Ca' Foscari
University
of Venice

Subtyping

Object oriented programming, module 1

Pietro Ferrara

pietro.ferrara@unive.it

Java typing

- Java has declared types
  - Fields, parameters and local variables declare their type
- This type is known at compile time
- The operations in the code must be compliant with the type
  - v1+v2 is not allowed
  - v1.getSpeed() is allowed
- Java 10+ supports not declared types
  - Not part of this module

```
int race(Vehicle v1, Vehicle v2, double length) {
  v1.fullStop();
  v2.fullStop();
  double distanceV1 = 0, distanceV2=0;
  while(true) {
   distanceV1 += v1.getSpeed();
   distanceV2 += v2.getSpeed();
   if(distanceV1 >= length || distanceV2 >= length) {
    if(distanceV1 > distanceV2) return 1;
    else return 2;
   }
   v1.accelerate(random());
   v2.accelerate(random());
  }
}
```

Java typing

- Java is strongly typed

"*whenever an object is passed from a calling function to a called function, its type must be compatible with the type declared in the called function*"[Liskov Zilles, 1974]

"*In a strongly typed language each data area will have a distinct type and each process will state its communication requirements in terms of these types*“ [Jackson, 1977]

- Everything is typed
  – Not only variables and fields!

https://en.wikipedia.org/wiki/Strong_and_weak_typing

```
char* ptr = (char*) malloc(100*sizeof(char));
printf( "%d", (int) ptr );
```

C (weakly typed)

It compiles and executes

123565787665

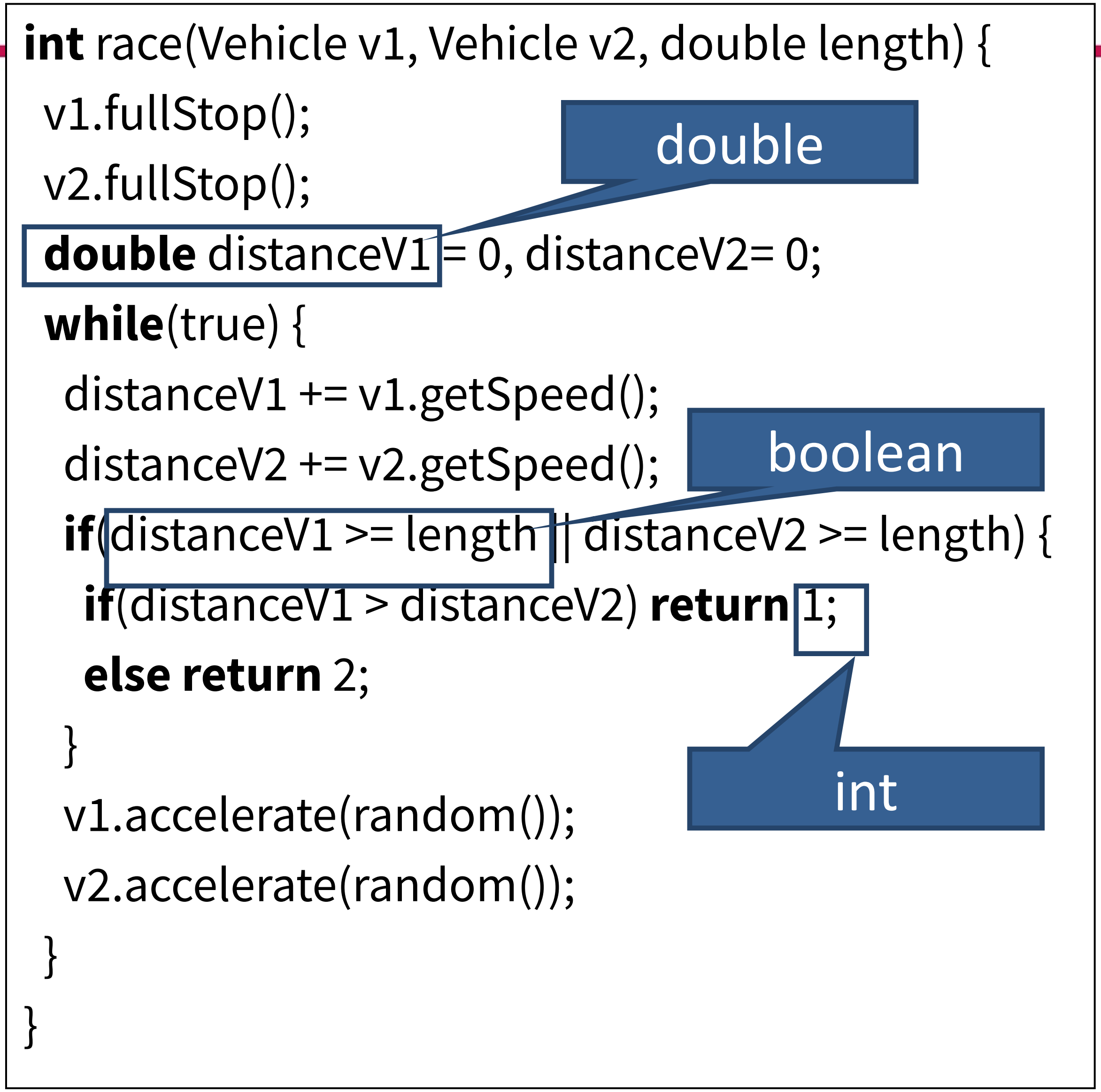
```
char[] arr = new char[100];
System.out. printf( "%d", (int) arr);
```

Java (strongly typed)

It does not compile

Java typing

- Java is statically typed
  - Types are verified at compile time
- Each expression has a type known during the compilation
  - That is, without executing the program
- Some of these types are declared
  - Variables, fields, return types, …
- Others are inferred, e.g.
  - <int>+<int> returns an int



```
int race(Vehicle v1, Vehicle v2, double length) {
 v1.fullStop();
 v2.fullStop();
 double distanceV1 = 0, distanceV2= 0;
 while(true) {
  distanceV1 += v1.getSpeed();
  distanceV2 += v2.getSpeed();
  if(distanceV1 >= length || distanceV2 >= length) {
   if(distanceV1 > distanceV2) return 1;
   else return 2;
  }
  v1.accelerate(random());
  v2.accelerate(random());
 }
}
```

https://en.wikipedia.org/wiki/Type_system#Static_type_checking

Substituting a class with an extension

- Subclasses extend the behavior
- An instance of the superclass can be substituted by a subclass
- If we have a Vehicle, we know we can accelerate or full brake
  - No need to know if it is a bike, a car, a truck, or something else!

```
race(new Car(), new Car(), 100);
race(new Truck(), new Truck(), 100);
race(new Bicycle(), new Bicycle(), 100);
race(new Car(), new Truck(), 100);
```

```
int race(Vehicle v1, Vehicle v2, double length) {
  v1.fullStop();
  v2.fullStop();
  double distanceV1 = 0, distanceV2=0;
  while(true) {
    distanceV1 += v1.getSpeed();
    distanceV2 += v2.getSpeed();
    if(distanceV1 >= length || distanceV2 >= length) {
      if(distanceV1 > distanceV2) return 1;
      else return 2;
    }
    v1.accelerate(random());
    v2.accelerate(random());
  }
}
```

The substitution principle

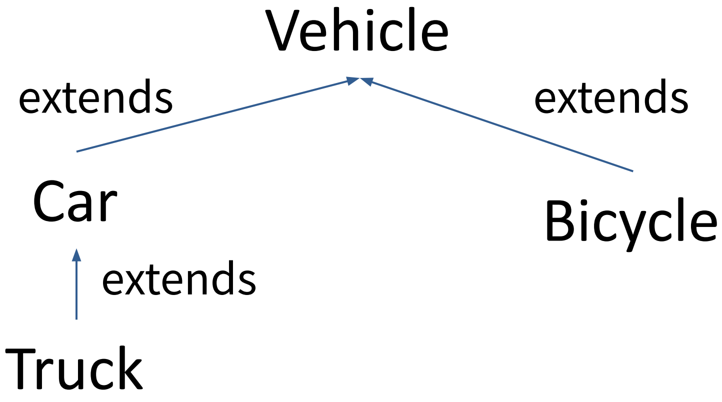
An object o1 instance of class C1 can be substituted by an object o2 of class C2 if class C2 provides the same or a wider interface (fields and methods) of class C1

- All the program points accessing a member of C1 can still access the same members with an instance of C2!
- To run a race, I need to accelerate
  - Cars, trucks and bicycles all provide the interface to accelerate!
    - Inherited from Vehicle!

```
class Vehicle {…}
class Car extends Vehicle {…}
class Truck extends Car {…}
class Bicycle extends Vehicle {…}

int race(Vehicle v1, Vehicle v2, double length) {…}

Car c1 = new Car(), c2 = new Car();
Truck t = new Truck();
Bicycle b = new Bicycle();

race(c1, c2, 100);
race(c1, t, 100);
race(t, b, 100);
race(c1, b, 100);
```

Subtyping

- Type of an object
  - The class it instantiates
- If a class C1 extends another class C2
  - Its instances provides a **<u>wider interface</u>**
- All occurrences of C2 can be substituted by C1
- **<u>C1 is a subtype of C2</u>**
  - E.g., we can assign instances of Car to Vehicle variables



```
Vehicle v1 = new Car();        ✓
Vehicle v2 = new Bicycle();    ✓
race(v1, v2);                  ✓
```

```
int race(Vehicle v1, Vehicle v2, double length) {
  v1.refuel(new FuelTank(…));   □ □
}
```

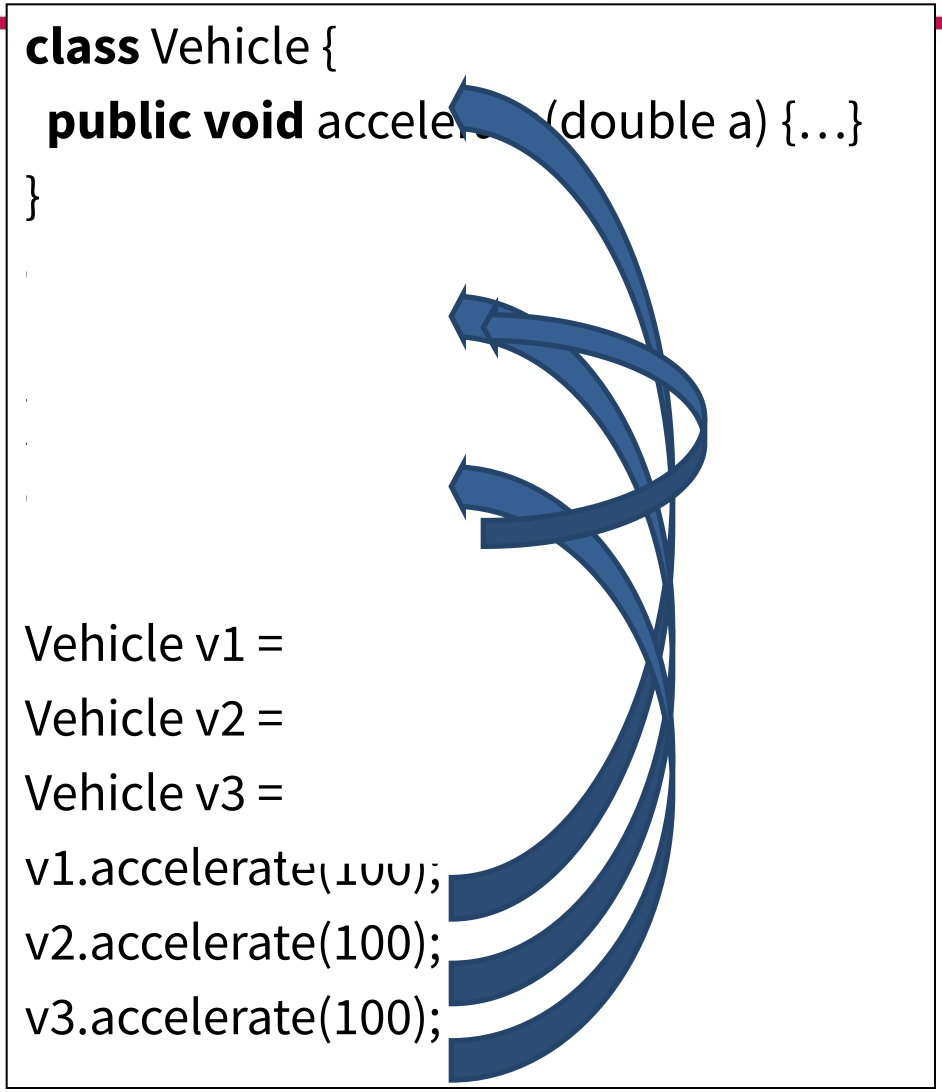
Does not compile

Polymorphism

An object of a given call can have multiple forms (textbook)

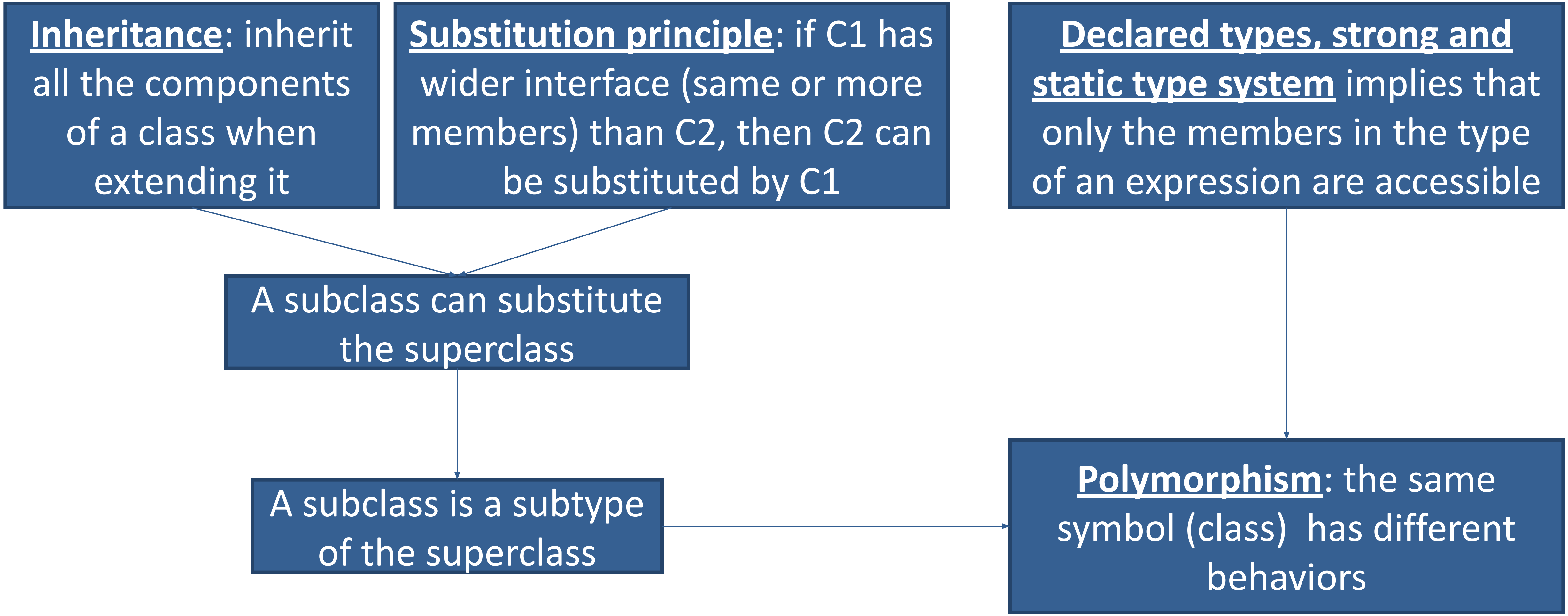
polymorphism is the provision of a single interface to entities of different types or the use of a single symbol to represent multiple different types (Wikipedia)

- Polymorphism enabled by
  - Inheritance
  - Subtyping



```
class Vehicle {
 public void accelerate(double a) {…}
}

Vehicle v1 =
Vehicle v2 =
Vehicle v3 =
v1.accelerate(100);
v2.accelerate(100);
v3.accelerate(100);
```

Summing up



Inheritance: inherit all the components of a class when extending it

Substitution principle: if C1 has wider interface (same or more members) than C2, then C2 can be substituted by C1

Declared types, strong and static type system implies that only the members in the type of an expression are accessible

A subclass can substitute the superclass

A subclass is a subtype of the superclass

Polymorphism: the same symbol (class) has different behaviors

Accessibility of overriding methods

- Modifiers are not part of the method signature
- Access modifiers can be relaxed
  - Wider visibility -> wider overriding
  - E.g., a protected method can be overridden with a public one
- Final methods cannot be overridden
- Static methods cannot be overridden
  - But we will discuss this later…

```
public class Vehicle {
  priva
  prote
   this.
  }
}
public class Car extends Vehicle {
  private FuelType fuelType;
  private double fuel;
  public void accelerate(double a) {
   super.accelerate(a);
   this.fuel -= a * fuelType.fuelConsumption;
  }
}
public class Bicycle extends Vehicle {}
```

This preserves the substitution principle!!!

Static and dynamic types

- Each expression has a
  - Static type determined at compile time
  - Dynamic type during execution
- The dynamic type can be a subtype of the static type
  - Exposing the same or more members
- Allowed to assign or pass a subtype
- Static strong typing
  - We know at compile time that the members we access exists at runtime

```
class Vehicle {…}
class Car extends Vehicle {…}
class Truck extends Car {…}
class Bicycle extends Vehicl

int race(Vehicle v1, Vehicle v2) {..

Car c1 = new Car(), c2 = new Car();
Truck t = new Truck();

Vehicle v = new Car();

race(c1, c2);
race(c1, t);
race(t, v);
```

static == dynamic type

dynamic subtype of static

Passed type is a subtype of the declared type

Types and the substitution principle

- If C1 (Car) extends C2 (Vehicle), C1 (Car) contains
  - All the members of C2 (Vehicle)
    - Potentially overridden (different behavior)
  - Additional members defined in C1 (Car)
- If we have an instance of C1 (Car)
  - We can access all the members of C2 (Vehicle)
  - We can substitute C2 (Vehicle) with C1 (Car)
- If we have an instance of C2 (Vehicle)
  - Additional member of C1 (Car) not defined!
  - We cannot substitute C1 (Car) with C2 (Vehicle)!

```
class Vehicle {…}
class Car extends Vehicle {…}
```

Car has all the members of Vehicle + something else

```
Vehicle v = new Car();
v.accelerate(…);
v.fullStop();
~~v.refuel(…);~~
```

If something is defined in Vehicle, it is in Car as well. refuel not part of Vehicle!

```
~~Car c = new Vehicle();~~
```

Forbidden by the compiler

```
c.refuel(…);
```

Not part of Vehicle

Type casting

- We can cast an expression to a subtype of its static type
  - (<type>) <expression>
- Useless to cast to a supertype
  - It is already a subtype, and we can already access the members of the supertype
- Forbidden to cast to a type not a subtype
- During the execution, if the dynamic type is not compatible an error is raised

```
Vehicle v = new Car();
v.accelerate(…);
v.fullStop();
Car c = (Car) v;
```

Casting v to Car

```
c.refuel(…);
Car c1 = new Car();
((Vehicle) c1).accelerate(…);
```

Useless! Allowed but warning

```
~~Bicycle b = (Bicycle) c1;~~
```

Forbidden! A car cannot be a bicycle

```
Bicycle b1 = (Bicycle) v;
```

Allowed but it crashes at runtime

instanceof

- How can we check dynamic types?
  - <expr> **instanceof** <type>
  - Returns true if and only if the dynamic type of the given expression is a [sub]type of the given type
- Useless to check a supertype
- Forbidden to check a type that is not a subtype

```
Vehicle v = null;
if(Math.random()>0.5)
  v = new Car();
else v = new Bicycle();
((Car) v).refuel(…);
```

It might crash

```
if(v instanceof Car)
  ((Car) v).refuel(…);
```

Safe, it will never crash

```
Car c = new Car();
if(c instanceof Vehicle)
  c.fullStop();
```

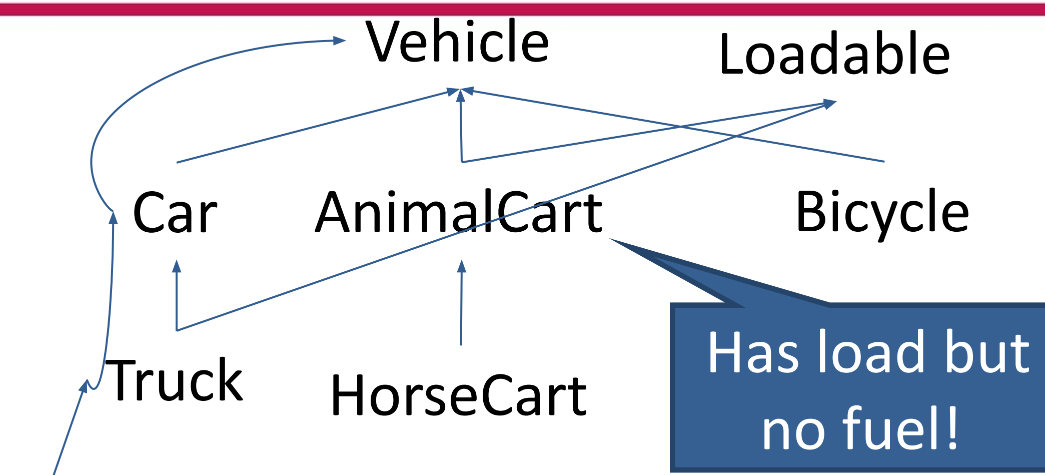
Useless, warning, always true!

```
if(c instanceof Bicycle)
  c.fullStop();
```

Useless, forbidden, always false!

Limits of extending classes

- Each class can extend one other class
  - Single inheritance
- Subtyping forms a tree
- Easy to detect what we are accessing
  - Univocally identified traversing the type tree
- We cannot mix together the implementation of different entities
  - Limit of single inheritance
- How could we have something that is subtypes of various non-related types?



Has load but no fuel!

```
Truck t = new Truck();
t.fullStop();
```

```
public class HorseCart extends Vehicle {
 public void chargeLoad(double l) {…}
}
```

Same of Truck!

No solution with classes!

- Implement a Loadable class
  - And then other methods rely on it
- Truck and HorseCart extends it
  - But they already extend another class
    - Car and Vehicle, respectively
- What about extending only Loadable?
  - Then how would they have a speed?
  - And the fuel of the truck?
  - We would just move the problem from Loadable to Vehicle/Car
- No solution with single inheritance!

```
class Vehicle {…}
class Car extends Vehicle {…}
class Loadable {
 private double load;
  public void chargeLoad(double l) {
    load += l;
  }
}
class Truck extends Car, Loadable {…}
class HorseCart extends Vehicle, Loadable {…}

void splitLoad(double load, Loadable[] v) {
  for(int i = 0; i < v.length; i++)
    v[i].chargeLoad(load/v.length);
}
```

Forbidden: each class can extend at most one other class!

Interfaces

- New concept: interfaces
  - Define only method signatures
    - No fields or implementations!
  - An interface define a type, like classes
- implements clause in class definition
  - Need to implement ALL methods
- In addition to extends clause
- Thus, each class can
  - Extend at most one other class
  - Implement many interfaces

```
interface Loadable {
 public void chargeLoad(double l);
}
class Truck extends Car
             implements Loadable {
 private double load;
 public void chargeLoad(double l) {
  load += l;
 }
}


void splitLoad(double load, Loadable[] v) {
 for(int i = 0; i < v.length; i++)
  v[i].chargeLoad(load/v.length);
}
```

Implementing interfaces

- Different implementations
  - Interface defines only the signature
  - Like abstract classes
- Interfaces documented like classes
  - Javadoc for interface and method
- Part of the capsule we deliver
  - Encapsulation exposes only signatures and documentation
- Duplicate code if we have the same implementation in several classes?
  - We can do better…

```
class HorseCart extends Vehicle
                    implements Loadable {
 private double load;
 final private double maxLoad;
 public void chargeLoad(double l) {
  if(l <= maxLoad)
       load += l;
  else System.out.println("Too much weight!");
 }
}

void splitLoad(double load, Loadable[] v) {
 for(int i = 0; i < v.length; i++)
  v[i].chargeLoad(load/v.length);
}
```

Default implementations (Java 8+)

- Java 8 added default implementations
  - **Limited** support for multiple inheritance
- Tag interface method as default
  - Then you can provide its implementation
- Interfaces can have fields
  - Only static, public and final
    - Constants, kind of useless ☺
- Interfaces can implement methods
- Default implementation can only rely on other declared methods
  - That are public!

```
interface Loadable {
 double getLoad();
 void setLoad(double l);
 default public void chargeLoad(double l) {
  this.setLoad(this.getLoad()+l);
 }
}

class Truck extends Car
            implements Loadable {}

class HorseCart extends Vehicle
            implements Loadable {}
```

Need to implement getLoad and setLoad, duplicating code!

Implementing multiple interfaces

- A class can implement many interfaces
  - Implement all the methods of all interfaces
- Support multiple subtyping
  - But no real multiple inheritance
  - Nothing is inherited!
- In this way, the type defined by a class is
  - Subtype of the superclass it extends
  - Subtype of all the interfaces it implements
- Build up complex type hierarchies
  - Direct acyclic graphs, not only trees!

```
interface Loadable {
 public void chargeLoad(double l);
}
interface Printable {
 public void print();
}
class Truck extends Car
        implements Loadable, Printable {
 private double load;
 public void chargeLoad(double l) {
  load += l;
 }
 public void print() {
   System.out.print(“Truck: “+load+“kg“);
 }
}
```

Ca' Foscari University of Venice

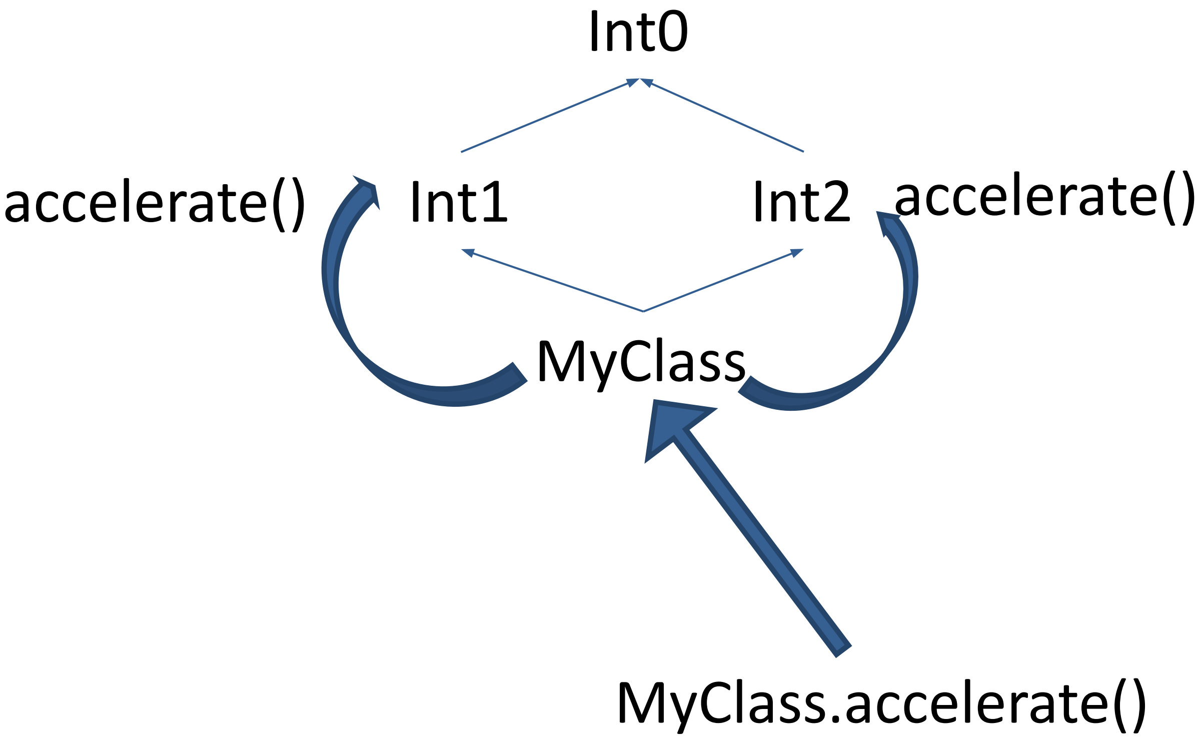
Implementing multiple interfaces with default implementations

- But then we might inherit the same default method multiple times!
- Forbidden by the Java compiler
  - When compiling the program, we know all the implemented interfaces
  - For each interface, we know its default methods
  - So we can check if a method is inherited twice
- Not a great support for multiple inheritance
  - Maybe a first step in the right direction?
  - Or maybe a step back from the wrong direction?
  - Indeed, does it exist a right direction?

```
interface FirstInterface {
   default void chargeLoad(double d) {…}
}

interface SecondInterface {
   default void chargeLoad(double d) {…}
}

class OnlyClass implements
         FirstInterface, SecondInterface {…}
```

class OnlyClass inherits unrelated defaults for chargeLoad() from types FirstInterface SecondInterface

Diamond problem



Extending interfaces

- An interface can extend another one
- Type hierarchy also among interfaces
  - Subinterfaces guarantee a wider interface
  - Substitute Loadable with LoadableUnloadable
- Possible to extend multiple interfaces

```
void splitLoad(double load, Loadable[] v) {…}
void splitAndHalfLoad(double load, LoadableUnloadable[] v) {
 splitLoad(load, v);
  for(int i = 0; i < v.length; i++)
   v[i].unchargeLoad(load/(2*v.length));
}
```

```
interface Loadable {
 public void chargeLoad(double l);
}
interface LoadableUnloadable
   extends Loadable {
 public void unchargeLoad(double l);
}
class Truck extends Car
      implements LoadableUnloadable {
~~private double load;~~
 public void chargeLoad(double l) {
   load += l;
 }
 public void unchargeLoad(double l) {
   load -= l;
 }
}
```

Abstract classes or interfaces?

- Pros of abstract classes
  - Abstract classes have a state
  - We can implement some methods
- Cons of abstract classes
  - We can extend at most one class
- Cons of interfaces
  - Interfaces cannot have a state
  - We cannot implement some methods
    - Except default implementations that have some limits
- Pros of interfaces
  - We can extend/implement multiple interfaces

```
interface Loadable {
 public void c  rgeLoad(double l);
}
```

We might have completely different vehicles that can load stuff.
And not only vehicles…

So many different classes, in totally different places in the type hierarchy, might be Loadable

Materials

- Lecture notes: Chapter 8 (subtyping) and 9 (interfaces)
- Arnold et al.: 3.4, 3.11, 3.12, chapter 4 (interfaces)
- Default implementations: https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html