Digression: Generic vs Particular

Consider a tool that lets a user draw shapes on a canvas: circles, squares, rectangles, ovals, etc.

Suppose the design of such a tool has reached a stage where the following classes have been identified:

- Canvas — the ‘drawing area’
- Color — to represent different colours for shapes
- AbstractShape — to represent the drawn shapes, with subclasses:
  - Circle
  - Square, etc.

Generic vs Particular

The AbstractShape class is to have the following fields:

- Color color
- Point anchorPoint

where anchorPoint represents some location representing the position of the shape on the canvas
this might be the centre of a Circle, the top-left corner of a Square, etc.

Generic vs Particular

The AbstractShape class is to have the following methods:

- void move(int dx, int dy) — to move a shape on the canvas
- void draw(Canvas c) — to actually draw the shape on the canvas

Generic vs Particular

The tool will store all the drawn shapes in an array:

```
AbstractShape[] shapes;
```

Whenever necessary (e.g., refreshing a screen), all the shapes on the canvas can be drawn by

```
for (int i=0; i < shapes.length; i++) {
  shapes[i].draw(theCanvas);
}
```

Thus all the shapes can be drawn in a *generic* way.

Generic vs Particular

However, actually drawing the shapes is *not* generic:

the code for drawing a Circle will be very different from the code for drawing a Square, and so on.

(Sound familiar?)

We might include a ‘dummy’ method in AbstractShape, which is *overridden* in the subclasses Circle, Square, etc.

Abstract Shapes

```
public class AbstractShape {
  private Color color;
  private Point anchorPoint;
  public void move(int dx, int dy) {
    anchorPoint.move(dx,dy);
  }
  public void draw(Canvas c) { }
}
```

Circles

This dummy method will be overridden in the subclasses. E.g.,

```
public class Circle extends AbstractShape {
  private int radius;
  public void draw(Canvas c) {
    // code to draw the circle
  }
}
```

Overriding

The method Circle.draw() *overrides* the method AbstractShape.draw()

if shapes[i] happens to be an instance of Circle, then

```
shapes[i].draw(theCanvas);
```

executes the code in Circle.draw().

Similarly, if Square overrides AbstractShape.draw(), and shapes[i] happens to be an instance of Square, then the Square.draw() code is executed.

This run-time determination of which code to execute is sometimes called dynamic binding.

Non-Generic Code

Having different classes, and dynamic binding allows us to avoid horrible code like:

in some not-very-nice class

```
if (shapes[i] instanceof Circle) {
  Circle c = (Circle)(shapes[i]);
  // code to draw the circle
} else if (shapes[i] instanceof Square) {
  Square s = (Square)(shapes[i]);
  // code to draw the square
} else // etc., etc.
```

Abstract Classes

An **abstract class** is a class that contains an *abstract method*.

An **abstract method** is a method declaration with no body; for example:

```
public void draw(Canvas c);
```

If a class contains an abstract method, it must declare itself and the method abstract:

abstract Shapes

```
public abstract class AbstractShape {

  private Color color;
  private Point anchorPoint;

  public void move(int dx, int dy) {

    anchorPoint.move(dx,dy);

  }

  public abstract void draw(Canvas c);
}
```

NB - you cannot create an instance of an abstract type.

‘Concrete’ Subclasses

Just as before, we want the subclasses of AbstractShape to contain the actual code for drawing circles, squares, etc.

```
public class Circle extends AbstractShape {

  private int radius;

  public void draw(Canvas c) {
    // code to draw the circle
  }
}
```

NB — this class is not abstract because it *does* provide a ‘concrete implementation’ of the abstract method draw().

Class Circle is not abstract, so we *can* create instances of Circle.

Polymorphism

The code

```
for (int i=0; i < shapes.length; i++) {
  shapes[i].draw(theCanvas);
}
```

works just as before.

This is an example of how inheritance, abstract classes and dynamic binding allow for **polymorphic code**
i.e., code that works for a number of classes in a generic way.

Remember Interfaces?

Interfaces are *completely abstract* classes, but interfaces and abstract classes are treated differently in Java

- The members of an interface can only be either:
  - constants (‘final’)
  - abstract methods
- the keyword abstract is not used in interfaces
- a class can implement any number of interfaces

  ```
  class DoItAll implements ActionListener,
    MouseListener, ComponentListener {
     ...
  }
  ```

  but can extend only *one* class, abstract or not