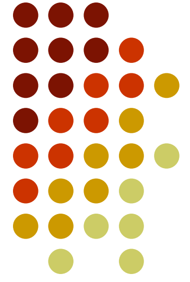
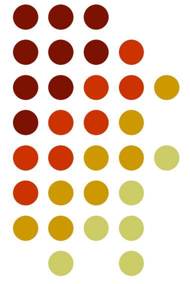
Inheritance and object compatibility



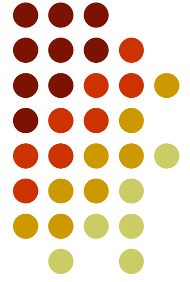
- Object type compatibility
  - An instance of a subclass can be used instead of an instance of the superclass, but not the other way around
- Examples:
  - reference/pointer can hold an object of a subclass
  - subclass can be passed to an function requiring superclass reference/pointer

Inheritance: is-a Relationships



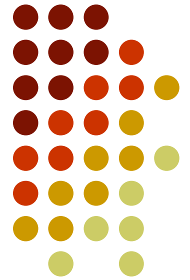
- **Inheritance** should only be used when an **is-a** relationship exists between the base and the derived class e.g.
  - A Ball is a Sphere, so a Ball class could be derived from a Sphere class, however
  - A Color is not a Ball (!!) so a Color class should not be derived from a Ball class (or vice versa)
- In programming terms a derived class should be **type-compatible** with all of its ancestor classes
  - That is, an instance of a derived class can be used instead of an instance of the base class

When to use inheritance?



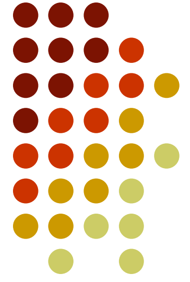
- An important principle in software design for writing code that is easy to maintain and re-use is the **Open-Closed** principle:
  - Software entities like classes, modules and functions should be *open* for extension but *closed* for modifications
  - In terms of classes, this means that creating a subclass should not necessitate changing the superclass
- Any subclass should comply with the **(Liskov) Substitution Principle** otherwise it may violate the Open-Closed Principle
  - Reference: http://www.objectmentor.com/resources/articles/lsp.pdf

(Liskov) Substitution Principle



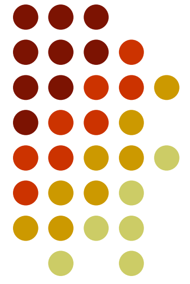
- “The supertype’s **behavior** must be supported by the subtypes: subtype objects can be substituted for supertype objects without affecting the behavior of the using code.”
- i.e., functions that use pointers/references to base class objects must be able to use objects of derived classes without knowing it.
- Example: Rectangle and Square classes

When to use inheritance?



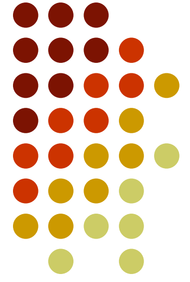
- The key is that an **is-a** relationship should apply to the behavior of objects, not to the real-world entities that the objects are modeling
  - A square is a rectangle but
  - The *behavior* of a `Square` object is not consistent with the behavior of a `Rectangle` object!
- The test, therefore, is that a subclass' public behavior should conform to the expected behavior of its base class
  - This is important because a client program may depend (or expect) that this conformity exists

Polymorphism



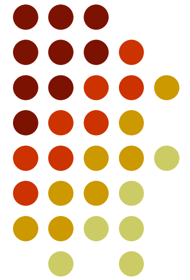
- Polymorphism means **many forms**
- A Subclass definition can include a redefinition of a superclass method
  - The subclass method then **overrides** the superclass method
    - If the calling object is a subclass object, the subclass' version of the method is used
  - This is useful if the subclass is required to perform the same action but in a different way
- Note that overriding is not the same as **overloading**
  - An overloaded method is one where there is another method with the same name, **but a different parameter list**

Dynamic Binding



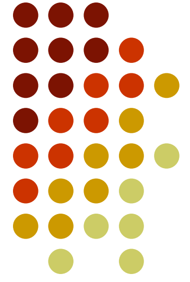
- The correct version of an overridden method is decided at execution time, not at compilation time, based on the type to which a pointer refers
  - This is known as **dynamic binding** or **late binding**
- This allows a superclass reference to be used to refer to a subclass object while still eliciting the appropriate behavior
  - which method is chosen is based on the object type rather than the pointer/reference type
  - this allows the old code (code of the superclass) to call new code (code of the subclass)!

Java and Dynamic Binding



- In Java, **all** object variables are **references** to objects and **all** methods have dynamic binding
- A base class reference variable that refers to a subclass object will use the subclass methods
  - But does not have access to subclass methods that do not exist in the base class
- Containers that store a base class can be used to store subclass objects
  - When retrieving objects from such a container the subclass specific methods can be accessed by first casting the object to a subclass reference
- Note that all Java classes are subclasses of **`Object`**

C++ and Dynamic Binding



- Unlike Java, C++ object variables are not references/pointers unless explicitly declared so, therefore a C++ object variable uses static memory
- Dynamic binding can happen only for pointers and references to objects (which method to call will be decided base on the type of object not on type of pointer or reference).
- However, dynamic binding is not automatically used as in Java!
  - Unless the method is declared `virtual` the static binding is used (i.e., depends on type of pointer or reference)