Chapter 1 Exercises Solutions

1.1 Varied Answers.

1.2 College courses may build on top of prerequisite knowledge, but teach well defined and systematically prepared information that focuses on specific material.

1.3 Possible points may include:

	top-down	***bottom-up***
Correct programs	Because the overall design is first considered, individual parts should integrate easier, reducing chances of errors in the integration process and improve correctness.	When addressing individual components first, there is less likelihood of overlooking intricacies of the individual components and likely to produce more correct components.
Efficient programs	Integration efficiency should increase.	Individual components should be well thought out and efficient.
General-purpose	Because interaction of components is considered from the beginning future general interaction will most likely also be considered.	Not havign a clear understanding of the total picture can force developers to build components that adapt to the needs of an application or system.
Rapidly developed	Focusing on one area (design, implementation, testing) at a time can improve efficiency.	Making small incremental steps through the cycle reduces the size or likelihood or large redesigns and reimplementation which could greatly increase development time.

1.4 Assertions should be part of the full development cycle. They should be established in the design phase, implemented and turned on and tested in the test phase.

1.5 Comments can initially be established from the criteria in our design phase. Further comments can be created to explain complex or unintuitive implementations. Finally comments can be used to establish correct use or states of components for testing or use of the components. Should defects be found in testing, comments may explain fixes that are added to the components. Used this way for quality assurance, the program's correctness, efficiency, generality, and development speed can be improved.

1.6 The class still compiles and run correctly, the difference is that the initial value for topFace is 0 (the default value for numbers) and not 1.

1.7 Constants by definition cannot be changed and therefore require no protection from manipulation. So in this case, direct access to the variable is appropriate.

1.8 ***System*** is a class. This class has a static variable called ***out*** that references a nonstatic object of class PrintStream which contains a nonstatic method println.

1.9

```
public void setTopFace(int topFace) {
  assert topFace >= 1 && topFace <=6 :
     "topFace must be a number between 1 and 6";
   this.topFace = topFace;
}
```

1.10

```
public boolean getBody()
{
      return body;
}

public int getEyes()
{
```

```
        return eyes;
    }

    public int getFeelers()
    {
        return feelers;
    }

    public boolean getHead()
    {
        return head;
    }

    public int getLegs()
    {
        return legs;
    }

    public boolean getTail()
    {
        return tail;
    }
```

1.11

```
    public void setBody(boolean body)
    {
        this.body = body;
    }

    public void setEyes(int eyes)
    {
        assert eyes >= 0 && eyes <=2 :
              "eyes must be a number between 0 and 2";
        this.eyes = eyes;
    }

    public void setFeelers(int feelers)
    {
        assert feelers >= 0 && feelers <=2 :
              "feelers must be a number between 0 and 2";
        this.feelers  = feelers;
    }

    public void setHead(boolean head)
    {
        this.head  = head;
    }

    public void setLegs (int legs)
    {
        assert legs >= 0 && legs <=6 :
          "legs must be a number between 0 and 6";
        this.legs  = legs;
    }

    public void setTail(boolean tail)
    {
```

```
            this.tail  = tail;
    }
```

1.12

```
public static void main(String[] args) {
        Beetle b = new Beetle();
        System.out.println("Start:\n"+b);
        b.addBody();
        System.out.println("Added Body:\n"+b);
        b.addHead();
        System.out.println("Added Head:\n"+b);
        b.addEye();
        System.out.println("Added Eye:\n"+b);
        b.addEye();
        System.out.println("Added Eye:\n"+b);
        b.addFeeler();
        System.out.println("Added Feeler:\n"+b);
        b.addFeeler();
        System.out.println("Added Feeler:\n"+b);
        b.addLeg();
        System.out.println("Added Leg:\n"+b);
        b.addLeg();
        System.out.println("Added Leg:\n"+b);
        b.addLeg();
        System.out.println("Added Leg:\n"+b);
        b.addLeg();
        System.out.println("Added Leg:\n"+b);
        b.addLeg();
        System.out.println("Added Leg:\n"+b);
        b.addLeg();
        System.out.println("Added Leg:\n"+b);
        b.addTail();
        System.out.println("Added Tail:\n"+b);
}
```

1.13 The only change that is required is in the getTopFace() method. this.topFace should be cast to an int to avoid the ***possible loss of precision*** error.

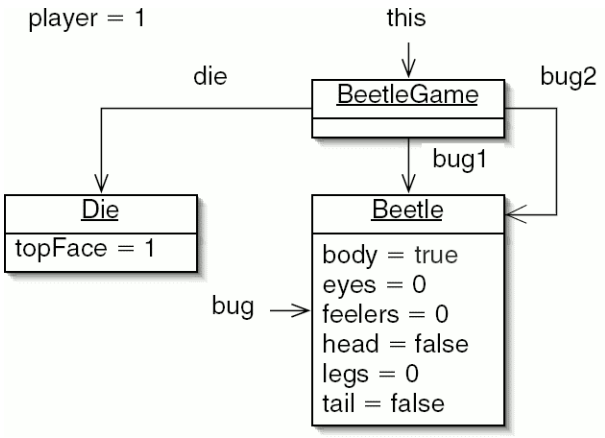
1.14 The Die class does compile but will now longer run on its own. The BeetleGame will still compile and run correctly because it uses the Die class as a component which does not require the Die's main method.

1.15 Places where *this* can be removed:

- The reference to topFace in the Constructor.
- The reference to topFace in the getTopFace() method.
- The reference to topFace in the roll() method.
- The reference to topFace in the toString() method.

1.16 The problem is that we lose control of the manipulation of the variable. In this case because addHead() is not called, there is no guarantee that there is a body prior to adding a head.

1.17 The program will compile but it will not run correctly because both players rolls will be acting on the same instance of Beetle.



1.18 Encapsulation is violated because variables x and y are publicly visible to everyone.

java.awt

Class Point

java.lang.Object
  └ java.awt.geom.Point2D
    └ **java.awt.Point**

All Implemented Interfaces:
Serializable, Cloneable

---

public class **Point**extends Point2Dimplements Serializable

A point representing a location in (x, y) coordinate space, specified in integer precision.

Since:
JDK1.0

See Also:
Serialized Form

Nested Class Summary

Nested classes/interfaces inherited from class java.awt.geom.Point2D
Point2D.Double, Point2D.Float

Field Summary

int	x The *x* coordinate.
int	y The *y* coordinate.

Constructor Summary

Point()	Constructs and initializes a point at the origin (0, 0) of the coordinate space.
Point(int x, int y)	Constructs and initializes a point at the specified (*x*, *y*) location in the coordinate space.
Point(Point p)	Constructs and initializes a point with the same location as the specified `Point` object.

Method Summary

boolean	equals(Object obj) Determines whether or not two points are equal.
Point	getLocation() Returns the location of this point.
double	getX() Returns the X coordinate of the point in double precision.
double	getY() Returns the Y coordinate of the point in double precision.
void	move(int x, int y) Moves this point to the specified location in the (*x*, *y*) coordinate plane.
void	setLocation(double x, double y) Sets the location of this point to the specified double coordinates.

void	setLocation(int x, int y) Changes the point to have the specified location.
void	setLocation(Point p) Sets the location of the point to the specified location.
String	toString() Returns a string representation of this point and its location in the (*x*, *y*) coordinate space.
void	translate(int dx, int dy) Translates this point, at location (*x*, *y*), by dx along the *x* axis and dy along the *y* axis so that it now represents the point (x + dx, y + dy).

Methods inherited from class java.awt.geom.Point2D
clone, distance, distance, distance, distanceSq, distanceSq, distanceSq, hashCode, setLocation

Methods inherited from class java.lang.Object
finalize, getClass, notify, notifyAll, wait, wait, wait