : String
- breed : String

+ getName() : String
+ getBreed() : String
+ *bark() : String*
+ setName(name : String)
# setBreed(breed : String)
+ toString() : String

Fields in middle compartment

Italics means abstract

Optional visibility:
+ for public
- for private
# for protected
~ for package (not used much)

Solid line, open triangle arrowhead for extends

GermanShephard

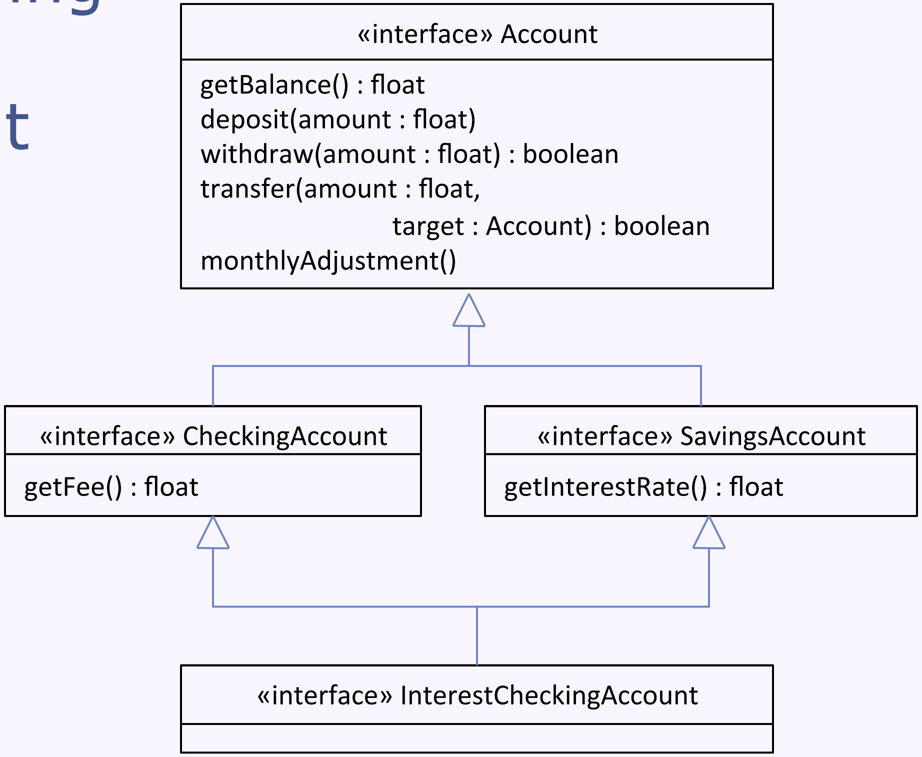
bark() : String
play()

Key concepts for today

- Inheritance and polymorphism, continued
  - Polymorphism and its alternatives
  - Java-specific inheritance details
  - Types and type-checking
  - Method dispatch, revisited
- The `java.lang.Object`

Recall: Is inheritance necessary?

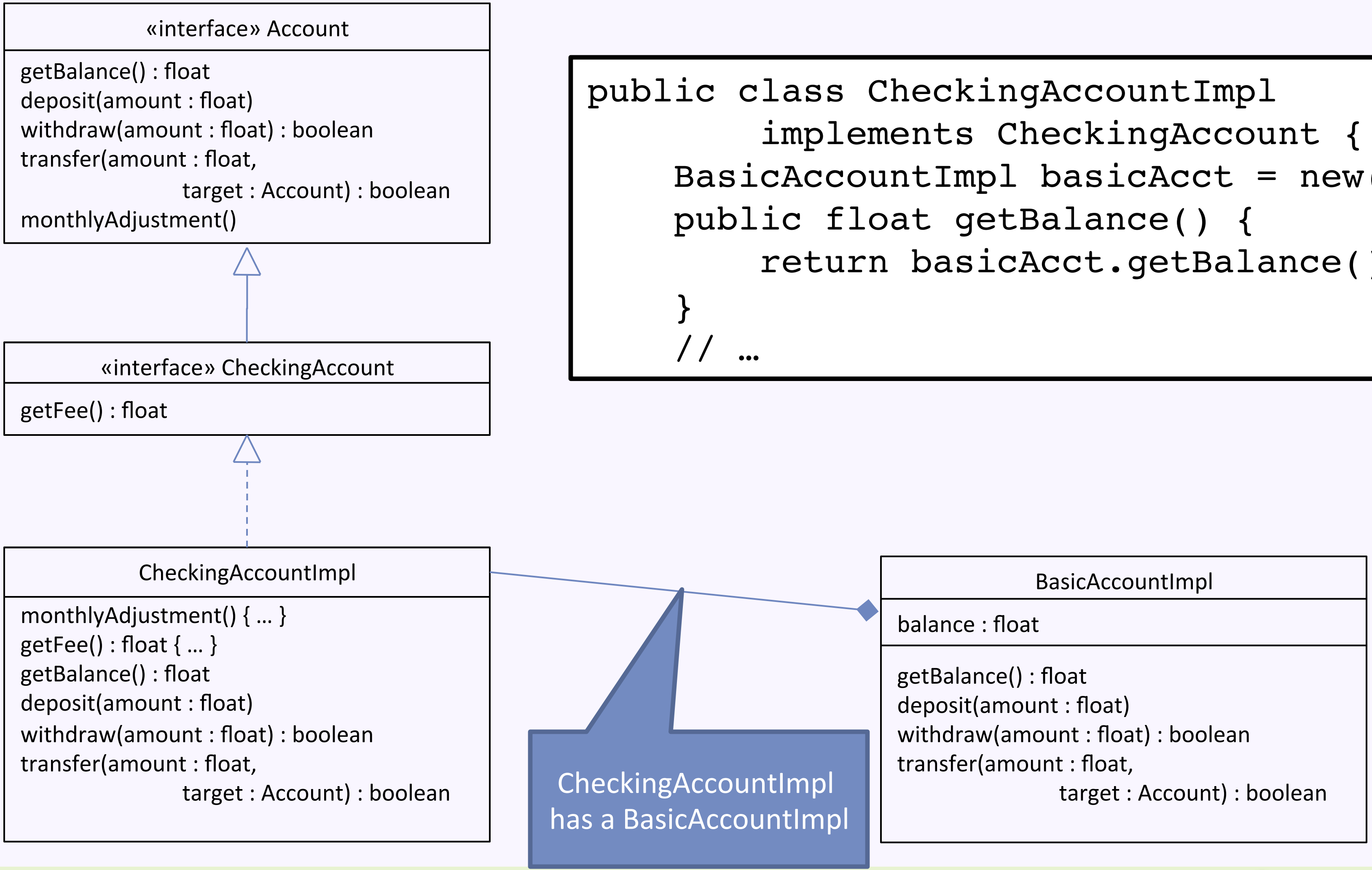
- Can we get the same amount of code reuse using only interfaces and class implementations, without using inheritance?



«interface» Account
getBalance() : float
deposit(amount : float)
withdraw(amount : float) : boolean
transfer(amount : float,
target : Account) : boolean
monthlyAdjustment()
«interface» CheckingAccount
getFee() : float
«interface» SavingsAccount
getInterestRate() : float
«interface» InterestCheckingAccount

Reuse via *composition* and *forwarding*

```
public class CheckingAccountImpl
        implements CheckingAccount {
    BasicAccountImpl basicAcct = new(…);
    public float getBalance() {
        return basicAcct.getBalance();
    }
    // …
```



«interface» Account

getBalance() : float
deposit(amount : float)
withdraw(amount : float) : boolean
transfer(amount : float,
target : Account) : boolean
monthlyAdjustment()

«interface» CheckingAccount

getFee() : float

CheckingAccountImpl

monthlyAdjustment() { … }
getFee() : float { … }
getBalance() : float
deposit(amount : float)
withdraw(amount : float) : boolean
transfer(amount : float,
target : Account) : boolean

BasicAccountImpl

balance : float

getBalance() : float
deposit(amount : float)
withdraw(amount : float) : boolean
transfer(amount : float,
target : Account) : boolean

CheckingAccountImpl has a BasicAccountImpl

Inheritance vs. composition

- Composition can be cleaner than inheritance
    - Reused code in a separate object
- Inheritance has less boilerplate code
    - No forwarding functions
    - Easier to avoid recursive dependencies
- Inheritance violates principles of encapsulation
    - Subclass dependent on superclass implementation
- Advice: Use inheritance sparingly
    - Before you define a class `Foo` to extend `Bar`, ask: "Is every Foo really a Bar?"

Extended re-use with `super`

```
public abstract class AbstractAccount implements Account {
  protected float balance = 0.0;
  public boolean withdraw(float amount) {
      // withdraws money from account (code not shown)
  }
}

public class ExpensiveCheckingAccountImpl
      extends AbstractAcount implements CheckingAccount {
  public boolean withdraw(float amount) {
      balance -= HUGE_ATM_FEE;
      boolean success = super.withdraw(amount)
      if (!success)
         balance += HUGE_ATM_FEE;
      return success;
  }
}
```

Overrides `withdraw` but also uses the superclass `withdraw` method

Constructor calls with `this` and `super`

```
public class CheckingAccountImpl
      extends AbstractAcount implements CheckingAccount {

  private float fee;

  public CheckingAccountImpl(float initialBalance, float fee) {
      super(initialBalance);
      this.fee = fee;
  }

  public CheckingAccountImpl(float initialBalance) {
      this(initialBalance, 5.00);
  }
  /* other methods… */ }
```

Invokes a constructor of the superclass. Must be the first statement of the constructor.

Invokes another constructor in this same class

Inheritance details: `final`

- A final class: cannot extend the class
  - e.g., `public final class CheckingAccountImpl { …`
- A final method: cannot override the method
- A final field: cannot assign to the field
  - (except to initialize it)

- Why might you want to use final in each of the above cases?

Type-casting in Java

- Sometimes you want a different type than you have
  - e.g.,

```
        float pi = 3.14;
        int indianaPi = (int) pi;
```

- Useful if you know you have a more specific subtype:
  - e.g.,

```
Account acct = …;
CheckingAccount checkingAcct =
                     (CheckingAccount) acct;
float fee = checkingAcct.getFee();
```

  - Will get a `ClassCastException` if types are incompatible

Inheritance details: `instanceof`

- Operator that tests whether an object is of a given class

```
Account acct = …;
float adj = 0.0;
if (acct instanceof CheckingAccount) {
    checkingAcct = (CheckingAccount) acct;
    adj = checkingAcct.getFee();
} else if (acct instanceof SavingsAccount) {
    savingsAcct = (SavingsAccount) acct;
    adj = savingsAcct.getInterest();
}
```

- Advice: avoid `instanceof` if possible

Avoiding `instanceof` with the Template Method pattern

```
public interface Account {
    …
    public float getMonthlyAdjustment();
}

public class CheckingAccount implements Account {
    …
    public float getMonthlyAdjustment() {
        return getFee();
    }
}

public class SavingsAccount implements Account {
    …
    public float getMonthlyAdjustment() {
        return getInterest();
    }
}
```

Avoiding `instanceof` with the Template Method pattern

```
~~Account acct = …;~~
~~float adj = 0.0;~~
~~if (acct instanceof CheckingAccount) {~~
~~    checkingAcct = (CheckingAccount) acct;~~
~~    adj = checkingAcct.getFee();~~
~~} else if (acct instanceof SavingsAccount) {~~
~~    savingsAcct = (SavingsAccount) acct;~~
~~    adj = savingsAcct.getInterest();~~
~~}~~
```

```
Account acct = …;
float adj = acct.getMonthlyAdjustment();
```

Typechecking

- The key idea: Analyze a program to determine whether each operation is applicable to the types it is invoked on

- Benefits:
  - Finds errors early
    - e.g., `int h = “hi” / 2;`
  - Helps document program code
    - e.g.,

```
baz(frob) { /* what am I supposed to do
               with a frob? */ }
void baz(Car frob) { /* oh, look,
                        I can drive it! */ }
```

Value flow and subtyping

- Value flow: assignments, passing parameters
  - e.g., `Foo f = expression;`
  - Determine the type T_{source} of the source expression
  - Determine the type T_{dest} of the destination variable `f`
  - Check that T_{source} is a subtype of T_{dest}
- Aside: The subtype relation *A* <: *B*
  - Base cases:
    - *A* <: B if *A* extends *B* or *A* implements *B*
    - *A* <: *A* (reflexivity)
  - Inductive case:
    - If *A* <: *B* and *B* <: *C* then *A* <: *C* (transitivity)

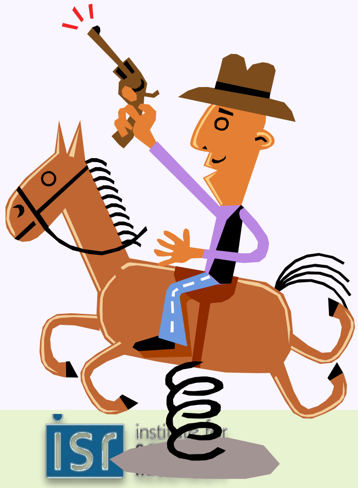
Typechecking expressions in Java

- Base cases:
  - variables and fields
    - the type is explicitly declared
  - Expressions using `new ...()`
    - the type is the class being created
  - Type-casting
    - the type is the type forced by the cast

- For method calls, e.g., e1.m(e2)
  1. Determine the type *T1* of the receiver expression e1
  2. Determine the type *T2* of the argument expression e2
  3. Find the method declaration m in type *T1* (or supertypes), using dispatch rules
  4. The type is the return type of the method declaration identified in step 3

Subtyping rules

- If a concrete class B extends type A
  - B inherits all concrete methods declared in A
    - B can override non-final inherited methods
  - B must override abstract or undefined interface methods
- If B overrides a method declared in type A
  - The argument types must be the same as in A
  - The result type must be subtype of result type from A
- Behavioral subtyping
  - If B overrides a method declared in A, it should conform to the *specification* from A
  - If `Cowboy.draw()` overrides `Circle.draw()` somebody gets hurt!



Method dispatch, revisited

e.g.: `x.foo(apple, 42)`

- Step 1 (compile time): determine which class to look in
  - Here, the static type of `x`
- Step 2 (compile time): determine the method signature to be executed
  - Find all accessible, applicable methods
  - Select the most specific method
    - `m1` is more specific than `m2` if each argument of `m1` is a subtype of the corresponding argument of `m2`

Method dispatch, revisited

e.g.: `x.foo(apple, 42)`

- Step 3 (run time): Determine the dynamic class of the receiver
  - The dynamic class of each object is stored in the heap
- Step 4 (run time): Locate the method to invoke
  - Starting at the run-time class, look for a method with the ***same signature*** found in step 2
    - If it is found in the run-time class, invoke it.
    - Otherwise, continue the search in the superclass of the run-time class and etc.
- I claim: this procedure will always find a method to invoke

Method dispatch practice

```
public class GenericAnimal {
   public String getNoise() { return “Noise”; }
}
```

```
public class Bird extends GenericAnimal {
   public String getNoise() { return “Chirp”; }
}
```

```
public class Cat extends GenericAnimal {
   public String getNoise() { return “Meow”; }
}
```

```
public class GenericDog extends GenericAnimal {
   // nothing special to hear here
}
```

```
public class Ewokian extends GenericDog {
   public String getNoise() { return “Oonga!”; }
}
```

Method dispatch practice, part A

```
public class GenericAnimal {
   public String getNoise() { return “Noise”; }
}
```

```
public class Bird extends GenericAnimal {
   public String getNoise() { return “Chirp”; }
}
```

```
public class Cat extends GenericAnimal {
   public String getNoise() { return “Meow”; }
}
```

```
public class GenericDog extends GenericAnimal {
   // nothing special to hear here
}
```

```
public class Ewokian extends GenericDog {
   public String getNoise() { return “Oonga!”; }
}
```

What is printed by:

```
GenericAnimal A = new GenericAnimal();
System.out.print(A.getNoise());
```

Method dispatch practice, part B-1

```
public class GenericAnimal {
   public String getNoise() { return “Noise”; }
}
```

```
public class Bird extends GenericAnimal {
   public String getNoise() { return “Chirp”; }
}
```

```
public class Cat extends GenericAnimal {
   public String getNoise() { return “Meow”; }
}
```

```
public class GenericDog extends GenericAnimal {
   // nothing special to hear here
}
```

```
public class Ewokian extends GenericDog {
   public String getNoise() { return “Oonga!”; }
}
```

What is printed by:

```
Bird B = new Bird();
System.out.print(B.getNoise());
```

Method dispatch practice, part B-2 (on paper!)

```
public class GenericAnimal {
   public String getNoise() { return “Noise”; }
}
```

```
public class Bird extends GenericAnimal {
   public String getNoise() { return “Chirp”; }
}
```

```
public class Cat extends GenericAnimal {
   public String getNoise() { return “Meow”; }
}
```

```
public class GenericDog extends GenericAnimal {
   // nothing special to hear here
}
```

```
public class Ewokian extends GenericDog {
   public String getNoise() { return “Oonga!”; }
}
```

What is printed by:

```
GenericAnimal B = new Bird();
System.out.print(B.getNoise());
```

Method dispatch practice, part C

```
public class GenericAnimal {
   public String getNoise() { return “Noise”; }
}
```

```
public class Bird extends GenericAnimal {
   public String getNoise() { return “Chirp”; }
}
```

```
public class Cat extends GenericAnimal {
   public String getNoise() { return “Meow”; }
}
```

```
public class GenericDog extends GenericAnimal {
   // nothing special to hear here
}
```

```
public class Ewokian extends GenericDog {
   public String getNoise() { return “Oonga!”; }
}
```

What is printed by:

```
GenericAnimal C = new Cat();
System.out.print(C.getNoise());
```

Method dispatch practice, part D

```
public class GenericAnimal {
   public String getNoise() { return “Noise”; }
}
```

```
public class Bird extends GenericAnimal {
   public String getNoise() { return “Chirp”; }
}
```

```
public class Cat extends GenericAnimal {
   public String getNoise() { return “Meow”; }
}
```

```
public class GenericDog extends GenericAnimal {
   // nothing special to hear here
}
```

```
public class Ewokian extends GenericDog {
   public String getNoise() { return “Oonga!”; }
}
```

What is printed by:

```
GenericAnimal D = new GenericDog();
System.out.print(D.getNoise());
```

Method dispatch practice, part E-1

```
public class GenericAnimal {
   public String getNoise() { return “Noise”; }
}
```

```
public class Bird extends GenericAnimal {
   public String getNoise() { return “Chirp”; }
}
```

```
public class Cat extends GenericAnimal {
   public String getNoise() { return “Meow”; }
}
```

```
public class GenericDog extends GenericAnimal {
   // nothing special to hear here
}
```

```
public class Ewokian extends GenericDog {
   public String getNoise() { return “Oonga!”; }
}
```

What is printed by:

```
GenericAnimal E = new Ewokian();
System.out.print(E.getNoise());
```

Method dispatch practice, part E-2

```
public class GenericAnimal {
   public String getNoise() { return “Noise”; }
}
```

```
public class Bird extends GenericAnimal {
   public String getNoise() { return “Chirp”; }
}
```

```
public class Cat extends GenericAnimal {
   public String getNoise() { return “Meow”; }
}
```

```
public class GenericDog extends GenericAnimal {
   // nothing special to hear here
}
```

```
public class Ewokian extends GenericDog {
   public String getNoise() { return “Oonga!”; }
}
```

What is printed by:

```
Ewokian E = new Ewokian();
GenericAnimal F = E;
System.out.print(F.getNoise());
```

The java.lang.Object

- All Java objects inherit from `java.lang.Object`
- Commonly-used/overridden public methods:

```
String     toString()
boolean    equals(Object obj)
int        hashCode()
Object     clone()
```

Method dispatch practice, part F

```
public class Object {
    String toString()                  { … }
    boolean       equals(Object obj)   { … }
    int           hashCode()           { … }
    Object clone()                     { … }
}

public class Point {
    private final int x, y;
    public Point(int px, int py) { x = px; y = py; }
    String  toString {
        return String.valueOf(x) + " " +
               String.valueOf(y);
    }
    boolean equals(Point p) {
      return x == p.x && y == p.y;
    }
    int     hashCode() {
      return toString().hashCode();
    }
    …
```

Method dispatch practice, part F (client from paper)

```java
public class Main {
    public static void check(Object a, Object b) {
        if (a.equals(b)) {
            System.out.println("True");
        } else {
            System.out.println("False");
        }
    }

    public static void main(String[] args) {
        Point p = new Point(1, 42);
        Point q = new Point(1, 42);
        check(p, q);
    }
}
```

Overriding java.lang.Object’s .equals

- The default .equals:

```
public class Object {
    public boolean equals(Object obj) {
        return this == obj;
    }
}
```

- An aside: Do you like:

```
public class CheckingAccountImpl
        implements CheckingAccount {
    @Override
    public boolean equals(Object obj) {
        return false;
    }
}
```

The `.equals(Object obj)` contract

- An equivalence relation
  - Reflexive: ∀`x` `x.equals(x)`
  - Symmetric: ∀`x,y` `x.equals(y)` if and only if `y.equals(x)`
  - Transitive: ∀`x,y,z` `x.equals(y)` and `y.equals(z)` implies `x.equals(z)`
- Consistent
  - Invoking `x.equals(y)` repeatedly returns the same value unless `x` or `y` is modified
- `x.equals(null)` is always false

The `.hashCode()` contract

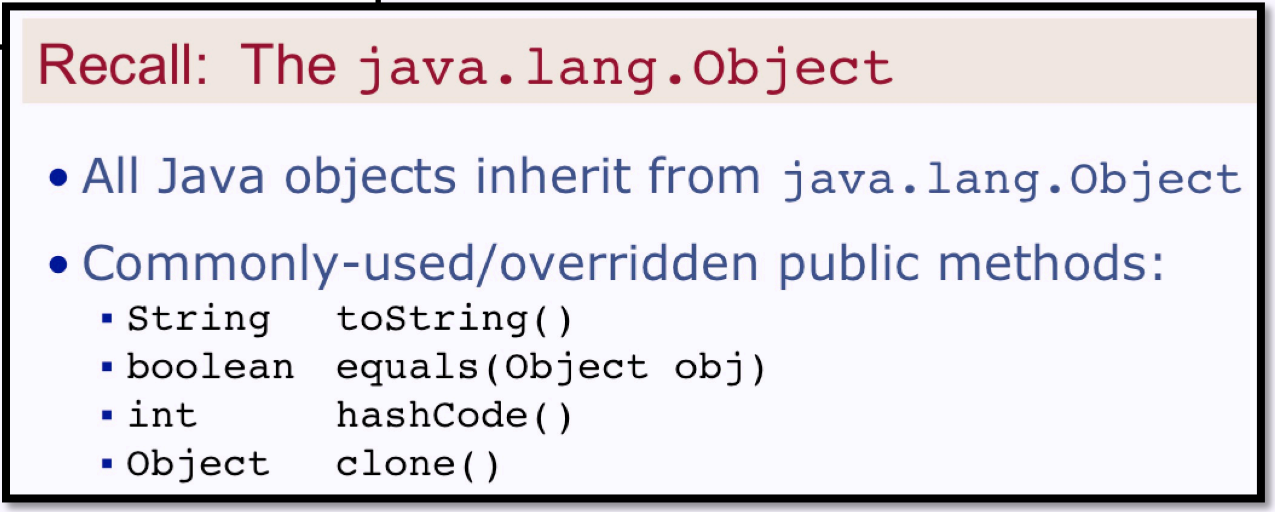
- Consistent
  - Invoking `x.hashCode()` repeatedly returns same value unless `x` is modified
- Equality implies `hashCode()` equality
  - i.e., `x.equals(y)` implies `x.hashCode() == y.hashCode()`
  - The reverse implication is not necessarily true:
    - `x.hashCode() == y.hashCode()` does not imply `x.equals(y)`
- Advice: You should override `.equals()` if and only if you override `.hashCode()`

The `.clone()` contract

- Returns a *deep copy* of an object
- Generally (but not necessarily!):
  - `x.clone() != x`
  - `x.clone().equals(x)`

A lesson in equality

```
public class Point {
  private final int x;
  private final int y;
  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
```



Recall: The `java.lang.Object`

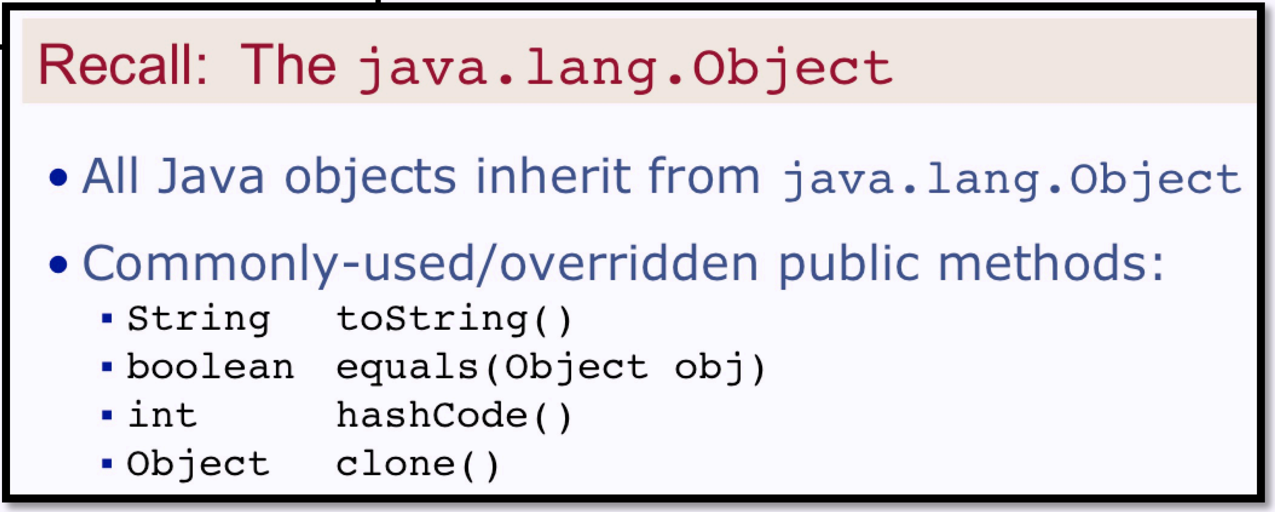
- All Java objects inherit from `java.lang.Object`
- Commonly-used/overridden public methods:
  - `String   toString()`
  - `boolean  equals(Object obj)`
  - `int      hashCode()`
  - `Object   clone()`

Implement the `.equals` method for the `Point` class.

A tempting but incorrect solution

```
public class Point {
   private final int x;
   private final int y;
   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }
}
```

```
public boolean equals(Point p) {
   return x == p.x && y == p.y;
}
```



Recall: The `java.lang.Object`

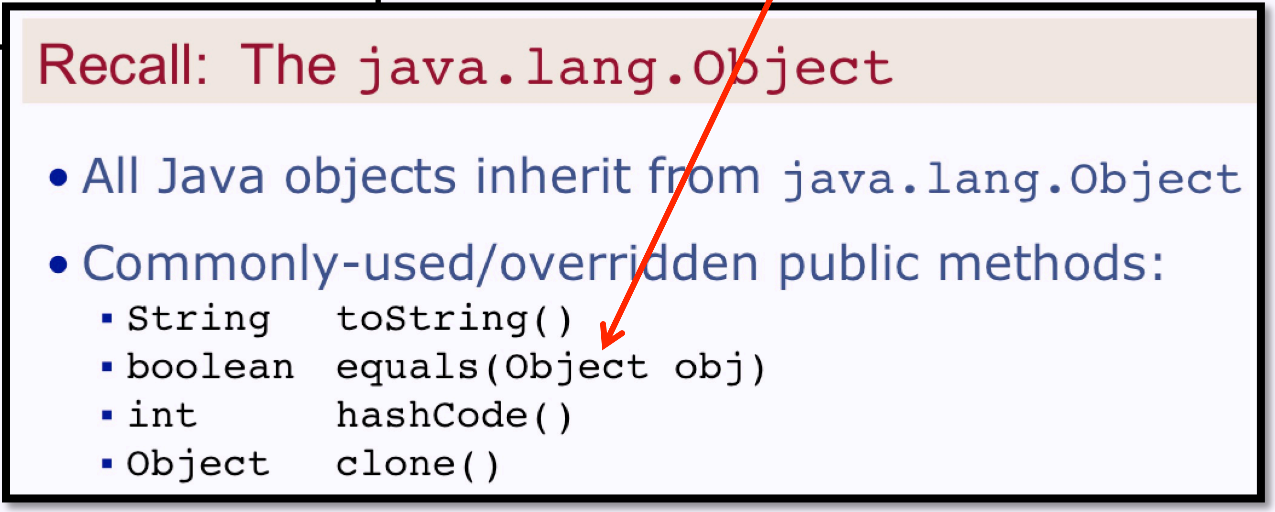
- All Java objects inherit from `java.lang.Object`
- Commonly-used/overridden public methods:
  - `String   toString()`
  - `boolean  equals(Object obj)`
  - `int      hashCode()`
  - `Object   clone()`

A tempting but incorrect solution

```
public class Point {
  private final int x;
  private final int y;
  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
```

```
public boolean equals(Point p) {
  return x == p.x && y == p.y;
}
```

Types must match



Recall: The `java.lang.Object`

- All Java objects inherit from `java.lang.Object`
- Commonly-used/overridden public methods:
  - `String   toString()`
  - `boolean  equals(Object obj)`
  - `int      hashCode()`
  - `Object   clone()`

`boolean equals(Point p)` does not override `boolean equals(Object obj)`

A correct solution

```
public class Point {
  private final int x;
  private final int y;
  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public boolean equals(Object obj) {
    if (!(obj instanceof Point)
      return false;
    Point p = (Point) obj;
    return x == p.x && y == p.y;
}
```

A new challenge

```
public class Point {
  private final int x;
  private final int y;
  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public boolean equals(Object obj) {
    if (!(obj instanceof Point))
      return false;
    Point p = (Point) obj;
    return x == p.x && y == p.y;
}
```

```
public class ColorPoint
      extends Point {
  private final Color color;

  public ColorPoint(int x,
                    int y,
                    Color color) {
    super(x, y);
    this.color = color;
  }
}
```

Implement `.equals` for the `ColorPoint` class.
You may assume `Color` correctly implements `.equals`

A tempting solution

```
public class Point {
  private final int x;
  private final int y;
  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public boolean equals(Object obj
    if (!(obj instanceof Point)
      return false;
    Point p = (Point) obj;
    return x == p.x && y == p.y;
}
```

```
public class ColorPoint
      extends Point {
  private final Color color;

  public ColorPoint(int x,
                    int y,
                    Color color) {
    super(x, y);
    this.color = color;
  }

  public boolean equals(Object obj) {
    if (!(obj instanceof ColorPoint))
       return false;
    ColorPoint cp = (ColorPoint) obj;
    return super.equals(cp) &&
           color.equals(cp.color);
}
```

A tempting solution

```
public class Point {
  private final int x;
  private final int y;
  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public boolean equals(Object obj
    if (!(obj instanceof Point))
      return false;
    Point p = (Point) obj;
    return x == p.x && y == p.y;
}
```

```
public class ColorPoint
      extends Point {
  private final Color color;

  public ColorPoint(int x,
                    int y,
                    Color color) {
    super(x, y);
    this.color = color;
  }

  public boolean equals(Object obj) {
    if (!(obj instanceof ColorPoint))
       return false;
    ColorPoint cp = (ColorPoint) obj;
    return super.equals(cp) &&
           color.equals(cp.color);
}
```

A problem: `p.equals(cp)` but `!cp.equals(p)`:

```
Point p = new Point(2, 42);
ColorPoint cp = new ColorPoint(2, 42, Color.BLUE);
```

More problems

```
public class Point {
  private final int x;
  private final int y;
  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public boolean equals(Object obj
    if (!(obj instanceof Point)
      return false;
    Point p = (Point) obj;
    return x == p.x && y == p.y;
}
```

```
public class ColorPoint
      extends Point {
  private final Color color;

  public ColorPoint(int x,
                    int y,
                    Color color) {
    super(x, y);
    this.color = color;
  }

  public boolean equals(Object obj) {
    if (!(obj instanceof Point))
       return false;
    if (!(obj instanceof ColorPoint))
       return super.equals(obj);
    ColorPoint cp = (ColorPoint) obj;
    return super.equals(cp) &&
           color.equals(cp.color);
}
```

Consider:

```
Point p = new Point(2, 42);
ColorPoint cp1 = new ColorPoint(2, 42, Color.BLUE);
ColorPoint cp2 = new ColorPoint(2, 42, Color.MAUVE);
```

An abstract solution

```
public abstract class Point {
  private final int x;
  private final int y;
  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public boolean equals(Object obj
    if (!(obj instanceof Point)
      return false;
    Point p = (Point) obj;
    return x == p.x && y == p.y;
}
```

```
public class ColorPoint
      extends Point {
  private final Color color;

  public ColorPoint(int x,
                    int y,
                    Color color) {
    super(x, y);
    this.color = color;
  }

  public boolean equals(Object obj) {
    if (!(obj instanceof ColorPoint))
       return false;
    ColorPoint cp = (ColorPoint) obj;
    return super.equals(cp) &&
           color.equals(cp.color);
}
```

```
public class PointImpl extends Point {
  public PointImpl(int x, int y) { super(x,y); }
  public boolean equals(Object obj) {
    if (!(obj instanceof PointImpl))
      return false;
    return super.equals(obj);
  }
```