Algorithms and Data Structures 1
CS 0445

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Lab 0 is due this Friday (not graded)
- Recitations start next week
- Homework 1 will be assigned this Friday
- JDB Example will be available on Canvas
- Draft slides and handouts available on Canvas

Today's Agenda

Java Review

- Shallow, deep, and deeper copying of objects
- Inheritance
  - Polymorphism

- Abstract Data Types
  - Java Interfaces
  - Generics
- File Operations

Invoking Constructors from Within Constructors

- Constructors typically initialize a class's data fields
- To call constructor of superclass explicitly:
  - Use **super( )** within definition of a constructor of a subclass
- If you omit **super( )**
  - Constructor of subclass automatically calls default constructor of superclass.

Invoking Constructors from Within Constructors

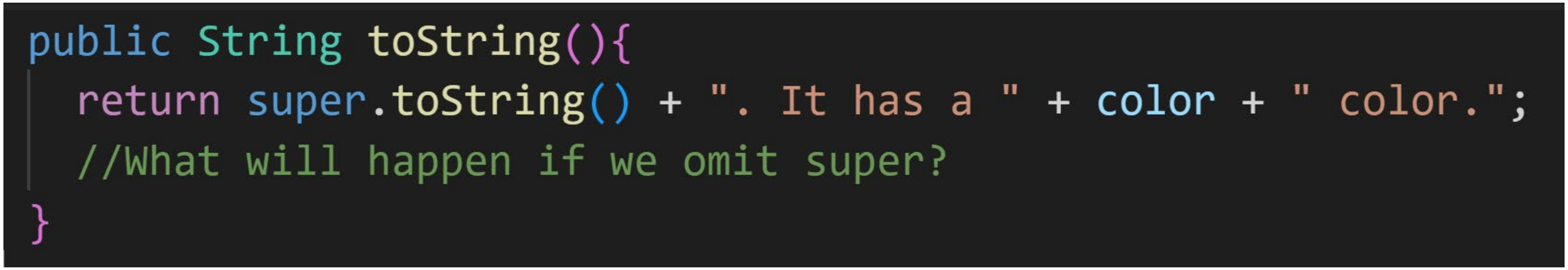
- Also possible to use **this** to invoke constructor of superclass
    - The subclass must not have a constructor with the same parameter list; otherwise, the subclass constructor will be called

Overriding and Overloading Methods

- When a subclass defines a method with
  - the same name
  - the same number and types of parameters
  - and the same return type as a method in the superclass …
- Example: toString() in Square and ColoredSquare
- Then, definition in the subclass is said to *override* the definition in the superclass
- You can use **super** in a subclass to call an overridden method of the superclass.
  - Check definition of toString in ColoredSquare

Overriding and Overloading Methods

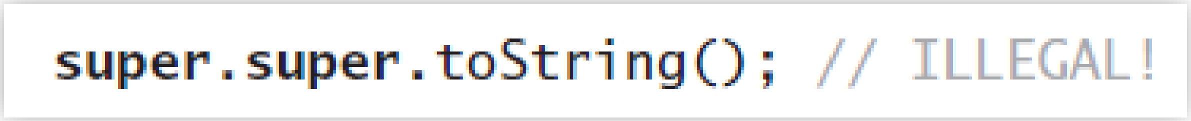
- Possible to have new method invoke the inherited method
  - Need to distinguish between the method for subclass and method from superclass



```java
public String toString(){
  return super.toString() + ". It has a " + color + " color.";
  //What will happen if we omit super?
}
```

Overriding and Overloading Methods

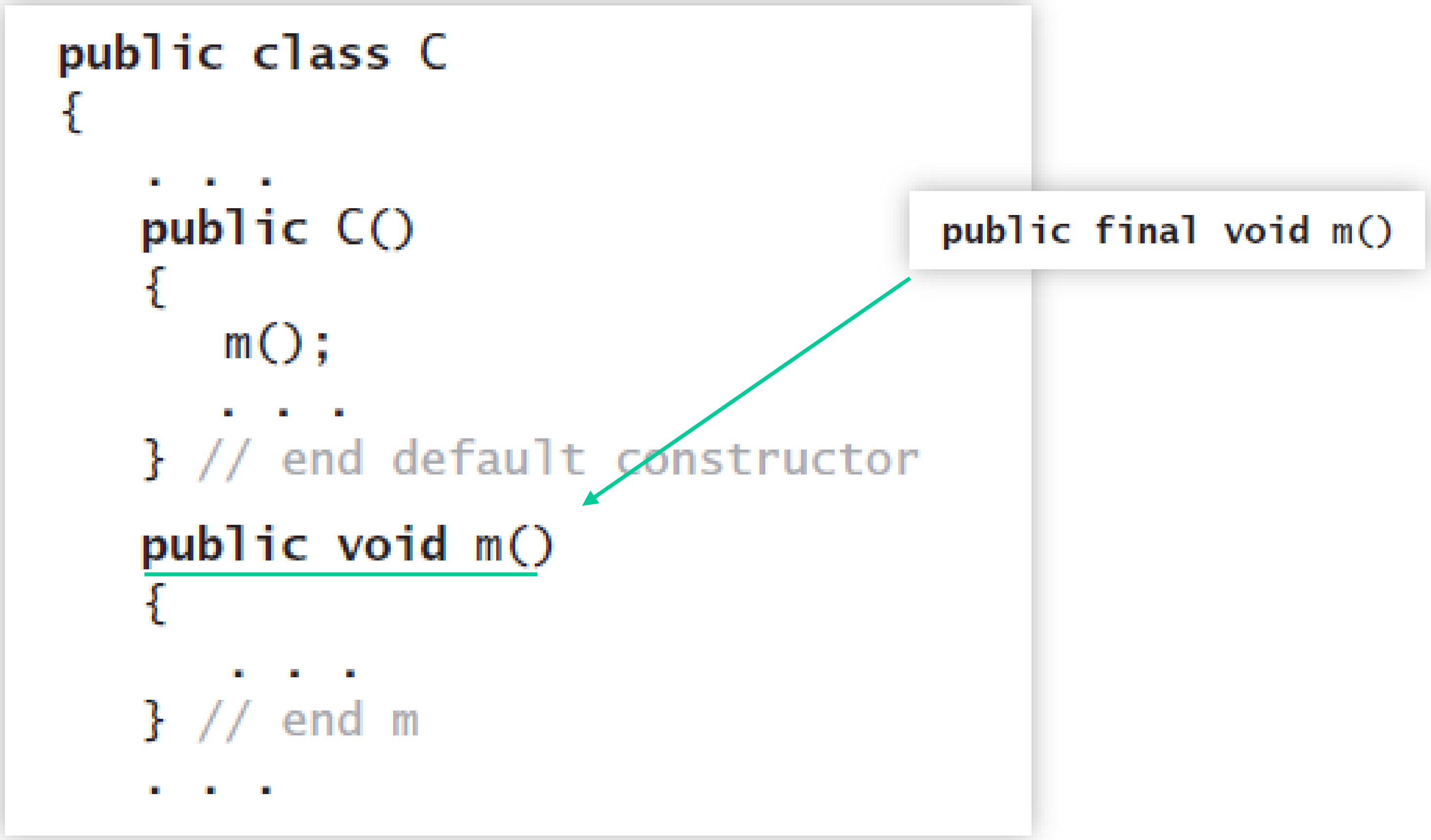
- But … repeated use of **super** is not allowed



```
super.super.toString(); // ILLEGAL!
```

Overriding and Overloading Methods

- To specify that a method definition cannot be overridden with a new definition in a subclass
  - Make it a final method by adding the **final** modifier to the method header.



```
public class C
{
   . . .
   public C()
   {
      m();
      . . .
   } // end default constructor
   public void m()
   {
      . . .
   } // end m
   . . .
```

```
public final void m()
```

The Class **Object**

- Java has a class—named **Object**
  - It is at the beginning of every chain of subclasses
  - An ancestor of every other class
- Class **Object** contains certain methods
  - Examples: **toString**, **equals**, **clone**
  - However, in most cases, you must override these methods

toString()

- Need to override the definition of **toString**
    - Cause it to produce an appropriate string for data in the class being defined

equals

- Object’s **equals** method compares the **addresses** of two objects
  - Overridden method, when added to the class **Square**, detects whether two **Square** objects are equal by comparing their data fields.
  - Check equals() method inside Square and ColoredSquare

clone

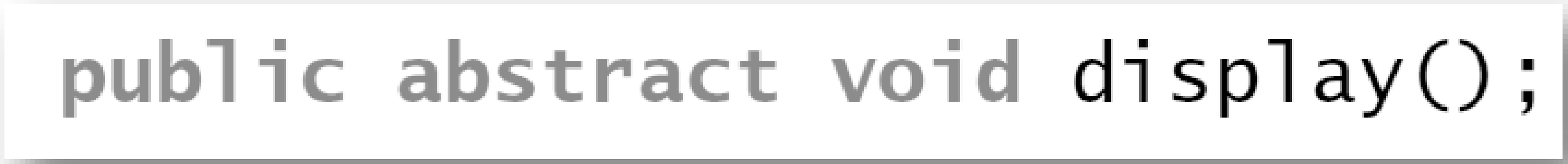
- A Method of the Class **Object**
  - Takes no arguments and returns a copy of the receiving object (this)
  - Check clone inside Square under Take2

Overloading Methods

- When subclass has a method with same name as a method in its superclass,
    - but the methods' parameters differ in number or data type …
- Method in subclass overloads method of superclass.
    - Java is able to distinguish between these methods
    - Signatures of the methods are different

Abstract Classes and Methods

- An abstract class will be the superclass of another class
- Thus, an abstract class is sometimes called an abstract superclass
- Declare abstract method by including reserved word abstract in header



```
public abstract void display();
```

Abstract Classes and Methods

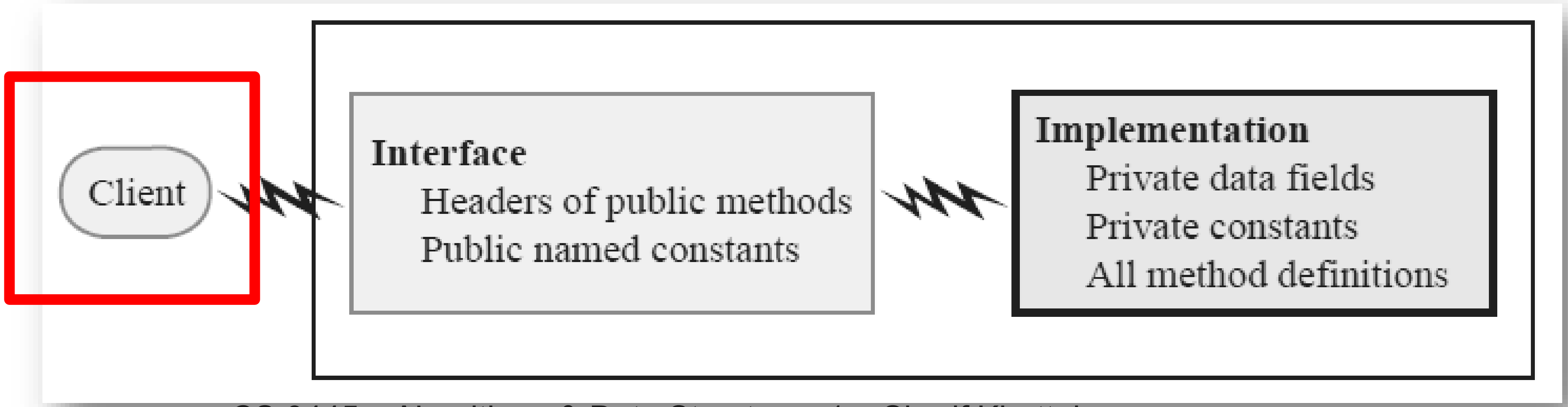
- Abstract method cannot be private, static, or final.
- Class with at least one abstract method must be declared as an abstract class
    - Abstract methods can appear only within an abstract class.
- Constructors cannot be abstract

Multiple Inheritance

- Some programming languages allow one class to be derived from two different super classes
  - This feature is not allowed in Java
- In Java, a subclass can have only one superclass
- Java Interfaces allow for multiple inheritance
- Let’s review Java Interfaces

Abstraction

- Focus on *what* instead of *how*
  - What needs to be done?
  - For the moment ignore how it will be done.
- Divide class into two parts
  - Interface
    - provides well-regulated communication between a hidden implementation and a client
  - Implementation



Specifying Method Headers in Interfaces

- **Preconditions**
  - What must be true before method executes
  - Implies responsibility for client
- **Postconditions**
  - Statement of what is true after method executes
  - Usually about the return value(s)
- Use **assertions**
  - In comments or with `assert` statement

Java Interfaces

- Program component that declares one or more public methods
    - Should include comments to inform programmer
    - Any data fields here should be `public`, `final`, `static`
    - May have `default` methods

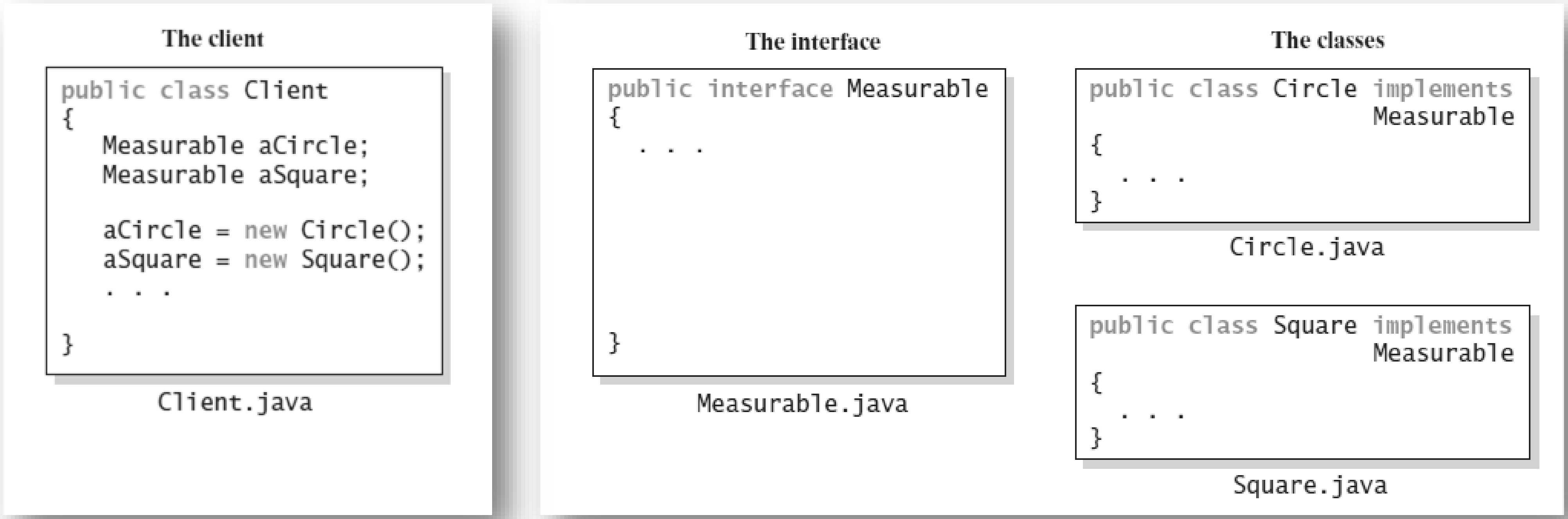
Named Constants Within an Interface

- An interface can contain named constants
  - Public data fields that you initialize and declare as final.
- Options:
  - Define the constants in an interface that the classes implement
  - Define your constants in a separate class instead of an interface

Implementing an Interface

- A way for programmer to guarantee a class has certain methods
- Several classes can implement the same interface
- A class can implement more than one interface
  - A form of multiple inheritance

Interface *vs.* Implementation *vs.* Client



The client

```
public class Client
{
   Measurable aCircle;
   Measurable aSquare;

   aCircle = new Circle();
   aSquare = new Square();
   . . .

}
```

Client.java

The interface

```
public interface Measurable
{
   . . .



}
```

Measurable.java

The classes

```
public class Circle implements
                    Measurable
{
   . . .
}
```

Circle.java

```
public class Square implements
                    Measurable
{
   . . .
}
```

Square.java

an interface, two implementations, and a client

Interface as a Data Type

- You can use a Java interface as you would use a data type
- Indicates that the variable can invoke a certain set of methods and only those methods.
- An interface type is a reference type
- An interface can be used to derive another interface by using inheritance

Interface vs. Abstract Class

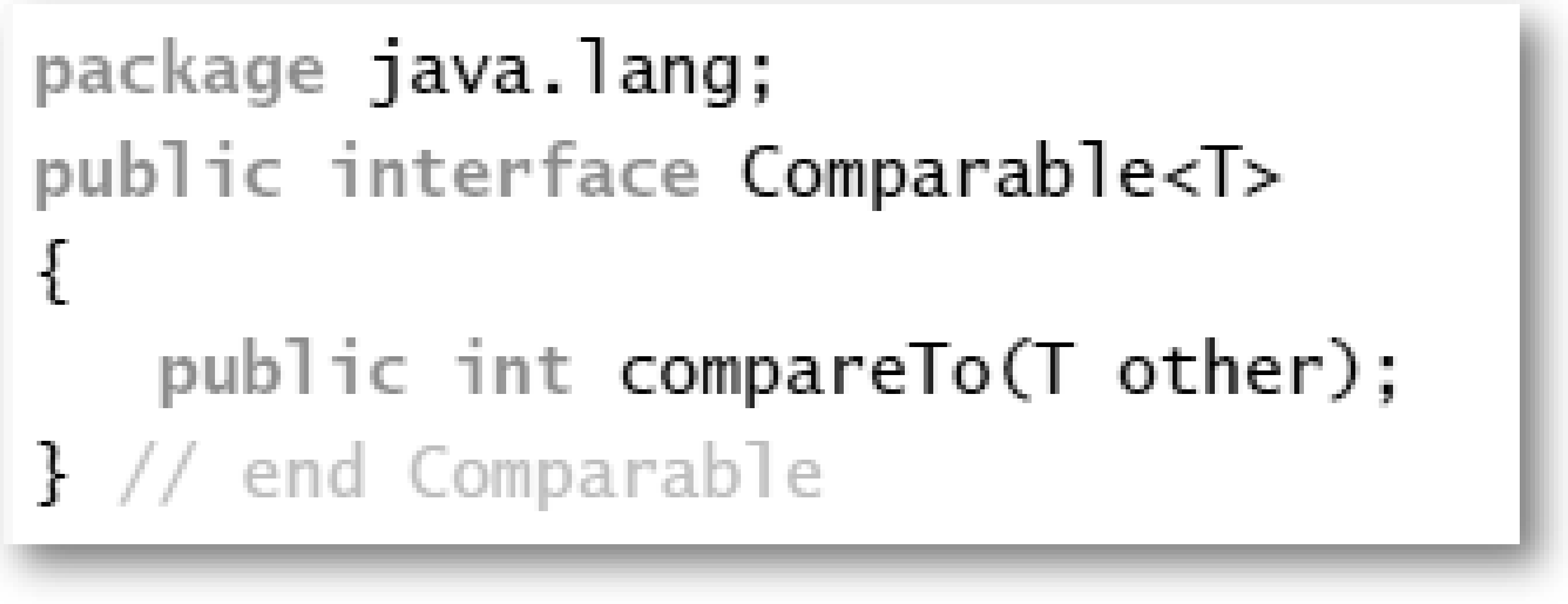
- Purpose of interface similar to that of abstract class
  - But an interface is *not* a class
- Use an abstract class …
  - If you want to declare a data field that your subclasses will have in common
- A class can implement several interfaces but can extend only one abstract class

Generic Data Types

- Enable you to write a placeholder instead of an actual class type
- The placeholder is called a type parameter
- Library developer defines a generic class
  - Client chooses data type of the objects in collection.
- Interfaces can be generic as well!

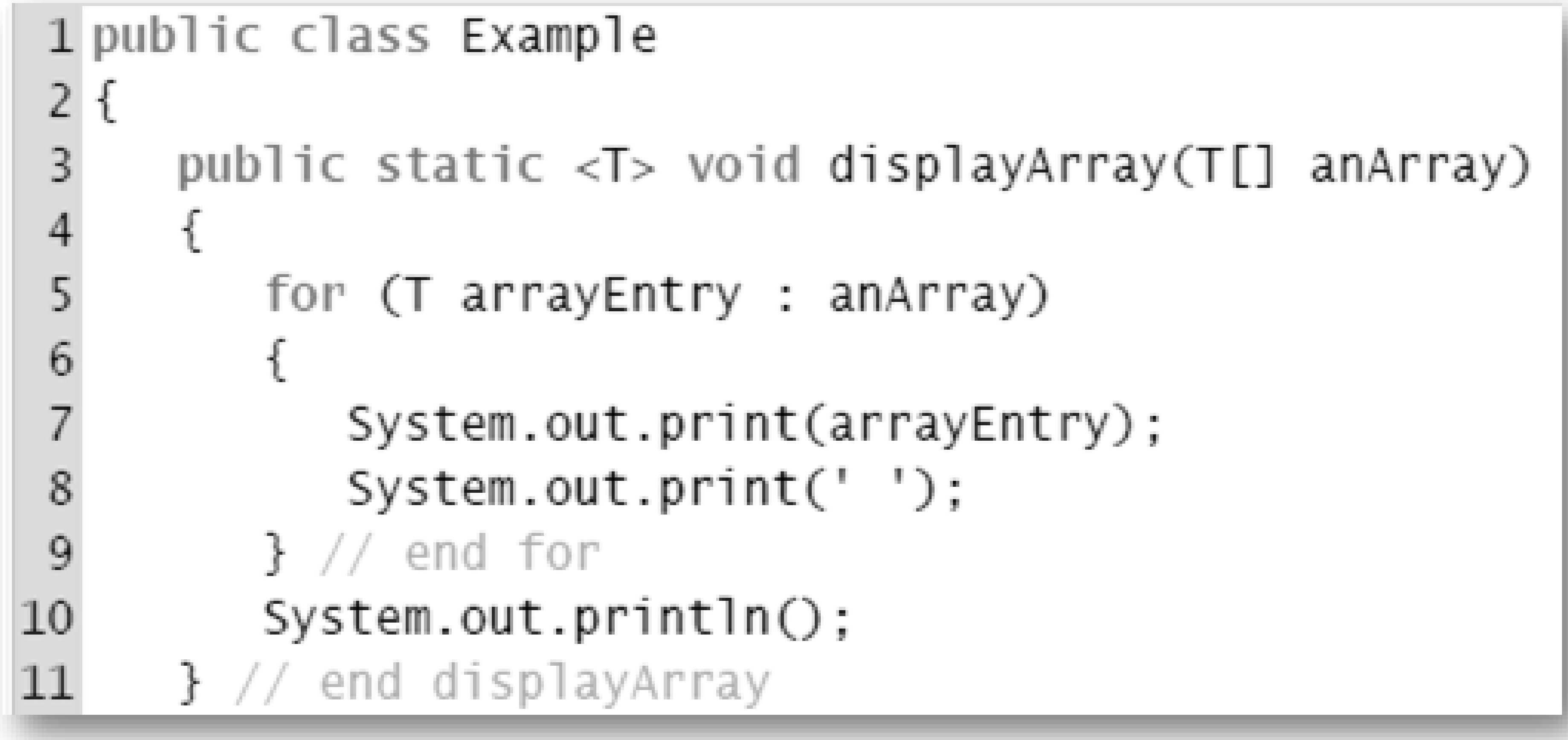
Generic Interface `Comparable`

- By invoking **`compareTo`**, you compare two objects of the class **`T`**. compareTo returns:
  - Negative: *this* < *other*
  - Zero: if *this* and *other* are equal
  - Positive: if *this* > *other*



```
package java.lang;
public interface Comparable<T>
{
    public int compareTo(T other);
} // end Comparable
```

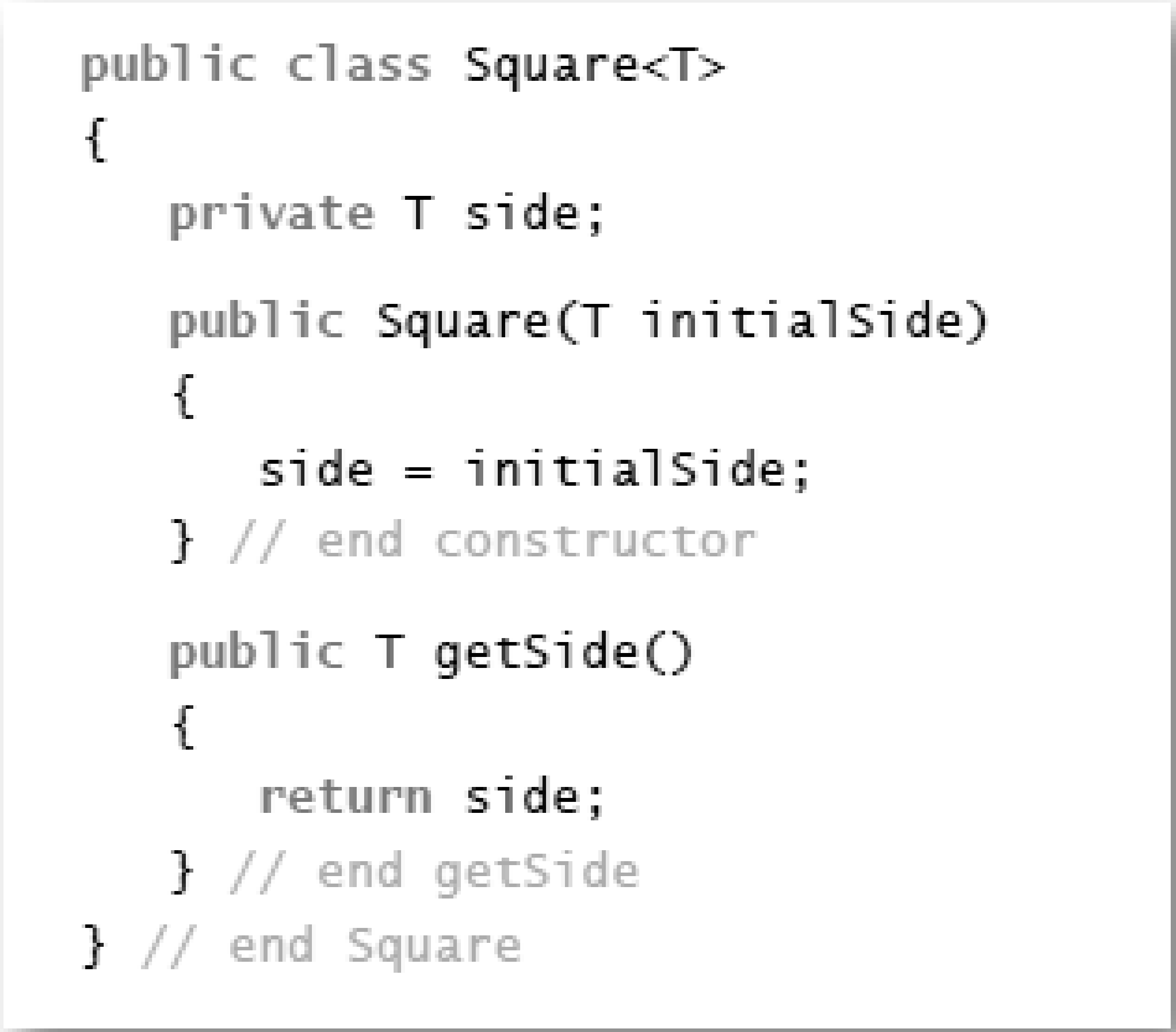
Generic Methods



```
public class Example
{
    public static <T> void displayArray(T[] anArray)
    {
        for (T arrayEntry : anArray)
        {
            System.out.print(arrayEntry);
            System.out.print(' ');
        } // end for
        System.out.println();
    } // end displayArray
```

Bounded Type Parameters

- Consider this simple class of squares:



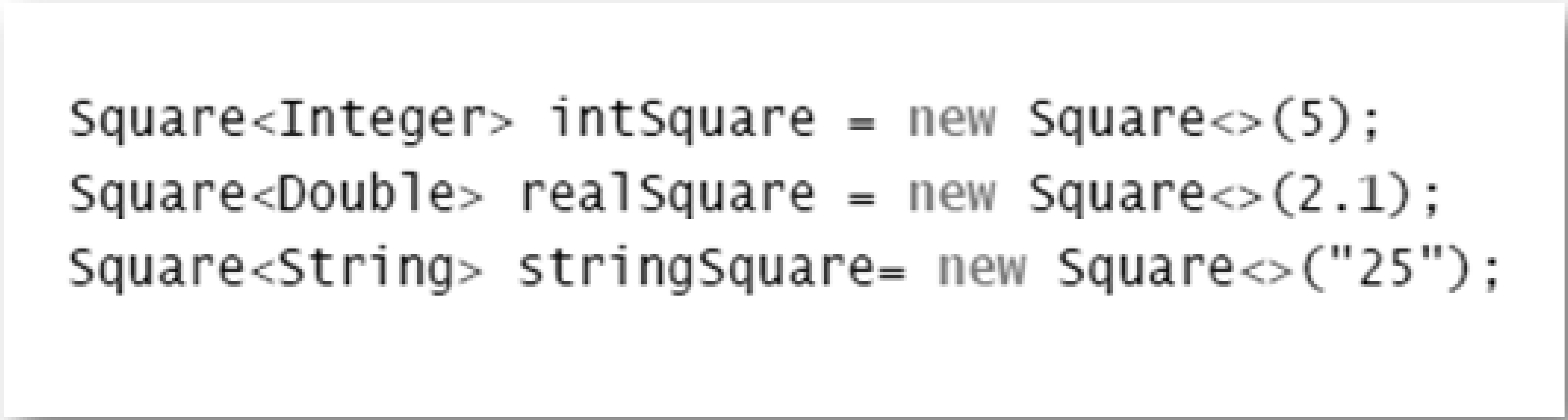
```
public class Square<T>
{
   private T side;

   public Square(T initialSide)
   {
      side = initialSide;
   } // end constructor

   public T getSide()
   {
      return side;
   } // end getSide
} // end Square
```

Bounded Type Parameters

- What is wrong here?



```
Square<Integer> intSquare = new Square<>(5);
Square<Double> realSquare = new Square<>(2.1);
Square<String> stringSquare= new Square<>("25");
```