

Unit 9: Inheritance

Unit 9 introduces inheritance, the mechanism that lets a class reuse and specialize another class. It covers superclasses and subclasses, constructor chaining with `super`, method overriding, polymorphic method dispatch, and the `Object` superclass that sits above every class.

AP Computer Science A

Inheritance

About 15 minutes to read

How to use this guide

Read the topics in order the first time because they build on each other. Inheritance defines the hierarchy, constructors show how objects in a hierarchy are built, overriding and `super` change inherited behavior, references and polymorphism decide which behavior runs, and `Object` sits at the top of it all. Exam questions usually show a small hierarchy and ask you to trace what prints.

After the first read, use the trap boxes and the Confusions table to review the distinctions that exam questions test most often. Then work the practice questions without looking back, and finish with the recall check on the last page out loud.

What this unit is worth. Unit 9 is about 7.5 to 10 percent of the AP CSA exam. Polymorphism and method dispatch are tested on both the multiple-choice and FRQ 2/3 every year. The `super()`-first rule and the private-members-are-not-inherited rule are the two traps that cost the most points.

9.1 Creating Superclasses and Subclasses

Inheritance models an **is-a relationship** between classes. A Dog is an Animal, so Dog can extend Animal and reuse what Animal already defines. The **superclass** holds the shared behavior, and each **subclass** adds or changes only what is specific to it.

```
public class Animal {
    private String name;
    public Animal(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
}

public class Dog extends Animal {
    public void bark() {
        System.out.println("woof");
    }
}
```

The keyword `extends` goes on the class header, and a class extends exactly one superclass. The subclass inherits the accessible members of the superclass: public, protected, and package-level members in the same package. Two things are never inherited. **Constructors are not inherited**, so the subclass declares its own. **Private members are not inherited** either, which means the subclass cannot name them directly. If the subclass needs a private superclass field, it goes through a public or protected accessor, like `getName()` above.

Trap. A subclass cannot read a private field of its superclass, even though every Dog object has a name field. Private means private to the class that declares it, and inheritance does not punch a hole in encapsulation. This is one of the two costliest traps in the unit.

9.2 Writing Constructors for Subclasses

A subclass constructor builds the superclass part of the object first, then the subclass part. The call `super( . . . )` invokes a superclass constructor, and it must be the **first statement** in the subclass constructor. Any statement before it is a compile error.

```
public class Dog extends Animal {
    private String breed;
    public Dog(String name, String breed) {
        super(name);
        this.breed = breed;
    }
}
```

If a constructor omits the super call, the compiler inserts `super( )` with no arguments as the first line. That only works if the superclass has a no-argument constructor. Once a superclass declares any constructor with parameters and no no-arg constructor, the default no-arg constructor disappears, and

every subclass constructor must call `super ( . . . )` with matching arguments. The rule is simple. The subclass constructor hands the superclass constructor whatever arguments it needs, on the first line.

Trap. Two common compile errors come from this rule. Putting an assignment before `super ( . . . )` fails because `super` must be first. Omitting `super ( . . . )` fails when the superclass has no no-arg constructor, because the inserted `super ( )` matches nothing. In both cases the fix goes on the first line of the constructor.

9.3 Overriding Methods

Overriding replaces an inherited method with a subclass version. The requirements are strict. The new method must have the same name and the exact same parameter list as the superclass method, the same signature. A different parameter list does not override. It creates an **overload**, a separate method that happens to share the name, and the inherited method still runs when the superclass signature is called.

```
public class Animal {
    public String speak() {
        return "generic sound";
    }
}
public class Dog extends Animal {
    @Override
    public String speak() {
        return "woof";
    }
}
```

Write `@Override` above every method you intend to override. The annotation is optional, but it turns a silent mistake into a compile error. Without it, a typo like `Speak` with a capital `S` compiles as a brand-new method while the inherited `speak` keeps running, and the bug shows up only in wrong output.

Three more limits apply. Private methods are not inherited, so a subclass cannot override them; a same-named method in the subclass is an unrelated method. Constructors cannot be overridden because they are not inherited. The overriding method cannot use a more restrictive access level than the original, so a public method stays public in the subclass.

Trap. The exam's favorite signature trap is `equals`. `public boolean equals(Dog other)` does not override `Object`'s `public boolean equals(Object other)`. The parameter type differs, so it is an overload. With `@Override` on it, the compiler rejects it and tells you the signature is wrong. Without `@Override`, it compiles, and calls made through an `Object` reference run the inherited identity version instead of your logic.

9.4 The super Keyword

`super` reaches up to the superclass from inside the subclass, and it has two jobs. The first is `super(...)`, the constructor call from topic 9.2, which is legal only as the first statement of a constructor. The second is `super.method()`, which calls the superclass version of an overridden method from inside the subclass.

```
public class Animal {
    public String getDescription() {
        return "an animal";
    }
}
public class Dog extends Animal {
    @Override
    public String getDescription() {
        return super.getDescription() + " that barks";
    }
}
```

Calling `super.getDescription()` runs the `Animal` version and lets `Dog` extend it instead of rewriting it. If the override called `getDescription()` without `super`, it would call itself and recurse forever. There is also `super.field` for reaching a hidden superclass field, but that situation is rare in AP code. The method form is the one the exam tests.

Trap. `super()` and `super.method()` are not interchangeable. `super(...)` belongs on the first line of a constructor and nowhere else. `super.method()` belongs in instance methods. Using `super(...)` inside an ordinary method is a compile error.

9.5 Creating References Using Inheritance Hierarchies

A reference variable has a **declared type**, from its declaration, and an **actual type**, the class of the object it points to at runtime. Java allows the actual type to be the declared type or any subclass of it.

```
Animal a = new Dog();           // legal: a Dog is an Animal
Dog d = new Animal();           // compile error: not every Animal is a Dog
```

Assigning a subclass object to a superclass reference is **upcasting**. It is always safe and needs no cast. The reverse is **downcasting** and needs an explicit cast, because the compiler cannot be sure the object is really a `Dog`.

```
Animal a = new Dog();
Dog d = (Dog) a;                // legal: the object really is a Dog
a.fetch();                      // compile error: Animal has no fetch method
((Dog) a).fetch();              // legal after the cast
```

The compiler checks casts against declared types. The JVM checks them against the actual object at runtime. A cast the compiler can prove impossible fails at compile time. A cast that looks possible but is wrong fails at runtime with a **ClassCastException**.

```
Animal a = new Animal();  
Dog d = (Dog) a;           // compiles, throws ClassCastException at runtime
```

There is one more compile-time rule. A reference can only call methods its declared type knows. `Animal a = new Dog ( )` can call `speak()`, which `Animal` declares, but not `fetch()`, which only `Dog` declares, until `a` is cast to `Dog`.

Trap. Upcasting is automatic, but downcasting needs a cast and an honest object. A downcast that compiles is not guaranteed to run, because the runtime check happens later. When you are unsure what an object really is, test with `instanceof` before casting.

9.6 Polymorphism

Polymorphism means the method that runs is chosen by the object's actual type at runtime, not by the reference's declared type. This choice is called **dynamic dispatch**.

```
Animal a = new Dog();  
System.out.println(a.speak()); // prints "woof"
```

The reference says `Animal`, the object is a `Dog`, and `Dog`'s version of the overridden instance method runs. This is why upcasting is useful. One reference type can hold many object types, and each object behaves like itself.

Three things do **not** dispatch polymorphically. Fields use the declared type at compile time.

```
public class Animal {  
    public String label = "animal";  
}  
public class Dog extends Animal {  
    public String label = "dog";  
}  
Animal a = new Dog();  
System.out.println(a.label); // prints "animal"
```

Static methods also bind to the declared type at compile time, and private methods and constructors are never overridden, so there is nothing to dispatch.

This is the unit's most tested idea. AP FRQs 2 and 3 use class hierarchies every year, and the tracing questions come down to one rule: for an overridden instance method, read the object's type. For a field or a static method, read the reference's type.

Trap. If a question asks which method runs, first check whether the method is an overridden instance method. If it is, the object decides. If the question asks about a field or a static method, the reference decides. Mixing those two rules up is how most polymorphism points are lost.

9.7 Object Superclass

Every class extends `Object`, directly or through its superclass chain, so every object inherits `Object`'s methods. Two of them appear on the exam.

`public String toString()` returns a string representation of the object. The default version returns the class name followed by the object's hash code, something like `Dog@1a2b3c`. Classes override it to produce readable output, and `println` calls it automatically when you print an object.

`public boolean equals(Object other)` compares two objects. The default version checks identity, whether both references point to the same object, which is exactly what `==` does. To compare by content, a class overrides `equals` with the exact signature `public boolean equals(Object other)`.

Trap. The parameter type is part of the signature. `equals(Dog other)` overloads `equals` instead of overriding it, so calls made through an `Object` reference still run the identity version. This is the case `@Override` exists for. Annotate the method and the compiler tells you the signature is wrong.

Confusions That Cost Points

PAIR	HOW TO KEEP THEM STRAIGHT
Overriding vs overloading	Overriding redefines an inherited method with the identical signature and is chosen at runtime by the object's type. Overloading defines a new method with the same name but different parameters and is chosen at compile time by the argument types.
<code>super()</code> vs <code>super.method()</code>	<code>super(...)</code> calls a superclass constructor and is legal only as the first statement of a subclass constructor. <code>super.method()</code> calls an overridden superclass method and is used inside instance methods.
Inherited vs accessible	A subclass inherits public, protected, and package-level members. Private members are not inherited and cannot be named in the subclass, although they still exist inside each object.
Upcasting vs downcasting	Upcasting assigns a subclass object to a superclass reference automatically and is always safe. Downcasting casts back toward the subclass, needs an explicit cast, and throws <code>ClassCastException</code> at runtime if the object is not of that type.
Method dispatch vs field access	Overridden instance methods dispatch on the object's actual type at runtime. Fields and static methods resolve on the reference's declared type at compile time.

Practice Questions

Original questions written for this guide in the style of the AP exam. Answers and explanations are on the next page, so complete the questions before checking them.

1. Consider the following classes, where Animal has only the constructor `Animal(String name)`.

```
public class Dog extends Animal {
    private String breed;
    public Dog(String name, String breed) {
        this.breed = breed;
        super(name);
    }
}
```

What is the result of compiling the Dog class?

1. It compiles, and `super(name)` runs before the assignment.
2. It fails to compile because the call to `super` must be the first statement in the constructor.
3. It compiles, and the compiler moves `super(name)` to the first line automatically.
4. It fails to compile because a subclass constructor cannot declare its own parameters.

2. Consider the following classes.

```
public class Animal {
    private String name;
    public Animal(String name) { this.name = name; }
    public String getName() { return name; }
}
public class Dog extends Animal {
    public Dog(String name) { super(name); }
    public void printName() {
        System.out.println("Dog: " + name);
    }
}
```

What is the result of compiling this code?

1. It compiles and prints "Dog: " followed by the dog's name when `printName` is called.
2. It fails to compile because `name` has private access in `Animal`.
3. It compiles but throws a `NullPointerException` when `printName` runs.
4. It fails to compile because `Dog` does not declare a `name` field.

3. Consider the following classes.

```
public class Animal {  
    public String speak() { return "generic sound"; }  
}  
public class Dog extends Animal {  
    @Override  
    public String speak() { return "woof"; }  
}
```

After `Animal a = new Dog();`, what is printed by `System.out.println(a.speak());`?

1. generic sound
2. woof
3. It fails to compile because `speak` is not defined for an `Animal` reference.
4. It fails to compile because a `Dog` cannot be assigned to an `Animal` variable.

4. Consider the following class.

```
public class Dog {  
    private String name;  
    public Dog(String name) { this.name = name; }  
    public boolean equals(Dog other) {  
        return name.equals(other.name);  
    }  
}
```

What is printed by the following code?

```
Object d1 = new Dog("Rex");  
Object d2 = new Dog("Rex");  
System.out.println(d1.equals(d2));
```

1. true, because the two dogs have equal names
2. false, because `equals(Dog)` does not override `Object`'s `equals(Object)`
3. It fails to compile because `equals` must take an `Object` parameter.
4. It fails to compile because `Dog` does not extend a class that defines `equals`.

5. Consider the following classes.

```
public class Animal {  
    public String label = "animal";  
}  
public class Dog extends Animal {  
    public String label = "dog";  
}
```

After `Animal a = new Dog();`, what is printed by `System.out.println(a.label);`?

1. animal
2. dog
3. It fails to compile because label is ambiguous.
4. null

6. Assume Dog extends Animal and both have no-argument constructors. What is the result of the following code?

```
Animal a = new Animal();  
Dog d = (Dog) a;
```

1. It compiles and runs; d refers to the same object as a.
2. It fails to compile because an Animal cannot be cast to a Dog.
3. It compiles but throws a ClassCastException at runtime.
4. It compiles but throws a NullPointerException at runtime.

7. Consider the following classes.

```
public class Animal {  
    public String getDescription() { return "an animal"; }  
}  
public class Dog extends Animal {  
    @Override  
    public String getDescription() {  
        return super.getDescription() + " that barks";  
    }  
}
```

After `Animal a = new Dog();`, what is printed by `System.out.println(a.getDescription());`?

1. an animal that barks
2. an animal
3. It recurses until the program crashes with a StackOverflowError.
4. It fails to compile because super cannot be used inside an ordinary method.

8. Consider the following classes.

```
public class Animal {  
    public String move() { return "moves"; }  
}  
public class Dog extends Animal {  
    @Override  
    public String move() { return "runs"; }  
}  
public class Fish extends Animal {  
    @Override  
    public String move() { return "swims"; }  
}
```

Consider `Animal[] animals = { new Dog(), new Fish(), new Animal() };`. What is printed by this loop?

```
for (Animal x : animals) {  
    System.out.println(x.move());  
}
```

1. moves
moves
moves
2. runs
swims
moves
3. runs
runs
runs
4. It fails to compile because Dog and Fish cannot both extend Animal.

Answer Key

- 1. B.** The call to super must be the first statement in the constructor, and here an assignment comes first, so the code does not compile. A is wrong because Java never reorders your statements. C is wrong because the compiler never moves code; it only inserts a no-arg super() when no super call is present, and that is not this case. D is wrong because subclass constructors routinely declare their own parameters.
- 2. B.** The field name is private in Animal, so printName cannot name it directly in the subclass. The error is reported at compile time. A assumes inheritance grants access to private fields, which it does not. C is wrong because the failure happens at compile time, so nothing ever runs. D gives the wrong reason; declaring a name field in Dog is legal, but the error here is access, not a missing declaration.
- 3. B.** speak is an overridden instance method, so the object's actual type decides. The object is a Dog, so "woof" prints. This is dynamic dispatch. A reads the reference type, which only decides fields and static methods. C is wrong because speak is declared in Animal, so it is callable through an Animal reference. D is wrong because assigning a subclass object to a superclass reference is legal upcasting.
- 4. B.** equals(Dog other) has a different parameter type than equals(Object other), so it overloads rather than overrides. Through the Object references d1 and d2, the call resolves to Object's equals, which compares identity, and two distinct objects are not identical, so it prints false. A assumes the name comparison runs, but it never gets called. C is wrong because the code compiles fine as an overload; only an override requires the Object parameter. D is wrong because every class extends Object, which defines equals(Object).
- 5. A.** Fields do not dispatch polymorphically. The access a.label uses the declared type Animal at compile time, which holds "animal". B applies the method-dispatch rule to a field, which is the trap this question tests. C is wrong because a hidden field is legal and unambiguous through a typed reference. D is wrong because label is initialized in both classes.
- 6. C.** The cast from Animal to Dog is plausible, so it compiles, but at runtime the object is a plain Animal, not a Dog, so the JVM throws a ClassCastException. A describes a successful downcast, which this is not. B is wrong because narrowing casts between related classes compile; the real check happens at runtime. D is wrong because nothing here is null.
- 7. A.** Dispatch picks Dog's override because the object is a Dog. The override calls super.getDescription() for "an animal" and appends " that barks". B ignores the override entirely. C would happen only if the method called getDescription() without super, recursing into itself. D is wrong because super.method() is legal in any instance method; only super(...) is restricted to constructors.
- 8. B.** Each object's own version of the overridden method runs: Dog prints "runs", Fish prints "swims", and the plain Animal prints "moves". A reads the reference type for all three elements. C assumes the first object's method applies to the whole array. D is wrong because any number of classes can extend one superclass; only multiple inheritance of classes is banned.

When you check your answers, note which distinction each miss came from. Make a flashcard for that distinction and drill it spaced out over the next few days instead of rereading the whole section. If you missed one of these questions, the same distinction is worth practicing again in Rycal, where the Inheritance deck has flashcards for it and more practice questions use the same kinds of traps.

One-Page Recall Check

Say each answer out loud before you look back, and mark the ones you cannot finish. Anything you cannot say out loud yet belongs in your flashcard deck. In Rycal, add those items to the Inheritance deck and let spaced review bring them back over the next few days.

- Explain the is-a relationship and what extends means in a class header.
- List what a subclass inherits and what it does not inherit.
- State the super()-first rule and what the compiler inserts when you omit super(...).
- Write a subclass constructor that passes arguments up through super(...).
- State the exact requirements for overriding a method.
- Explain what @Override buys you.
- Distinguish overriding from overloading.
- Explain both uses of super.
- State which reference-to-object assignments compile and which need a cast.
- Trace a method call through a superclass reference to a subclass object.
- List what does not dispatch polymorphically.
- Explain what happens at runtime when a downcast is wrong.
- State what toString() and equals() do by default, and how to override each correctly.

Where to go next. Turn every missed item above into flashcards and drill them spaced out over several days rather than in one sitting. In Rycal, open the Inheritance deck under AP Computer Science A. The deck covers the terms in this guide, and its practice questions target the same traps named here. If you have a test date, add it in the Test Planner. You can also start your next review with a Brain Dump, then check what you missed against this guide.

Key terms for this unit

Inheritance, is-a relationship, Superclass, Subclass, extends, Constructor, super(), Overriding, @Override annotation, Overloading, Signature, Dynamic dispatch, Polymorphism, Declared type, Actual type, Upcasting, Downcasting, ClassCastException, instanceof, super.method(), Object superclass, toString(), equals(Object).

About this guide. Written for Rycal and aligned to the College Board AP Computer Science A course framework, Unit 9. All questions and explanations are original Rycal writing. Rycal is independent and is not affiliated with or endorsed by the College Board.