Lecture 4

Primitives, References
and Class Inheritance
Roadmap

• Last time
  – Objects, methods, public/private

• Today
  – Primitives vs. References
  – Inheritance

• Next time
  – Advanced Inheritance (Interfaces)
  – “static”
  – Generics
Lecture 3 Popular Issues

The deal with null and when to use it
Primitives vs. References

• **Primitive** types are basic Java types
  – int, long, double, boolean, char, short, byte, float
  – The actual *values* are stored in the variable

• **Reference** types are arrays and objects
  – String, int[], Baby, ...
How java stores *primitives*

- Variables are like fixed size cups
- Primitives are small enough that they just fit into the cup

![Diagram showing int, double, char, and boolean going into a cup]

- int
- double
- char
- boolean
How java stores **objects**

- Objects are too big to fit in a variable
  - Stored somewhere else
  - Variable stores a number that locates the object
How java stores **objects**

- Objects are too big to fit in a variable
  - Stored somewhere else
  - Variable stores a number that locates the object
References

• The object’s location is called a reference

• == compares the references

  Baby shiloh1 = new Baby(“shiloh”);
  Baby shiloh2 = new Baby(“shiloh”);

Does shiloh1 == shiloh2?
References

• The object’s location is called a reference
• == compares the references
  Baby shiloh1 = new Baby("shiloh");
  Baby shiloh2 = new Baby("shiloh");
Does shiloh1 == shiloh2?

no
References

Baby shiloh1 = new Baby("shiloh");
Baby shiloh2 = new Baby("shiloh");
Baby mybaby = new Baby("davy", true)
mybaby.name = "david"

mybaby’s location

name = "davy"
isMale = true
...

References
References

```java
Baby mybaby = new Baby("davy", true)
mybaby.name = "david"
```
• Using = updates the reference.

```
baby1 = baby2;
```
• Using = updates the reference.

\[
baby1 = baby2;
\]
References

• using [ ] or •
  – Follows the reference to the object
  – May modify the object, but never the reference

• Imagine
  – Following directions to a house
  – Moving the furniture around

• Analogous to
  – Following the reference to an object
  – Changing fields in the object
void doSomething(int x, int[] ys, Baby b) {
    x = 99;
    ys[0] = 99;
    b.name = "99";
}

... 

int i = 0;
int[] j = {0};
Baby k = new Baby("50", true);
doSOMething(i, j, k);

i=? j=? k=?
Back to null

- Null is used to leave a reference uninitialized
- Why would we do this?

```java
int[] runners = each runner’s time;

int bestRunnerTime = Integer.MAX_VALUE;
int bestRunnerIndex = -1;

for (int i = 0; i < runners.length; i++) {
    if (runners[i] < bestRunnerTime) {
        bestRunnerTime = runners[i];
        bestRunnerIndex = i;
    }
}
```
Back to null

- Null is used to leave a reference uninitialized
- Why would we do this?

```java
Runner[] runners = an array of Runner objects;

int bestRunnerTime = Integer.MAX_VALUE;
Runner bestRunnerRef = null;

for (Runner r : runners) {
    if (r.getResult() < bestRunnerTime) {
        bestRunnerTime = r.getResult();
        bestRunnerRef = r;
    }
}
```
INHERITANCE
Inheritance

“The industrial revolution for software engineering”

What’s the point?

- Minimize code duplication
- Maximize code reuse
Code duplication: what gives?

• The more code you have...
  – The more bugs
  – The more difficult to maintain

• Need to change one version?
  – Remember to update all versions!
Motivating Example

• Fantasy adventure game

```java
public class Player {
    public String name;
    public int hp = 100;

    public void sayName() {
        System.out.println(name);
    }

    public void punchFace(Player target) {
        target.hp -= 10;
    }
}
```
Without Inheritance

• Now create a Warrior...

```java
public class Warrior {
    // ugh, gotta copy and paste
    // Player's stuff

    public void swordSlash(Player target) {
        target.hp -= 25;
    }

    public void swordSlash(Warrior target) {
        target.hp -= 25;
    }
}
```
With Inheritance

• What do we want to say?

That a Warrior *is a type of* Player.

```java
public class Warrior extends Player {
    public void swordSlash(Player target) {
        target.hp -= 25;
    }
}
```
Inheritance Relationships

By “public class Warrior extends Player”

• We mean that a Warrior is a type of Player

• Warrior gets all Player fields (hp, name)
• Warrior gets all Player methods (sayName()...
Class Diagram

Player

Warrior

Super/parent class

Sub/child Class
Other types of Relationships?

Magician

Player

Warrior

SwordWarrior

ShieldWarrior

SwordShieldWarrior
public class Game {
    public static void main(String[] args) {
        Warrior thrall = new Warrior();
        Warrior arthas = new Warrior();
        thrall.swordSlash(arthas);
        arthas.punchFace(thrall);
    }
}

Inherited from Player

Player
    sayName()
    punchFace()

Warrior
    swordSlash()
How does Java do that?

What Java does when it sees 

arthas.punchFace(thrall)

1. Look for punchFace() in the Warrior class
2. It’s not there! Does Warrior have a parent?
3. Look for punchFace() in Player class
4. Found it! Call punchFace()
5. Deduct hp from thrall.
public class Game {

    public static void main(String[] args) {
        Player thrall = new Player();
        Warrior arthas = new Warrior();
        thrall.swordSlash(arthas);
    }

}
public class Player {
    // ...
    public void punchFace(Player target) {
        target.hp -= 10;
    }
}

public class Warrior extends Player {
    public void punchFace(Player target) {
        // what the Player did, and then some!
    }
}
public class Warrior extends Player {

    public void punchFace(Player target) {
        System.out.println("ROAR");
    }
}

in main():

    Warrior thrall = new Warrior();
    Warrior arthas = new Warrior();
    thrall.punchFace(arthas);

> "ROAR"
public class Warrior extends Player {

    public void punchFace(Player target) {
        super.punchFace(target);
        System.out.println("ROAR");
    }
}

in main():

    Warrior thrall = new Warrior();
    Warrior arthas = new Warrior();
    thrall.punchFace(arthas);
> "ROAR" and -10 HP
Illegal Method Overriding

public class Warrior extends Player {

    public void punchFace(Player target) {
        super.super.punchFace();
    }

}  

in main():

    Warrior thrall = new Warrior();
    Warrior arthas = new Warrior();

    // want to use the Player version of punchFace...
    ((Player) thrall).punchFace(arthas);

    Error!

    You will still use the Warrior’s version!
Cool Trick: Java Annotations

```java
public class Player {
    // ...
    public void punchFace(Player target) {
        target.hp -= 10;
    }
}
```

```java
public class Warrior extends Player {
    @Override
    public void punchFace(int howHard) {
    }
}
```

Method signature doesn’t match.

Eclipse will throw an error for you!
Inheritance Summary

- **class A extends B** means: **A** is a subclass of **B**
- **A** has all the fields and methods that **B** has
- **A** can add it’s own fields and methods
- **A** can have at most 1 parent
- **A** can replace a parent’s method by overriding it
- **If A doesn’t implement something Java searches ancestors**
- **To override methods properly, match signatures exactly!**