THE TWO MEANINGS OF INTERFACE

JAVA INTERFACES VERSUS GRAPHIC USER INTERFACES

Tate Redding

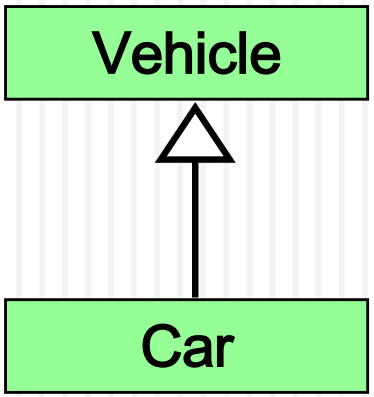
IS413 GUI Systems using Java

Java Inheritance and Interfaces

- Chapter 11 deals with Inheritance, Chapter 13 with Interfaces
- Inheritance Revisited
  - Inheritance is a fundamental object-oriented design technique used to create and organize reusable classes
- *Inheritance* allows a software developer to derive a new class from an existing one
- The existing class is called the *parent class,* or *superclass*, or *base class*
- The derived class is called the *child class* or *subclass*
- As the name implies, the child inherits characteristics of the parent
- That is, the child class inherits the methods and data defined by the parent class

Inheritance

- Inheritance relationships are shown in a UML class diagram using a solid arrow with an unfilled triangular arrowhead pointing to the parent class



- Proper inheritance creates an *is-a* relationship, meaning the child *is a* more specific version of the parent

Inheritance

- A programmer can tailor a derived class as needed by adding new variables or methods, or by modifying the inherited ones
- *Software reuse* is a fundamental benefit of inheritance
- By using existing software components to create new ones, we capitalize on all the effort that went into the design, implementation, and testing of the existing software

Deriving Subclasses

- In Java, we use the reserved word `extends` to establish an inheritance relationship

```
class Car extends Vehicle {
   // class contents }
```

Visibility modifiers affect the way that class members can be used in a child class. Variables and methods declared with *private visibility cannot be referenced by name in a child class*.

They can be referenced in the child class if they are declared with public visibility -- but *public variables violate the principle of encapsulation*

There is a third visibility modifier that helps in inheritance situations: `protected`

The protected Modifier

- The `protected` modifier allows a child class to reference a variable or method directly in the child class
- It provides more encapsulation than public visibility, but is not as tightly encapsulated as private visibility
- A protected variable is visible to any class in the same package as the parent class
- Protected variables and methods can be shown with a # symbol preceding them in UML diagrams

The super Reference

- Constructors are not inherited, even though they have public visibility
- Yet we often want to use the parent's constructor to set up the "parent's part" of the object
- The `super` reference can be used to refer to the parent class, and often is used to invoke the parent's constructor
- A child's constructor is responsible for calling the parent's constructor
- The first line of a child's constructor should use the `super` reference to call the parent's constructor
- The `super` reference can also be used to reference other variables and methods defined in the parent's class

Overriding Methods

- A child class can *override* the definition of an inherited method in favor of its own
- The new method must have the same signature as the parent's method, but can have a different body
- The type of the object executing the method determines which version of the method is invoked
- A method in the parent class can be invoked explicitly using the `super` reference
- If a method is declared with the `final` modifier, it cannot be overridden
- The concept of overriding can be applied to data and is called *shadowing variables*
- Shadowing variables should be avoided because it tends to cause unnecessarily confusing code

Overloading vs. Overriding

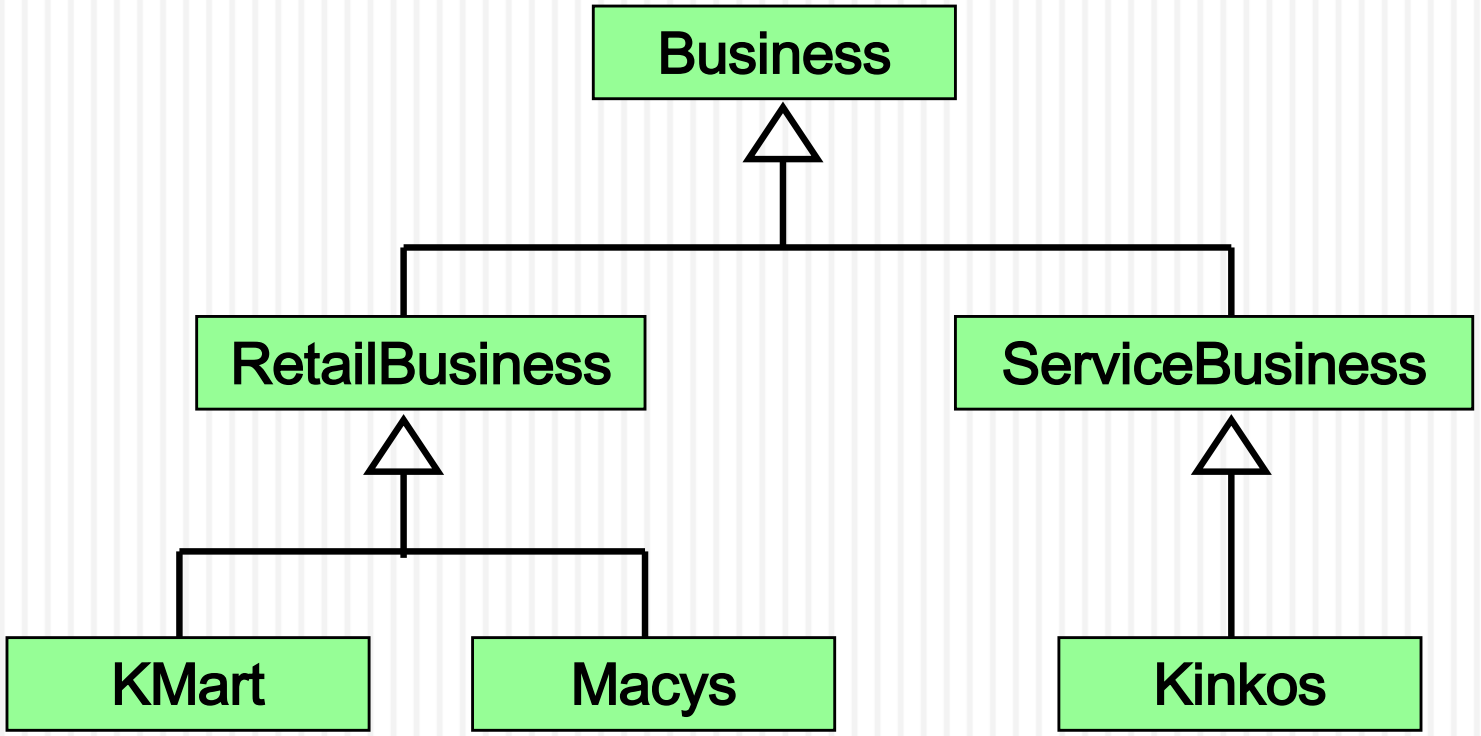
- Overloading deals with multiple methods with the same name in the same class, but with different signatures
- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature
- Overloading lets you define a similar operation in different ways for different parameters
- Overriding lets you define a similar operation in different ways for different object types

Multiple Inheritance-NO Interfaces- YES!

- Java supports *single inheritance*, meaning that a derived class can have only one parent class
- *Multiple inheritance* allows a class to be derived from two or more classes, inheriting the members of all parents. Collisions, such as the same variable name in two parents, have to be resolved.
- Java does not support multiple inheritance
- In most cases, the use of interfaces gives us aspects of multiple inheritance without the overhead

Class Hierarchies

- A child class of one parent can be the parent of another child, forming a *class hierarchy*



Abstract Classes

- An *abstract class* is a placeholder in a class hierarchy that represents a generic concept
- An abstract class cannot be instantiated
- We use the modifier `abstract` on the class header to declare a class as abstract:

```
public abstract class Product
{
    // contents
}
```

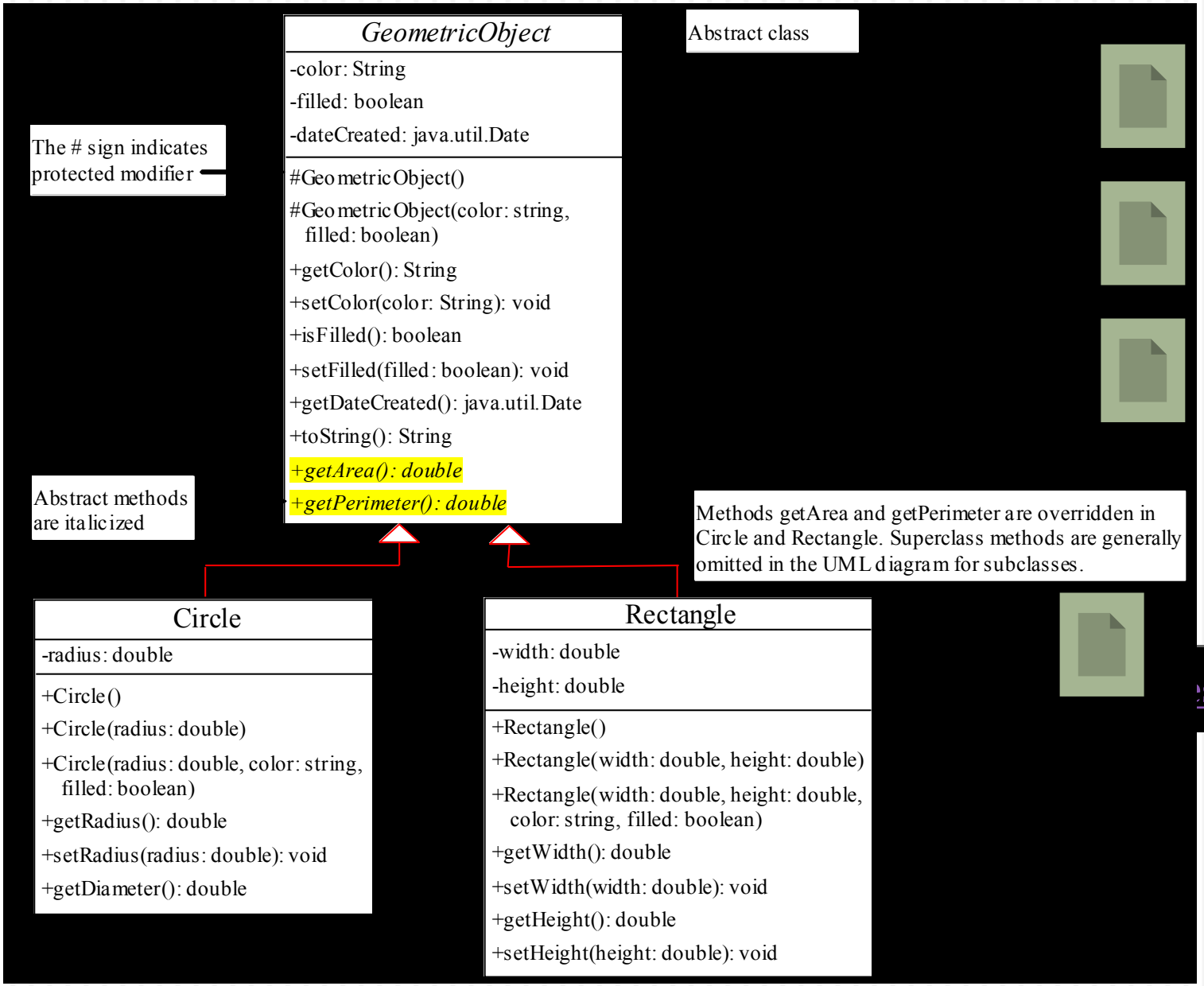
Abstract Classes

- An abstract class often contains abstract methods with no definitions (like an interface)
- Unlike an interface, the `abstract` modifier must be applied to each abstract method
- Also, an abstract class typically contains non-abstract methods with full definitions
- A class declared as abstract does not have to contain abstract methods -- simply declaring it as abstract makes it so

Abstract Classes

- The child of an abstract class must override the abstract methods of the parent, or it too will be considered abstract
- An abstract method cannot be defined as `final` or `static`
- The use of abstract classes is an important element of software design – it allows us to establish common elements in a hierarchy that are too generic to instantiate

Abstract Classes and Abstract Methods



GeometricObject

Circle

Rectangle

stGeometricObject

object cannot be created from abstract class

An abstract class cannot be instantiated using the new operator, but you can still define its constructors, which are invoked in the constructors of its subclasses. For instance, the constructors of GeometricObject are invoked in the Circle class and the Rectangle class.

16

abstract class without abstract method

A class that contains abstract methods must be abstract. However, it is possible to define an abstract class that contains no abstract methods. In this case, you cannot create instances of the class using the new operator. This class is used as a base class for defining a new subclass.

17

abstract class as type

You cannot create an instance from an abstract class using the new operator, but an abstract class can be used as a data type. Therefore, the following statement, which creates an array whose elements are of GeometricObject type, is correct.

```
GeometricObject[] geo = new    GeometricObject[10];
```


18

Interface Hierarchies

- Inheritance can be applied to interfaces as well as classes
- That is, one interface can be derived from another interface
- The child interface inherits all abstract methods of the parent
- A class implementing the child interface must define all methods from both the ancestor and child interfaces
- Note that class hierarchies and interface hierarchies are distinct (they do not overlap)

Java Interfaces

What is an interface?

Why is an interface useful?

How do you define an interface?

How do you use an interface?

What is an interface? Why is an interface useful?

An interface is a classlike construct that contains only constants and abstract methods. In many ways, an interface is similar to an abstract class, but the intent of an interface is to specify common behavior for objects. For example, you can specify that the objects are comparable, edible, cloneable using appropriate interfaces.

21

Define an Interface

To distinguish an interface from a class, Java uses the following syntax to define an interface:

```
public interface InterfaceName {
  constant declarations;
  method signatures;
}
```

Example:

```
public interface Edible {
  /** Describe how to eat */
  public abstract String howToEat();
}
```


22

Interface is a Special Class

An interface is treated like a special class in Java. Each interface is compiled into a separate bytecode file, just like a regular class. Like an abstract class, you cannot create an instance from an interface using the new operator, but in most cases you can use an interface more or less the same way you use an abstract class. For example, you can use an interface as a data type for a variable, as the result of casting, and so on.

23

Omitting Modifiers in Interfaces

All data fields are *public final static* and all methods are *public abstract* in an interface. For this reason, these modifiers can be omitted, as shown below:

```
public interface T1 {
  public static final int K = 1;

  public abstract void p();
}
```

Equivalent

```
public interface T1 {
  int K = 1;

  void p();
}
```

A constant defined in an interface can be accessed using syntax InterfaceName.CONSTANT_NAME (e.g., T1.K).

25

Interfaces vs. Abstract Classes

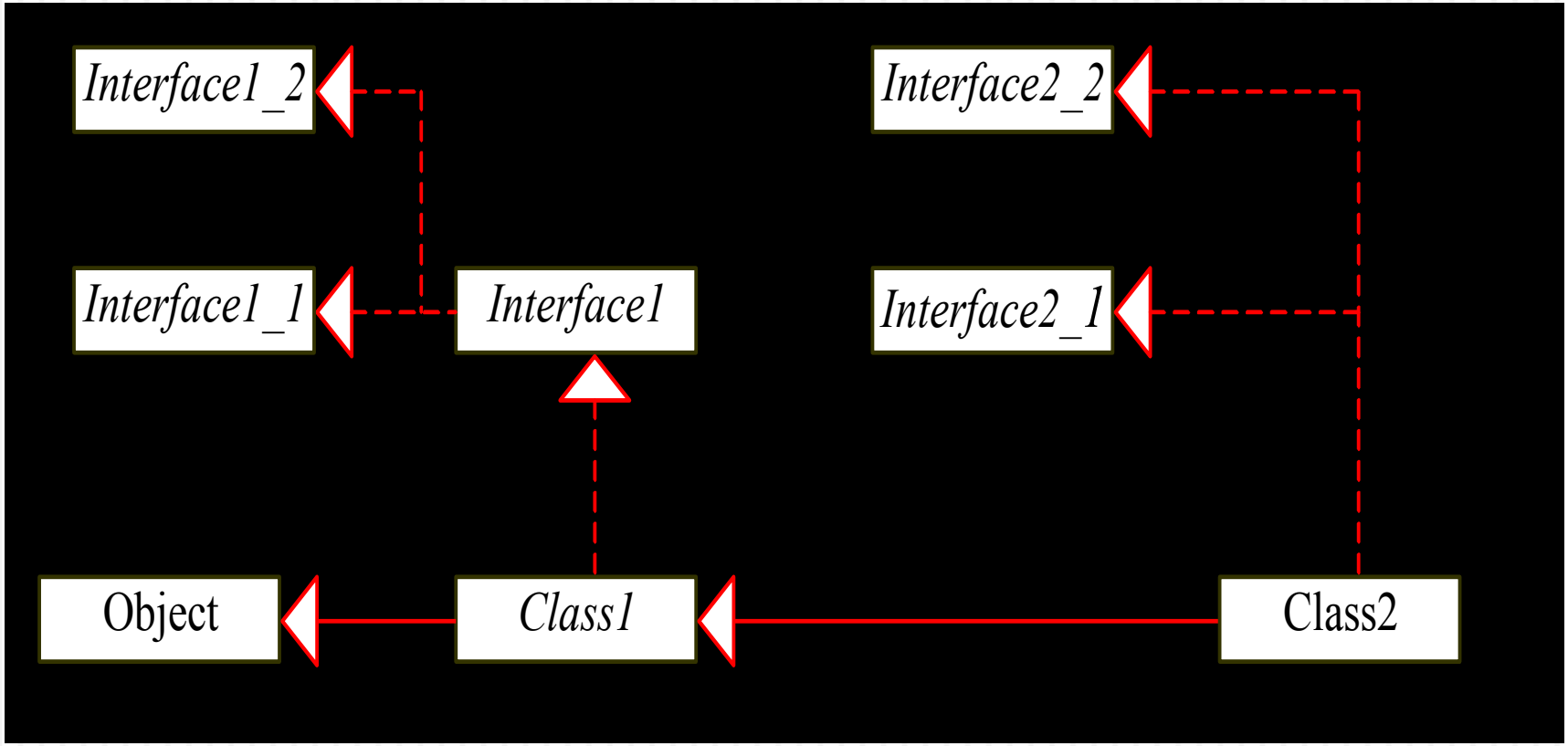
In an interface, the data must be constants; an abstract class can have all types of data.

Each method in an interface has only a signature without implementation; an abstract class can have concrete methods.

	Variables	Constructors	Methods
Abstract class	No restrictions	Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the new operator.	No restrictions.
Interface	All variables must be public static final	No constructors. An interface cannot be instantiated using the new operator.	All methods must be public abstract instance methods

26

Java Interfaces



Whether to use an interface or a class?

Abstract classes and interfaces can both be used to model common features.

In general, a strong is-a relationship that clearly describes a parent-child relationship should be modeled using classes- a staff member is a person so their relationship should be class inheritance.

A weak is-a relationship or is-kind-of relationship can be modeled using interfaces. For example, all strings are comparable, so the String class implements the Comparable interface.

Interfaces can be used to circumvent single inheritance restriction if multiple inheritance is desired. In the case of multiple inheritance, you have to design one as a superclass, and others as interface. See

Example: The Comparable Interface

```
// This interface is defined in
// java.lang package
package java.lang;

public interface Comparable<E> {
  public int compareTo(E o);
}
```


29

The toString, equals, and hashCode Methods

Each wrapper class overrides the toString, equals, and hashCode methods defined in the Object class. Since all the numeric wrapper classes and the Character class implement the Comparable interface, the compareTo method is implemented in these classes.

30

Integer and BigInteger Classes

```
public class Integer extends Number
    implements Comparable<Integer> {
  // class body omitted

  @Override
  public int compareTo(Integer o) {
    // Implementation omitted
  }
}
```

```
public class BigInteger extends Number
    implements Comparable<BigInteger> {
  // class body omitted

  @Override
  public int compareTo(BigInteger o) {
    // Implementation omitted
  }
}
```

String and Date Classes

```
public class String extends Object
    implements Comparable<String> {
  // class body omitted

  @Override
  public int compareTo(String o) {
    // Implementation omitted
  }
}
```

```
public class Date extends Object
    implements Comparable<Date> {
  // class body omitted

  @Override
  public int compareTo(Date o) {
    // Implementation omitted
  }
}
```


31

The `Cloneable` Interfaces

Marker Interface: An empty interface.

A marker interface does not contain constants or methods. It is used to denote that a class possesses certain desirable properties. A class that implements the Cloneable interface is marked cloneable, and its objects can be cloned using the clone() method defined in the Object class.

```
package java.lang;
public interface Cloneable {
}
```


33

Examples

Many classes (e.g., Date and Calendar) in the Java library implement Cloneable. Thus, the instances of these classes can be cloned. For example, the following code

```
Calendar calendar = new GregorianCalendar(2003, 2, 1);
Calendar calendarCopy = (Calendar)calendar.clone();
System.out.println("calendar == calendarCopy is " +
  (calendar == calendarCopy));
System.out.println("calendar.equals(calendarCopy) is " +
  calendar.equals(calendarCopy));
```

displays

calendar == calendarCopy is false

calendar.equals(calendarCopy) is true

34

Implementing Cloneable Interface

To define a custom class that implements the Cloneable interface, the class must override the clone() method in the Object class. The following code defines a class named House that implements Cloneable and Comparable.

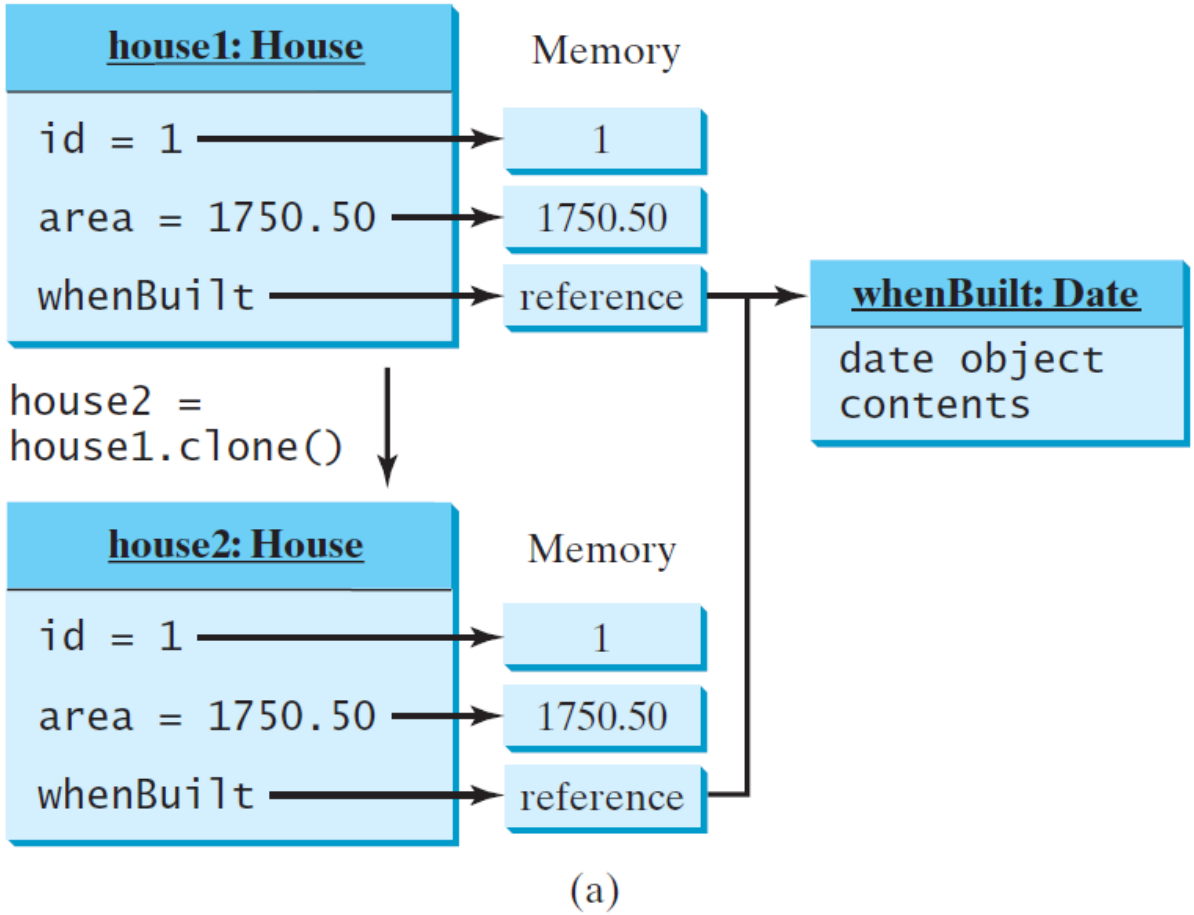
35

Shallow vs. Deep Copy

House house1 = new House(1, 1750.50);

House house2 = (House)house1.clone();

Shallow Copy

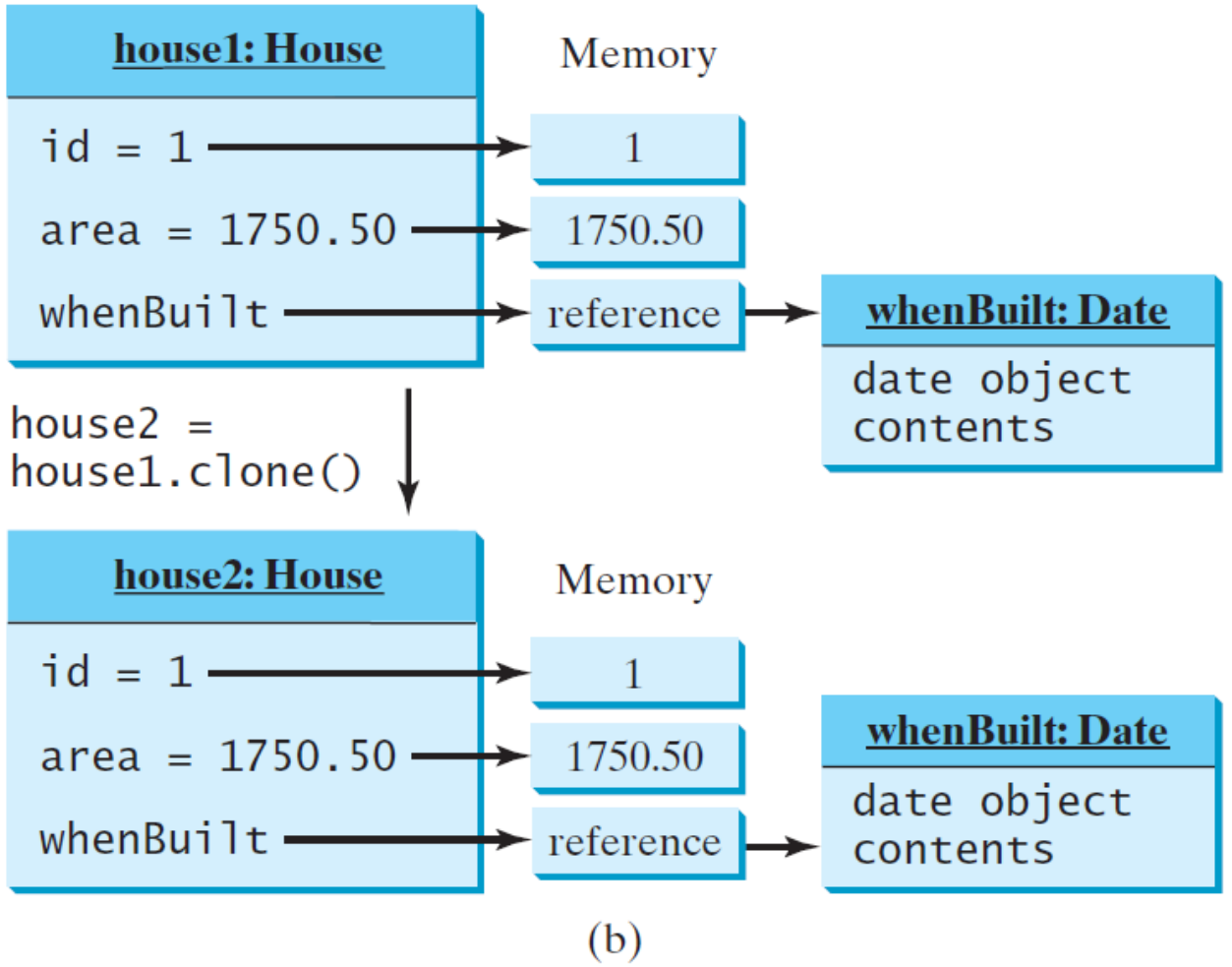


Shallow vs. Deep Copy

House house1 = new House(1, 1750.50);

House house2 = (House)house1.clone();

Deep Copy



Graphic User Interfaces

When we talk about GUI's we are often referring to how the users of our programs interact with an application.

For this course you will be writing programs that may or may not include Java Interfaces to implement User Interfaces.

We will use the tools of the Java graphics classes to create more exciting and richer user experiences.

Later we will apply these concepts to creating Android apps as the Graphic Interface for underlying Java code.

GUI Design

- Keep in mind our goal is to solve a problem
- Fundamental ideas of good GUI design include
  - knowing the user
  - preventing user errors
  - optimizing user abilities
  - being consistent
- We should design interfaces so that the user can make as few mistakes as possible

Know the User

- Knowing the user implies an understanding of:
  - the user's true needs
  - the user's common activities
  - the user's level of expertise in the problem domain and in computer processing
- We should also realize these issues may differ for different users
- Remember, to the user, the interface is the program

Prevent User Errors

- Whenever possible, we should design user interfaces that minimize possible user mistakes
- We should choose the best GUI components for each task
- For example, in a situation where there are only a few valid options, using a menu or radio buttons would be better than an open text field
- Error messages should guide the user appropriately

Optimize User Abilities

- Not all users are alike – some may be more familiar with the system than others
- Knowledgeable users are sometimes called *power users*
- We should provide multiple ways to accomplish a task whenever reasonable
  - "wizards" to walk a user through a process
  - short cuts for power users
- Help facilities should be available but not intrusive

Be Consistent

- Consistency is important – users get used to things appearing and working in certain ways
- Colors should be used consistently to indicate similar types of information or processing
- Screen layout should be consistent from one part of a system to another
- For example, error messages should appear in consistent locations

Assignment- 2

- By the next class
- Exercises 13.1,3,5,7,9,11,13,15
  - See BlackBoard for hints and downloads.
  - Read Chapter 14 and complete CheckPoints and Self Test Reviews