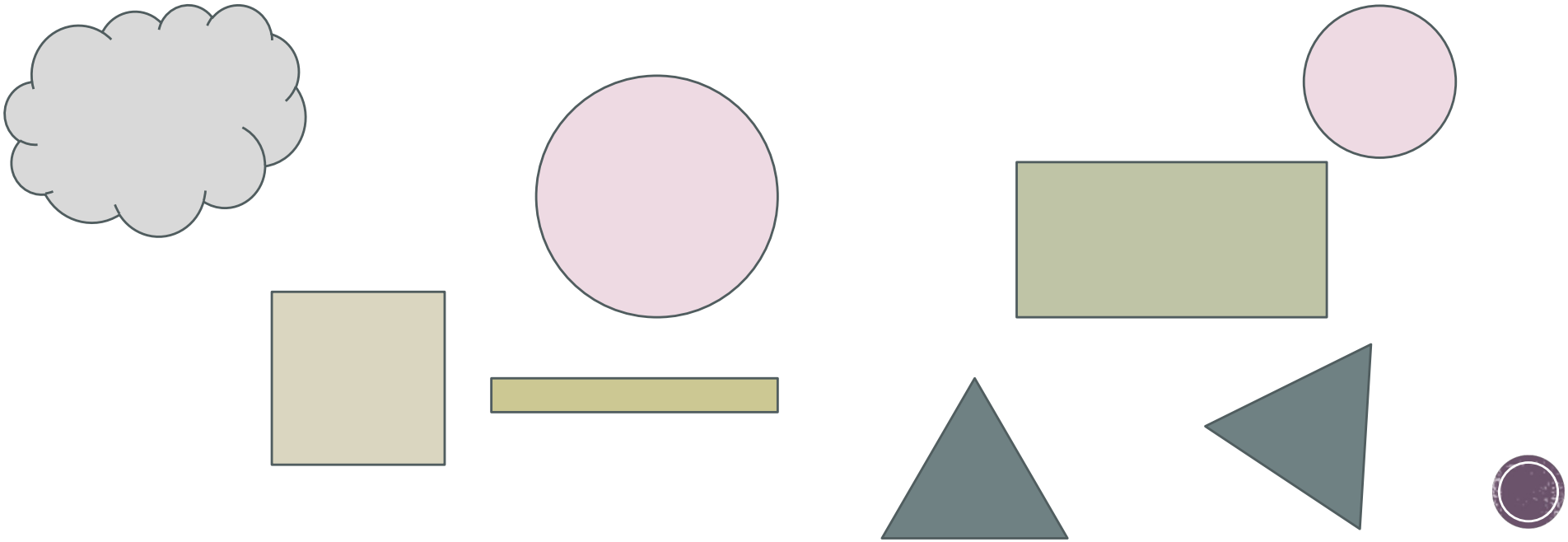

Inheritance
Reusability of Object-Oriented Programs

Simulating Geometric Shapes

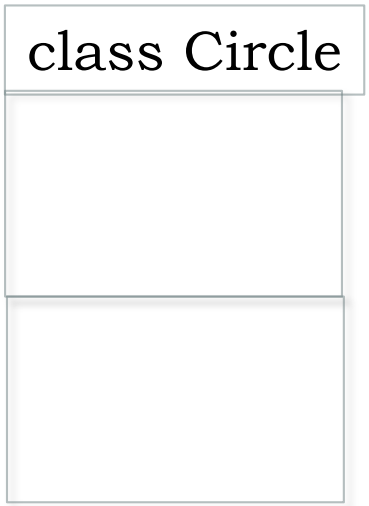
- The “canonical” example, introduced by Alan Kay, 1967
- Basic idea: Reuse code by creating a hierarchy of classes
- Common features (variables, methods) move UP in the hierarchy
- Features relevant to a single class stay in class

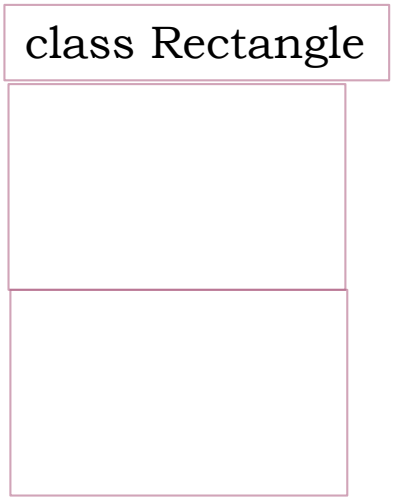


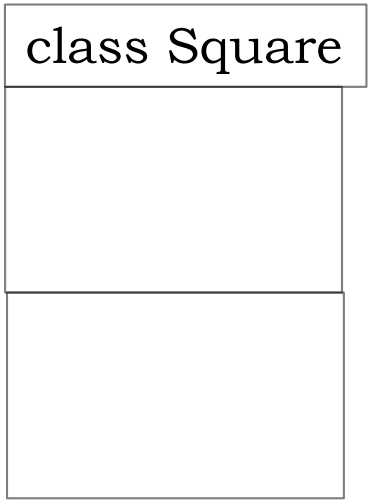
Let's design the classes with UML.

- What instance variables we need for a Circle? That is, what are the basic pieces of information to keep about a Circle?
- How about a Rectangle?
- How about a Square?

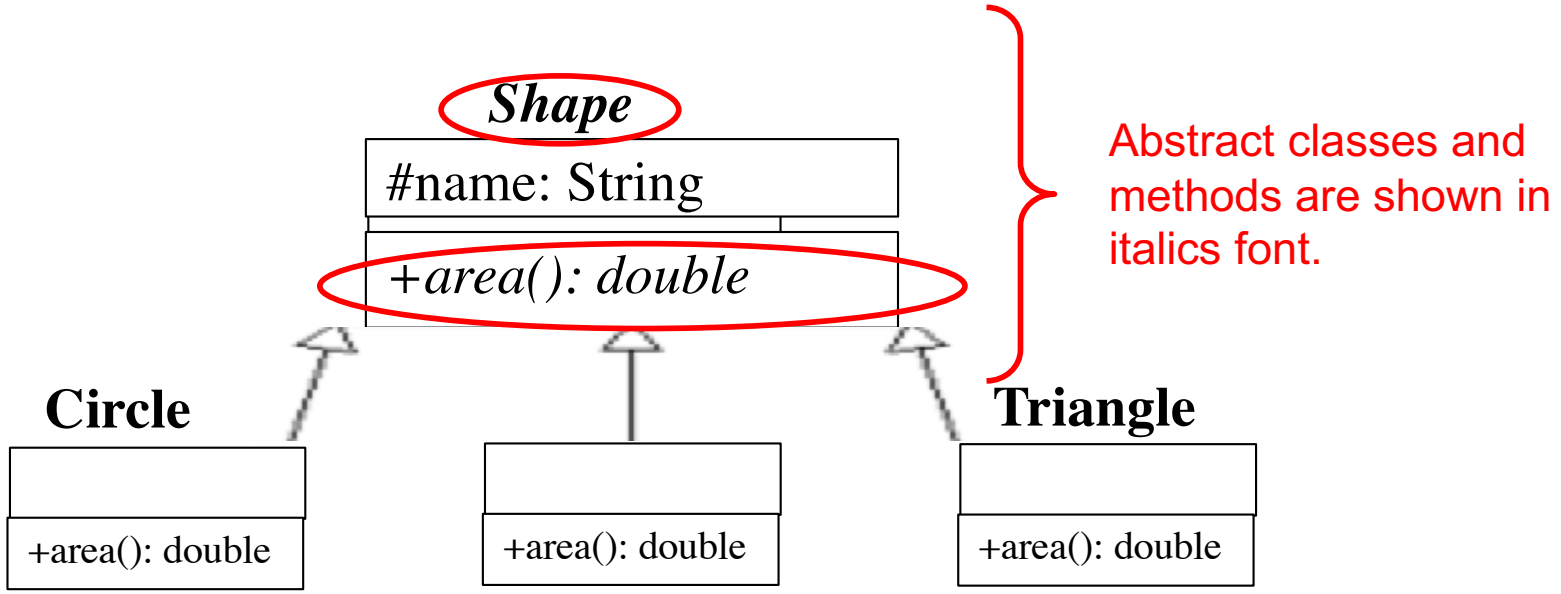
- What methods would make sense to have in a Circle?
- How about a Rectangle?
- How about a Square?



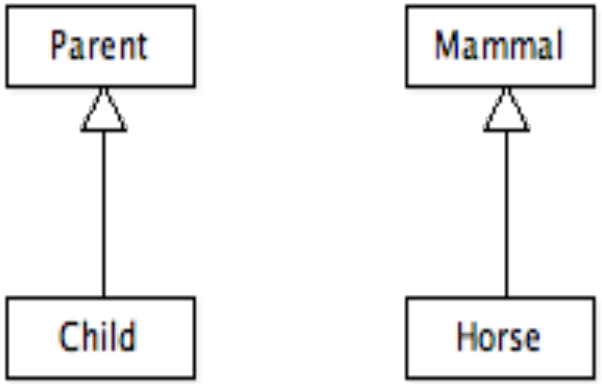




Need for Abstract Classes



Inheritance

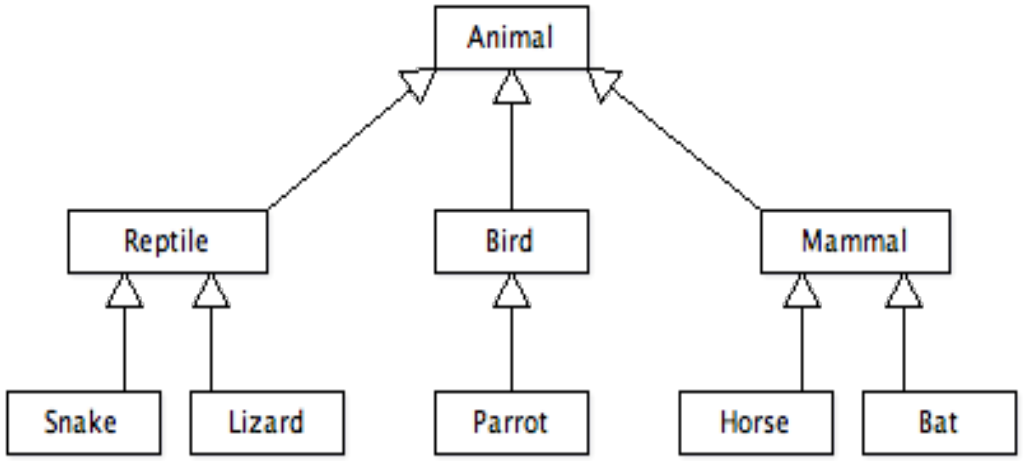


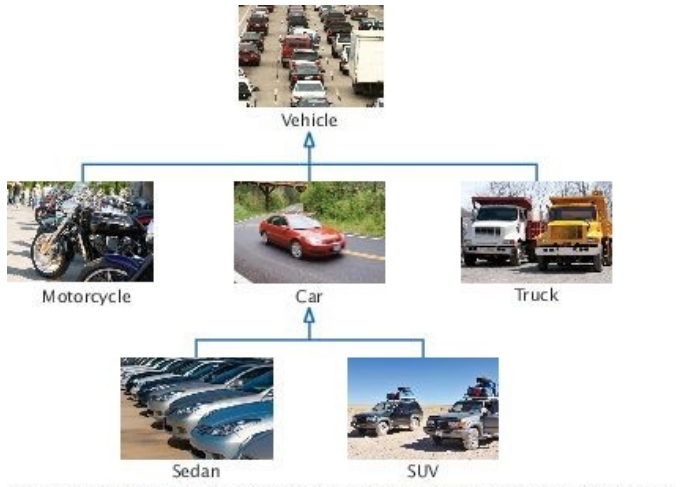
- Inheritance is a fundamental technique used to
- The child more specific version of parent
- The child **inherits** characteristics of the parent (methods and data defined by the parent class)
- Tailor derived class by **adding** new variables or methods, or by **modifying** the inherited ones
- Keyword is used to establish an inheritance (is-a) relationship

```
class Child extends Parent {
   // class contents
}
```

Class Hierarchies

- A child class can be the parent of another child, forming a *class hierarchy*
- Two children of the same parent are called *siblings*
- Common features should be put as high in the hierarchy as is reasonable
- An inherited member is passed continually down the line
  - Therefore, a child class inherits from all its ancestor classes





Self-Check

Consider classes `Manager` and `Employee`. Which should be the superclass and which should be the subclass?

Answer: Because every manager is an employee but not the other way around, the `Manager` class is more specialized. It is the subclass, and `Employee` is the superclass.

Self-Check

What are the inheritance relationships between classes `BankAccount`, `CheckingAccount`, and `SavingsAccount`?

Answer: `CheckingAccount` and `SavingsAccount` both inherit from the more general class `BankAccount`.

The `protected` Modifier

- A **protected** variable is visible to any class in the same [ ] as the parent class
- The **protected** modifier allows a child class to reference a [ ] or [ ] directly in the parent class
- It provides more **encapsulation** than [ ] visibility, but is not as tightly encapsulated as [ ] visibility

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 keyword `super` can be used to refer to the parent class, including the parent's constructor
- A child's constructor should:
  - Call the parent's constructor as it's first line: `super();`
  - If it does not call `super()`,
    a 0-parameters `super()` constructor will be called anyway!
- The `super` reference can also be used to reference other variables & methods defined in parent's class

Overriding

- What happens when a parent and a child class have methods with the same name?

- A child class can *override* the definition of an inherited method in favor of its own
- A method in the parent class can be invoked explicitly using the super reference, as in:

```
super.toString()
```

Overriding

- A child class can *override* the definition of an inherited method in favor of its own
- A method in the parent class can be invoked explicitly using the super reference, as in:

  super.toString()
- If a method is declared with the **final** modifier, it **cannot** be overridden
- The concept of overriding can be applied to variables and is called *shadowing variables*.
  It is
  Shadowing variables should be **avoided** because it tends to cause unnecessarily confusing code

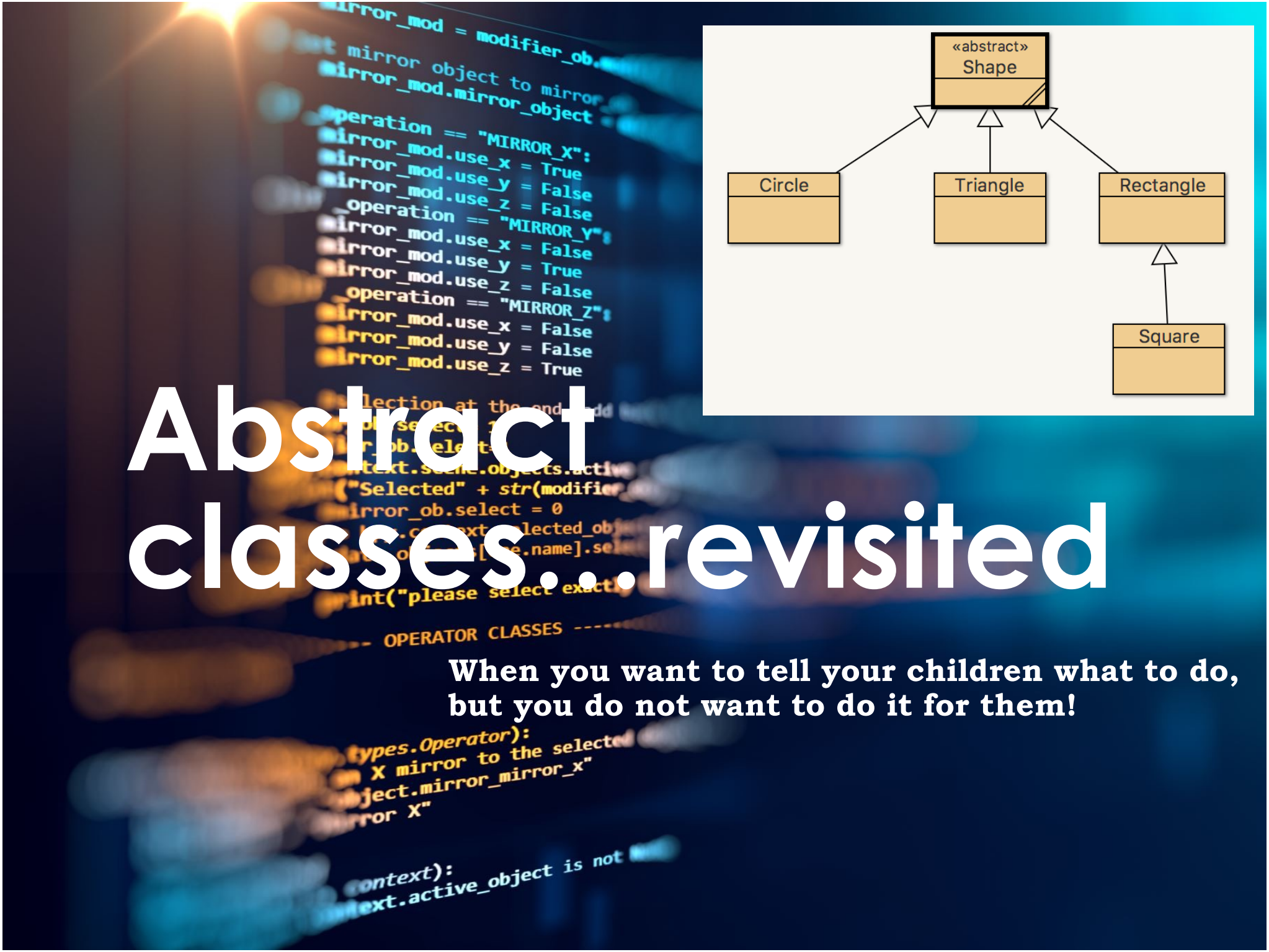
Overloading vs. Overriding

- **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

Self-Check

What happens if in the class `Circle.java`,
in the call `super.toString();` we omit the `super.`?

Answer: The method toString() calls itself! An infinite loop!



Abstract classes...revisited

When you want to tell your children what to do, but you do not want to do it for them!

Abstract Classes

- An *abstract class* is a **placeholder** in a class hierarchy that represents a **generic** concept
- An abstract class **cannot** be instantiated
- To declare a class as abstract:

```
public abstract class Shape
{
   // contents

}
```

- Abstract classes are an important element of software design: they allow us to establish **common** elements in a hierarchy that are too **generic** to instantiate

Abstract Classes: Rules

- An abstract class often contains abstract methods with **no definitions**
  - The abstract modifier **must** be applied to each abstract method
- 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 the class abstract
- 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 Object Class

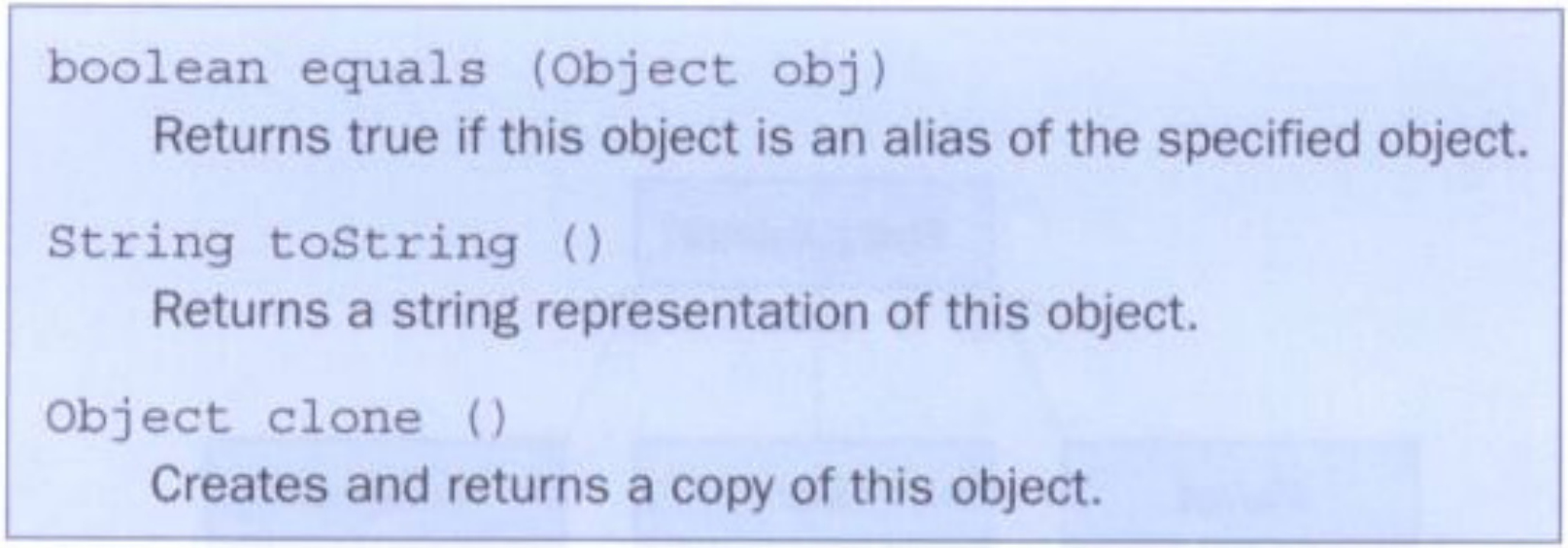
The Mother of all Classes

The `Object` Class

- A class called Object is defined in the java.lang package of the Java standard class library
- All classes are derived from the Object class
- If a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the Object class
- Therefore, the Object class is the ultimate root of all class hierarchies

The Object Class

- Some methods of the `Object` class:



The `Object` Class Methods

- The `Object` class contains a few useful methods,
  which are inherited by all classes
- I.e., the `toString()` method is defined in the `Object` class
- Every time we define the `toString` method,
  we are actually **overriding** an inherited definition
- The `toString` method in the `Object` class is defined to return a string that contains the name of the objects class along with some other information
- Also in `Object` :
- `equals()` returns T if and only if _____
- `clone()` returns _____

The Object Class

- The `equals` method of the `Object` class returns true if two references are aliases
- We can override `equals` in any class to define equality in some more appropriate way
- As we've seen, the `String` class defines the `equals` method to return true if two `String` objects contain the same characters
- The designers of the `String` class have overridden the `equals` method inherited from `Object` in favor of a more useful version

Overriding the equals Method

© Ken Brown/iStockphoto.

`equals` method checks whether two objects have the same content:

```
if (stamp1.equals(stamp2)) . . .
   // Contents are the same
```

== operator tests whether two references are identical - referring to the same object:

```
if (stamp1 == stamp2) . . .
   // Objects are the same
```

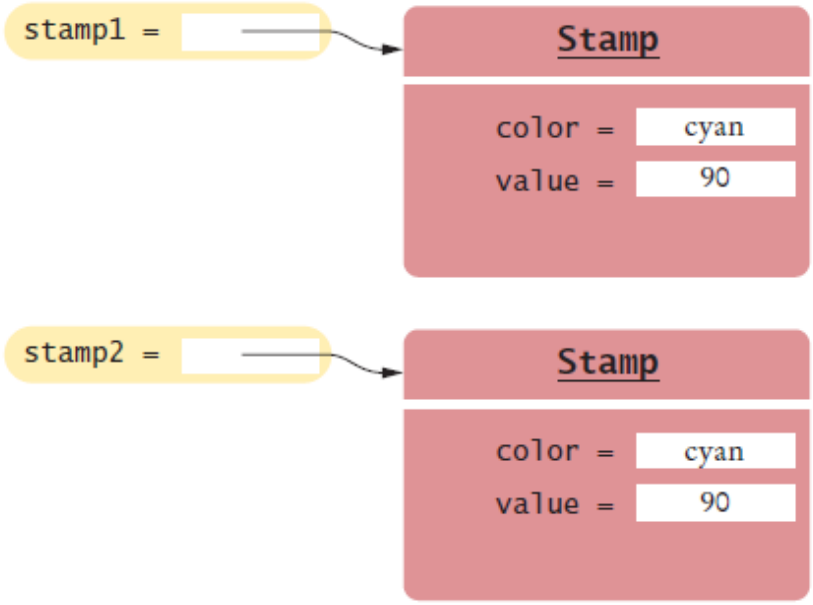


Figure 10 Two References to Equal Objects

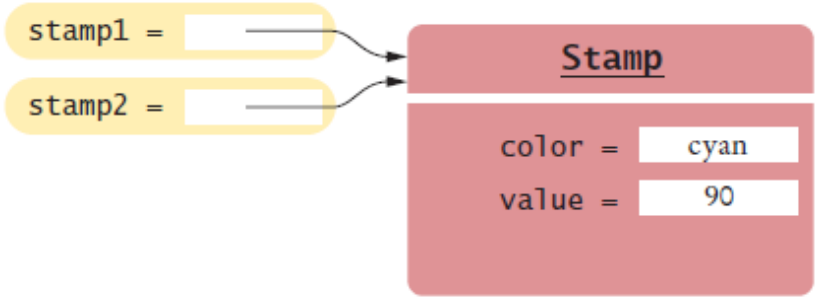


Figure 11 Two References to the Same Object

Overriding the equals Method

- To implement the equals method for a `Stamp` class -

  Override the `equals` method of the `Object` class:

```
public class Stamp
{
   private String
   color;  private
   int value;
   . . .
            public boolean equals(Object otherObject)
            {
                        ...
            }
. . .
}
```

- Cannot change parameter type of the `equals` method - it must be Object
- Cast the parameter variable to the class `Stamp` instead:

```
Stamp other = (Stamp) otherObject;
```