Inheritance and Polymorphism

Shing-Chi Cheung

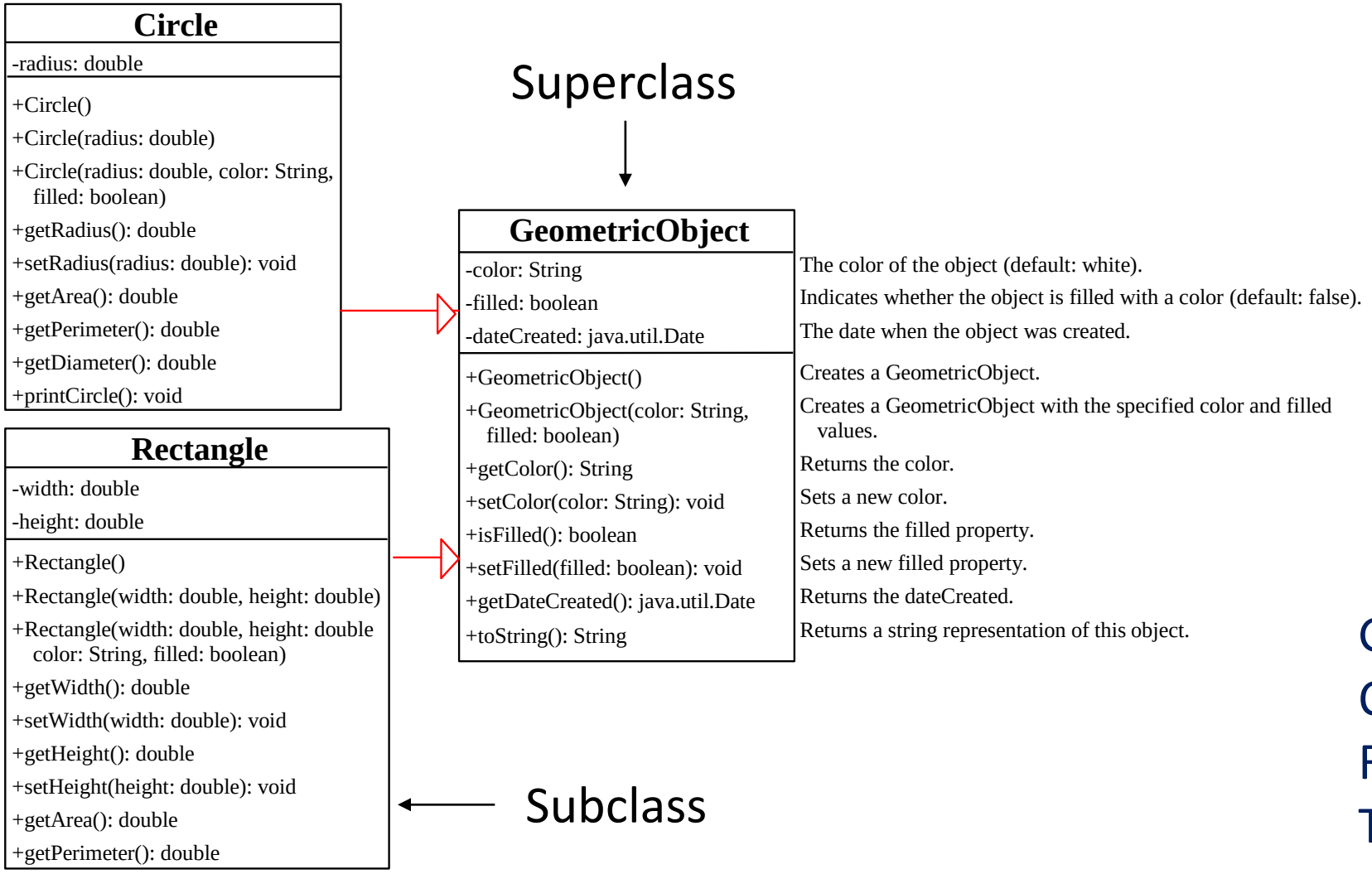
Computer Science and Engineering

HKUST

Motivations

- Suppose we want to define classes to model geometric shapes like circles, rectangles, and triangles.
- Classes of these shapes share many common features (i.e., common instance variables and methods)
  - instance variables: color, filled, etc.
  - instance methods: getColor(), setColor(), isFilled(), setFilled(), etc.
- Define a superclass GeometricObject for these common features.
- Allow Circle, Rectangle and Triangle to extend GeometricObject

Subclasses and Superclasses



GeometricObject.java
Circle.java
Rectangle.java
TestCircleRectangle.java

Are superclass constructors inherited?

- Unlike properties and methods, a superclass constructors are not inherited by subclasses.

- Two ways to invoke a superclass constructor.
  - Explicit invocation: Superclass constructor can be invoked from a subclass constructor using the keyword super.
  - Implicit invocation: If the keyword super is not explicitly used in a subclass constructor, its superclass' no-arg constructor is automatically invoked.

ConstructorInheritanceTest.java

Superclass constructor is always invoked

- A constructor may invoke another overloaded constructor using keyword this or its superclass's constructor using keyword super.
- If none of them is invoked explicitly, Java compiler implicitly adds super() as the first statement in the constructor. For example,

```
public A() {
}
```

is equivalent to

```
public A() {
  super();
}
```

```
public A(double d) {
  //some statements
}
```

is equivalent to

```
public A(double d) {
  super();
  //some statements
}
```

Using the keyword super

The keyword **super** keyword can be used in two ways:

- To call a superclass constructor
  - super(0); *// call the superclass constructor that accepts an integer*
- To call a superclass method or access a superclass field
  - super.getColor(); *// call the superclass getColor() method*
- Never call superclass' constructors or its instance methods by its class name, as this will cause compilation errors
  - GeometricObject.getColor(); *// compilation error*

```
public class Faculty extends Employee {
  public static void main(String[ ] args) {
    new Faculty();
  }
  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
}}
class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }
  public Employee(String s) {
    System.out.println(s);
}}
class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
}}
```

When a class' constructor is invoked, Java will invoke its superclass' constructor iteratively along the class' inheritance chain. This mechanism is known as Constructor Chaining.

Faculty.java

```
public class Faculty extends Employee {
  public static void main(String[ ] args) {
    new Faculty();
  }
  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
}}
class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }
  public Employee(String s) {
    System.out.println(s);
}}
class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
}}
```

```
public class Faculty extends Employee {
  public static void main(String[ ] args) {
    new Faculty();
  }
  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
}}
class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }
  public Employee(String s) {
    System.out.println(s);
}}
class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
}}
```

```
public class Faculty extends Employee {
  public static void main(String[ ] args) {
    new Faculty();
  }
  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
}}
class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }
  public Employee(String s) {
    System.out.println(s);
}}
class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
}}
```

```
public class Faculty extends Employee {
  public static void main(String[ ] args) {
    new Faculty();
  }
  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
}}
class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }
  public Employee(String s) {
    System.out.println(s);
}}
class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
}}
```

```
public class Faculty extends Employee {
  public static void main(String[ ] args) {
    new Faculty();
  }
  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
}}
class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }
  public Employee(String s) {
    System.out.println(s);
}}
class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
}}
```

```
public class Faculty extends Employee {
  public static void main(String[ ] args) {
    new Faculty();
  }
  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
}}
class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }
  public Employee(String s) {
    System.out.println(s);
}}
class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
}}
```

```
public class Faculty extends Employee {
  public static void main(String[ ] args) {
    new Faculty();
  }
  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
}}
class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }
  public Employee(String s) {
    System.out.println(s);
}}
class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
}}
```

```
public class Faculty extends Employee {
  public static void main(String[ ] args) {
    new Faculty();
  }
  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
}}
class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }
  public Employee(String s) {
    System.out.println(s);
}}
class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
}}
```

```
public class Faculty extends Employee {
  public static void main(String[ ] args) {
    new Faculty();
  }
  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
}}
class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }
  public Employee(String s) {
    System.out.println(s);
}}
class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
}}
```

```
public class Faculty extends Employee {
  public static void main(String[ ] args) {
    new Faculty();
  }
  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
}}
class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }
  public Employee(String s) {
    System.out.println(s);
}}
class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
}}
```

```
public class Faculty extends Employee {
  public static void main(String[ ] args) {
    new Faculty();
  }
  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
}}
class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }
  public Employee(String s) {
    System.out.println(s);
}}
class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
}}
```

```
public class Faculty extends Employee {
  public static void main(String[ ] args) {
    new Faculty();
  }
  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
}}
class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }
  public Employee(String s) {
    System.out.println(s);
}}
class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
}}
```

Example on the Impact of a Superclass without no-arg Constructor

```
public class Apple extends Fruit {
}

class Fruit {
  public Fruit(String name) {
    System.out.println("Fruit's constructor is invoked");
  }
}
```

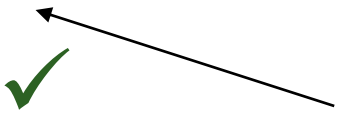
Are there errors in this program?

Example on the Impact of a Superclass without no-arg Constructor

```
public class Apple extends Fruit {
  public Apple() { super(); } // implicitly inserted by Java compiler
}
class Fruit {
  public Fruit(String name) {
    System.out.println("Fruit's constructor is invoked");
  }
}
```

The call to super must be the first statement in a constructor

```
public Boop() {
  super();
}
```



```
public Boop(int i) {
  super();
  size = i;
}
```

These are OK because a call to superclass constructor is explicitly coded in the first statement

```
public Boop() {
}
```

```
public Boop(int i) {
  size = i;
}
```

These are OK because a call to super() is automatically inserted as the first statement

```
public Boop(int i) {
  size = i;
  super();
}
```

Compilation error because a call to constructor is not the first statement

The call to this must be the first statement in a constructor

```
public Boop() {
  this(0);
}
```

✓

```
public Boop(int i) {
  this();
  size = i;
}
```

✓

These are OK because a call to another overloading constructor is explicitly coded in the first statement

```
public Boop() {
}
```

✓

```
public Boop(int i) {
  size = i;
}
```

✓

These are OK

```
public Boop(int i) {
  size = i;
  this();
}
```

✘

Compilation error because a call to another overloading constructor is not the first statement

First Statement Rule in Constructor

- The call to super must be the first statement in a constructor
- The call to this must be the first statement in a constructor

- How is the rule related to constructor chaining?
  - Which constructor body must be completely executed once before initializing any instance variables of the new object? Why?
  - May a superclass constructor be called more than once? So?

Defining a Subclass

- A subclass inherits all fields and methods (except constructors) from its superclass.
- We can also:
  - Add new fields
  - Add new methods
  - Override superclass methods

Questions

- Can a class have no superclass?
- Can a class extend more than one superclass?
- How is it connected with the language design?

Calling Superclass Methods

- We could rewrite the printCircle() method in the Circle class as follows:

```
public void printCircle() {
  System.out.println("The circle is created " +
    super.getDateCreated() + " and the radius is " + radius);
}
```

Circle.java

Overriding Superclass Methods

- A subclass overrides an inherited method from its superclass when it provides another implementation of the method.

```
public class Circle extends GeometricObject {
  // Other methods are omitted here
  // Override the toString method defined in GeometricObject

  @Override
  public String toString() {  // overriding method
    return super.toString() + "\nradius is " + radius;
  } }
```

NOTE

- An instance method of a superclass can be overridden only if it is accessible.
  - A private instance method cannot be overridden because it is inaccessible outside its own class.
  - A superclass private method and a subclass method are completely unrelated even they share the same signature.

OverrideTest.java

NOTE

- Like an instance method, a static method can be inherited.
- However, a static method cannot be overridden.
- If a superclass static method is redefined in a subclass, the method is hidden.

StaticTest.java

Overriding vs. Overloading

```
public class Test {
  public static void main(String[] args) {
    var a = new A();
    a.p(10);
    a.p(10.0);
  }
}
class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}
class A extends B {
  // This method overrides the method in B
  public void p(double i) {
    System.out.println(i);
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    var a = new A();
    a.p(10);
    a.p(10.0);
  }
}
class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}
class A extends B {
  // This method overloads the method in B
  public void p(int i) {
    System.out.println(i);
  }
}
```

Overloading

```
class B {
  public Object m(B o) { … }
}

class D extends B {
  public Object m(D o) { … }
}
```

Are they overloading?

```
public static void main(String args[]) {
  var d= new D ();
  B b = d;
  b.m(d);
  d.m(d);
  d.m(null);
}
```

Which method m is called?

OverloadingTest.java

Overrides vs Overload

When method f overrides method g	When method f overloads method g
The class defining f descends from the class defining g	f and g can be defined in the same class
f has the same parameter types and number of parameters as g	f has different parameter types and/or number of parameters as g
f's return type must be the same as or a subclass of g's return type	Their return types can differ
f cannot reduce the accessibility of g	Their accessibility can differ
Dynamic binding applies	Dynamic binding does not apply

What is Polymorphism?

- “Poly” means many.
- “Morph” means form.
- Polymorphism exists in the nature.

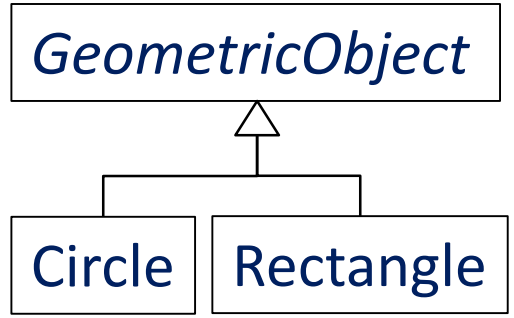
http://sgugenetics.pbworks.com/w/page/52776473/Polymorphism

Polymorphism in Java?

Transitivity: If A is a subtype of B and B is a subtype of C, A is a subtype of C. The transitivity rule also applies to supertype.

- A class defines a type.
  - A type defined by a subclass C is called a *subtype of C*
  - A type defined by its superclass C is called a *supertype of C*
- Polymorphism allows assigning (or passing) an instance of subtype to its supertype variable (or parameter)
  - Example: GeometricObject o = **new** Circle();

supertype variable ← o = **new** Circle(); → instance of subtype



  - Circle is the actual type of the instance
  - GeometricObject is the declared type of the instance when it is accessed via o

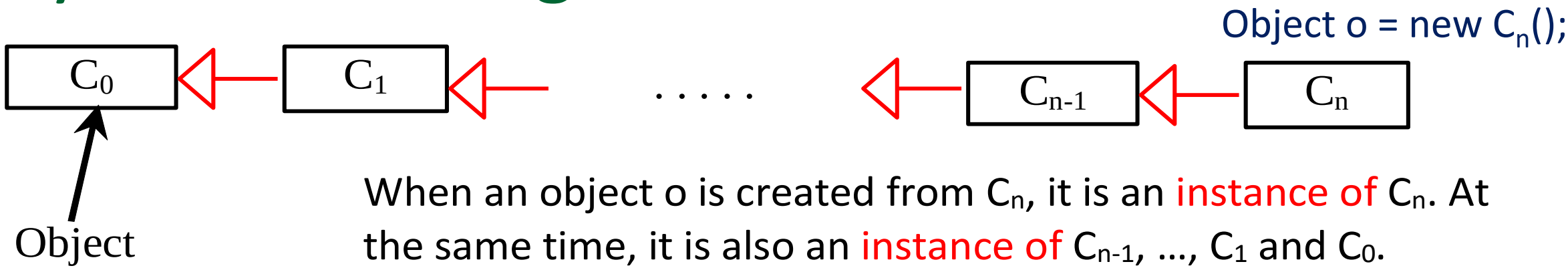
Liskov Substitution Principle (LSP)

Liskov Substitution Principle (LSP): ***An instance of subtype must accept a method call if it is accepted by an instance of the supertype. Overriding rules are defined to observe LSP.***

When method f overrides method g
The class defining f is a subtype of the class defining g
f has the same parameter types and number of parameters as g
f's return type must be the same as or a subclass of g's return type
f cannot reduce the accessibility of g
Dynamic binding applies

We will see more rules that observe LSP when discussing other Java features.
Further reading: https://reflectoring.io/lsp-explained/

Dynamic Binding



When an object o is created from C_n, it is an instance of C_n. At the same time, it is also an instance of C_{n-1}, …, C_1 and C_0.

- **Dynamic binding**
  - When an instance method m is invoked in o's context (i.e., o.m), JVM searches its implementation in the order of C_n, C_{n-1}, …, C_1 and C_0.
  - Once an implementation for m is found, the search stops and m is executed. This is the most specific implementation for m.

Method Matching vs. Dynamic Binding

- Matching a method signature and binding a method implementation are two separate issues.

- Java compiler finds a matching method according to its signature (parameter type, number of parameters, and order of the parameters) at compilation time. OverloadingTest.java
- JVM dynamically binds a method to its most specific implementation at runtime.

OverrideTest.java

Generic Programming

- Generic programming advocates methods to be implemented generically for a wide range of object reference arguments
- It is facilitated by
  - Polymorphism: Allows passing a subtype instance to a supertype parameter
  - Dynamic binding: When an instant's method is invoked, it is dynamically binded to its most specific implementation
  - Classes GraduateStudent, Student, Person, and Object have their own implementation of the toString() method

```
public class PolymorphismDemo {
  public static void main(String[] args) {
    m(new Student());
    m(new Person());
    m(new Object());
  }
  public static void m(Object x) {
    System.out.println(x);
  }}
class Student extends Person {
  public String toString() {
    return "Student";
  }}
class Person extends Object {
  public String toString() {
    return "Person";
}}
```

PolymorphismDemo.java

Casting Objects

We can use the casting operator to convert an instance of a type to another within an inheritance hierarchy.

```
m(new Student());
```

Assigns a Student object to a parameter of Object.

is the same as:

```
Object o = new Student(); // Implicit casting
m(o);
```

The implicit casting works because an instance of Student is also an instance of Object.

When do we use explicit casting?

- A compilation error occurs if we want to assign the reference value held by o to a variable of the Student type:

```
Object o = new Student(); // ok
Student b = o; // compile error
```

- Although o references a Student object, the compiler is not so clever to know it when compiling the second statement.
- To tell the compiler that o is a Student object, use explicit casting.

```
Student b = (Student) o; // Explicit casting
```

The instanceof Operator

- Use the instanceof operator to test whether an object is an instance of a reference type (class/interface):

we will discuss interface soon

```
Object myObj = new Circle();
  …
/** Perform casting if myObj is an instance of Circle */
if (myObj instanceof Circle) {
  var c = (Circle) myObj;
  System.out.println("The diameter is " + c.getDiameter());
  …
}
```

TIP in Understanding Casting

- To help understand casting, we may also consider the analogy of fruit, apple, and orange with the Fruit class as the superclass for Apple and Orange
- An apple is a fruit, so we can always safely assign an instance of Apple to a variable for Fruit
- However, a fruit is not necessarily an apple, so we have to use explicit casting to assign an instance of Fruit to a variable of Apple

Casting from Supertype to Subtype

- Explicit casting must be used when casting an object from a supertype to its subtype.
- Explicit casting may fail in some situations.

```
Apple x = (Apple) fruit;  // fail if fruit not references an apple
Orange x = (Orange) fruit; // fail if fruit not references an orange
```

Example: Demonstrating Polymorphism and Casting

- This example creates two geometric objects: a circle, and a rectangle, invokes the displayGeometricObject method to display the objects
- The displayGeometricObject displays the area and diameter if the object is a circle, and displays area if the object is a rectangle

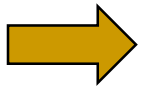
CastingDemo.java

Pattern Matching for **instanceof** operator

- Java 14 preview feature

```
if (myObj instanceof Circle) {
  var c = (Circle) myObj;
  System.out.println("The diameter
is " + c.getDiameter());
  ...
}
```

Such use of the instanceof operator is frequent, which always involve a redundant casting statement.



Perform matching based on the referenced instance's actual type

```
if (myObj instanceof Circle c) {
  System.out.println("The diameter
is " + c.getDiameter());
  ...
}
```

Strong demand by developers to eliminate the redundancy using pattern matching. The local variable c references an instance pattern-matched to Circle.

CastingDemo.java

The equals() versus ==

- The equals() method compares two objects.
- The inherited implementation of equals() from the Object class compares if two variables share the same object references.

```
public boolean equals(Object obj) { // implementation in Object
  return (this == obj);
}
```

compare object references

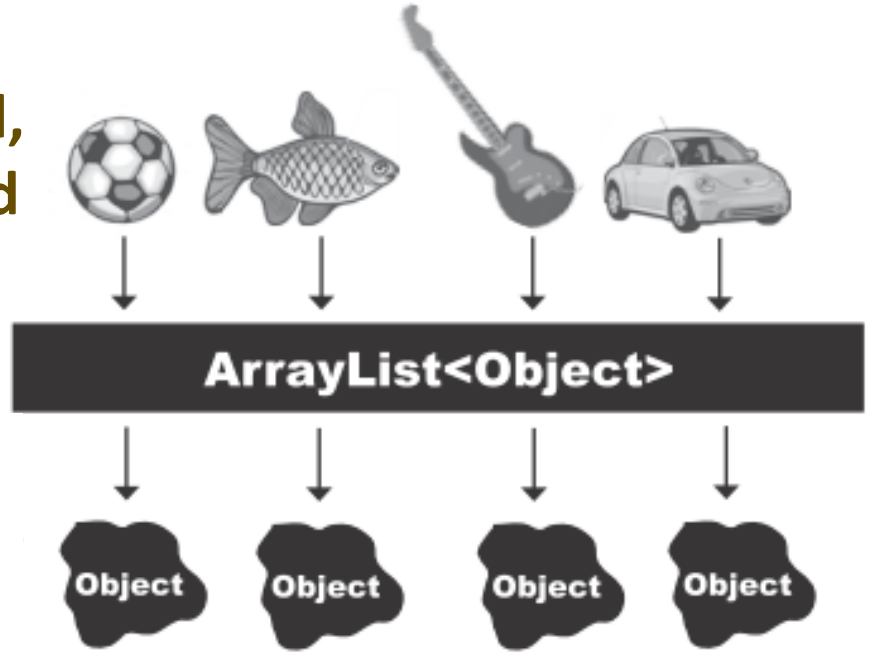
A better practice is:
Override equals() in our defined Java class to compare contents instead of object references.

```
class Circle {
// …
  public boolean equals(Object o) {
    if (o instanceof Circle c)
      return radius == c.radius;
    else
      return false;
} }
```

Good practice: avoid using ArrayList<Object>

- Everything comes out of an ArrayList<Object> has the type of Object, regardless of what its actual type is

The objects go IN as SoccerBall, Fish, Guitar, and Car.



But they come OUT as though they were of type Object.

Objects come out of an ArrayList<Object> acting like they're generic instances of class Object. The Compiler cannot assume the object that comes out is of any type other than Object.

extract from Head First Java

The protected modifier

- The protected modifier can be applied on data and methods in a class. A protected data or a protected method in a public class can be accessed by any class in the same package or its subclasses, even if the subclasses are in a different package.
- private, default, protected, public

Visibility increases

private, none (if no modifier is used), protected, public

Accessibility Summary

Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass	Accessed from a different package
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default	✓	✓	–	–
private	✓	–	–	–

Visibility Modifiers

A protected member grants two permissions:

1. default accessibility (under the same package);
2. to be inherited by a subclass and accessible by this subclass (under a different package)

p1.B; p2.C, p2.D

package p1;

```
public class C1 {
  public int x;
  protected int y;
  int z;
  private int u;

  protected void m() {
  }
}
```

```
public class C2 {
  C1 o = new C1();
  can access o.x;
  can access o.y;
  can access o.z;
  cannot access o.u;

  can invoke o.m();
}
```

```
public class C3
          extends C1 {
  can access x;
  can access y;
  can access z;
  cannot access u;

  can invoke m();
}
```

package p2;

```
public class C4
          extends C1 {
  can access x;
  can access y;
  cannot access z;
  cannot access u;

  can invoke m();
}
```

```
public class C5 {
  C1 o = new C1();
  can access o.x;
  cannot access o.y;
  cannot access o.z;
  cannot access o.u;

  cannot invoke o.m();
}
```

Cannot reduce accessibility of inherited methods

- A subclass may inherit a protected method from its superclass and change its visibility to public
- However, the subclass cannot reduce the accessibility of the inherited method declared in superclass

Liskov Substitution Principle (LSP): ***An instance of subtype must accept a method call if it is accepted by an instance of the supertype. Method inheritance rules are defined to observe LSP.***

```
class Super {
  protected void myMethod() {
    …
  }
}
class Sub extends Super {
  public void myMethod() {
    …
  }
}
```

cannot declare private or leave it empty

Cannot reduce accessibility of inherited methods

```
public class Super {
  public void myMethod() { … }
}

public class Sub extends Super {
  protected void myMethod() { … } // Compilation Error!
}
```

```
/* In another package */
void useMyMethod (Super obj) {
  obj.myMethod();
  …
}
```

Cannot reduce accessibility of inherited methods

```
public class Super {
  public static void myMethod() { … }
}

public class Sub extends Super {
  protected static void myMethod() { … } // Compilation Error!
}
```

NOTE

- The modifiers are used on classes and class members (data and methods), except that the final modifier can also be used on local variables in a method. A final local variable is a constant inside a method.

The final Modifier

- A final class cannot be extended:

```
final class Math { ... }
```

- A final variable is a constant:

```
final static double PI = 3.14159;
```

- A final method cannot be overridden by its subclasses.

```
final void myFinalMethod() { … }
```

Notes on Inheritance related to Arrays

- Arrays are objects.
- An array is an instance of the Object class
- If A is a subclass of B, every instance of A[] is an instance of B[]
- All the following statements are true:
  - **new** Integer(10) **instanceof** Object
  - **new** Integer[10] **instanceof** Object
  - **new** Integer[10] **instanceof** Object[]

Notes on Inheritance related to Arrays

- **int** and **double** are two compatible primitive types
  - We can assign an **int** value to a **double** type variable
- BUT **int**[] and **double**[] are two incompatible types
  - We cannot assign an **int**[] array to a variable of **double**[] or **Object**[] type
  - **int** is not a subclass of **Object**.

Three ways to initialize an instance field

- Field initializer
- Instance Initializer block
- Constructor

Field Initializer

```
class Student {
  double cga = 3.7;  // explicit field initializer
  // …
}
```

Instance Initializer Block

```
class Student {
  double cga = 3.7;
  { cga = 4.0; } // instance initializer block
  // …
}
```

Constructor

```
class Student {
  double cga = 3.7;
  { cga = 4.0; } // instance initializer block
  Student() { cga = 4.3; } // constructor
  // …
}
```

Instance Field Initialization

What happens when an object is created?

- All instance variables initialized to their default value (0, false, null)
- Field initializers and instance initializer blocks are executed according to their order of appearance
- Body of the constructor is executed after the body of its superclass' constructor
  - Note that a constructor might call another constructor at its first statement

What is the cga after a Student object is created?

```
class Student {
  double cga = 3.7;
  { cga = 4.0; } // instance initializer block
  Student() { cga = 4.3; } // constructor
  // …
}
```

InstanceFieldInitialization.java

Instance Field Initialization Order with Superclass

When a subclass object is created, it will be initialized in the following order:

- Field initializers and instance initializer blocks of its superclass
- Constructor of its superclass
- Field initializers and instance initializer blocks of the current class
- Constructor of the current class

Account.java
SavingsAccount.java
Bank.java

Constructors can invoke methods.
Why need instance initializer blocks?

Static Initializer Block

```
class Student {
  static double cga = 3.7;  // static field initializer
  static { cga = 4.0; } // static initializer block
  // …
}
```

Account.java
SavingsAccount.java
Bank.java

What happens when a class is loaded by JVM?

- All static variables are initialized to their default value (0, false, null)
- Static field initializers and static initializer blocks are executed in the order of their appearance
- Note that static variables are initialized once when its parent class is loaded.

StaticFieldInitialization.java

True or False?

extracted from Head First Java

1. To use the Math class, the first step is to make an instance of it.
2. You can mark a constructor with the keyword 'static'.
3. Static methods don't have access to an object's instance variables.
4. It is good practice to call a static method using a reference variable.
5. Static variables could be used to count the instances of a class.
6. Constructors are called before static variables are initialized.
7. MAX_SIZE would be a good name for a static final variable.
8. A static initializer block runs before a class's constructor runs.
9. If a class is marked final, all of its methods must be marked final.
10. A final method can only be overridden if its class is extended.
11. There is no wrapper class for boolean primitives.
12. A wrapper is used when you want to treat a primitive like an object.
13. The parseXxx methods always return a String.
14. Formatting classes (which are decoupled from I/O), are in the java.format package.

Ans: FFTFT FTTFF FTFF