Creating a 2D Platform Game

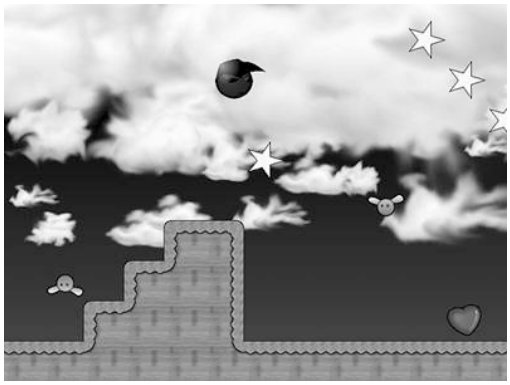
Chapter 5

Content

- Creating a Tile-Based Map
- Collision Detection
- Finishing Things Up and Making It Fast
- Creating an Executable .jar File
- Ideas to Expand the Game
- Summary

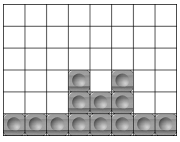
Introduction

- This chapter: tile-based maps, map files, collision detection, power-ups, simple bad guys, and parallax background scrolling



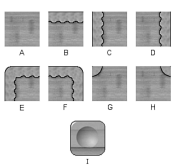
1. Creating a Tile-Based Map

- In a 2D platform game, the map of an entire level, or the game map, is usually several screens wide. Some maps are 20 screens wide; others are 100 or more. As the main character walks across the screen, the map scrolls.
- As you can imagine, using one huge image for the game map is not the best idea in this situation. This would take up so much memory that most machines wouldn't be capable of loading the map. Plus, using a huge image doesn't help define which parts of the map the player can and cannot walk through—in other words, which parts are "solid" and which are "empty."
- Instead of using a huge image for the entire map, you'll create a tile-based map. Tile-based maps break down the map into a grid, as shown in the figure. Each cell in the grid contains either a small tile image or nothing.



Creating a Tile-Based Map (2)

- Tile-based maps are like creating a game with building blocks. Only a few different block colors are used, but you have an unlimited amount of each color.
- The tile map contains references to which image belongs to each cell in the grid. This way, you need only a few small images for the tiles, and you can make the maps as big as you want without worrying too much about memory constraints.
- How big the tiles are is up to you. Most tile-based games use a tile size that is a power of 2, such as 16 or 32. In our game, you'll use a tile size of 64.



Creating a Tile-Based Map (3)

- Tile-based maps also have the nice side effect of being able to easily determine what's "solid" and what's "empty" in a map.
- That way, you know which part of the map the player can jump on, and you can make sure the player can't magically walk through walls.
- You will use this later in this chapter, in the "Collision Detection" section.

Implementing the Tile Map

- The TileMap class contains the tile map you'll use in the game. It holds a two-dimensional array of Image objects that define the tile map. Empty tiles are null.
- Keep in mind that each entry in the Image array isn't a unique object; it's just a reference to an existing object. If one tile is in the map 12 times, the same Image object will be in the array 12 times.
- Object references are only 4 bytes on a 32-bit Java VM, so a one-dimensional 5,000-Image array takes up only about 20KB.

Source code: TileMap.java

Implementing the Tile Map (2)

- Besides the tiles, the TileMap contains the sprites in the game.
- Sprites can be anywhere on the map, not just on tile boundaries.
- The TileMap class also treats the player as a special sprite because you usually want to treat it quite differently from the rest of the sprites.

The Hero The Fly The Grub

The Goal Music Note The Star

Loading Tile Maps

- Tile-based games always have more than one map or level. You'll want an easy way to create multiple maps so that when the player finishes one map, the player can then start the next map.
- You could create maps by calling TileMap's **addTile()** and **addSprite()** methods for every tile and sprite in the game. As you can imagine, this technique isn't very flexible. It makes editing levels far too difficult, and the code itself would not be very pretty to look at.
- Many tile-based games have their own map editors for creating maps. These tile-based editors enable you to visually add tiles and sprites to the game, and are quick and easy to use. They usually store maps in an intermediate map file that the game can parse.

Loading Tile Maps (2)

- Creating a map editor is a bit of overkill in this case. Instead, you'll just define a text-based map file format that can be edited in an everyday text editor.
- Because tile maps are defined on a grid, each character in the text file will be either a tile or a sprite.

```
# Map file for tile-based game
# (Lines that start with '#' are comments)
# The tiles are:
#   (Space) Empty tile
#   A..Z    Tiles A through Z
#   o       Star
#   !       Music Note
#   *       Goal
#   1       Bad Guy 1 (grub)
#   2       Bad Guy 2 (fly)
AD       IIIIIIII              IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
AD                    o o o                 I                  E
AD                   IIIIIII                I                  C
AD                                          I              o o C
AD                  2   !        2                      IIIII  C
AD                    IIIII                                    C
AD          III                       1   1  1          2     C
AD                                1       IIIIIIIIIIII        C
AD      1       EBBBBBBBBBBBBBBF     o o o o o o              * C
AHBBBBBBBBBBBBBBBGAAAAAAAAAAAAAAAAAHBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBG
```


Loading Tile Maps (3)

- Lines that start with # are comments, and all other lines define a row of tiles. The size of the map isn't fixed, so you can make maps bigger by adding more lines or making the lines longer.
- Parsing the map file is easy and basically takes three steps:

```
# Map file for tile-based game
# (Lines that start with '#' are comments)
# The tiles are:
#   (Space) Empty tile
#   A..Z    Tiles A through Z
#   o       Star
#   !       Music Note
#   *       Goal
#   1       Bad Guy 1 (grub)
#   2       Bad Guy 2 (fly)
AD       IIIIIIII              IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
AD                    o o o                 I                  E
AD                   IIIIIII                I                  C
AD                                          I              o o C
AD                  2   !        2                      IIIII  C
AD                    IIIII                                    C
AD          III                       1   1  1          2     C
AD                                1       IIIIIIIIIIII        C
AD      1       EBBBBBBBBBBBBBBF     o o o o o o              * C
AHBBBBBBBBBBBBBBBGAAAAAAAAAAAAAAAAAHBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBG
```

1. Read every line, ignoring commented lines, and put each line in a list.
2. Create a **TileMap** object. The width of the TileMap is the length of the longest line in the list, and the height is the number of lines in the list.
3. Parse each character in each line, and add the appropriate tile or sprite to the map, depending on that character.

- If a character is encountered that is "illegal," the tile is considered to be empty.


```
private TileMap loadMap(String filename) throws IOException {

    ArrayList lines = new ArrayList();
    int width = 0;
    int height = 0;

    // read every line in the text file into the list
    BufferedReader reader = new BufferedReader(
        new FileReader(filename));
    while (true) {
        String line = reader.readLine();
        // no more lines to read
        if (line == null) {
            reader.close();
            break;
        }

        // add every line except for comments
        if (!line.startsWith("#")) {
            lines.add(line);
            width = Math.max(width, line.length());
        }
    }

    // parse the lines to create a TileEngine
    height = lines.size();

    TileMap newMap = new TileMap(width, height);
    for (int y=0; y<height; y++) {
        String line = (String)lines.get(y);
        for (int x=0; x<line.length(); x++) {
            char ch = line.charAt(x);

            // check if the char represents tile A, B,C, etc.
            int tile = ch - 'A';
            if (tile >= 0 && tile < tiles.size()) {
                newMap.setTile(x, y, (Image)tiles.get(tile));
            }

            // check if the char represents a sprite
            else if (ch == 'o') {
                addSprite(newMap, coinSprite, x, y);
            }
            else if (ch == '!') {
                addSprite(newMap, musicSprite, x, y);
            }
```

```
            else if (ch == '*') {
                addSprite(newMap, goalSprite, x, y);
            }
            else if (ch == '1') {
                addSprite(newMap, grubSprite, x, y);
            }
            else if (ch == '2') {
                addSprite(newMap, flySprite, x, y);
            }
        }
    }

    // add the player to the map
    Sprite player = (Sprite)playerSprite.clone();
    player.setX(TileMapRenderer.tilesToPixels(3));
    player.setY(0);
    newMap.setPlayer(player);

    return newMap;
}
```

Loading Tile Maps (5)

- One thing to note is the special case of **adding sprites** to the TileMap. For starters, you need to create a different Sprite object for each sprite in the game. To do this, you can clone sprites from a "host" sprite.
- Second, a sprite might not necessarily be the same size as the tile size. So, in this case, you'll center and bottom-justify each sprite in the tile it's in. All this is taken care of in the addSprite() method

```
private void addSprite(TileMap map,
    Sprite hostSprite, int tileX, int tileY)
{
    if (hostSprite != null) {
        // clone the sprite from the "host"
        Sprite sprite = (Sprite)hostSprite.clone();

        // center the sprite
        sprite.setX(
            TileMapRenderer.tilesToPixels(tileX) +
            (TileMapRenderer.tilesToPixels(1) -
            sprite.getWidth()) / 2);

        // bottom-justify the sprite
        sprite.setY(
            TileMapRenderer.tilesToPixels(tileY + 1) -
            sprite.getHeight());

        // add it to the map
        map.addSprite(sprite);
    }
}
```


Loading Tile Maps (6)

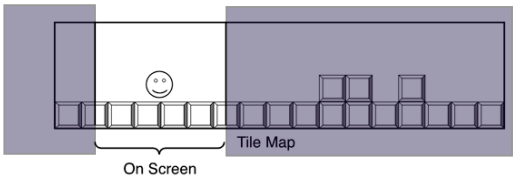
- In earlier demos in the book, the sprite's position was relative to the screen, but in this game, the sprite's position is relative to the tile map. You can use the **TileMapRender.tilesToPixels()** static function to convert tile positions to pixel positions in the map. This function multiplies the number of tiles by the tile size:

```
int pixelSize = numTiles * TILE_SIZE;
```

- This way, sprites can move around to any position in the map and don't have to be justified with the tile boundaries.

Drawing Tile Maps

- Tile maps are much larger than the screen, so only a portion of the map is shown on the screen at a time.
- As the player moves, the map scrolls to keep the player in the middle of the screen.



Drawing Tile Maps (2)

- Therefore, before you draw the tiles, you need to figure out the position of the map onscreen. Start off by keeping the player in the middle of the screen.

```
int offsetX = screenWidth / 2 -
    Math.round(player.getX()) - TILE_SIZE;
```

- This formula assigns **offsetX** the horizontal position of the map onscreen. It's an easy formula, but you also need to make sure that when the player is near the far left or far right edges of the map, the scrolling stops so that the "void" beyond the edges of the map isn't shown. To do this, give the offsetX value a limit:

```
int mapWidth = tilesToPixels(map.getWidth());
offsetX = Math.min(offsetX, 0);
offsetX = Math.max(offsetX, screenWidth - mapWidth);
```


Drawing Tile Maps (3)

- For convenience, also create the offsetY variable for the vertical scroll position. It keeps the map flush with the bottom of the screen, no matter how big the screen is:

```
int offsetY = screenHeight - tilesToPixels(map.getHeight());
```

- We are now ready to draw the tiles. You could just draw every single tile in the map, but instead, you just need to draw the visible tiles. Here, you get the first horizontal tile to draw based on offsetX and then calculate the last horizontal tile to draw based on the width of the screen. Then you draw the visible tiles:

```
int firstTileX = pixelsToTiles(-offsetX);
int lastTileX = firstTileX +
    pixelsToTiles(screenWidth) + 1;
for (int y=0; y<map.getHeight(); y++) {
    for (int x=firstTileX; x <= lastTileX; x++) {
        Image image = map.getTile(x, y);
        if (image != null) {
            g.drawImage(image,
                tilesToPixels(x) + offsetX,
                tilesToPixels(y) + offsetY,
                null);
        }
    }
}
```


Drawing Sprites

- After drawing the tiles, you'll draw the sprites. Because you drew only the visible tiles, let's look into only drawing the visible sprites as well. Here are a few ideas:
  - Partition the sprites into screen-size sections. Draw only the sprites that are in the sections visible on the screen. As the sprites move, make sure they are stored in the appropriate section.
  - Keep the sprites in a list ordered by the sprites' horizontal position, from left to right. Keep track of the first visible sprite in the list, which can change as bad guys die or the screen scrolls. Draw sprites in the list from the first visible sprite until one is found that is not visible. As the sprites move, make sure the list is sorted.
  - Implement the brute-force method of running through every sprite in the list, checking whether it's visible.

Drawing Sprites (2)

- The first two ideas no doubt are useful if there are a lot of sprites in the map. However, in this case, there are not very many sprites in each map, so you can use the brute-force method and check to see if each sprite is visible. Actually, you can just run through the list, drawing every sprite:

```
Iterator i = map.getSprites();
while (i.hasNext()) {
    Sprite sprite = (Sprite)i.next();
    int x = Math.round(sprite.getX()) + offsetX;
    int y = Math.round(sprite.getY()) + offsetY;
    g.drawImage(sprite.getImage(), x, y, null);
}
```

- This way, the graphics engine does the work, checking to see whether each image is visible before drawing it. It's not the most efficient solution, but it works.

Parallax Scrolling

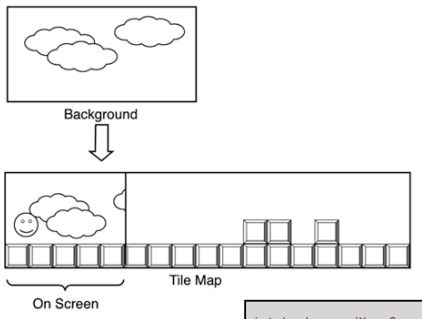
- Now that you've got the tiles and sprites drawn, you just need to draw one more thing: the **background**. When you draw the background, you need to decide how the background is drawn in comparison to the map. Here are a few ways to do this:
  - Keep the background static so it doesn't scroll when the map scrolls.
  - Scroll the background at the same rate as the map.
  - Scroll the background at a slower rate than the map so the background appears to be farther away.

Parallax Scrolling (2)

- The third method is called parallax scrolling and is the method you'll use for the game. Parallax is the apparent change in position of an object when seen from a different point of view. For example, when you're driving in a car and look out the side window, the nearby objects, such as traffic signs, fly by rather quickly, but farther objects, such as mountains, slowly creep by. The farther away an object is, the less it appears to move as you move.
- If you make the background move slower than the map, it will appear farther away and will give the game a bit of perspective.
- As mentioned before, you don't want to use a huge image for the entire map. Likewise, you don't want to use a huge image for the entire background of the map. Because you're using parallax scrolling, you don't need to. You'll create a background image that is two screens wide, and it will scroll from the first screen to the second screen as the map scrolls from the left to the right. This is the key to implementing parallax scrolling in the game.

Parallax Scrolling (3)

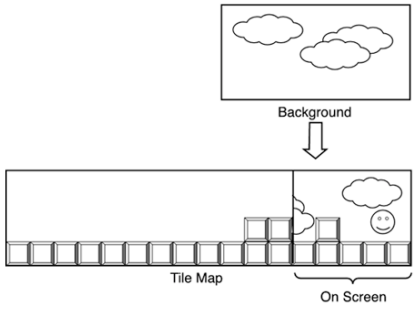
- When the player is on the left part of the map, the leftmost part of the map is shown and the left part of the background is drawn.




```
int backgroundX = 0;
g.drawImage(background, backgroundX, 0, null);
```


Parallax Scrolling (4)

- Likewise, when the player is on the far right side of the map, the rightmost part of the background is shown.




```
int backgroundX = screenWidth - background.getWidth(null);
g.drawImage(background, backgroundX, 0, null);
```


Parallax Scrolling (5)

- Previously you calculated offsetX, which is the position of the map that the screen is drawing, so you just need a formula to convert offsetX to **backgroundX**.
- The range for offsetX is from 0 (left part of map drawn) to **screenWidth–mapWidth** (right part of map drawn). This matches with the range of backgroundX from 0 to **screenWidth–background.getWidth(null)**. So, all you have to do interpolate these two discrete points:

```
int backgroundX = offsetX *
    (screenWidth - background.getWidth(null)) /
    (screenWidth - mapWidth);
g.drawImage(background, backgroundX, 0, null);
```

Parallax Scrolling (6)

- This formula assumes two things, however:
  - The background is wider than the screen.
  - The map is wider than the background.
- One last thing to note is that you don't have to use an image as a background. You could just as easily use another TileMap so that you could be free to create as large a background as you want.
- Also, there doesn't have to be just one scrolling background. There could be two or more, each scrolling at different speeds, with the front layers having transparency so the back layers show through.
- The background image itself could also be tiled. If you do this, make sure the right edge of the background matches up with the left edges, so the background appears seamless.

Power-Ups

- A **power-up** is a sprite that the player can pick up, often giving the player points, extra powers, or the ability to perform some other action.
- The PowerUp class defines the power-ups. It is an abstract class that extends PowerUp, but it contains static inner subclasses for each power-up in the game: the star, the music note, and the goal.

Source code: PowerUp.java


```
package com.brackeen.javagamebook.tilegame.sprites;

import java.lang.reflect.Constructor;
import com.brackeen.javagamebook.graphics.*;

/**
    A PowerUp class is a Sprite that the player can pick up.
*/
public abstract class PowerUp extends Sprite {

    public PowerUp(Animation anim) {
        super(anim);
    }


    public Object clone() {
        // use reflection to create the correct subclass
        Constructor constr = getClass().getConstructors()[0];
        try {
            return constr.newInstance(
                new Object[] {(Animation)anim.clone()});
        }
        catch (Exception ex) {
            // should never happen
            ex.printStackTrace();
            return null;
        }
    }


    /**
        A Star PowerUp. Gives the player points.
    */
    public static class Star extends PowerUp {
        public Star(Animation anim) {
            super(anim);
        }
    }
```

```
    /**
        A Music PowerUp. Changes the game music.
    */
    public static class Music extends PowerUp {
        public Music(Animation anim) {
            super(anim);
        }
    }


    /**
        A Goal PowerUp. Advances to the next map.
    */
    public static class Goal extends PowerUp {
        public Goal(Animation anim) {
            super(anim);
        }
    }
}
```


Power-Ups (3)

- **clone()** method, a way to make many copies of the same sprite.
- PowerUp class contains a generic clone() method that uses reflection to clone the object.
- It selects the first constructor of the object's class and then creates a new instance of that object using that constructor.
- Power-Ups
  - When the player acquires a star, a sound is played, but no other action is taken.
  - When the player acquires a music note, the drum track in the background MIDI music is toggled on or off.
  - Finally, when the player acquires the "goal" power-up, the next map is loaded.

Power-Ups (4)

- All of the power-up actions take place in the collision-detection code. Whenever the player collides with a PowerUp, the **acquirePowerUp()** method is called to determine what to do with it

```
public void acquirePowerUp(PowerUp powerUp) {
    // remove it from the map
    map.removeSprite(powerUp);

    if (powerUp instanceof PowerUp.Star) {
        // do something here, like give the player points
        soundManager.play(prizeSound);
    }
    else if (powerUp instanceof PowerUp.Music) {
        // change the music
        soundManager.play(prizeSound);
        toggleDrumPlayback();
    }
    else if (powerUp instanceof PowerUp.Goal) {
        // advance to next map
        soundManager.play(prizeSound,
            new EchoFilter(2000, .7f), false);
        map = resourceManager.loadNextMap();
    }
}
```


Simple bad guys

- Animations
  - The player and the two baddies are facing only left, mirroring instead of creating more PNGs
  - We used the **AffineTransform** class to mirror an image whenever it is drawn. This time, we'll use the AffineTransform class to create mirror images on startup, saving them to another image.

```
public Image getMirrorImage(Image image) {
    return getScaledImage(image, -1, 1);
}

public Image getFlippedImage(Image image) {
    return getScaledImage(image, 1, -1);
}

private Image getScaledImage(Image image, float x, float y) {

    // set up the transform
    AffineTransform transform = new AffineTransform();
    transform.scale(x, y);
    transform.translate(
        (x-1) * image.getWidth(null) / 2,
        (y-1) * image.getHeight(null) / 2);

    // create a transparent (not translucent) image
    Image newImage = gc.createCompatibleImage(
        image.getWidth(null),
        image.getHeight(null),
        Transparency.BITMASK);

    // draw the transformed image
    Graphics2D g = (Graphics2D)newImage.getGraphics();
    g.drawImage(image, transform, null);
    g.dispose();

    return newImage;
}
```

Creature Class

- Bad guy's animations:
  - Move left, right
  - Dead, facing left, right
- To accommodate this, you need a new type of sprite object that can change its underlying animation whenever it changes direction or dies.

Creature Class (2)

- The Creature class contains several methods to add functionality to the Sprite class:
  - The **wakeUp()** method can be called when the baddie first appears on screen. In this case, it calls *setVelocityX(-getMaxSpeed())* to start the baddie moving, so baddies don't move until you first see them.
  - The **isAlive()** and **isFlying()** methods are convenience methods to check the state of the baddie. Baddies that aren't alive don't hurt the player, and gravity doesn't apply to baddies that are flying.
  - Finally, the methods **collideVertical()** and **collideHorizontal()** are called when the baddie collides with a tile. In the case of a vertical collision, the vertical velocity of the baddie is set to 0. In the case of a horizontal collision, the baddie simply changes direction. That is the extent of the baddies' intelligence.

Source code: Creature.java

Grub

```
package com.brackeen.javagamebook.tilegame.sprites;

import com.brackeen.javagamebook.graphics.Animation;

/**
    A Grub is a Creature that moves slowly on the ground.
*/
public class Grub extends Creature {

    public Grub(Animation left, Animation right,
        Animation deadLeft, Animation deadRight)
    {
        super(left, right, deadLeft, deadRight);
    }


    public float getMaxSpeed() {
        return 0.05f;
    }

}
```


Fly

```
package com.brackeen.javagamebook.tilegame.sprites;

import com.brackeen.javagamebook.graphics.Animation;

/**
    A Fly is a Creature that flies slowly in the air.
*/
public class Fly extends Creature {

    public Fly(Animation left, Animation right,
        Animation deadLeft, Animation deadRight)
    {
        super(left, right, deadLeft, deadRight);
    }


    public float getMaxSpeed() {
        return 0.2f;
    }


    public boolean isFlying() {
        return isAlive();
    }

}
```


Player

```
package com.brackeen.javagamebook.tilegame.sprites;

import com.brackeen.javagamebook.graphics.Animation;

/**
    The Player.
*/
public class Player extends Creature {

    private static final float JUMP_SPEED = -.95f;

    private boolean onGround;

    public Player(Animation left, Animation right,
        Animation deadLeft, Animation deadRight)
    {
        super(left, right, deadLeft, deadRight);
    }


    public void collideHorizontal() {
        setVelocityX(0);
    }


    public void collideVertical() {
        // check if collided with ground
        if (getVelocityY() > 0) {
            onGround = true;
        }
        setVelocityY(0);
    }


    public void setY(float y) {
        // check if falling
        if (Math.round(y) > Math.round(getY())) {
            onGround = false;
        }
        super.setY(y);
    }
```

```
    public void wakeUp() {
        // do nothing
    }


    /**
        Makes the player jump if the player is on the ground or
        if forceJump is true.
    */
    public void jump(boolean forceJump) {
        if (onGround || forceJump) {
            onGround = false;
            setVelocityY(JUMP_SPEED);
        }
    }


    public float getMaxSpeed() {
        return 0.5f;
    }
}
```


2. Collision Detection

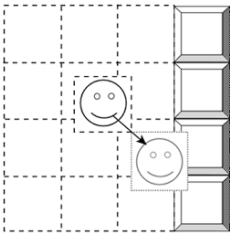
- Break this process down into parts:
  - Detecting a tile collision and
  - Correcting a sprite's position to avoid a collision
- Detecting a Collision
  - Theoretically, the sprite could move across several tiles at once and could be located in up to four different tiles at any one time. Check every tile the sprite is currently in and every sprite the tile is going to be in.

Collision Detection (2)

```
Point pointCache = new Point();

...

/**
    If a collision is found, returns the tile location of the
    collision. Otherwise, returns null.
*/
public Point getTileCollision(Sprite sprite,
        float newX, float newY)
{
    float fromX = Math.min(sprite.getX(), newX);
    float fromY = Math.min(sprite.getY(), newY);
    float toX = Math.max(sprite.getX(), newX);
    float toY = Math.max(sprite.getY(), newY);

    // get the tile locations
    int fromTileX = TileMapRenderer.pixelsToTiles(fromX);
    int fromTileY = TileMapRenderer.pixelsToTiles(fromY);
    int toTileX = TileMapRenderer.pixelsToTiles(
        toX + sprite.getWidth() - 1);
    int toTileY = TileMapRenderer.pixelsToTiles(
        toY + sprite.getHeight() - 1);

    // check each tile for a collision
    for (int x=fromTileX; x<=toTileX; x++) {
        for (int y=fromTileY; y<=toTileY; y++) {
            if (x < 0 || x >= map.getWidth() ||
                map.getTile(x, y) != null)
            {
                // collision found, return the tile
                pointCache.setLocation(x, y);
                return pointCache;
            }
        }
    }

    // no collision found
    return null;
}
```

- **getTileCollision()** treats movement off the left or right edge of the map as a collision, to keep creatures on the map
- This method isn't perfect when a sprite moves across several tiles in between frames (a case for a large amount of time between frames).

Collision Detection (3)

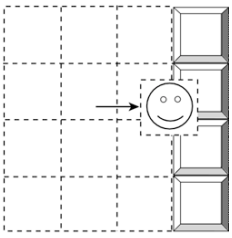


The sprite collides with a wall when it moves too far.

- Handling a collision
  - Correct the sprite's position after a collision is detected.
  - In this case, the sprite moves diagonally and collides with two sprites at the same time. Visually, it looks like an easy fix: Just scoot the sprite over to the left.

Collision Detection (4)

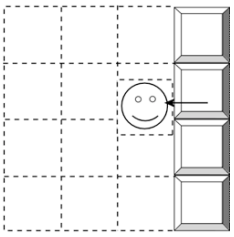
- Handling a collision (cont.)
  - To calculate this, break the movement of the sprite into two parts: ***moving horizontally*** and ***moving vertically***. First, move the sprite horizontally...



First, move the sprite horizontally. A collision is detected

Collision Detection (5)

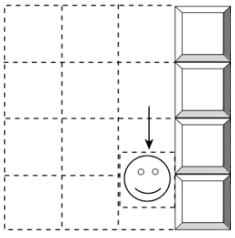
- Handling a collision (cont.)
  - To correct this, just move the sprite back the opposite way it came, lining up the sprite with the edge of the tile



Second, correct the sprite's horizontal position so it doesn't collide with any tiles.

Collision Detection (6)

- Handling a collision (cont.)
  - Now the player has moved horizontally, and its position has been corrected to avoid a collision. Next, apply the same technique for the vertical movement, no collision here...



inally, move the sprite vertically. In this case, there is no collision.

updateCreate method()

```
private void updateCreature(Creature creature, long
elapsedTime) {

    // apply gravity
    if (!creature.isFlying()) {
        creature.setVelocityY(creature.getVelocityY() +
            GRAVITY * elapsedTime);
    }

    // change x
    float dx = creature.getVelocityX();
    float oldX = creature.getX();
    float newX = oldX + dx * elapsedTime;
    Point tile =
        getTileCollision(creature, newX, creature.getY());
    if (tile == null) {
        creature.setX(newX);
    }
    else {
        // line up with the tile boundary
        if (dx > 0) {
            creature.setX(
                TileMapRenderer.tilesToPixels(tile.x) -
                creature.getWidth());
        }
        else if (dx < 0) {
            creature.setX(
                TileMapRenderer.tilesToPixels(tile.x + 1));
        }
        creature.collideHorizontal();
    }
```

```
// change y
    float dy = creature.getVelocityY();
    float oldY = creature.getY();
    float newY = oldY + dy * elapsedTime;
    tile = getTileCollision(creature, creature.getX(), newY);
    if (tile == null) {
        creature.setY(newY);
    }
    else {
        // line up with the tile boundary
        if (dy > 0) {
            creature.setY(
                TileMapRenderer.tilesToPixels(tile.y) -
                creature.getHeight());
        }
        else if (dy < 0) {
            creature.setY(
                TileMapRenderer.tilesToPixels(tile.y + 1));
        }
        creature.collideVertical();
    }
}
```

Collision Detection (7)

- The updateCreature() method also applies gravity to creatures that aren't flying. Gravity always affects the creatures, but if a creature is standing on a tile, the effect isn't visible because the creature collides with the tile it is standing on.
- When a collision is detected and corrected for, the creature's collideVertical() or collideHorizontal() methods are called. Usually these methods change or halt the velocity of the creature so the collision won't happen again any time soon.

Collision Detection (8)

- Next, you need to detect when the player **collides with other sprites**, such as power-ups and bad guys. In this game, you'll ignore collisions between creatures and just detect collisions with the player.
- This is simply a matter of seeing whether the player's sprite boundary intersects with another sprite's boundary.

```
public boolean isCollision(Sprite s1, Sprite s2) {
    // if the Sprites are the same, return false.
    if (s1 == s2) {
        return false;
    }
    // if one of the Sprites is a dead Creature, return false
    if (s1 instanceof Creature && !((Creature)s1).isAlive())
{
        return false;
    }
    if (s2 instanceof Creature && !((Creature)s2).isAlive())
{
        return false;
    }

    // get the pixel location of the Sprites
    int s1x = Math.round(s1.getX());
    int s1y = Math.round(s1.getY());
    int s2x = Math.round(s2.getX());
    int s2y = Math.round(s2.getY());

    // check if the two Sprites' boundaries intersect
    return (s1x < s2x + s2.getWidth() &&
        s2x < s1x + s1.getWidth() &&
        s1y < s2y + s2.getHeight() &&
        s2y < s1y + s1.getHeight());
}
```


Collision Detection (9)

- TileMap contains all the sprites in a list, you can just run through the list checking every sprite to see if it collided with the player

```
public Sprite getSpriteCollision(Sprite sprite) {

    // run through the list of Sprites
    Iterator i = map.getSprites();
    while (i.hasNext()) {
        Sprite otherSprite = (Sprite)i.next();
        if (isCollision(sprite, otherSprite)) {
            // collision found, return the Sprite
            return otherSprite;
        }
    }

    // no collision found
    return null;
}
```


Collision Detection (10)

- What happens when a collision with a sprite is made is entirely up to you. If the sprite is a power-up, you can just give the player points, play a sound, or do whatever else the power-up is supposed to do.
- If the sprite is a bad guy, you can either kill the bad guy or kill the player. In this case, you kill the creature if the player falls or jumps on it, or, in other words, when the vertical movement of the player is increasing:

```
boolean canKill = (oldY < player.getY());
```

- In all other cases, such as when the player just walks up and touches a bad guy, the player dies.
- Future, you might want to add other ways to kill a bad guy, such as with an "invincible" power-up or by pushing a creature off a cliff.

3. Finishing Things Up and Making It Fast

- Now you have just about everything you need for the game. Of course, some of the basics, such as keyboard input, sound, and music, are in the game, too, but these functions exist in classes not listed in this chapter. Besides the classes already listed in this chapter, there are three other classes:
  - GameManager Handles keyboard input, updates sprites, provides collision detection, and plays sound and music.
  - TileMapRenderer Draws the tile map, parallax background, and sprites.
  - ResourceManager Loads images, creates animations and sprites, and loads levels.

Creating an Executable .jar File

- Finally, when you're ready to pass out your game to friends, the last thing you want is to give them arcane instructions on how to run the code. Telling another programmer to type something like this at the command line might be okay:
- *java com.brackeen.javagamebook.tilegame.GameManager*
- A good idea to make it easier is to create an executable .jar file. With an executable .jar file, all the user has to do is double-click the .jar file, and the game starts right up.
- If you're unfamiliar with what a .jar file is, it's a Java archive file—basically just a container for a group of classes. All the classes for your game are stored in the .jar file, which is usually compressed.

Creating an Executable .jar File (2)

- To make the .jar file executable, you specify which class to run in the .jar's manifest file. The manifest file is called META-INF/MANIFEST.MF inside the .jar. For this platform game, the manifest file looks something like this:
- *Manifest-Version: 1.0*
  Main-Class: com.brackeen.javagamebook.tilegame.GameManager
- When you double-click a .jar file, the Java VM starts up and looks at the .jar file's manifest. If the Main-Class attribute is found, it runs the specified class. If there is no manifest or the Main-Class attribute doesn't exist, the VM just pretends that nothing happened and exits.
- If you're using the jar tool to create .jar files, first create a manifest file in a text editor and then use the -m option to add the manifest file to your .jar.

5. Ideas to Expand the Game

- The tile game isn't perfect, and there are lots of ways to make it better. First, a few problems could be fixed:
  - On Windows machines, the granularity of the system timer isn't accurate enough for smooth scrolling. You'll create a workaround for this in Chapter 16, "Optimization Techniques."
  - Using sprite bounds for collision detection isn't accurate enough. Smaller collision bounds could be used instead. We present more ideas for this problem in Chapter 11.
  - Also, you could use better collision handling for times when the sprite travels across large distances between frames. You'll implement this in Chapter 11 as well.

Ideas to Expand the Game (2)

- Furthermore, just by adding levels and more tile images, you could make the gameplay last much longer. You could also tweak the engine to make it faster and taller, with weaker gravity, or whatever suits your desires. Here are some other ideas that you could add to it:
  - Add interactive tiles, such as being able to break tiles or open doors. This could add a puzzle element to the game.
  - Add "walk-through" tiles that have a graphic but aren't solid. This could be useful for map decoration.
  - Add more than one parallax-scrolling background, such as a transparent background of trees or mountains.
  - Use a different sprite animation when the player is jumping.

Ideas to Expand the Game (2)

- Here are some other ideas that you could add to it: (cont.)
  - Create a more destructive environment by adding falling rocks or bottomless pits.
  - Give the player variable-height jumping and running acceleration. The longer the jump key is held down, the higher the player jumps. Likewise, the longer the player moves, the faster the player goes.
  - Add creature-to-creature collisions so that bad guys bump into one another instead of walk through each other.
  - Use 3D sound to make the fly buzz when it gets closer to you, or make all sounds echo as if you're in a cave.
  - Provide a better map-to-map transition, such as summing up the stars collected for the previous map and providing a "get ready" screen before the next map.
  - Add more standard game options, such as pausing, key configuration, and menus.
  - Add more bad guys, more graphics, more levels—more, more, more!

Summary

- It's only the fifth chapter of the book, but already you've created a fun 2D platform game with
  - a tile-based map
  - collision detection
  - bad guys
  - power-ups, and
  - parallax scrolling.