Introduction to Programming

Week 10

Magnus Madsen

Week 10: Outline

- Interfaces and Implementations
- Inheritance and Overriding
- Abstract Classes
- The `Object` Class
- Packages

Prologue

Quote of the Week

"The main activity of programming is not the origination of new independent programs, but in the integration, modification, and explanation of existing ones."

— Terry Winograd

Epigram of the Week

"Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it."

— Alan Perlis

Interfaces

What is an interface?

An **interface** is a **contract** that specifies what operations a class must provide.

Key Concepts:

- An interface defines **method signatures** without method bodies.
- A class **implements** one or more interfaces filling out the methods.
- Interfaces enables **subtype polymorphism**.

Example: Shape

We can use an **interface** to define a **type** for shapes:

```
public interface Shape {
    double getX();
    double getY();
    void setX(double x);
    void setY(double y);
    void draw();
}
```

The interface specifies the *method signatures*, but omits the *method bodies*.

We say that the methods are **abstract methods**.

Example: Square (1/2)

We can write an **implementation** of an interface using the `implements` keyword.

```
public class Square implements Shape {
    private double x;
    private double y;
    private double side;

    public Square(double x, double y, double side) {
        this.x = x;
        this.y = y;
        this.side = side;
    }
    ...
}
```

Example: Square (2/2)

A class must define **all** abstract methods from the interface:

```
public class Square implements Shape {
    // ... from previous slide ...
    double getX() { return x; }
    double getY() { return y; }
    void setX(double x) { this.x = x; }
    void setY(double y) { this.y = y; }
    void draw() {
        StdDraw.square(x, y, side / 2);
    }
}
```

We have now implemented `Shape` for `Square`. We say that `Square` is a **subtype** of `Shape`.

Example: Circle

We can also write a `Circle` class that implements the `Shape` interface:

```
public class Circle implements Shape {
    private double x;
    private double y;
    private double radius;

    public Circle(double x, double y, double radius) {
        this.x = x;
        this.y = y;
        this.radius = radius;
    }
    // ... getters and setters ...
    public void draw() {
        StdDraw.circle(x, y, radius);
    }
}
```

Example: Triangle

And we can write a `Triangle` class that implements the `Shape` interface:

```
public class Triangle implements Shape {
    private double x;
    private double y;
    private double size;

    public Triangle(double x, double y, double size) {
        this.x = x;
        this.y = y;
        this.size = size;
    }
    // ... getters and setters ...
    public void draw() {
        double[] xCoords = {x, x - size / 2, x + size / 2};
        double[] yCoords = {y + size / 2, y - size / 2, y - size / 2};
        StdDraw.polygon(xCoords, yCoords);
    }
}
```

Subtype Polymorphism (1/3)

We can write code like:

```
Square s = new Square(5, 5, 10);
Circle c = new Circle(5, 5, 10);
s.draw();
c.draw();
```

No surprises here.

Subtype Polymorphism (2/3)

But more interestingly we can write:

```
Shape s = new Square(5, 5, 10);
Shape c = new Circle(5, 5, 10);
r.draw();
c.draw();
```

We call this idea **subtype polymorphism**.

- Shape takes many forms; it is polymorphic.

Key Concepts:

- The **static type** of `s` and `c` is `Shape`.
- The **dynamic type** of `s` and `c` is `Rectangle` and `Circle`.

Remark: At runtime there are no shapes! There are only rectangles and circles.

Subtype Polymorphism (3/3)

What we can do with a `Shape`:

We can call the methods defined on `Shape`:

```
Shape s = ...
s.getX();
s.draw();
// ...
```

The interface guarantees that all implementations provide these methods.

Subtype Polymorphism (3/3)

What we can do with a `Shape`:

We can call the methods defined on `Shape`:

```
Shape s = ...
s.getX();
s.draw();
// ...
```

The interface guarantees that all implementations provide these methods.

What we cannot do with a `Shape`:

We *cannot* call methods that are only available on a specific shape:

```
class Pyramid implements Shape {
    // ... fields, getters, setters ...
    public int getHeight() {
        return this.height;
    }
}
```

This won't work:

```
Shape s = new Pyramid(...);
s.getHeight(); // Error!
```

Example: Using an Interface

We can use interfaces to define functions that are polymorphic.

Here is a function that can move *any* shape:

```
public static void move(Shape shape, double dx, double dy) {
    shape.setX(shape.getX() + dx);
    shape.setY(shape.getY() + dy);
}
```

We can use it as follows:

```
Square s = new Square(0, 0, 2);
move(c, 5, 5);
```

Example: An Array of Shapes (1/2)

We can use the `Shape` interface to create an array of shapes:

```
Shape[] shapes = new Shape[3];
shapes[0] = new Square(0.3, 0.5, 0.2);
shapes[1] = new Circle(0.5, 0.5, 0.1);
shapes[2] = new Triangle(0.7, 0.5, 0.15);
```

Example: An Array of Shapes (2/2)

We can create an animation of shapes moving together:

```
public static void main(String[] args) {
    Shape[] shapes = new Shape[3];
    shapes[0] = new Square(0.3, 0.5, 0.2);
    shapes[1] = new Circle(0.5, 0.5, 0.1);
    shapes[2] = new Triangle(0.7, 0.5, 0.15);

    StdDraw.enableDoubleBuffering();
    for (int i = 0; i < 100; i++) {
        StdDraw.clear();
        for (Shape shape : shapes) {
            move(shape, 0.002, 0.001);
            shape.draw();
        }
        StdDraw.show();
        StdDraw.pause(20);
    }
}
```

Q:

What can you say about this program?

```
public class Main {
    public static void main(String[] args) {
        Shape s = new Shape();
        s.setX(0.5);
        s.setY(0.5);
        s.draw();
    }
}
```

Q:

What are the static and dynamic types of the variables?

```
Square u = new Square(0, 0, 5);
Circle v = new Circle(0, 0, 5);
Shape w = u;
w = v;
w = null;
```

Subtyping and Assignability (1/2)

We say there is a **subtyping relationship** between `Shape` and `Circle`:

- `Circle` is a subtype of `Shape`.
- But `Shape` is most definitely *not* a subtype of `Circle`.

All humans are animals, but not all animals are humans!

Subtyping and Assignability (2/2)

We must respect the subtyping relation during assignment.

Valid assignments (subtype to supertype):

```
Circle c = new Circle(0.5, 0.5, 0.1);
Shape s = c;  // OK: Circle is a subtype of Shape
```

Invalid assignments (supertype to subtype):

```
Shape s = new Circle(0.5, 0.5, 0.1);
Circle c = s;  // Error: Shape is not a subtype of Circle
```

Multiple Interfaces (1/2)

Java allows a class to implement *multiple* interfaces.

We can define three interfaces:

```
public interface Shape {
    double getX();
    double getY();
    void draw();
}

public interface Movable {
    void move(double dx, double dy);
}

public interface Measurable {
    double getArea();
}
```

and have `Circle` implement all of them:

```
public class Circle implements Shape, Movable, Measurable {
    // ... fields and constructors ...
    public double getX() { return x; }
    public double getY() { return y; }

    public void draw() {
        StdDraw.circle(x, y, radius);
    }

    public void move(double dx, double dy) {
        this.x += dx;
        this.y += dy;
    }

    public double getArea() {
        return Math.PI * radius * radius;
    }
}
```

Multiple Interfaces (2/2)

All of the following assignments are now valid:

```
Circle c = new Circle(0.5, 0.5, 0.1);

Shape s = c;          // OK: Circle implements Shape
Movable m = c;        // OK: Circle implements Movable
Measurable a = c;     // OK: Circle implements Measurable
```

Each variable gives access to different methods:

```
s.draw();             // Shape methods
m.move(0.1, 0.2);     // Movable methods
a.getArea();          // Measurable methods
```

Q:

Which of the following assignments are valid?

```
Circle c = new Circle(...);
Shape s = new Circle(...);

Shape s1 = c;          // (1)
Circle c1 = s;         // (2)
Movable m1 = c;        // (3)
Measurable a1 = s;     // (4)
Shape s2 = m1;         // (5)
```

Q:

Which of the following method calls are valid?

```
Shape s = new Circle(...);
Movable m = new Circle(...);

s.draw();              // (1)
s.move(0.1, 0.2);      // (2)
s.getArea();           // (3)
m.draw();              // (4)
m.move(0.1, 0.2);      // (5)
m.getArea();           // (6)
```

Example: Animals (1/2)

Different animals have different capabilities. We can model this with multiple interfaces:

```
public interface Animal {
    String makeSound();
}

public interface Swimmable {
    void swim();
}

public interface Flyable {
    void fly();
}
```

Example: Animals (1/2)

Different animals have different capabilities. We can model this with multiple interfaces:

```java
public interface Animal {
    String makeSound();
}

public interface Swimmable {
    void swim();
}

public interface Flyable {
    void fly();
}
```

```java
public class Duck implements Animal, Swimmable, Flyable {
    public String makeSound() {
        return "Quack!";
    }

    public void swim() {
        System.out.println("Duck is swimming");
    }

    public void fly() {
        System.out.println("Duck is flying");
    }
}
```

Example: Animals (2/2)

Different animals implement different combinations of interfaces:

```
public class Fish implements Animal, Swimmable {
    public String makeSound() { return "Blub!"; }
    public void swim() { System.out.println("Fish is swimming"); }
}

public class Bird implements Animal, Flyable {
    public String makeSound() { return "Tweet!"; }
    public void fly() { System.out.println("Bird is flying"); }
}

public class Cat implements Animal {
    public String makeSound() { return "Meow!"; }
}
```

Inheritance

What is Inheritance?

Inheritance is a mechanism where one class **inherits** fields and methods from another class.

Key Concepts:

- A **subclass** (or child class) inherits from a **superclass** (or parent class).
- The subclass gains all the fields and methods of the superclass.
- The subclass can **extend** the superclass by adding new fields and methods.
- The subclass can **override** methods to provide specialized behavior.
- Inheritance is form of **subtype polymorphism**.

Example: Point and ColoredPoint (1/2)

We can write a **superclass**:

```
public class Point {
    private int x;
    private int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return this.x;
    }

    public int getY() {
        return this.y;
    }
}
```

Example: Point and ColoredPoint (1/2)

We can write a **superclass**:

```
public class Point {
    private int x;
    private int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return this.x;
    }

    public int getY() {
        return this.y;
    }
}
```

and we can write a **subclass** that extends `Point`:

```
public class ColoredPoint extends Point {
    private Color color;

    public ColoredPoint(int x, int y, Color color) {
        super(x, y);
        this.color = color;
    }

    public Color getColor() {
        return this.color;
    }
}
```

Example: Point and ColoredPoint (2/2)

We can create objects of both the superclass and the subclass:

```
Point p = new Point(10, 20);
ColoredPoint cp = new ColoredPoint(5, 15, Color.RED);
```

We can call inherited methods on the subclass:

```
System.out.println(cp.getX());      // 5
System.out.println(cp.getY());      // 15
System.out.println(cp.getColor());  // RED
```

We can also use subtype polymorphism:

```
Point p1 = new Point(10, 20);
Point p2 = new ColoredPoint(5, 15, Color.BLUE);  // OK!
```

What is Dynamic Method Lookup?

Dynamic method lookup is the mechanism where Java determines **which method to call** based on the actual type of the object at runtime.

Recall:

- The **static type** is the declared type of a reference (variable, formal parameter, field).
- The **dynamic type** is the actual type of the object at runtime.
- Java uses the **dynamic type** to determine which method to call.

Example: Dynamic Method Lookup

We can create an array of `Point` objects that contains both `Point` and `ColoredPoint`:

```
Point[] points = new Point[3];
points[0] = new Point(0, 0);
points[1] = new ColoredPoint(1, 1, Color.RED);
points[2] = new ColoredPoint(2, 2, Color.BLUE);
```

We can call methods and draw each point:

```
for (Point p : points) {
    StdDraw.point(p.getX(), p.getY());  // Dynamic method lookup!
}
```

Java determines which `getX()` and `getY()` to call based on the **dynamic type** of each object.

The `instanceof` Operator

The `instanceof` operator tests whether an object is an instance of a specific class or interface.

```
Point p1 = new Point(0, 0);
Point p2 = new ColoredPoint(1, 1, Color.RED);
```

```
System.out.println(p1 instanceof Point);          // true
System.out.println(p1 instanceof ColoredPoint);   // false
System.out.println(p2 instanceof Point);          // true
System.out.println(p2 instanceof ColoredPoint);   // true
```

Example: `instanceof`

We can use `instanceof` to check if a point is colored before drawing it:

```
Point[] points = new Point[3];
points[0] = new Point(0, 0);
points[1] = new ColoredPoint(1, 1, Color.RED);
points[2] = new ColoredPoint(2, 2, Color.BLUE);

for (Point p : points) {
    if (p instanceof ColoredPoint) {
        ColoredPoint cp = (ColoredPoint) p; // Cast
        StdDraw.setPenColor(cp.getColor());
    } else {
        StdDraw.setPenColor(Color.BLACK);
    }
    StdDraw.point(p.getX(), p.getY());
}
```

Remark: Using `instanceof` is bad style and againsts proper object-oriented design.

Safe and Unsafe Casts

Recall that we can cast between **primitive types**:

```
double x = 3.14;
int y = (int) x;  // y is 3
```

Safe and Unsafe Casts

Recall that we can cast between **primitive types**:

```
double x = 3.14;
int y = (int) x;  // y is 3
```

We can also cast between **reference types** in the inheritance hierarchy:

Valid cast (subclass to superclass, then back):

```
Point p = new ColoredPoint(1, 1, Color.RED);
ColoredPoint cp = (ColoredPoint) p;  // OK!
```

Invalid cast (unrelated types):

```
Point p = new Point(0, 0);
Square s = (Square) p;
```

```
Exception in thread "main"
java.lang.ClassCastException
```

Overriding

What is Overriding?

Method overriding allows a subclass to provide a **specific implementation** of a method that is already defined in its superclass.

Key Concepts:

- A subclass can **override** a method from its superclass by redefining it.
- The overriding method must have the **same signature** (name, parameters, return type).
- The overriding method is called instead of the superclass method at runtime.
- Use the `@Override` annotation to ensure you are actually overriding a method.

Example: Overriding (1/2)

The **superclass** defines a method:

```
public class Employee {
    private String name;
    private double salary;

    public Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
    }

    public double getSalary() {
        return this.salary;
    }
}
```

and the **subclass** overrides it:

```
public class Manager extends Employee {

    public Manager(String name, double salary) {
        super(name, salary);
    }

    @Override
    public double getSalary() {
        return 2.5 * super.getSalary();
    }
}
```

Example: Overriding (2/2)

We can create instances of `Employee` and `Manager`, and call `getSalary()` on each:

```
Employee e = new Employee("Alice", 100_000);
Employee m = new Manager("Bob", 100_000);

System.out.println(e.getSalary());  // 100_000.0
System.out.println(m.getSalary());  // 250_000.0
```

The `super` keyword is used to **refer to the superclass** of the current object.

Key uses of `super`:

- Call **superclass constructor**: `super(args)` must be first in subclass constructor.
  - A subclass **must** call one of its superclass constructors.
- Call **superclass method**: `super.methodName()` invokes the superclass version.
- Access **superclass field**: `super.fieldName` (very very rarely used).

Example: `super`

Earlier, we saw a call to a **super constructor**:

```
public Manager(String name, double salary) {
    super(name, salary);            // Call Employee constructor
}
```

and a call to a **super method**:

```
@Override
public double getSalary() {
    return 2.5 * super.getSalary();  // Call Employee's getSalary()
}
```

Q:

What does this program print?

```
class Beverage {
    private int temperature;
    public Beverage(int temp) { this.temperature = temp; }
    public String serve() { return "Serving at " + temperature + "°C"; }
}

class Coffee extends Beverage {
    public Coffee(int temp) { super(temp); }
    public String serve() { return super.serve() + " with caffeine!"; }
}

public static void main(String[] args) {
    Beverage b = new Coffee(85);
    System.out.println(b.serve());
}
```

What does this program print?

Q:

```
class Book {
    private String title;
    private String author;
    public Book(String title, String author) {
        this.title = title; this.author = author;
    }
    public String describe() { return title + " by " + author; }
}
class AudioBook extends Book {
    private double duration;
    public AudioBook(String title, String author, double duration) {
        super(author, title); this.duration = duration;
    }
    public String descirbe() { return super.describe() + " (" + duration + "h)"; }
}
public static void main(String[] args) {
    Book b = new AudioBook("1984", "Orwell", 11.5);
    System.out.println(b.describe());
}
```

The `@Override` Annotation

The `@Override` **annotation** instructs the compiler that a method is **intended to override** a method from a superclass or interface.

If we had written:

```
@Override
public String descirbe() {
    return super.describe() + " (" + duration + "h)";
}
```

The compiler would have caught our typo.

Abstract Classes

What is an Abstract Class?

An **abstract class** is a class that **cannot be instantiated** and may contain abstract methods.

Key Concepts:

- An abstract class is declared with the `abstract` keyword.
- It can have both **abstract methods** (no body) and **concrete methods** (with body).
- Subclasses must implement all abstract methods or be abstract themselves.
- Abstract classes can have fields, constructors, and concrete methods like normal classes.

Remark: Abstract classes sit between interfaces and concrete classes: they provide partial implementation while still allowing abstract behavior.

Example: Abstract Class (1/3)

As earlier, we can write an interface for `Animal`:

```
public interface Animal {
    String makeSound();
}
```

And we can write implementations:

```
public class Dog implements Animal {
    public String makeSound() {
        return "Woof!";
    }
}
```

```
public class Cat implements Animal {
    public String makeSound() {
        return "Meow!";
    }
}
```

```
public class Cow implements Animal {
    public String makeSound() {
        return "Moo!";
    }
}
```

But what happens when we want every animal to have an age?

Example: Abstract Class (2/3)

We want to avoid **code duplication**, so we change `Animal` to a proper **class**:

```java
public class Animal {
    private int age;

    public Animal(int age) {
        this.age = age;
    }

    public int getAge() {
        return this.age;
    }

    public String makeSound() {
        return null;
    }
}
```

```java
public class Dog extends Animal {
    public Dog(int age) {
        super(age);
    }

    public String makeSound() {
        return "Woof!";
    }
}
```

Example: Abstract Class (2/3)

We want to avoid **code duplication**, so we change `Animal` to a proper **class**:

```
public class Animal {
    private int age;

    public Animal(int age) {
        this.age = age;
    }

    public int getAge() {
        return this.age;
    }

    public String makeSound() {
        return null;
    }
}
```

```
public class Dog extends Animal {
    public Dog(int age) {
        super(age);
    }

    public String makeSound() {
        return "Woof!";
    }
}
```

Problem 1: We can now create `Animal` objects!

Problem 2: Forgetting to override `makeSound` is a bomb waiting to go off.

Example: Abstract Class (3/3)

The solution is to make `Animal` into an **abstract class** and to declare `makeSound` as an **abstract method**:

```
public abstract class Animal {
    private int age;

    public Animal(int age) {
        this.age = age;
    }

    public int getAge() {
        return this.age;
    }

    public abstract String makeSound();
}
```

```
public class Dog extends Animal {
    public Dog(int age) {
        super(age);
    }

    @Override
    public String makeSound() {
        return "Woof!";
    }
}
```

Solution: We cannot create `Animal` objects, and subclasses **must** implement `makeSound()`. Moreover, we have avoided code duplication.

Overview: Interfaces, Abstract Classes, and Classes

Feature	Interface	Abstract Class	Concrete Class
Can be instantiated?	**No**	**No**	**Yes**
Can have fields?	**No**	**Yes**	**Yes**
Can have constructors?	**No**	**Yes**	**Yes**
Can have abstract methods?	**Yes** (all)	**Yes** (some)	**No**
Can have concrete methods?	**No**	**Yes**	**Yes**
Multiple inheritance?	**Yes** (`implements`)	**No** (`extends`)	**No** (`extends`)
Use case	Define contract	Share code	Implementation

What to use?

The Good: Interfaces and classes that implement them.

- Promotes flexibility and loose coupling.
- Easy to test and maintain.

The Bad: Class-to-class inheritance.

- Creates tight coupling between parent and child.
- Hard to change without breaking subclasses.

The Ugly: Abstract classes.

- Use sparingly and only when you need shared implementation.
- Prefer interfaces when possible.

Q:

What can you say about the following program?

```
public interface Animal {
    String makeSound() {
        return "Heresy!";
    }
}

public class Cat extends Animal {
    public String makeSound() {
        return "Meow!";
    }
}
```

The `Object` Class

In Java, **every class implicitly extends** the `Object` class.

Consequently, every Java class has these methods:

`boolean equals(Object obj)`	compares two objects for equality
`int hashCode()`	returns a hash code value for the object
`String toString()`	returns a string representation of the object

and more ...

The `toString` Method

```
class Point {
    private int x;
    private int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public String toString() {
        return "Point(" + this.x + ",
          " + this.y + ")";
    }
}
```

Without `toString`:

```
Point p = new Point(3, 4);
System.out.println(p);
// Prints: Point@5ca881b5
```

With `toString`:

```
Point p = new Point(3, 4);
System.out.println(p);
// Prints: Point(3, 4)
```

The `equals` Method

```
class Point {
    private int x;
    private int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null) return false;
        if (getClass() != obj.getClass())
            return false;
        Point other = (Point) obj;
        return x == other.x && y == other.y;
    }
}
```

Without `equals`:

```
Point p1 = new Point(3, 4);
Point p2 = new Point(3, 4);

// Prints: true
System.out.println(p1 == p1);

// Prints: false
System.out.println(p1.equals(p2));
```

With `equals`:

```
Point p1 = new Point(3, 4);
Point p2 = new Point(3, 4);
// Prints: true
System.out.println(p1.equals(p2));
```

The `hashCode()` Method

The `hashCode()` method returns an integer hash code for an object.

Key Points:

- Used by hash-based collections (`HashMap`, `HashSet`, etc.)
- **Objects that are equal must have the same hash code**

❗ **Important:** If you override `equals()`, you **must** also override `hashCode()`!

Example: Overriding `equals` and `hashCode`

```java
class Point {
    private int x;
    private int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null) return false;
        if (getClass() != obj.getClass())
            return false;
        Point other = (Point) obj;
        return x == other.x
            && y == other.y;
    }
}
```

```java
class Point {
    // ... continued ...
    public int hashCode() {
        return 3 * x + 7 * y;
    }
}
```

A better way:

```java
class Point {
    // ... continued ...
    public int hashCode() {
        return Objects.hash(x, y);
    }
}
```

Example: Equality and HashCode

```
class Person {
    private String firstName;
    private String lastName;
    private int age = 0;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        Person other = (Person) obj;
        return firstName.equals(other.firstName) && lastName.equals(other.lastName);
    }

    public int hashCode() {
        return Objects.hash(firstName, lastName);
    }
}
```

What can you say about this class?

Q:

```
class Email {
    private String subject;
    private String sender;
    private String text;

    // ... constructors and methods ...

    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        Email other = (Email) obj;
        return subject == other.subject && sender == other.sender;
    }

    public int hashCode() { return Objects.hash(subject, sender); }
}
```

Packages

What is a Package?

A **package** is a namespace mechanism for organizing related classes and interfaces.

Packages provide:

Modularity: Related types are grouped into cohesive units.

Namespaces: Classes with the same name can coexist in different packages.

Access control: Package-private visibility controls member accessibility.

Package names use *reverse domain names* to ensure global uniqueness:

- The domain `google.com` becomes the package `com.google`.
- The domain `cs.au.dk` becomes the package `dk.au.cs`.

Example: Declaring and Using Packages

We can declare a class in a package:

```java
package dk.au.cs;

public class Circle {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public double getArea() {
        return Math.PI * radius * radius;
    }
}
```

The class must be stored in `dk/au/cs/Circle.java`.

Example: Declaring and Using Packages

We can declare a class in a package:

```java
package dk.au.cs;

public class Circle {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public double getArea() {
        return Math.PI * radius * radius;
    }
}
```

The class must be stored in `dk/au/cs/Circle.java`.

and then we can then **import** it:

```java
import dk.au.cs.Circle;

public class Main {
    public static void main(String[] args) {
        Circle c = new Circle(5.0);
        System.out.println(c.getArea());
    }
}
```

The `Main` class can be stored in any package.

Example: Wildcard Imports

We can import as many classes as we want:

```
import dk.au.cs.Circle;
import dk.au.cs.Rectangle;
import dk.au.cs.Triangle;
```

Or use a **wildcard import**:

```
import dk.au.cs.*;
```

Name Clash

What if two packages have classes with the same name?

```
import dk.au.cs.game.Pool;
import dk.au.cs.swim.Pool;

public class Main {
    public static void main(String[] args) {
        Pool p = new Pool();  // Which Pool?
    }
}
```

```
Error: Pool is already defined
```

Resolving Name Clashes

We can use **fully qualified names** to resolve the clash:

```
import dk.au.cs.game.Pool;

public class Main {
    public static void main(String[] args) {
        Pool gamePool = new Pool();
        dk.au.cs.swim.Pool swimPool = new dk.au.cs.swim.Pool();
    }
}
```

Epilogue

Error of the Week (1/3)

What is wrong here?

```
public interface Shape {
    double getX();
    double getY();
    void draw();
}

public class Rectangle implements Shape {
    private double x;
    private double y;

    public double getX() { return x; }
    public double getY() { return y; }
    public void darw() { /* ... */ }
}
```

Error of the Week (1/3)

What is wrong here?

```
public interface Shape {
    double getX();
    double getY();
    void draw();
}

public class Rectangle implements Shape {
    private double x;
    private double y;

    public double getX() { return x; }
    public double getY() { return y; }
    public void darw() { /* ... */ }
}
```

```
Rectangle is not abstract
and does not override
abstract method draw() in
Shape
```

Error of the Week (2/3)

What is wrong here?

```
public interface Movable {
    void move(double dx, double dy);
}

public class Ball implements Movable {
    public void move(int dx, int dy) {
        // ...
    }
}
```

Error of the Week (2/3)

What is wrong here?

```
public interface Movable {
    void move(double dx, double dy);
}

public class Ball implements Movable {
    public void move(int dx, int dy) {
        // ...
    }
}
```

```
Ball is not abstract and
does not override abstract
method move(double,double)
in Movable
```

Error of the Week (3/3)

What is wrong here?

```
public abstract class Animal {
    private String name;
    private int age;

    public Animal(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

public class Giraffe extends Animal { }

public class Elephant extends Animal { }
```

Error of the Week (3/3)

What is wrong here?

```
public abstract class Animal {
    private String name;
    private int age;

    public Animal(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

public class Giraffe extends Animal { }

public class Elephant extends Animal { }
```

```
There is no default
constructor available
in Animal

There is no default
constructor available
in Animal
```

Live Programming

Live Programming

- Interfaces, Abstract Classes, and Classes
  - Assignability.
  - Dynamic method dispatch.
- The `@Override` annotation and `instanceof` operator.
- Example: CalendarEvent, Birthday, Meeting, PhysicalMeeting, OnlineMeeting