AP® Computer Science AB
2004 Sample Student Responses

AB 3 A1

3. This question involves reasoning about the code from the Marine Biology Simulation case study. A copy of the code is provided as part of this exam.

Consider defining a new type of fish, called `PredatorFish`, that eats other fish. At the beginning of a simulation step, a `PredatorFish` examines the environment location directly in front of it. If a fish of any kind is present, the `PredatorFish` eats that fish, removing it from the environment. Otherwise, if this is the fifth consecutive turn in which the `PredatorFish` has not eaten, it dies. After the attempt to eat, if the `PredatorFish` is still alive, it takes an ordinary turn, acting as an ordinary `Fish` in terms of breeding, moving, and dying.

A `PredatorFish` can be defined by inheriting behavior from the `Fish` class and adding or overriding methods as appropriate.

(a) Write a partial class declaration for `PredatorFish` that includes the heading for the class, any instance variables that must be declared in the class, and a two-parameter constructor that takes the environment and the initial location of the `PredatorFish` as parameters. Do not show other constructors and methods.

```
public class PredatorFish extends Fish {
    private int turnsOfFasting;
    private int fastingLimit;
    public PredatorFish (Environment env, Location loc) {
        super(env, loc);
        turnsOfFasting = 0;
        fastingLimit = 5;
    }
```

GO ON TO THE NEXT PAGE.

A2

(b) Write the new `eat` method for the `PredatorFish` class. This method will remove a `Fish` that is directly in front of the `PredatorFish` if that environment location is not empty. The `PredatorFish` remains in the same location. The `boolean` result that is returned from the `eat` method indicates whether the `PredatorFish` was able to eat.

Complete the `PredatorFish` method `eat` below.

```
// if the PredatorFish is able to eat, the eaten fish is removed and
// true is returned; otherwise, false is returned
protected boolean eat() {
    Environment env = environment(); //faster lookups
    Location oneInFront =        env.getNeighbor(location(), direction());

    Locatable prey = env.objectAt(oneInFront);
    if (prey == null) // no fish there or invalid location
        return false;
    //assume that this Locatable object is a fish
    // insert Jaws theme here
    ((Fish) prey).die();
    return true;
}
```

Part (c) begins on page 16.

(c) One way to implement `PredatorFish` is to override the `act` method. A `PredatorFish` will first eat if possible. If it is the fifth consecutive turn in which the `PredatorFish` has not eaten, the `PredatorFish` dies. Otherwise, it acts as an ordinary `Fish` in terms of breeding, moving, and dying.

You may use any of the accessible methods of the `PredatorFish` class and other public classes from the case study program. Solutions that reimplement functionality provided by these methods, rather than invoking these methods, will not receive full credit. Assume that the `eat` method works correctly, regardless of what you wrote in part (b).

Complete the `PredatorFish` method `act` below.

```
// acts for one step in the simulation
public void act() {
    if (!isInEnv())
        return;
    if (eat())
        turnsOfFasting = 0;
    else {
        turnsOfFasting++;
        if (turnsOfFasting >= fastingLimit) {
            die();
            return;
        }
    }
    super.act();
}
```

AB3 B1

3. This question involves reasoning about the code from the Marine Biology Simulation case study. A copy of the code is provided as part of this exam.

Consider defining a new type of fish, called `PredatorFish`, that eats other fish. At the beginning of a simulation step, a `PredatorFish` examines the environment location directly in front of it. If a fish of any kind is present, the `PredatorFish` eats that fish, removing it from the environment. Otherwise, if this is the fifth consecutive turn in which the `PredatorFish` has not eaten, it dies. After the attempt to eat, if the `PredatorFish` is still alive, it takes an ordinary turn, acting as an ordinary `Fish` in terms of breeding, moving, and dying.

A `PredatorFish` can be defined by inheriting behavior from the `Fish` class and adding or overriding methods as appropriate.

(a) Write a partial class declaration for `PredatorFish` that includes the heading for the class, any instance variables that must be declared in the class, and a two-parameter constructor that takes the environment and the initial location of the `PredatorFish` as parameters. Do not show other constructors and methods.

```
public class PredatorFish extends Fish
{ private int timeSinceEaten;
  public PredatorFish(Environment env, Location loc) {...}

}
```

GO ON TO THE NEXT PAGE.

B2

(b) Write the new eat method for the PredatorFish class. This method will remove a Fish that is directly in front of the PredatorFish if that environment location is not empty. The PredatorFish remains in the same location. The boolean result that is returned from the eat method indicates whether the PredatorFish was able to eat.

Complete the PredatorFish method eat below.

```
// if the PredatorFish is able to eat, the eaten fish is removed and
// true is returned; otherwise, false is returned
protected boolean eat()
{
  Environment env = environment();

  Location inFront = env.getNeighbor(location(), direction());
  if(env.isEmpty(inFront))
      return false;
  else
      env.remove((Locatable) inFront);
      return true;

  }
```

Part (c) begins on page 16.

GO ON TO THE NEXT PAGE.

(c) One way to implement `PredatorFish` is to override the `act` method. A `PredatorFish` will first eat if possible. If it is the fifth consecutive turn in which the `PredatorFish` has not eaten, the `PredatorFish` dies. Otherwise, it acts as an ordinary `Fish` in terms of breeding, moving, and dying.

You may use any of the accessible methods of the `PredatorFish` class and other public classes from the case study program. Solutions that reimplement functionality provided by these methods, rather than invoking these methods, will not receive full credit. Assume that the `eat` method works correctly, regardless of what you wrote in part (b).

Complete the `PredatorFish` method `act` below.

```
// acts for one step in the simulation
public void act()
    if(!isInEnv())
      return;
    boolean boo = eat();
    if( boo)
     {
       timeSinceEaten = 0;
     }
    else if(!boo)
     {
       timeSinceEaten++;
     }
    if(timeSinceEaten == 5)
      die();

    if ( !breed())
      move();
    Random randNumber = RandNumGenerator.getInstance();
    if (randNumGen.nextDouble() < probOfDying)
    if
```

AB3 C1

3. This question involves reasoning about the code from the Marine Biology Simulation case study. A copy of the code is provided as part of this exam.

Consider defining a new type of fish, called `PredatorFish`, that eats other fish. At the beginning of a simulation step, a `PredatorFish` examines the environment location directly in front of it. If a fish of any kind is present, the `PredatorFish` eats that fish, removing it from the environment. Otherwise, if this is the fifth consecutive turn in which the `PredatorFish` has not eaten, it dies. After the attempt to eat, if the `PredatorFish` is still alive, it takes an ordinary turn, acting as an ordinary `Fish` in terms of breeding, moving, and dying.

A `PredatorFish` can be defined by inheriting behavior from the `Fish` class and adding or overriding methods as appropriate.

(a) Write a partial class declaration for `PredatorFish` that includes the heading for the class, any instance variables that must be declared in the class, and a two-parameter constructor that takes the environment and the initial location of the `PredatorFish` as parameters. Do not show other constructors and methods.

```
public class PredatorFish
{
    int die = 0; // integer that keeps track of number of moves without eat

    public PredatorFish (Environment env, Location loc) { }

}
```

GO ON TO THE NEXT PAGE.

(b) Write the new `eat` method for the `PredatorFish` class. This method will remove a `Fish` that is directly in front of the `PredatorFish` if that environment location is not empty. The `PredatorFish` remains in the same location. The `boolean` result that is returned from the `eat` method indicates whether the `PredatorFish` was able to eat.

— Complete the `PredatorFish` method `eat` below.

```
// if the PredatorFish is able to eat, the eaten fish is removed and
// true is returned; otherwise, false is returned
protected boolean eat()
{
    boolean ate = false;
    Location front = env.getNeighbor(location(), direction());

    if (!env.isEmpty(front))
    {
        env.remove(env.objectAt(front));
        ate = true;
    }

    return ate;
}
```

Part (c) begins on page 16.

C3

(c) One way to implement `PredatorFish` is to override the `act` method. A `PredatorFish` will first eat if possible. If it is the fifth consecutive turn in which the `PredatorFish` has not eaten, the `PredatorFish` dies. Otherwise, it acts as an ordinary `Fish` in terms of breeding, moving, and dying.

You may use any of the accessible methods of the `PredatorFish` class and other public classes from the case study program. Solutions that reimplement functionality provided by these methods, rather than invoking these methods, will not receive full credit. Assume that the `eat` method works correctly, regardless of what you wrote in part (b).

Complete the `PredatorFish` method `act` below.

```
// acts for one step in the simulation
public void act()
{
    if (! isInEnv())
        return ;
    if ( ! eat() )
        die++;
    if ( die == 5 )
        die();
    else if (! breed())
        move();
    Random randN = RandNumGenerator.getInstance();
    if (randN.nextDouble() < probOfDying)
        die();
}
```