

---

POLYMORHISM



# Polymorphism

## polymorphism

/ˌpɒlɪˈmɔːfɪz(ə)m/ 

*noun*

the condition of occurring in several different forms.

"the complexity and polymorphism of human cognition"

- **BIOLOGY**

the occurrence of different forms among the members of a population or colony, or in the life cycle of an individual organism.

"the workers of this species exhibit polymorphism, specialized physical castes"

- **GENETICS**

the presence of genetic variation within a population, upon which natural selection can operate.

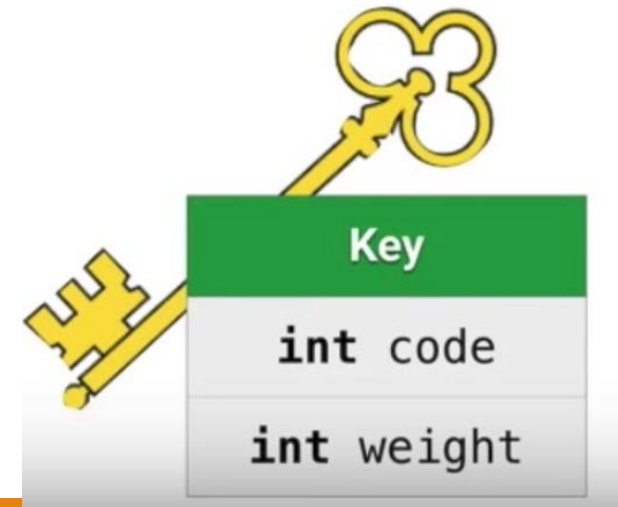
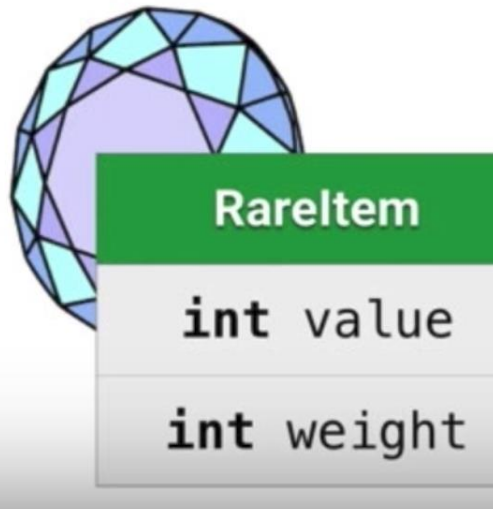
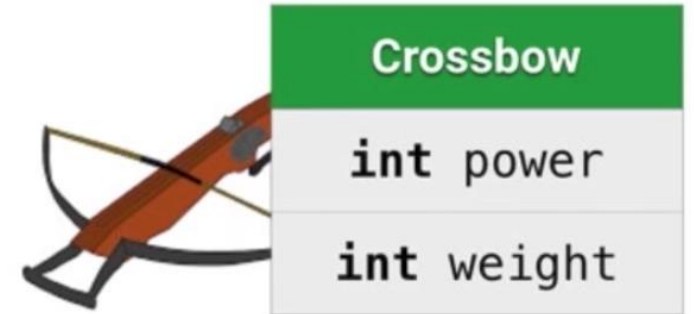


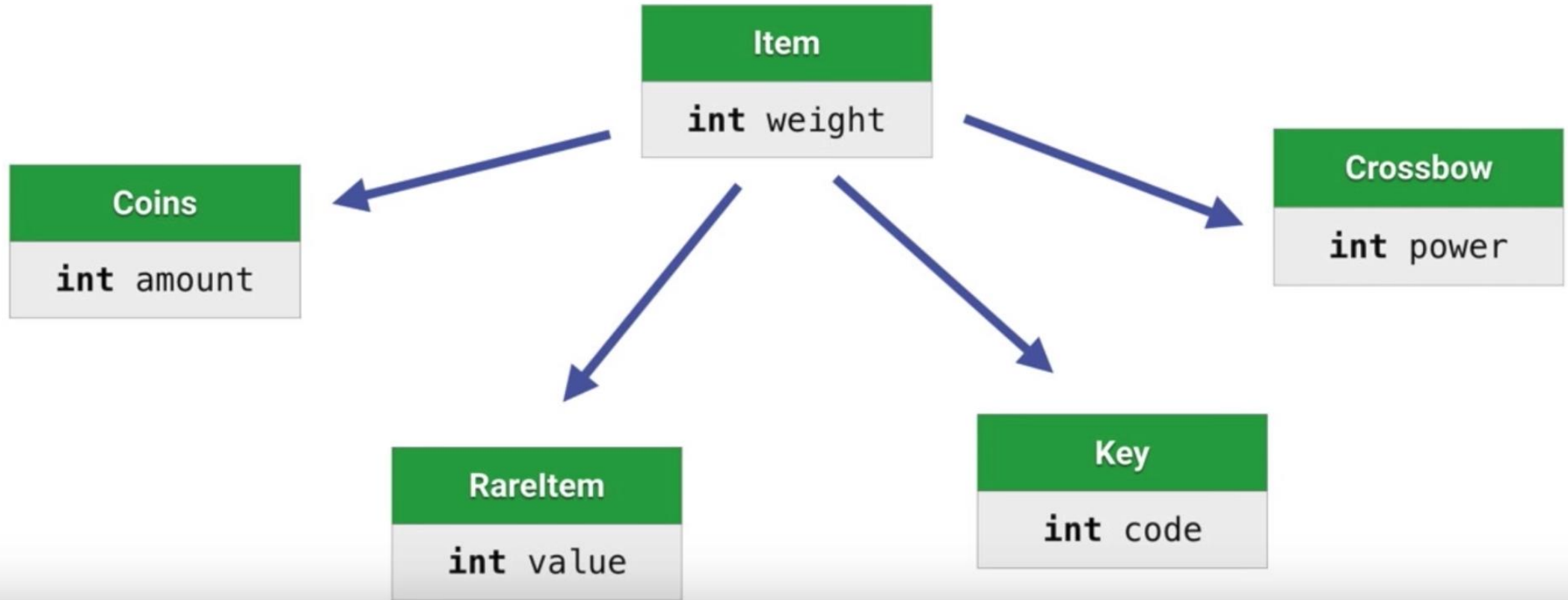
Translations, word origin, and more definitions

## Bag & Items Example



# Bag & Items Example





# Bag & Items Example

---

```
public class Bag{  
    int currentWeight;  
    boolean canAddItem(Item item);  
}
```

```
boolean canAddItem(Item item){  
    if(currentWeight + item.weight > 20){  
        return false;  
    }  
    else{  
        return true;  
    }  
}
```

# Bag & Items Example

---

```
public static void main(String [] args){  
    ...  
    Crossbow crossbow = new Crossbow();  
    if(bag.canAddItem(crossbow)){  
        bag.addItem(crossbow);  
    }  
    ...  
}
```



# Bag & Items Example

---

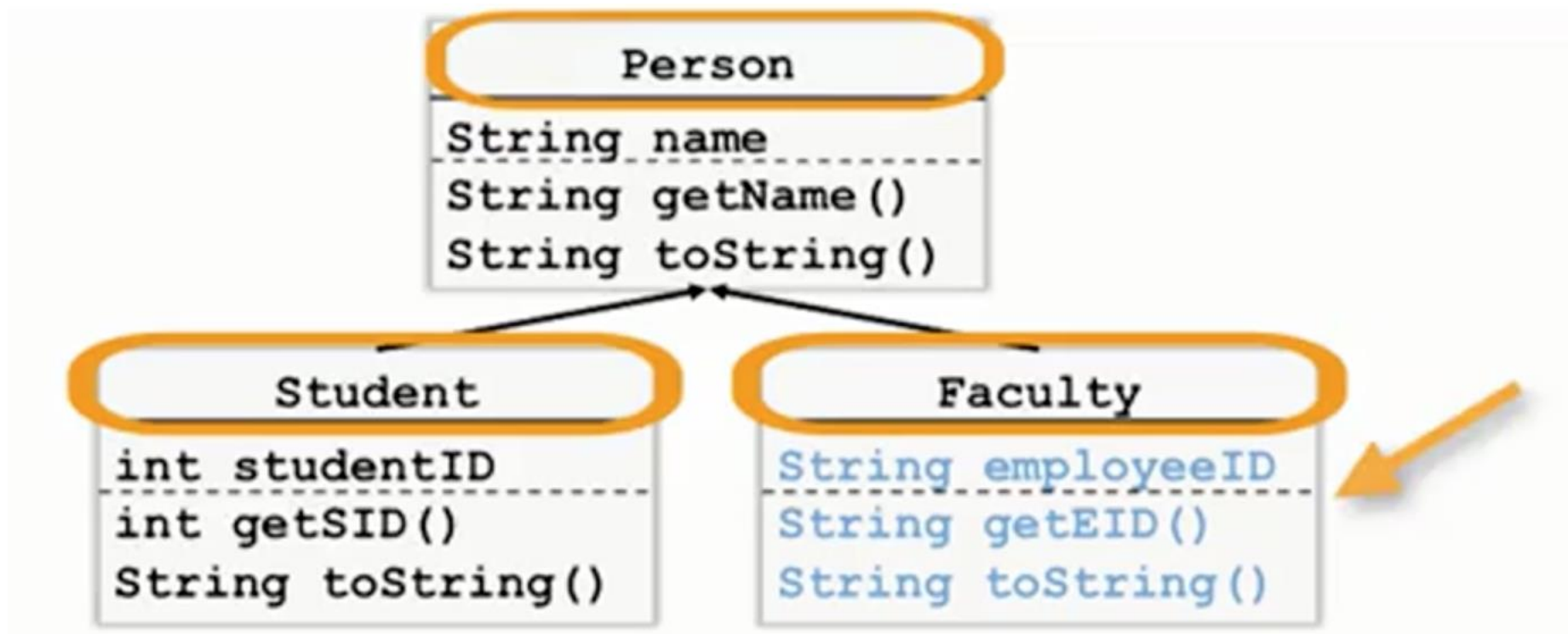
```
public static void main(String [] args){  
    ...  
    Map treasureMap = new Map();  
    if(bag.canAddItem(treasureMap)){  
        bag.addItem(treasureMap);  
    }  
    ...  
}
```

# Polymorphism

---

Superclass reference to sub class object

```
Person s=new Student("Cara", 1234);
```



```
// assume appropriate ctors
Person p[] = new Person[3];
p[0] = new Person( "Tim" );
p[1] = new Student( "Cara", 1234 );
p[2] = new Faculty( "Mia", "ABCD" );

for(int i = 0; i < p.length; i++)
{
    System.out.println( p[i] );
}
```

For the code below:

```
1 // in main
2 Person p[] = new Person[3];
3 p[0] = new Person( "Tim" );
4 p[1] = new Student( "Cara", 1234);
5 p[2] = new Faculty( "Mia", "ABCD" );
6
7 for(int i = 0; i < p.length; i++)
8 {
9     System.out.println( p[i] );
10 }
```

Do you think the method "toString" in the Person class will be called when p[1] is printed or do you think the "toString" method in the Student class will be called when p[1] is printed?

- ☒ The "toString" method in Student will be called for p[1]
- ☐ The "toString" method in Person will be called for p[1]

# Compile Time Rules

---

Compiler only knows reference type,

Can only look in reference type class for method

Outputs a method signature

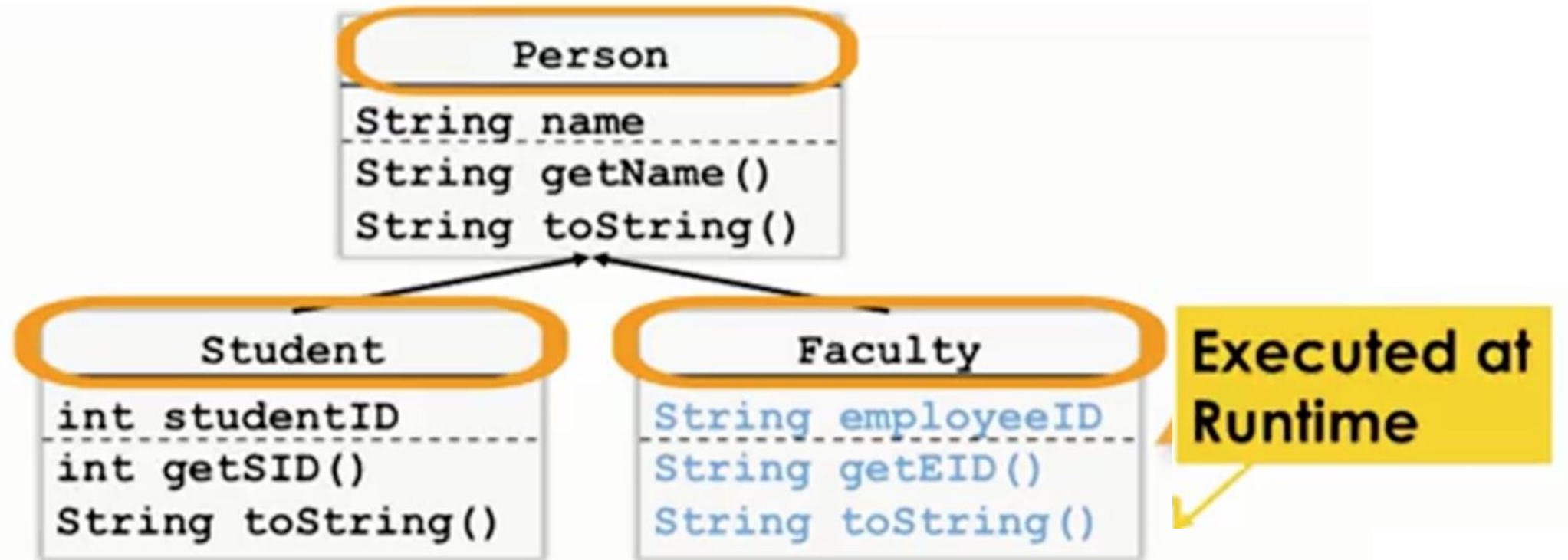
# Run Time Rules

---

Follow exact runtime type of object to find method.

Must match compile time method signature to appropriate method in actual object's class.

```
Person s = new Student("Cara", 1234);  
s.toString();
```



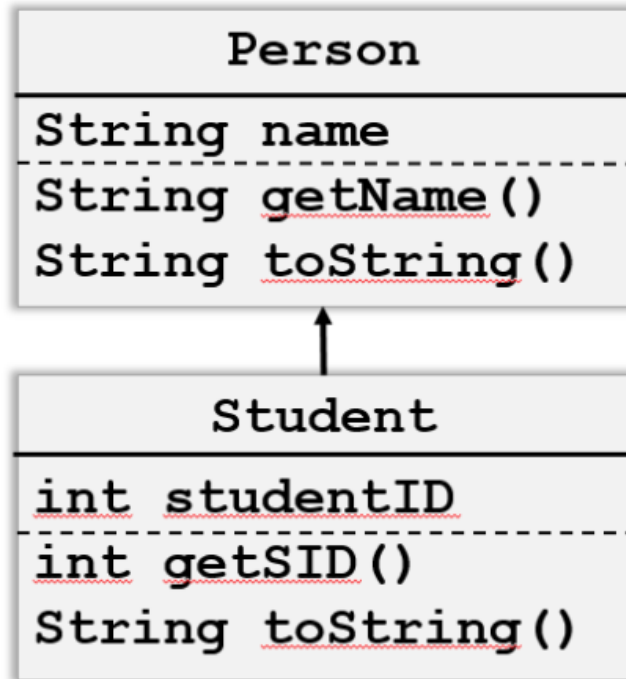


# Casting

---

Step through decisions made at compile time and runtime

Use casting of objects to aid the compiler



```
Person s = new Student("Cara",1234) ;  
s.getSID() ;
```

When we call `s.getSID()`, what do you think will happen?

# Casting

---

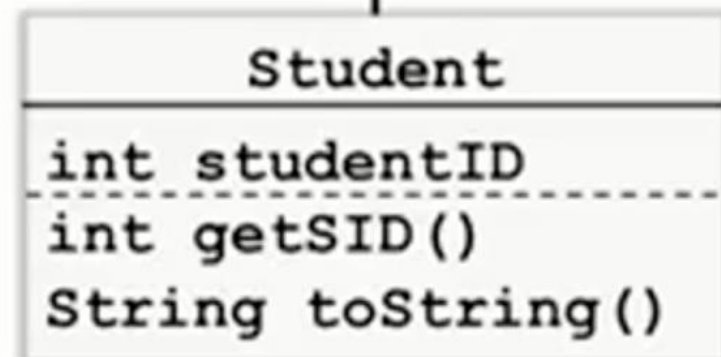
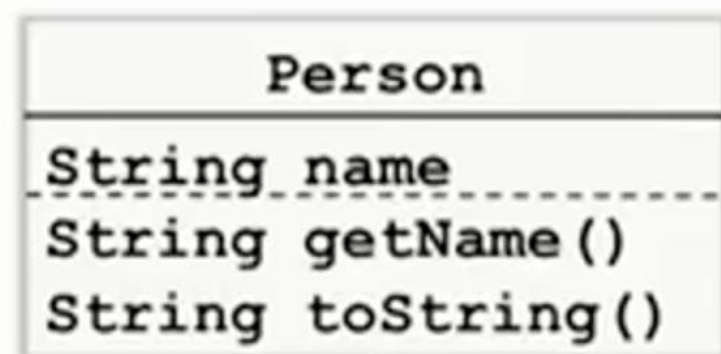
Automatic type promotion (like int to double)

- Superclass ref=new Subclass(); (widening)

Explicit casting (like double to int)

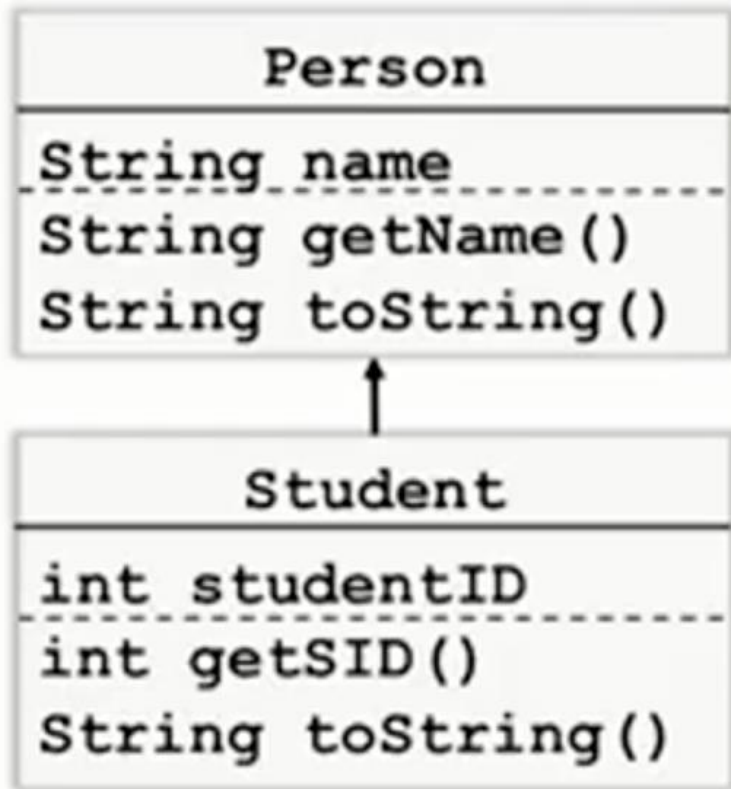
Subclass ref=(Subclass) superRef; (narrowing)

**Be careful! Compiler trusts  
you**



```
Person s = new Student("Cara",1234);  
s.getSID();  
( (Student)s ).getSID();
```

**This works!**



```
Person s = new Person("Tim");  
( (Student)s ).getSID();
```

**Runtime Error!**  
**java.lang.ClassCastException:**  
**From Person to Student**

How about this?

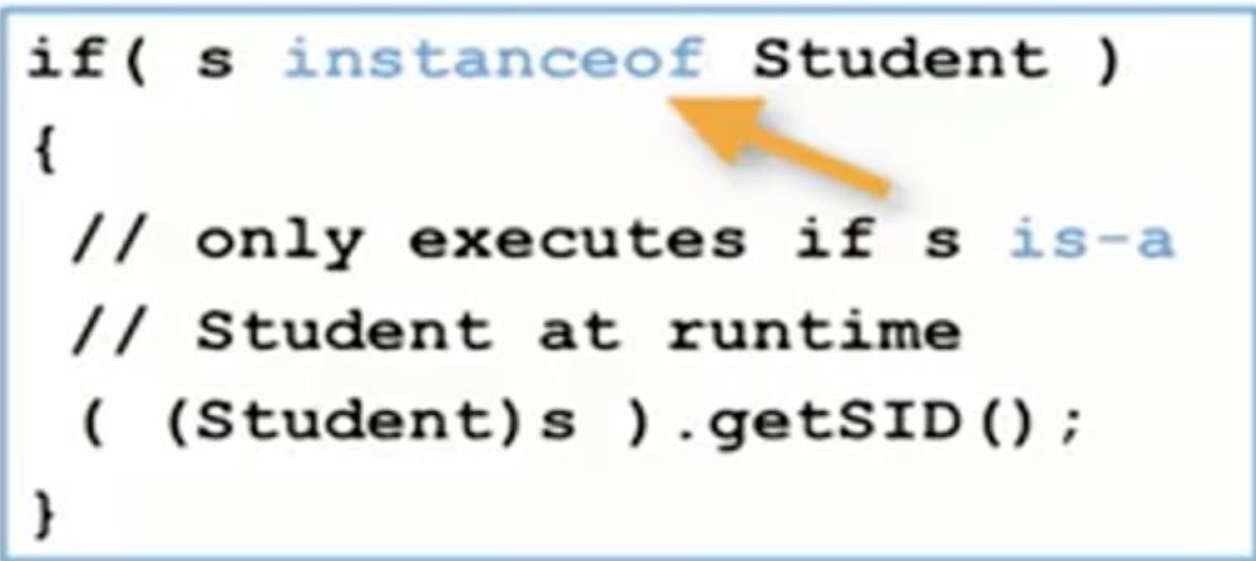
# Runtime Type Check

---

## InstanceOf

- Provides runtime check of **is-a** relationship

```
if( s instanceof Student )
{
    // only executes if s is-a
    // Student at runtime
    ( Student ) s .getSID();
}
```



# Polymorphism

---

Compiler Time Decisions

Runtime Decisions

# Polymorphism Challenge 1

```
public class Person
{
    private String name;

    public Person(String name) { this.name = name; }
    public boolean isAsleep(int hr) { return 22 < hr || 7 > hr; }
    public String toString() { return name; }

    public void status( int hr )
    {
        if ( this.isAsleep( hr ) )
            System.out.println( "Now offline: " + this );
        else
            System.out.println( "Now online: " + this );
    }
}

public class Student extends Person
{
    public Student(String name) {
        super(name);
    }

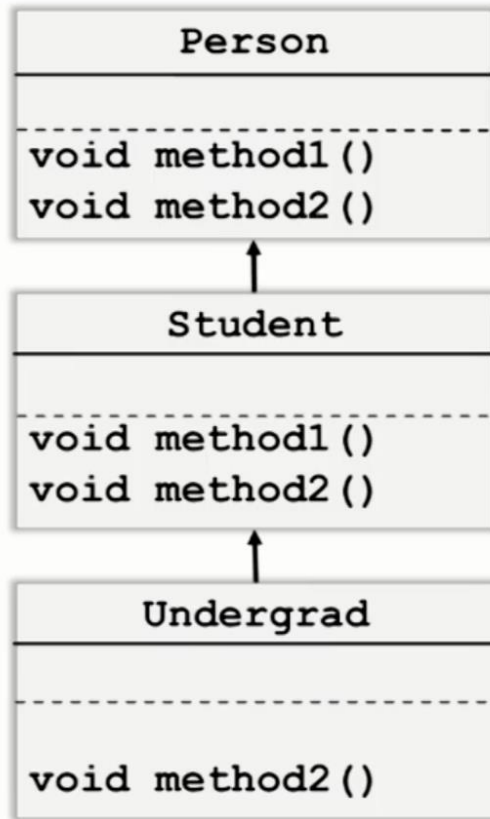
    public boolean isAsleep( int hr ) // override
    { return 2 < hr && 8 > hr; }
}
```

```
Person p;
p = new Student("Sally");
p.status(1);
```



# Polymorphism Challenge 2

---



# Polymorphism Challenge 2

```
public class Person {  
    public void method1() {  
        System.out.print("Person 1 ");  
    }  
    public void method2() {  
        System.out.print("Person 2 ");  
    }  
}  
public class Student extends Person {  
    public void method1() {  
        System.out.print("Student 1 ");  
        super.method1();  
        method2();  
    }  
    public void method2() {  
        System.out.print("Student 2 ");  
    }  
}
```

```
public class Undergrad extends Student {  
    public void method2() {  
        System.out.print("Undergrad 2 ");  
    }  
}
```

```
Person u = new Undergrad();  
u.method1();
```

# Abstract classes and Interfaces

---

Use the keyword Abstract

Compare “inheritance of implementation” and “Inheritance of interface”

Decide between Abstract classes and Interfaces

# Person- Campus Accounts

---

Add method “monthlyStatement”

“Person” objects no longer make sense

How do we?

1. Force subclasses to have this method
2. Stop having actual Person objects
3. Keep having Person references
4. Retain common Person code

Abstract Classes

# Abstract

---

Can make any class abstract with keyword:

```
public abstract class Person{
```

Cannot create objects of this  
type

Class must be abstract if any methods are:

```
public abstract class void monthlyStatement()
```

# Implementation vs. Interface

---

Abstract classes offer inheritance of both!

**Implementation:** instance variables and methods which define common behaviour

**Interface:** method signatures which define required behaviours. You get to inherit both implementation and interface

# Person- Campus Accounts

---

Add method “monthlyStatement”

“Person” objects no longer make sense

How do we?

1. Force subclasses to have this method
2. Stop having actual Person objects
3. Keep having Person references
4. ~~Retain common Person code~~

Then use an Interface!

Abstract Classes

# Interfaces

---



Interface



Class

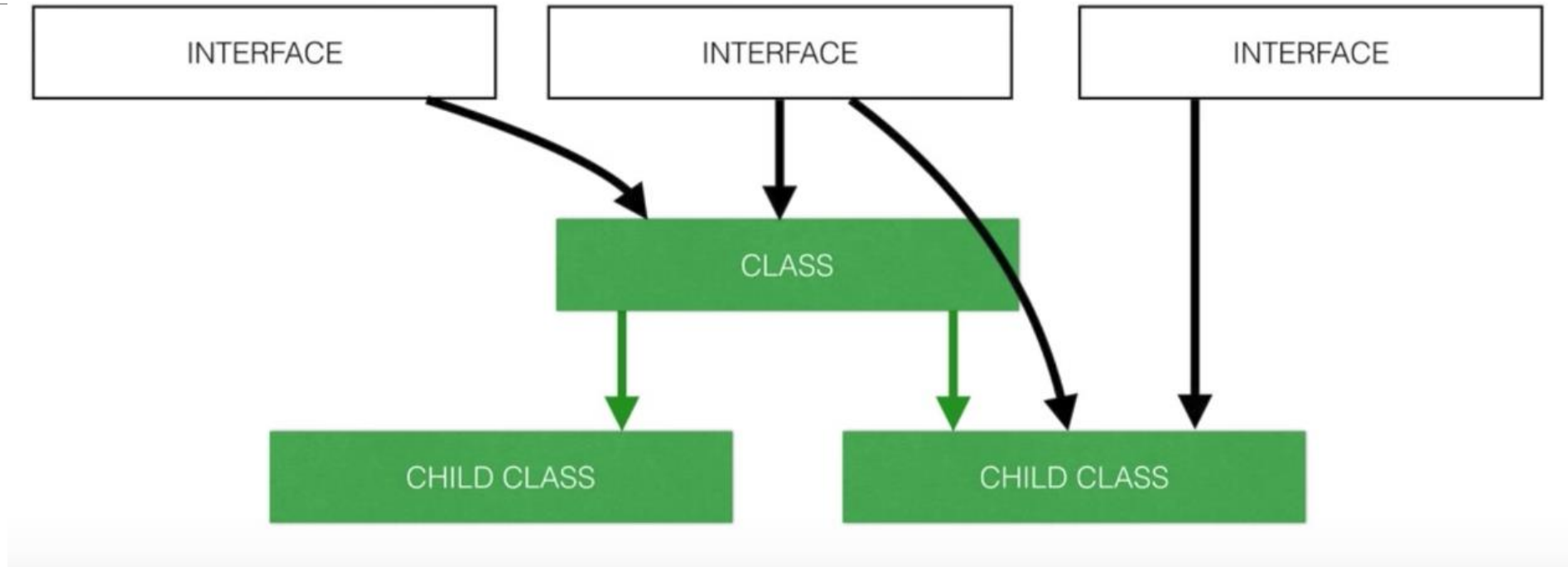


Object



# Interfaces

---



# Interfaces

---

Interfaces only define required methods

Classes can inherit from multiple Interfaces

```
// Defined in java.lang.Comparable
package java.lang;

public interface Comparable<E> {
    // Compare this object's name to o's name
    // Return < 0, 0, > 0 if this object compares
    //    less than, equal to, greater than o.
    public abstract int compareTo( E o );
}
```

```
public class Person implements Comparable<Person> {  
    private String name;  
    ...
```

```
@Override
```

```
public int compareTo( Person o ) {  
    return this.getName().compareTo( o.getName() );  
}
```

# Interfaces Summary

---

- Interfaces define what a class should **do** but not **how** to do it.
- Creating an **interface** is very similar to creating a **class**.
- An interface's sole purpose is to be **implemented** by one or more classes.
- You **cannot** create an instance (Object) from an interface.
- It's not reducing code repetition, it's more about **enforcing a good design**.

# Abstract class or Interface

---

If you just want to define a required method:

- Interface

If you want to define potentially required methods AND common behaviour:

- Abstract Classes

# Polymorphism Examples

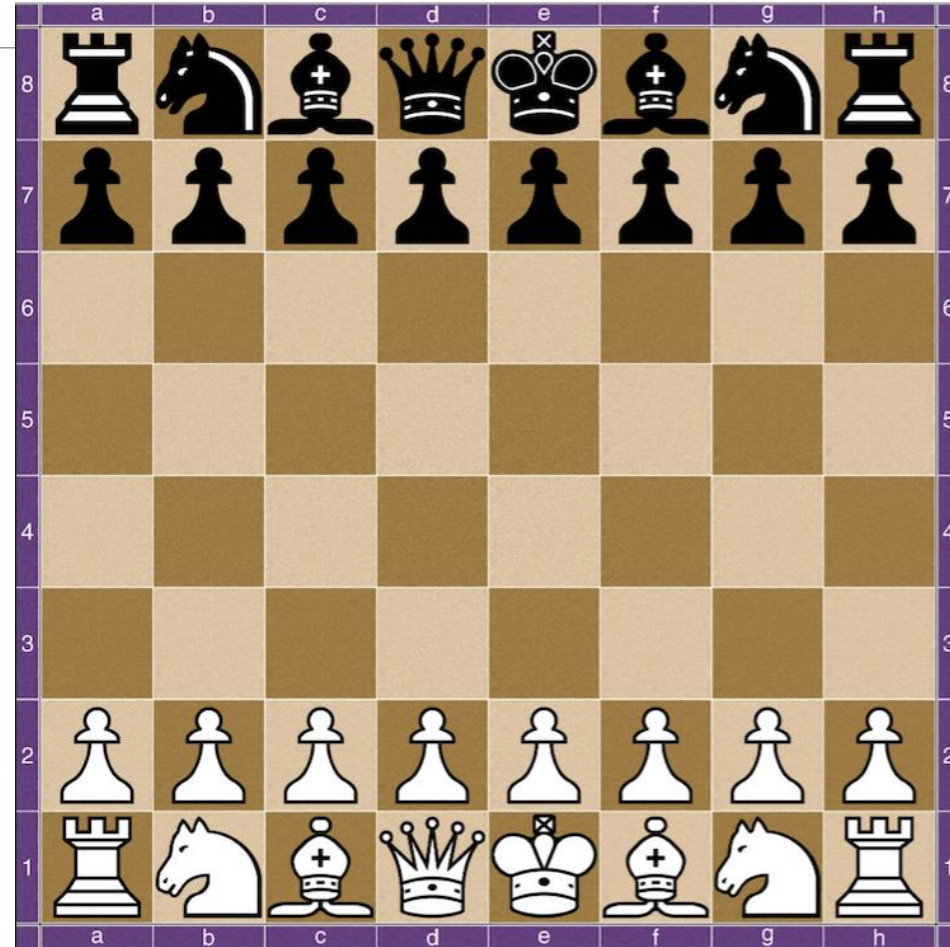
---

P

## The Chess Example

We've seen how Inheritance allows you to extend classes and add more functionality to them.

Sometimes you not only want to extend the functionality of a class, but also modify it slightly in the child class. For example, say you're building a Java chess game.





---

A good Java design will have a **class** for each piece type:



King



Queen



Rock



Bishop



Knight



Pawn

And they should all inherit from a common base class: **Piece**

```
class Game{  
    Piece [][] board;  
    // Constructor creates an empty board  
    Game(){  
        board = new Piece[8][8];  
    }  
}
```

---

```
class Position{
    int row;
    int column;
    // Constructor using row and column values
    Position(int r, int c){
        this.row = r;
        this.column = c;
    }
}
```

```
class Piece{
    Position position;
}
```

```
class Piece{
    boolean isValidMove(Position newPosition){
        if(position.row>0 && position.column>0
            && position.row<8 && position.column<8){
            return true;
        }
        else{
            return false;
        }
    }
}
```

```
Queen queen = new Queen();  
Position testPosition = new Position(3,10);  
if(queen.isValidMove(testPosition)){  
    System.out.println("Yes, I can move there.");  
}  
else{  
    System.out.println("Nope, can't do!");  
}
```

```
class Rock extends Piece{
    boolean isValidMove(Position newPosition){
        if(newPosition.column == this.column || newPosition.row == this.row){
            return true;
        }
        else{
            return false;
        }
    }
}
```

```
class Bishop extends Piece{
    boolean isValidMove(Position newPosition){
        if(Math.abs(newPosition.column - this.column) == Math.abs(newPosition.row - this.
            return true;
        }
        else{
            return false;
        }
    }
}
```