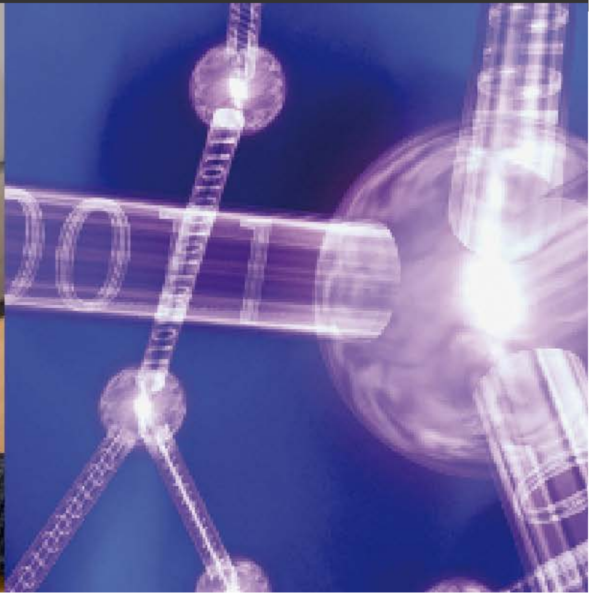
MONASH University

Information Technology

Module 7:
Inheritance and Polymorphism

FIT2034 Computer Programming 2
Faculty of Information Technology



Inheritance

Structural Hierarchies (Non-programming)

- Consider a law firm’s employees
  - Lawyers
  - Secretaries
  - Legal Secretaries
  - Marketer
- Whenever a new worker arrives at the company, they undergo an orientation/induction session
  - They are all provided with general information manual
- Depending on the department they will work for, they get a specialist induction about their job

Structural Hierarchies – Non-programming

- To help them remember things, we give everyone a booklet.
  - 20 pages explains the basic common information for everyone
    - Employees work 40 hours per week
- Their department will give them another few pages of information specific to the department.
  - Better than maintaining 4 separate documents that have significant commonality
  - These departmental pages may overrule some provisions in the 20 page document, for example:
    - Lawyers are to use the ‘pink’ form instead of the ‘yellow’ form to apply for holidays.
    - Marketing department employees get paid more money.
  - The people’s jobs mean that they can do different actions to other people.

Modeling in Code – Employee class

Assume the following class:

```
public class Employee {
        public void applyForVacation() {
                System.out.println("Use the yellow form");
        }
        public void showHours() {
                System.out.println("I work 40 hours per week");
        }
        public void showSalary() {
                System.out.println("My Salary is $40,000");
        }
        public void showVacation() {
                System.out.println("I receive 2 weeks vacation");
        }
}
```

How should we represent a secretary as an employee that has the additional ability to ‘take dictation’?

Secretary – “Bad” Implementation

```
public class Secretary {
        public void applyForVacation() {
                System.out.println("Use the yellow form");
        }
        public void showHours() {
                System.out.println("I work 40 hours per week");
        }
        public void showSalary() {
                System.out.println("My Salary is $40,000");
        }
        public void showVacation() {
                System.out.println("I receive 2 weeks vacation");
        }

        public void takeDictation() {
                System.out.println("I know how to take dictation");
        }
}
```

Why might this be bad?

What happens when we want to write a Lawyer class, where lawyers have an additional ability to sue people?

Inheritance

- **Inheritance** is a technique in programming where a derived class extends the functionality of a base class, inheriting all of its fields and methods
  - So by default it is capable of the same states and behaviors
- **Superclass** – the parent class, the basis class
- **Subclass** – the child class, the newly formed class

- A Subclass usually adds new features or functionality that was absent in the superclass

Secretary – Preferred Implementation

The keyword extends in Java, means a new class is being defined by using some other class as the starting basis

We must state which class is to be used as the basis class

```
public class Secretary extends Employee
{
        public void takeDictation() {
                System.out.println("I know how to take dictation");
        }
}
```

We provide only the new behaviors that are special or specific to this class

Example Main Program – What happens here?

```
public class EmployeeMain {
    public static void main(String[] args) {
        System.out.println("Employee:");

        Employee emp1 = new Employee();
        emp1.applyForVacation();
        emp1.showHours();
        emp1.showSalary();
        emp1.showVacation();

        System.out.println("Secretary:");

        Secretary emp2 = new Secretary ();
        emp2.applyForVacation();
        emp2.showHours();
        emp2.showSalary();
        emp2.showVacation();
        emp2.takeDictation();
    }
}
```

We instantiate an Employee object, and use emp1 to refer to it.

The method code in the Employee class operates for the object that emp1 refers to

We instantiate a Secretary object, and use emp2 to refer to it.

The method code in the Employee class operates for the object that emp2 refers to, because of inheritance

The method code in the Secretary class operates for the object that emp2 refers to

Further Example: Lawyers

- A lawyer is allowed to have an extra week of vacation
- A lawyer should use the pink paper form instead of yellow paper form to apply for their vacation
- An ability unique to lawyers is to sue people. (The secretary cannot sue people).

Can we still use inheritance?

Lawyer class

```
public class Lawyer extends Employee
{
```

We provide the new behaviors that are special or specific to this class:

```
        public void sue() {
                System.out.println("See you in court!!!");
        }
```

And can **re-define existing behaviors** that were inherited from the superclass:

```
        public void applyForVacation() {
                System.out.println("Use the pink form");
        }

        public void showVacation() {
                System.out.println("I receive 3 weeks vacation");
        }
}
```

Overriding Methods

- **Override** – to replace the inherited version of a method with a new implementation
- To override a method, <u>must</u> use same signature as original definition
  - We should also mark it using: @Override
    (explained later today)

This contrasts with:

- **Overloading** – means we define multiple methods with the same name, but *different* signatures, in the one class.

Generalisation Hierarchies

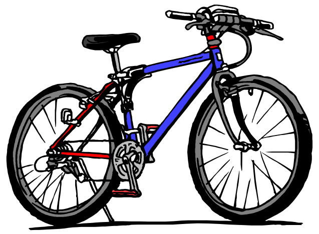
- Generalisation is another type of relationship in OO
- It arises where you have a sub-class of another class.
- We refer to sub-classes as children, descendants, of some class
- We refer to super-classes as parent, ancestors, of some class
- One class may be a sub-class of another class, but in turn could be the parent of other classes, leading to a hierarchy of relationships between classes
  - A class with no parents is a **base class**
  - A class with no children is a **leaf class**
- UML Notation for Generalisation: a line connecting two classes with a hollow triangle at the super-class end

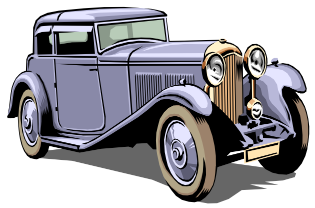
Substitutability

- Objects of the child class can be substituted for the parent but not the reverse

Generalisation – Categorising objects

- Sometimes a group of objects shares some common attributes and/or methods, but other attributes and/or methods are distinct
  - Bicycles and cars are both vehicles,
    both have some number of wheels
    and both can stop, start and turn
  - Cars have doors, windows and fuel tank, but bicycles don't
  - Bicycles have pedals and handlebars but cars don't
  - Cars are powered by fuel burning whereas bicycles are powered by pedalling





Generalisation vs Specialisation

- Forming a new class by considering commonalities amongst a set of classes, is called **Generalisation**
  - Move the common things out of the classes into the new class
  - Make the other classes now be sub-classes of the new class

- If we make a new sub-class from an existing class, (i.e. the opposite to generalisation) is **Specialisation**
  - Deciding what new methods are needed, which methods need to be overridden, to form a more refined class than the existing one
  - Deciding what new fields are required.

Identifying Potential Generalisations

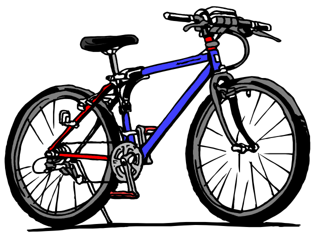
- Consider there are two classes Bicycle and Car
- Should the common members (attributes, methods) of different classes be duplicated in completely separate classes?

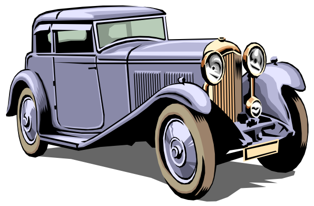
Bicycle
wheels **gears** handles pedals

start()
stop()
turn()
pedal()

Car
wheels **gears** windows doors tank

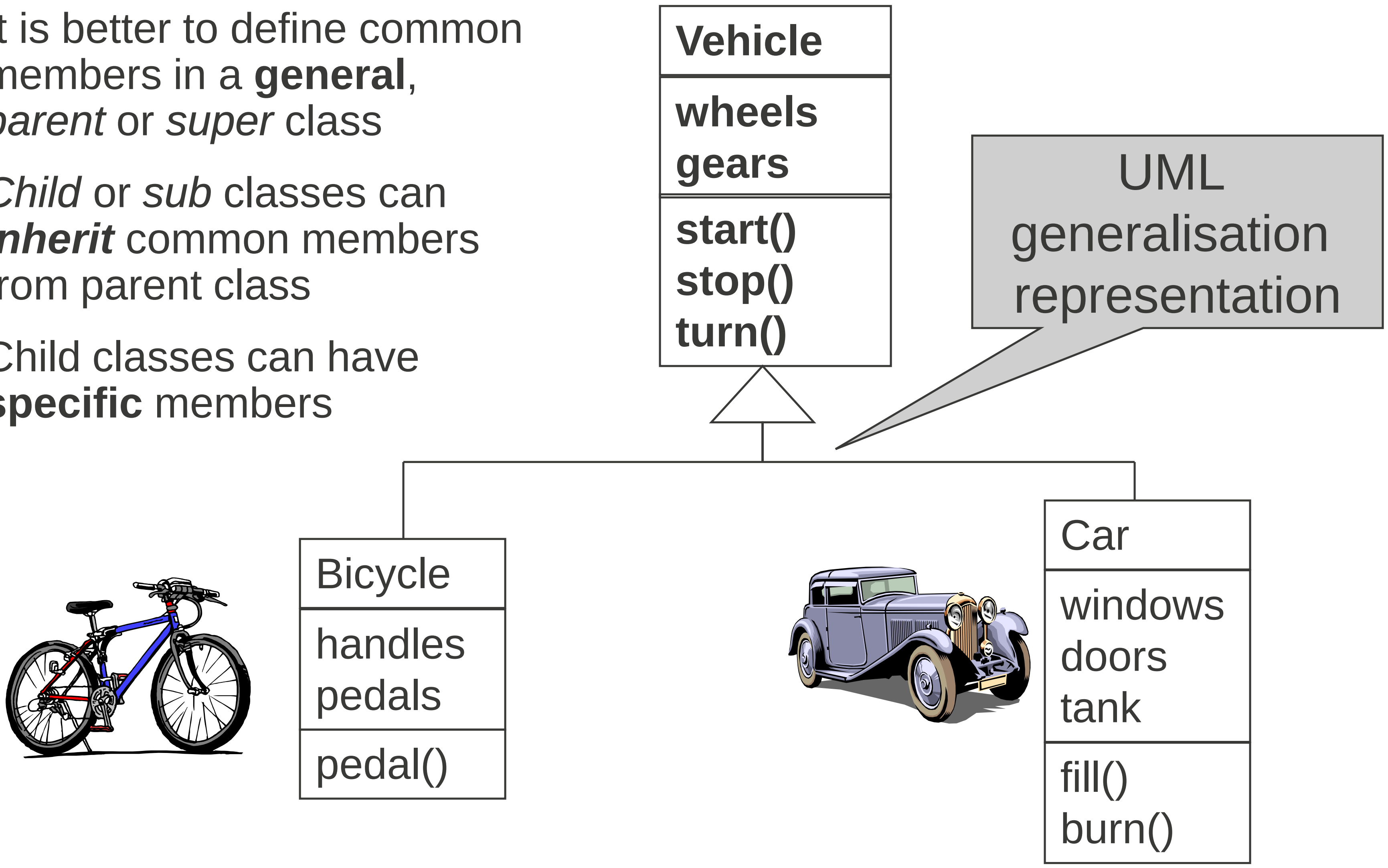
start()
stop()
turn()
burn()





Forming a class hierarchy

- It is better to define common members in a **general**, *parent* or *super* class
- *Child* or *sub* classes can ***inherit*** common members from parent class
- Child classes can have **specific** members



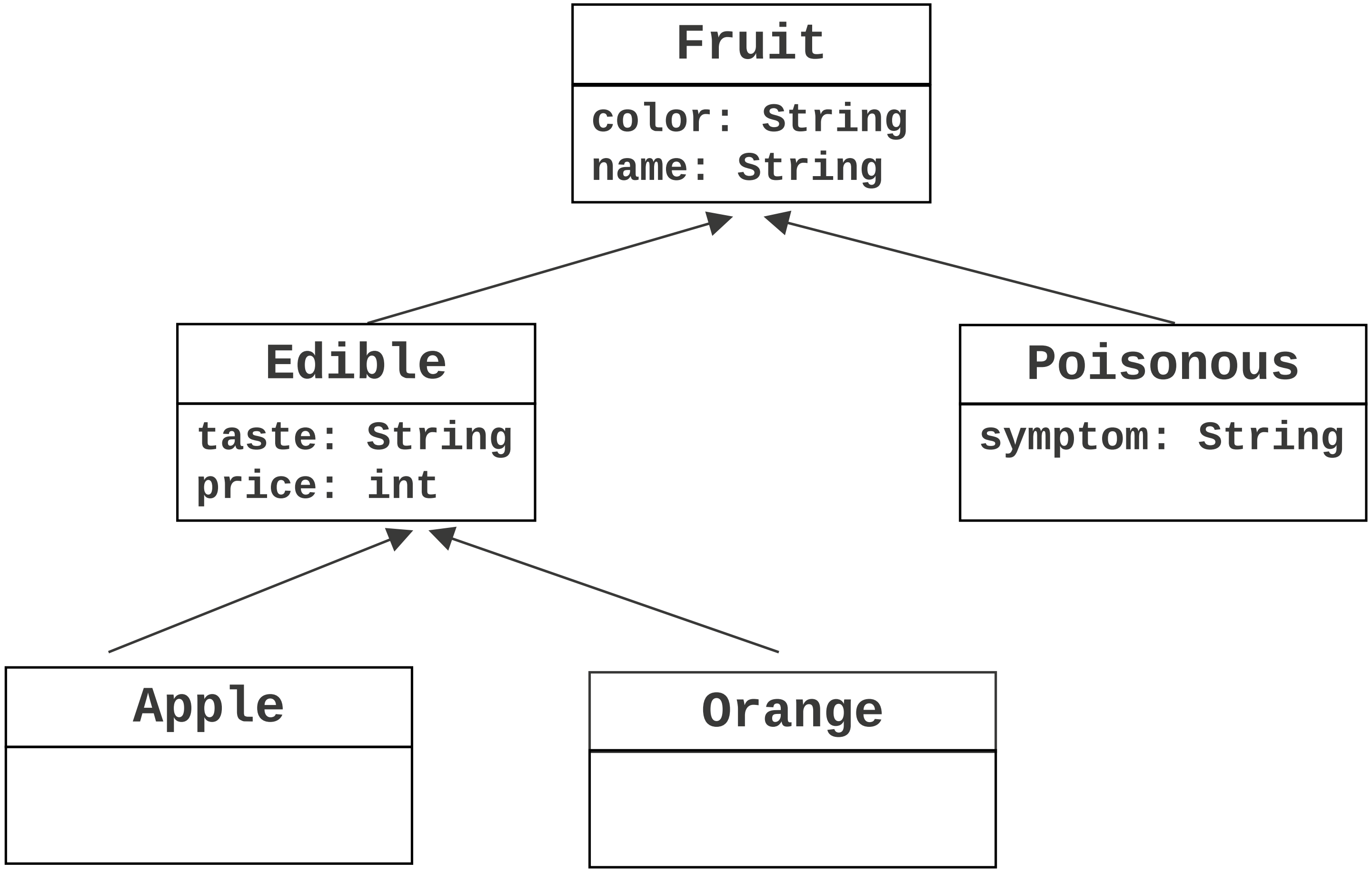
Basic Inheritance Hierarchies

- Inheritance Hierarchies are not restricted to a single level.
- They can be arbitrarily deep (and are transitive)
  - The Java API Classes sometimes are very deep in a hierarchy
- They cannot contain cycles
  - If C is a subclass of B, and B is a subclass of A, then we cannot make A be a subclass of C
  - Whereas, associations allowed cycles through bi-directional links.

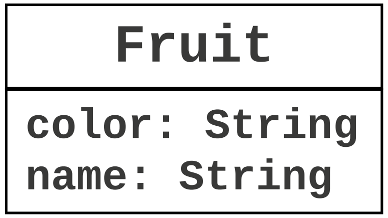
Multiple Inheritance

- In some situations, we might like a class to inherit from more than one other class.
  - For example, we may have a **Boat** class and a **Plane** class. Boat would have behaviours such as *dropAnchor*. Plane would have behaviours such as *land* and *takeOff*. They would have some behaviours in common, e.g. *goForward*. That might be in a superclass called **Vehicle**.
- A **seaplane** is a Plane, and it is also a Boat. It could inherit from both, so that it could do all of these things.
- Some programming languages allow for multiple inheritance. <u>Java does not</u>.
  - This is due to the complications involved in resolving potential *conflicts* in multiple inheritance (which version of ‘goForward’ would we inherit).

Basic Inheritance – Fruit Hierarchy

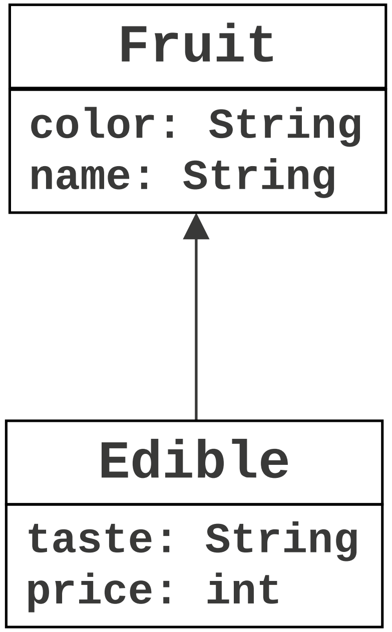


Basic Inheritance – Code for Fruit class



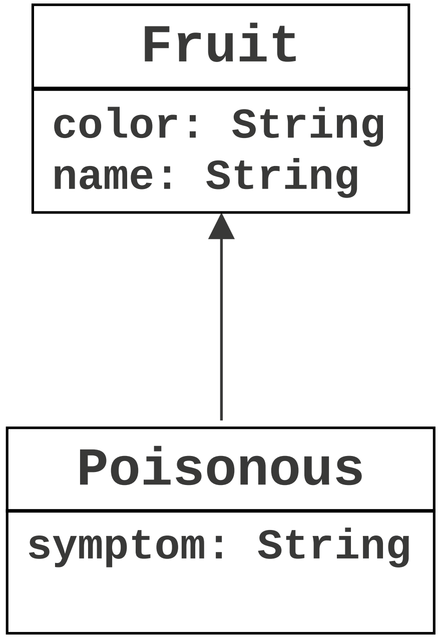
```
public class Fruit
{
        private String color;
        private String name;

        public String getName() {
            return name;
        }

        public void setName(String aName) {
            this.name = aName;
        }

        public String getColor() {
            return color;
        }

        public void setColor(String aColor) {
            this.color = aColor;
        }
}
```

Basic Inheritance – Sub class code



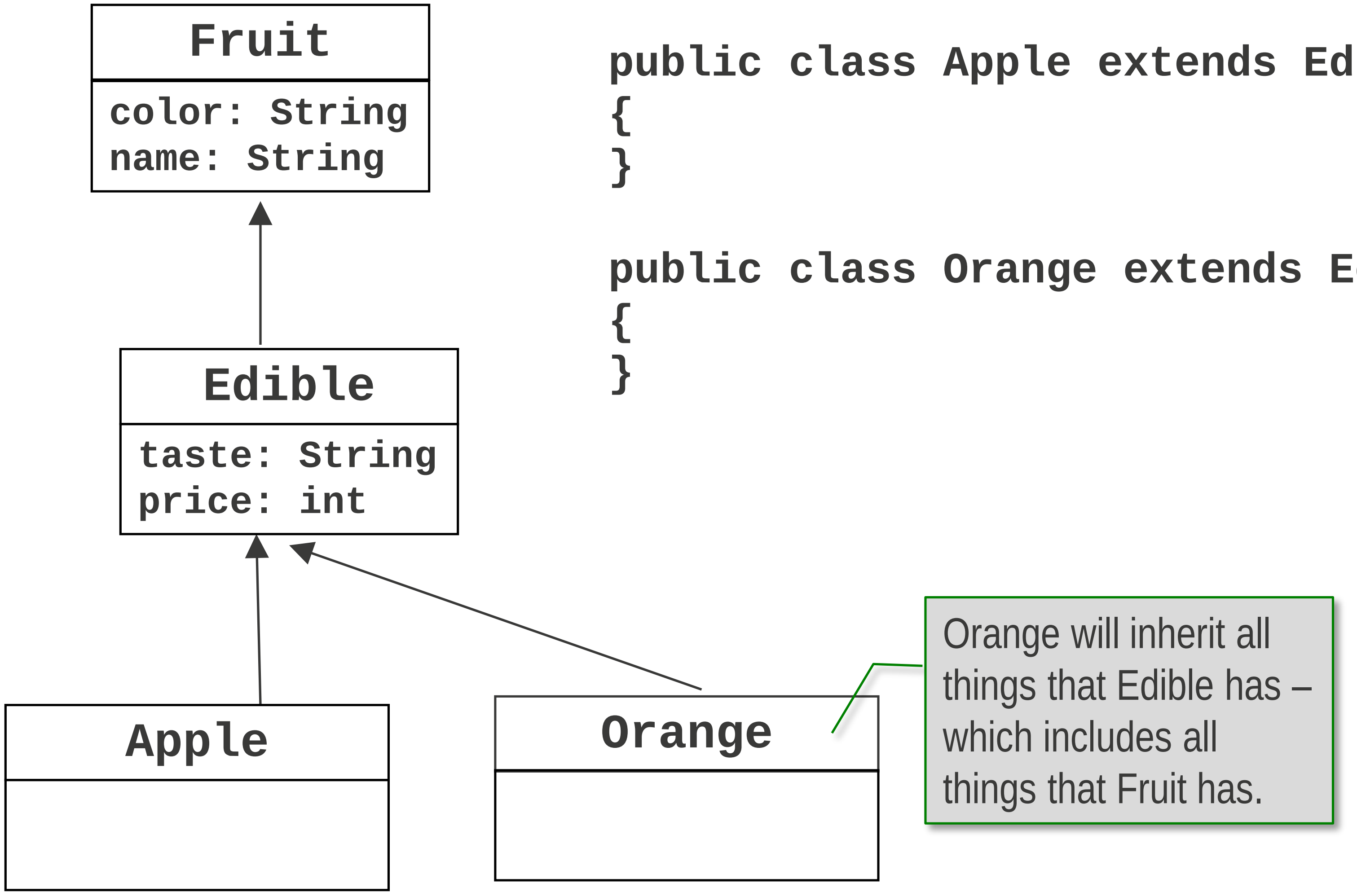
```
public class Edible extends Fruit {
        private String taste;
        private int price;

        public String getTaste() {
            return taste;
        }

        public void setTaste(String aTaste) {
            this.taste = aTaste;
        }

        public int getPrince() {
            return price;
        }

        public void setPrice(int aPrice) {
            this.price = aPrice;
        }
}
```

Basic Inheritance – Sub class code



```
public class Poisonous extends Fruit
{
        private String symptom;

        public String getSymptom() {
            return symptom;
        }

        public void setSymptom(
                     String aSymptom)
       {
            this.symptom=aSymptom;
        }

}
```

Basic Inheritance – Apple & Orange


Fruit
color: String
name: String
Edible
taste: String
price: int
Apple
Orange
Orange will inherit all things that Edible has – which includes all things that Fruit has.

```
public class Apple extends Edible
{
}

public class Orange extends Edible
{
}
```

Inheritance and Private Members

- Any members (i.e. fields or methods) which are declared as "private" are **not** directly accessible in a sub-class
  - The object still has these members, but our code in the sub-class cannot access them by using their name – because they are private!

- If we want to be able to directly access the inherited private instance variables, we would need to declare them public:

```
public class Edible extends Fruit
{
        public String taste;
        public float price;
        …
}
```

- But that makes it public to *all* classes, which we don't want.

Inheritance and Private Members (2)

- If we used public for the instance variables, we would have broken the rule of encapsulation
- A better way is to define proper set/get methods (as before).
- We can then access these instance variables with the get/set methods in the subclasses, even though we cannot access them directly as an instance variable:

```
public class Fruit
{
   private String color;
   private String name;

   public String getName() {
      return name;
   }
```

```
public void setName(String aName)
{
   this.name = aName;
}
…
```

Inheritance and Private

- Example: The following access attempt will now work and is safer than making instance variables public:

```
Apple a = new Apple();

a.setTaste("Delicious");
System.out.println(a.getTaste());
```

- However, because the methods are public, it still means that *all* classes can call the set/get methods, which may not be desired
  - We want just the subclasses to be able to call the ‘set’ methods

Inheritance and Protected

- Java provides the ‘**Protected**’ accessibility modifier for the case where we want a class’ members to be accessible only to subclasses, but not to classes outside the hierarchy

```
public class Edible extends Fruit
{
        protected String taste;
        protected float price;
}

public class Apple extends Edible
{
        public Apple()
        {
                taste = “Yummy";
                price = 2.25f;
        }
}
```

Now the subclass can directly access the taste and price, as though declared inside the Apple class.
Other classes (like Student) cannot access them.

Inheritance and Protected

- In FIT2034, we want you to avoid using protected accessibility *for attributes* (instance variables)
- Instead, you should still make attributes private, but define mutators and accessors that are either public or protected
  - Preserves any checking that mutators are performing – the subclass won't be able to bypass the checks.
  - It is quite alright for mutators and accessors to be 'protected'
  - Mutators should be protected if you want subclasses to be able to call it, but not other classes outside of the hierarchy defining the class
  - Mutators should be public if you want any class to be able to call it
    - Same rules for accessors.

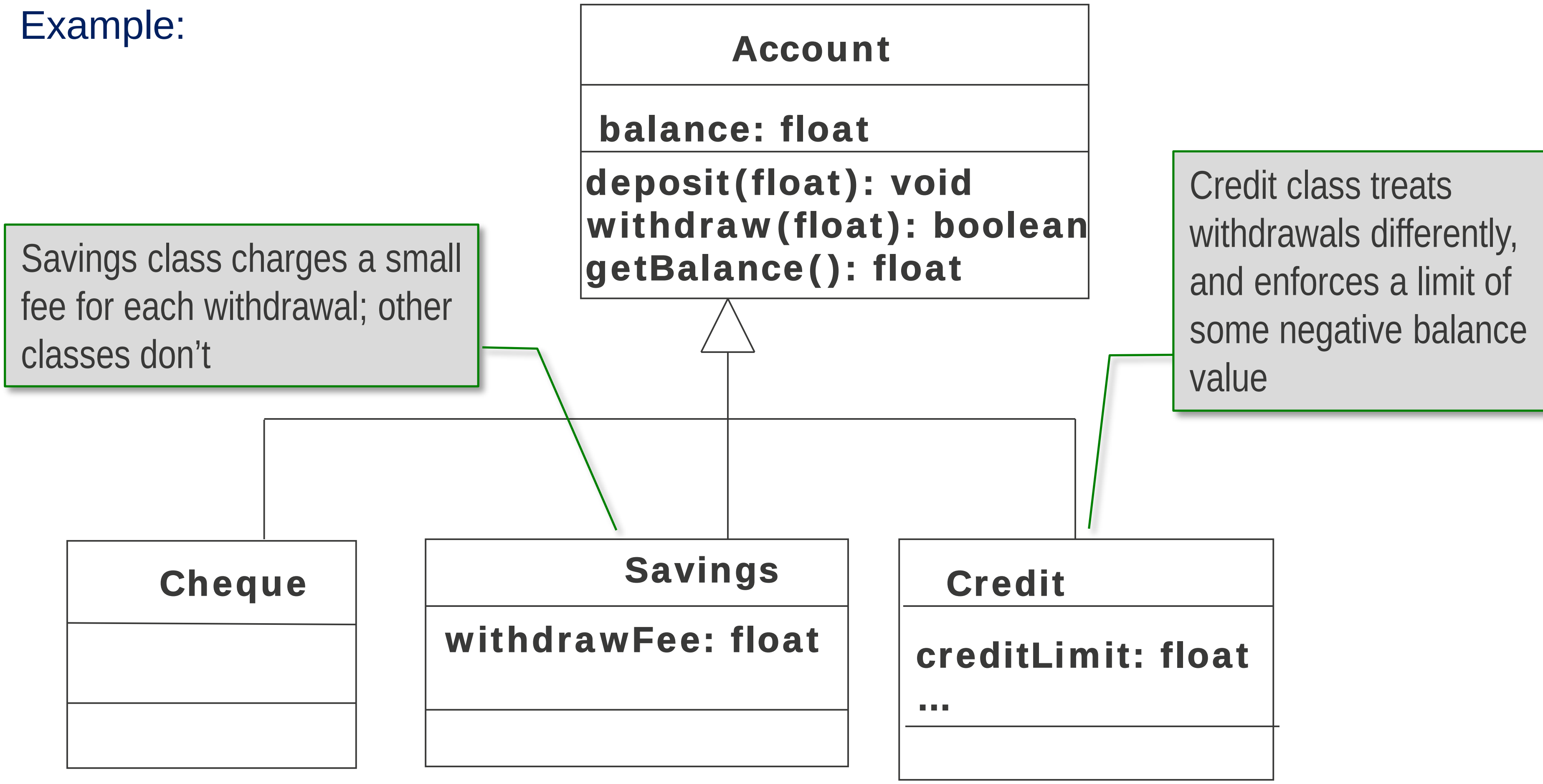
Overriding Methods

Overriding a Method

- Method definitions are inherited from superclasses in the same way as attributes.
- However, Java allows us to re-define the behaviour of inherited methods
  - Unless the method was declared using the keyword: **`final`** placed before the return-type
- It also can allow us to re-define the return type in some cases

Overriding a Method

Example:


Account
balance: float
deposit(float): void
withdraw(float): boolean
getBalance(): float
Savings class charges a small fee for each withdrawal; other classes don't
Credit class treats withdrawals differently, and enforces a limit of some negative balance value
Cheque
Savings
withdrawFee: float
Credit
creditLimit: float
…

Overriding a Method – Parent definition

```
public class Account  {               // The parent class
    float balance = 0.0f;
    …
    public boolean withdraw(float amount)   // Original
    {
             if ( balance - amount >= 0.0f )
             {
                    balance -= amount;
                    return true;
             }
             else return false;
    }
    …
```

Overriding a Method in the Subclass

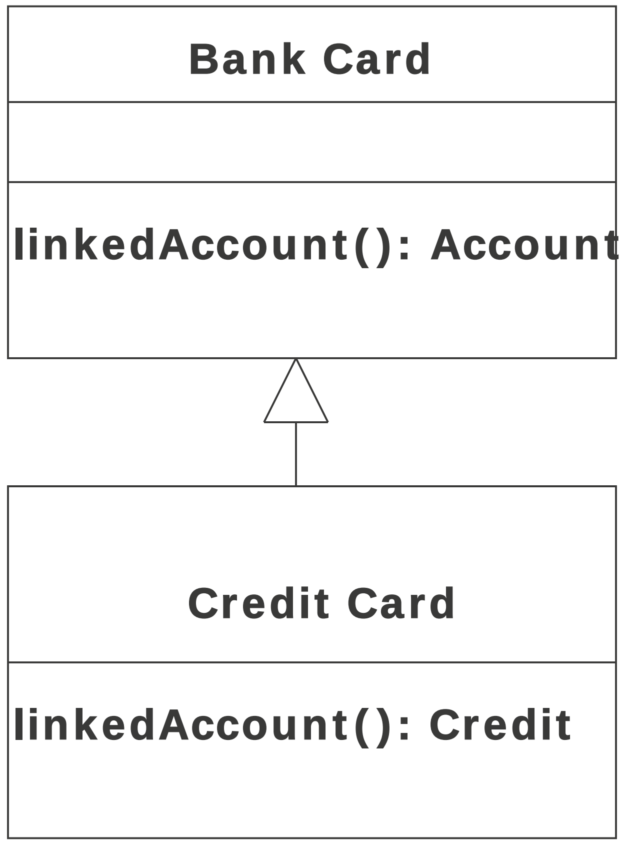
```
public class Savings extends Account {
    private float bankFee;
    …
    // Replacement method - in subclass
    public boolean withdraw (float amount)
    {
        if (balance - amount - bankFee >= 0.0f)
        {
            balance = balance - amount - bankFee;
            return true;
        }
        return false;
     }
   …
```

Method Signature and Overriding

- To override a method it must have exactly the same signature:
  - same name
  - same parameters (and parameter types) in same order
- it must also have the same return type
  - The only exception to this rule (from Java 5 onwards only) is that the return-type of an overriding method in a subclass may be **more specific** than the return type given in the superclass.
    - For example, a method in some class that expects to return a Fruit object, can be redefined in a subclass to return a descendant class of Fruit
- Otherwise, if there are different parameters, it is method overloading.

Method Signature and Overriding

Example: A general Bank Card can be linked to any account, but a Credit Card only to a Credit Account.



Calling (Invoking) a Super Class Method

- Overriding doesn't totally replace an inherited method. It merely *shadows* or hides it from view.
  - We can still invoke the original method as it was defined in the super-class from code written in the sub-class.
- The syntax for this is:

```
super.methodname(...parameters...)
```

- This is often used when we wish to "extend" the functionality of the super-class method.
  - Because it is sometimes necessary to maintain the semantics of the parent class method.
  - For example, things were done by the original method which still need to be done, and the "extra" things go in the subclass' version.

Calling (Invoking) a Super Class Method

- Example: withdrawing an amount *x* from a Savings Account is the same as withdrawing (*x*+*bank fees*) from a generic account:

```
public class Savings extends Account
{
    private float bankFee;
    ...
    public boolean withdraw (float amount)
    {
       boolean result;
       result = super.withdraw(amount+bankFee);
       return result;
    }
    ...
}
```

Causes the version of withdraw as it exists in the superclass (Account) to be invoked.

“Final” Methods

- Normally we can override inherited methods
- Sometimes you want to prevent a method from ever being overridden (e.g. for security reasons it is not desirable)
- By declaring the method as **final** the compiler knows you cannot re-define it in any subclass.
- For example, the following will prevent the getTaste method from being overridden by Apple or Orange:

```
public class Edible extends Fruit {
    public final String getTaste()
    {
          return taste;
    }
}
```

“Final” Classes

- We can designate a class as not allowed to be subclassed by declaring the class as final:

```
public final class Apple {
    ...
}
```

- Now nobody can extend the Apple class.

- The String class in the Java API is an example of a class that has been declared as Final
  - So you cannot make subclasses of String

@Override

- This is one type of **annotation** that Java provides (since Java 5)
  - An Annotation is a special marker that can be attached to code for various tools to use. In this case, the compiler is the tool.
- Include it in the method header line of the replacement version, e.g.:

```
@Override public boolean withdraw (float amount)
```

- Its purpose is to provide us with some basic level of checking when compiling:
  - If we mark a method as @Override when it is not actually overriding or not allowed to be overriding, the compiler will tell you.
  - Without it, you could mistakenly believe you were overriding a method when you might have made an error with the signature and had instead overloaded.

Constructors and Inheritance

Overriding an Initialization Value

- If an inherited instance variable needs to have different default initialization values in different subclasses, it must be initialized in a constructor.
- Note: to be on the safe side, all initializations should be performed in constructors.

```
public class Apple extends Edible
{
    public Apple(int aPrice) {
          setColor("red");
          setName("Red Delicious");
          setTaste("sweet");
          setPrice(aPrice);
    }
}
```

Constructors when inheritance is used

- Constructors are not inherited
  - (this is obvious when you think about the fact that they must have different names and create objects of different classes).
- However, when you construct an object, it must complete all its ancestors’ constructors.
  - Instantiating an Apple will cause Edible’s constructor to occur because an Apple **is an** Edible.
  - This in turn causes Fruit’s constructor to occur, because an Edible **is a** Fruit.
- By default, Java will assume you want the superclass’ zero-parameter constructor to be done
  - If you haven’t written any constructor, Java provides one by default
  - If you have written a constructor, but not one with zero-parameters, the compiler will complain

Constructors and Inheritance

```
public class Person {
    private int age;
    private String name;

   public Person() {
   }

   public Person( int anAge ) {
       if (anAge>0) this.age = anAge;
       else throw new Error(“Invalid Age”);
   }
}
public class Student extends Person {
   private int studentID;
}
```

A programmer-provided zero-parameter constructor, (which does nothing in its body).

Since no programmer-provided constructors are written here, Java provides a default zero-parameter constructor for us.

It will invoke the inherited zero-parameter constructor of Person.

Constructors and Inheritance

- Sub-class constructors implicitly call the zero-parameter super-class constructor first

```
public class Person {
   private int age;
   private String name;

   public Person() {
      this.name = "John Doe";
   }
   ...
}

public class Student extends Person {
   private int studentID;
   public Student() { }
}
```

The code:

```
new Student()
```

generates a Student object with name set to “John Doe”

Constructors and Inheritance

- Note: if you want to use a default constructor in some sub-class, its immediate super-class must have a zero-parameter constructor because `super()` is implicitly called by the default constructor Java will provide when we don’t write one in the sub-class.

- Therefore you should always include a zero-parameter constructor if you plan to use inheritance.

Using a Specific Super-Class Constructor

- It is possible for us to specify a particular constructor of the super-class to be performed
  - E.g. when the super-class has only constructors with parameters we must do this
  - E.g. if we don't want to write zero-parameter constructors.
- To choose which constructor to use, we must use:

  `super(` *`parameters`* **`)`**

  as the **first statement** in the sub-class constructor
  - Otherwise the compiler will try to use the zero-parameter constructor of the superclass

Using a Specific Super-Class Constructor

```
public class Person {
   private int age;
   private String name;

   public Person( int anAge, String aName ) {
       this.age = anAge;
       this.name = aName;
   }
}

public class Student extends Person {
   private int studentID;

   public Student(int anAge, String aName, int anID) {
      super(anAge, aName); // call to Person constructor that takes 2 parameters
      studentID = anID;
   }
}
```

Polymorphism
and
Static & Dynamic Binding

Polymorphism

- Inheritance in a class hierarchy, affords polymorphic code
- **Poly** = Many, **Morph** = form
- Since an Apple is an Edible, the following is valid in Java:

```
Edible myEdibleFruit = new Apple();
```

- Different classes are named on each side of the assignment
  - But the Apple is a descendant of Edible
- Later we could change the value of the variable:

```
myEdibleFruit = new Watermelon();
```

Polymorphic Variables

- A variable could be declared as being of a class type for which there exists subclasses.
- An object of any subclass type can be assigned to that variable – making a ***polymorphic variable***
- In the previous slide, **`myEdibleFruit`** is a polymorphic variable.
- A polymorphic variable is limited in what messages can be sent to it
  - Only those messages defined by the variable’s declared type can be sent (e.g. only those things that an Edible knows how to do)

Casting Object

- A class specifies the type for objects, so you would expect to be able to use type casts. This is possible in two directions.
- **Upcasting** – Changing the type to a superclass/ancestor

```
Savings s = new Savings();
Account a = (Account) s;       // upcast
```

- Upcasting is safe to do

Casting Object

- A class specifies the type for objects, so you would expect to be able to use type casts. This is possible in two directions.
- **Downcasting** – Changing the type to a subclass

```
Savings s = new Savings();
Account a = (Account) s;      // upcast
Savings s2 = (Savings) a;     // downcast
```

- Downcasting is dangerous – unless the object was originally created as an object of the target class it may not have all instance variables and may not know how to respond to some messages.

instanceof

- Before you try to do a downcast, you should first check whether the object is capable of being cast to that more specific class type.
- This can be achieved using the special `instanceof` operator, as in the following examples:

```
Savings s1 = new Savings();
Account a1 = new Savings();
Account a2 = new Cheque();
if (a1 instanceof Savings) {          ➔ true
      Savings s2 = (Savings) a1;
}
```

- Also, given the above assignments:

```
(a2 instanceof Savings)      ➔ false
(s1 instanceof Savings)      ➔ true
(a2 instanceof Account)      ➔ true
```

Polymorphism and Sending Messages

- Consider the following code and then the issue on the next slide…

```
public class Color {
    public String test() {
        return "I'm a generic color";
    }
}

public class Blue extends Color {
    public String test() {
        return "I'm blue";
    }
}

public class Red extends Color {
    public String test() {
        return "I'm red";
    }
}
```

Polymorphism and Sending Messages

- If a message is sent to a polymorphic variable, and the object it refers to is of a subclass which has redefined the behaviour of the method, which method will be performed?
  - In particular, if test() is called with 0, what will it print?
  - What will it print if test() is called with 1 as the parameter?

```
public class Test
{
      public void test(int which) {
          Color c = null;

          if (which==0) c = new Blue();
          else c = new Red();

          System.out.println( c.test() );
      }
}
```

Polymorphism and Sending Messages

- c is a **Color** object, but (depending on input) is actually instantiated as either a **Blue** or a **Red** object.
- Each of these 3 classes implements the test method uniquely.
  - So which one is done, and why?

```
public class Test
{
        public void test(int which) {
            Color c = null;

            if (which==0) c = new Blue();
            else c = new Red();

            System.out.println( c.test() );
        }
}
```

Polymorphism and Sending Messages

- When c.test() is reached, the computer first looks at the object which c is referring to *at that moment in execution*.
  - For value 0 for the ‘which’ parameter the code below will print “I’m blue”
  - For any other value for the ‘which’ parameter, it prints “I’m red”.

```
public class Test
{
        public void test(int which) {
            Color c = null;

            if (which==0) c = new Blue();
            else c = new Red();

            System.out.println( c.test() );
        }
}
```

Late Binding vs Early Binding

- There is no way how Java could have decided at compile time which method implementation to use.
  - It could have been Red or it could have been Blue, based on user's choice
  - Could possibly even have been null in some programs.
- The method code to run has to be decided at run time.
- This mechanism is called "**late binding**" or "**dynamic binding"** and is used to achieve **method polymorphism** (method overriding).
- This binding decision contrasts to the decision made between *overloaded* methods (methods with the same name but *different* parameter signatures), which can always be decided at compile time – and is called "**early binding**".

Dynamic Binding – Another Example

```
Vehicle vehicle;                               // Polymorphic Variable
Car car = new Car(4,5,6,3);
Bicycle bike = new Bicycle();
 ……
 if (!fineWeather) {  // if weather not fine, take the car
    vehicle = car;         // Refer to the car object
    …………
 }
 else {                  // else ride a bicycle
    vehicle = bike;         // Refer to the bike object
    …………
 }
 if (road goes to the left)
    vehicle.turn(LEFT);   // Dynamic binding occurs now
```

- Selection of the correct <u>turn</u> method will be made at run time – i.e. depending on the weather!

Example of Early Binding

- In this example, the decide() method is overloaded. Since we invoke decide using c, **whose type is Color**, and there is a version of decide that wants to receive a Color reference, it is therefore chosen.
  - The test() method will always return “I am a generic color”

```
public class DifferentTest {
      public void test(int which) {
          Color c = null;
          if (which == 0) c = new Blue();
          else c = new Red();
          System.out.println( decide(c) );
      }

      public String decide(Color c) {
        return "I am a generic color";
      }

      public String decide(Blue c) { return "I am blue"; }
      public String decide(Red c) { return "I am red"; }
}
```

Early Binding

- If the compiler can know definitely which method is to be invoked, then it will choose (bind) the exact method at compilation time – called ***early*** or ***static binding***
- Java uses early binding in these cases:
  - For cases like on previous slide – where the reference's declared type matches the parameter's type
  - For calling of a method declared as **final** (because no subclass will be able to override it)
  - For calling of private method (because no subclass will be able to see it)
  - For static methods, i.e. class-level methods, since these do not require an object for invocation – using a class name before the dot operator.
    - Example from wrapper class **`Integer`**:
      `int number = Integer.parseInt(“123”);`

Some Special Classes

The ‘Object’ class

- All Java classes are descendants of the java.lang.Object class
  - We don’t have to say “extends Object” – it is automatically done

- This class defines a set of methods, which are inherited by all classes, some of which are useful:
  - public String **toString**() – returns a **String** representation of the object, suitable for displaying.
    By default, this method returns a string equal to the value of:

    ```
    getClass().getName()+'@'+Integer.toHexString(hashCode())
    ```

    * *Recall this string from when we tried to print an object of a class without providing our own toString() for that class (Week 4)*
  - You should ***override*** this with your own toString() method for your class to display useful information.

The ‘Object’ class (2)

- More methods:
  - public boolean **equals**(Object other) – allows you to check whether some other object is equivalent to the object on which you invoke the method.
    We can override this in our own class to compare something useful.
  - public Class **getClass**() – returns an object of type **Class** (next slide), which is an object that describes the actual type for the actual object on which the method is invoked.
  - protected Object **clone**() - creates and returns a copy of this object. Note that the object must be cloneable* in the first place.
    - More about the Cloneable interface next week
- See MoneyDemo program on Moodle for example of how you can override the equals() method to check for the equivalence of two objects.
  - In this case, equivalent money amounts

The ‘Class’ class

- The java.lang.Class class is used to represent information *about* the various class types used in a program.
- The most useful method is:
  - public String **getName**() – reports the fully-qualified name of the class which is being represented in the Class object.
- Example:

```
String welcome = "Hello";
System.out.println(welcome.getClass().getName() )
```

Displays:

```
java.lang.String
```

Demo Programs

- ShapeDemo – demonstrates inheritance and polymorphism
- MoneyDemo – demonstrates .equals()